ESCUELA POLITÉCNICA SUPERIOR

DEPARTAMENTO DE ELECTRÓNICA

Arquitectura basada en FPGAs para la detección de objetos en movimiento, utilizando visión computacional y técnicas PCA.

Autor:

Ignacio Bravo Muñoz

Directores:

Dr. D. José Luis Lázaro Galilea

Dr. D. Manuel Mazo Quintas

Tesis doctoral

2007

RESUMEN

En esta tesis se plantea una nueva propuesta para la detección de objetos en movimiento dentro de una escena, mediante una plataforma de propósito específico construida en base a una FPGA (*Field Programmable Gate Array*). Este dispositivo además de encargarse de la captación y gestión de las imágenes en blanco y negro procedentes de un sensor CMOS, de resolución 1.2M píxeles, realiza la detección de nuevos objetos presentes en la escena mediante la aplicación de la técnica PCA (Análisis de las Componentes Principales).

Los problemas más importantes que se plantean en una solución como la descrita en esta tesis son los relacionados con la implementación de la técnica PCA de forma íntegra en una FPGA. El elevado número de operaciones que demanda la técnica PCA conlleva elevados tiempos de ejecución si se utilizan plataformas de procesamiento convencionales, normalmente de tipo secuencial. Por esta razón se ha desencadenado la adaptación de la técnica PCA para hardware reconfigurable. Para ello, se ha paralelizado la ejecución de las diferentes fases que forman el algoritmo PCA: cálculo de la matriz de covarianza, cálculo de autovectores, proyección y recuperación de imágenes al y desde el espacio transformado y detección de nuevos objetos. Con la solución propuesta e implementada en esta tesis se ha conseguido un sistema segmentado que alcanza un alto ratio de imágenes procesadas por segundo (aproximadamente 120 imágenes/sg).

Dentro del algoritmo PCA se deben realizar cuatro tipos diferentes de multiplicaciones de matrices (cálculo de la matriz de covarianza, obtención de la matriz de transformación, proyección y recuperación al/desde el espacio transformado) por lo que ha sido necesario la implementación de una nueva propuesta para la multiplicación de matrices en FPGAs. Concretamente el sistema desarrollado en esta tesis está construido en base a un *array* semi-sistólico de propósito específico, el cual posee una estructura modular que permite su extensión y reutilización para los distintos productos de matrices que requiere la técnica PCA. Aunque las dimensiones de las matrices a multiplicar son diferentes, debido a la estructura regular del array, éste adapta su ejecución en función del producto a realizar.

El cálculo de autovalores y autovectores es una de las operaciones que demanda mayor carga computacional dentro de la técnica PCA, condicionando notablemente el tiempo total consumido en la ejecución del algoritmo PCA. Para resolver el cálculo de los autovalores y autovectores en la FPGA, se ha desarrollado un novedoso y eficiente método basado en el algoritmo de Jacobi. La nueva propuesta realizada consume muy pocos recursos internos de la FPGA en comparación con otras alternativas desarrolladas anteriormente, sin apenas penalizar su tiempo de ejecución. El tiempo de ejecución del mismo es claramente inferior al empleado por un PC de altas prestaciones tal y como se justifica en esta tesis. Otra

importante aportación de esta nueva alternativa es la facilidad para la expansión y adaptación del cálculo de autovalores y autovectores a matrices de diferentes tamaños sin apenas incrementar el número de recursos internos. Para resolver las diferentes operaciones trigonométricas que aparecen en el cálculo de autovalores y autovectores, el sistema posee dos unidades CORDIC que resuelven dichas operaciones. Con idea de dotar al nuevo sistema de la mayor eficiencia posible se ha diseñado un módulo CORDIC de propósito específico que permite trabajar en coordendas circulares tanto para el modo vectorización como para el de rotación.

Para la detección de los nuevos objetos presentes en la escena captada, a partir de la información del mapa de distancias entre la imagen actual y la recuperada del espacio transformado, se propone una nueva alternativa basada en la construcción de un histograma de los máximos de cada una de las columnas del mapa de distancias. Gracias a este histograma se detecta claramente la aparición de nuevos objetos dentro de la escena mediante la umbralización de dicho mapa. El umbral necesario para cada nueva imagen es calculado dinámicamente a partir de la información aportada por el histograma ya que los nuevos objetos se manifiestan en éste mediante la aparición de un nuevo lóbulo. A diferencia de otras propuestas realizadas previamente, esta alternativa alcanza una alta eficiencia en una FPGA.

Para conseguir el diseño óptimo, tanto desde el punto de vista de exactitud como de velocidad, ha sido necesario encontrar el tamaño óptimo de los datos manejados por la FPGA. Para lograr esto, se ha realizado un estudio detallado de las diferentes posibilidades concluyendo que para una implementación de 18 bits codificados en coma fija se consigue una alta exactitud con un consumo moderado de recursos internos de la FPGA.

Todo el diseño implementado en la FPGA ha sido codificado en VHDL, sin emplear ningún *core* comercial. Esto permite la portabilidad del sistema diseñado a cualquier FPGA.

La plataforma hardware desarrollada en esta tesis está basada en un sensor CMOS de alta velocidad (hasta 500 imágenes/sg con la máxima resolución) cuyo interfaz de control es configurable desde un dispositivo externo. Además la plataforma dispone de un banco de memoria externa de tipo SDRAM, que es gestionado desde la misma FPGA, permitiendo almacenar en él las imágenes captadas, procesadas o datos temporales. Estos periféricos dotan al sistema de una alta versatilidad y flexibilidad, con lo que su empleo en diferentes algoritmos de visión para FPGAs es factible.

En nuestro caso la FPGA empleada (una XC2VP7 de Xilinx) no dispone de un excesivo número de recursos internos ya que sólo dispone de aproximadamente 5000 *slices*, por lo que la optimización en el código generado ha sido otra de los objetivos principales de esta tesis. Todos los módulos han sido optimizados al máximo tanto desde el punto de vista de recursos consumidos como de la máxima

frecuencia de reloj admisible. Esto ha permitido que toda la FPGA funcione internamente con una frecuencia patrón de 100MHz.

Los resultados reales alcanzados en esta tesis se pueden considerar como excelentes ya que además de conseguir un alto ratio de imágenes procesadas, el sistema desarrollado logra una alta eficiencia a la hora de detectar nuevos objetos dentro de una escena. Así, para el peor de los casos el número de falsos positivos (detección de nuevo objeto cuando no lo hay) asciende a un 17% y el de falsos negativos (omisión de detección de un nuevo objeto) a un 9%. En el mejor de los casos el índice de falsos positivos y falsos negativos es inferior al 1%.

ABSTRACT

This thesis deals with a new proposal for the detection of moving objects inside a scene, by means of a specific-purpose platform based on a FPGA (Field Programmable Gate Array). This device, not only takes charge of the reception and management of binary images coming from a sensor CMOS with a resolution 1.2M pixels, but also carries out the detection of new objects in the scene by applying PCA techniques (Principal Component Analysis).

The most important problems about a solution, as the one described in this thesis, are those related to the implementation of the whole PCA technique in an FPGA. The high number of operations required by PCA techniques implies high computing times if conventional computing platforms are used, typically of sequential type. For this reason the adaptation of PCA technique for hardware reconfigurable has been achieved. The execution of the different phases from PCA algorithm has been parallelized: calculation of the covariance matrix, eigenvetor computing, projection and recovery of images to and from the transformed space and detection of new objects. With the solution proposed and implemented in this thesis, a pipelined system has been designed achieving a high ratio of processed images per second (approximately 120 frames/s).

Inside the PCA algorithm four different types of matrix multiplications should be carried out (calculation of the covariance matrix, obtaining of the transformation matrix, projection and recovery to and from the transformed space), so it has been necessary the implementation of a new proposal for matrix multiplication in FPGAs. The system developed in this thesis is based on a specific-purpose semi-systolic array, which has a modular structure to allow extending and reusing of the different products in matrices required by PCA technique. Although the dimensions of matrices to be multiplied are different, the proposed array can adapt execution according to product to be computed, thanks to its regular structure.

The eigenvalue and eigenvector calculation is one of the computing tasks that requires higher computational load in PCA techniques, notably conditioning the total time consumed in the execution of PCA algorithm. To solve the eigenvalue and eigenvector calculation in the FPGA, a novel and efficient method has been developed based on the Jacobi algorithm. The new proposal requires very few internal resources of the FPGA in comparison with other alternatives previously developed, without significantly increasing the execution time. Its execution time is less than the required by a high-performance PC as is detailed in this thesis. Another important contribution of this new alternative is the easy expansion and adaptation to the eigenvalue and eigenvector calculation of matrices with different sizes without significantly increasing the number of internal resources. In order to solve the different trigonometrical operations appearing in eigenvalue and eigenvector calculation, the system has two CORDIC units. With the purpose of providing the

new system with the largest possible efficiency, a specific-purpose CORDIC module has been designed that allows working in circular coordinates, not only for the vector mode but also for the rotation one.

For the detection of the new objects in the captured scene, starting from the information from the distance map between the current image and the recovered one from the transformed space, a new alternative is proposed, based on the construction of a histogram with the maximum values from every column of the distance map. Thanks to this histogram, the appearance of new objects inside the scene is detected, by thresholding this map. The necessary threshold for every new image is dynamically computed from the information given by the histogram, since the new objects are represented in it by appearing a new lobe. Contrary to other previous proposals, this alternative achieves a high efficiency in a FPGA.

To yield an optimal design, not only from the point of view of accuracy but also from speed, it has been necessary to find the optimal size of data managed by the FPGA. A detailed study of the different possibilities has been carried out, concluding that, for a 18-bit fixed-point implementation, a high accuracy is obtained with a moderate consumption of internal resources in the FPGA.

The overall design implemented in the FPGA has been specified in VHDL, without using any core. This allows the portability of the designed system to any FPGA.

The platform hardware developed in this thesis is based on a high-speed CMOS sensor (up to 500 frames/s with the maximum resolution) whose control interface is configurable from an external device. The platform also has an external SDRAM memory bank managed from the same FPGA, used to store captured images, processed ones or temporary data. These peripherals provide a high versatility and flexibility to the system, so it can be used in different vision algorithms for FPGAs.

In our case, the used FPGA (a Xilinx XC2VP7) does not have many internal resources (approximately 5000 slices), so the optimization of the generated code has been another of the main objectives of this thesis. All the modules have been optimized about required resources, and about the maximum clock frequency. This has allowed the overall FPGA to internally work with a master frequency of 100MHz.

The experimental results reached in this thesis can be considered as excellent, since they show a high ratio of processed images. Furthermore, the developed system achieves a high efficiency when detecting new objects inside a scene. In this way, for the worst case, the number of false positives (detection of new object when it is not) is 17% and the number of false negative (non detection of a new object) is 9%. In the best case, the index of false positives and negatives are below 1%.

"El trabajo que nunca se empieza es el que tarda más en finalizarse"

J. R. R. Tolkien (1892-1973)

"Lo que con mucho trabajo se adquiere, más se ama"

Aristóteles (384 AC-322 AC)

A Silvia y ….

AGRADECIMIENTOS

Cuando se finaliza un trabajo, como es esta tesis, en el que se ha invertido mucho tiempo y esfuerzo, es el momento en el que se evoca a las personas que han colaborado en el desarrollo de la presente. Quizás en los primeros en quién uno piensa son los que han estado en tu "barco" en los últimos meses, pero el barco salió a la mar hace mucho tiempo y en él han estado muchos marineros. Primeramente me gustaría agradecer a Pedro Jiménez, su tiempo y sacrificio. Sin duda alguna, capitán del barco en numerosas ocasiones donde sus geniales ideas han servido muchas veces para poder encontrar la mejor ruta. Gracias por estar tantos ratos de tu tiempo libre ahí, aportando tus ideas y tu experiencia. También quiero agradecer a mis almirantes, José Luis y Manolo, sus sabios consejos y su tiempo. Gracias Manolo por "enseñarme" a escribir y a ti José Luis por estar siempre ahí, animándome y ayudándome.

No me puedo olvidar de Silvia, quién pacientemente siempre ha estado esperando en la orilla a que el barco atracara. Gracias por tu paciencia, por animarme cuando llegaba a casa hundido o cansado de navegar y por sacarme a flote tantas veces.

Gracias a Fernando Méndez por ser el primer tripulante de mi barco. Tus primeros pasos nos sirvieron para poder partir hacia nuestro destino final.

También quiero agradecer al resto de compañeros del pasillo 21 y del 22 que siempre han estado mostrando continuo interés por mi trabajo. Gracias a Raúl y a Álvaro, por poder trabajar con vosotros en el área de las FPGAs, de ambos he aprendido mucho. Muchas gracias también a Alfredo, tu ayuda en las últimas millas ha sido importantísima.

Además de mi barco, hay otros barcos que partieron casi a la vez y que deseo de todo corazón que lleguen a puerto rápidamente. Gracias a Marta, José Luis y a mis compañeros de pasillo Ernesto y Miguel Ángel, por esas charlas improvisadas en las que nos animábamos mutuamente.

Gracias a mis amigos de toda la vida por comprender la importancia que esta tesis tenía para mí y por ello condicionar el desarrollo de muchos fines de semana. Con vosotros comencé mis estudios de primaria, y con vosotros también he acabado esta otra etapa.

Por último y no por ello menos importantes, quiero agradecer su ánimo incondicional al resto de mi familia: Gracias mamá por dejarme estudiar siempre lo que he querido. Gracias Begoña por tu ánimo e interés. Gracias Juan y Felisa por todo vuestro cariño.

Glosario

ADC: Analog Digital Converter.

AMF: Actualización del Modelo de Fondo.

ASIC: Application-Specific Integrated Circuit.

ASMF: Actuallización Selectiva del Modelo de Fondo.

ASTRB: Address Strobe.

BRAM: Block Random Access Memory.

CCM: Custom Computing Machine.

CMOS: Complementary Metal Oxide Semiconductor.

CORDIC: COordinate Rotation DIgital Computer.

CPU: Control Processing Unit.

DFFC: Data-Flow Functional Computer.

DIMM: *Dual In-line Memory Module.*

DP: Dual-Port Memory.

DSP: Digital Signal Processing/Processor.

DSTRB: Data Strobe.

EP: Elemento de Procesado.

EPP: Enhanced Parallel Port.

FIFO: First Input First Output.

FPGA: Field Programmable Gate Array.

FPOA: Field Programmable Operator Arrays.

FSM: Finite State Machine.

IOB: Input Output Block.

IP: Intellectual Property

LUT: Look Up Table.

MAC: Multiply and ACcumulate

MD: Mapa de Distancias.

Mult. Mat.: Multiplicación de Matrices

MUSIC: MUltiple SIgnal Classification.

PC: Personal Computer.

PCA: Principal Component Analysis.

PCI: Peripheral Component Interconnect

PD: Procreator Diagonal.

PND: Procreator No Diagonal.

PV: Procreator de autoVectores.

RAM: Random Access Memory

RMSE: Root Mean Squared Error

ROM: Read Only Memory.

RSR: Right Shift Register.

SDRAM: Synchronous Dynamic Random Access Memory

SIMD: Single Instruction Multiple Data.

SOC: System On Chip.

SPMD: Single Program Multiple Data.

SRAM: Static Random Access Memory

SVD: Single Value Decomposition

USB: Universal Serial Bus.

V2P: Virtex II Pro.

VHDL: Very High Speed Integrated Circuit Hardware Description Language.

XSG: Xilinx System Generator.

Notación

$\mathbf{A}$: Matriz que contiene en sus columnas los vectores $\mathbf{\Phi}_i$.

$\alpha_{i,j}^{(k)}$: Ángulo de rotación generado en $P_{i,j}$ siendo $i = j$ en la iteración k .

B : Factor de promediado (binning).

b : Número de imágenes actualizadas desde la última actualización de la matriz $\mathbf{U}_t$.

$\mathbf{C}$: Matriz de covarianza.

$\mathbf{C}^T$: Matriz de covarianza traspuesta.

c : Número de ventanas en las que se divide $\mathbf{I}_i$ para la ASMF.

$\mathbf{D}$: Matriz $\mathbf{C}^T$ diagonalizada.

Di : Factor de diezmado.

ε : Error de recuperación.

$\varepsilon_{w,i}$: Elemento del mapa de distancias $\mathbf{MD}_{V1}$.

$\varepsilon'_{v_{i,w}}$: Elemento i, w del $\mathbf{MD}_{V2}$, donde ($i, w[\in 1,..., N]$).

$error_i$: Error cometido entre datos en coma fija (FPGA) y coma flotante (MATLAB).

e_{MAX} : Error máximo de $error_i$

e_{SQRT} : Error cuadrático medio de $error_i$.

F : Número de elementos de $\mathbf{S}_T$.

f : Número de intervalos del histograma de los máximos de cada columna.

$\mathbf{\Phi}_j$: Vector que contiene la diferencia entre $\mathbf{I}_j$ y $\mathbf{\Psi}$.

$\hat{\mathbf{\Phi}}_j$: Vector de la imagen $\mathbf{I}_j$ recuperada del espacio transformado.

h : Número total de iteraciones del algoritmo de Jacobi.

$\mathbf{I}$: Matriz identidad.

$\mathbf{I}_{max}$: Vector que contiene los valores medios máximos de las M imágenes.

$\mathbf{I}_{\min}$: Vector que contiene los valores medios mínimos de las M imágenes.

$\mathbf{I}_i$: Vector de la imagen captada $i \in [1, M]$.

$\mathbf{I}_j$: Vector que contiene la imagen sobre la que determinar nuevos objetos.

(k): Iteración actual del algoritmo de Jacobi.

K_m : Factor de corrección de CORDIC.

$\Lambda\upsilon_i$: Valor medio de cada una de las c ventanas.

λ_i : Autovalor i.

M : Número de imágenes captadas para construir la matriz de autovectores.

N^2 : Tamaño imágenes captadas.

n: Tamaño patrón de los datos internos de la FPGA y número de etapas internas de CORDIC.

n_j : Vector norma del autovector $\mathbf{u}_j$ ($j = 1,...,t$) de la matriz U_t.

P: Porcentaje de autovectores significativos.

$P_{i,j}$: Procesador (i,j) del sistema sistólico para autovalores y autovectores.

p : Número de bits empleados para codificar un píxel.

$\mathbf{\Psi}$: Vector media.

$\mathbf{\Omega}$: Vector de $\mathbf{I}_j$ proyectado.

q^2 : Dimensión de la matriz de promediado del mapa de distancias.

$\mathbf{R}\left(\alpha^{(k)}\right)$: Matriz de Rotación o de Givens.

r : Porcentaje de ventanas cuyo $\Lambda\upsilon_i$ no está dentro de los límites.

$\mathbf{S}$: Matriz de la covarianza transpuesta.

$\mathbf{S}_T$: Matriz triangular superior de S.

$\mathbf{S}_{i,j}^{(k)}$: Submatriz de 2x2 elementos de $\mathbf{S}$ donde $i = j = 1,..., M/2$.

$T_{CAPT\_M\_IMGS}$: Tiempo de captación y escritura de las M imágenes en memoria.

T_{CLK} : Periodo de la señal de reloj de la FPGA.

T_{E1} : Tiempo consumido en la FPGA para el cálculo de $\mathbf{\Psi}$ y $\mathbf{A}$.

T_{E2} : Tiempo consumido en la FPGA para generar $\mathbf{S}$.

T_{E3} : Tiempo consumido en la FPGA para generar $\mathbf{V}$.

T_{E4} : Tiempo consumido en la FPGA para obtener $\mathbf{U}_t$.

T_{E5} : Latencia final en la FPGA para obtener las normas de $\mathbf{U}_t$.

$T_{GEN\_WR\_U}$: Tiempo consumido en la FPGA para generar $\mathbf{U}_t$.

Th_{MD} : Umbral (*Threshold*) para la determinación de nuevos objetos sobre el mapa de distancias $\mathbf{MD}_{V1}$.

Th_H : Umbral (*Threshold*) para la determinación de nuevos objetos sobre el mapa de distancias promediado $\mathbf{MD}_{V2}$.

T_{IMAGE} : Tiempo de captación y escritura en memoria de una imagen en la FPGA.

T_{MD} : Tiempo de cómputo en la FPGA del Mapa de Distancias.

T_{OBJ} Tiempo consumido en la FPGA para detectar nuevos objetos desde la captación de $\mathbf{I}_j$.

T_{PCA_TOTAL} : Tiempo consumido por la propuesta de esta tesis una vez captadas las primeras

T_{VECT} : Tiempo consumido por la FPGA en el modo vectorización de CORDIC.

M Imágenes.

t : Último autovalor significativo.

$\mathbf{U}$: Matriz de autovectores de $\mathbf{C}$.

$\mathbf{U}_t$: Matriz de autovectores de $\mathbf{C}$ reducida.

$\mathbf{U}_{tn}$: Matriz de autovectores reducida y normalizada.

u : Valor de cada uno de los f intervalos del histograma de los máximos de las columnas del $\mathbf{MD}_{V2}$.

$\mathbf{u}_i$: Autovector asociado a λ_i .

$\mathbf{V}$: Matriz de autovectores de $\mathbf{S}$.

$\mathbf{V}_t$: Matriz de autovectores $\mathbf{V}$ reducida.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

1.1. ASPECTOS GENERALES.

Uno de los principales objetivos de las aplicaciones de visión artificial, es la descripción de forma automática de una determinada escena. Así, se entiende como descripción, a la identificación y localización de los objetos que la forman en base a sus características [Ratha, 1999]. Frecuentemente, el principal problema de las aplicaciones de visión artificial es el tiempo de ejecución de los algoritmos y el coste de los equipos. La mayor demanda de prestaciones de los algoritmos de procesamiento de imágenes, unido a la mayor resolución espacial, hace que cada vez sean mayores las demandas de carga computacional. Si además se desea trabajar en tiempo real, las exigencias de los tiempos de ejecución de los algoritmos son aún más críticas.

Habitualmente, las plataformas elegidas para realizar estos algoritmos son las basadas en programas secuenciales. Sin embargo, éstas no son las óptimas en muchas aplicaciones desde un punto de vista de rendimiento. Por tanto, se justifica la búsqueda de nuevos sistemas segmentados para procesamiento de imágenes en paralelo. Así, según [Hamid, 1994], en base a las características de computación, la mayoría de los sistemas de visión se pueden dividir en tres niveles tal y como se muestra en la Figura 1.1: en el nivel inferior (bajo nivel), se incluyen operaciones a nivel de píxel, tales como filtrado o detección de bordes. Este nivel se caracteriza por manejar un elevado número de datos (píxeles) y de operaciones sencillas, como sumas y multiplicaciones. El procesado de nivel medio, se caracteriza por realizar

operaciones más complejas de bloques de píxeles. Dentro de este nivel se pueden incluir operaciones de segmentación o etiquetado de regiones. Por último, las tareas pertenecientes al nivel alto, se caracterizan por una alta complejidad en la algoritmia. Dentro de este último nivel, se podrían incluir, por ejemplo, todos los algoritmos de *matching*. Se puede observar en la Figura 1.1, cómo a medida que crece la altura de la pirámide, el volumen de datos a manejar disminuye, incrementándose la complejidad computacional de los algoritmos. La Tabla 1.1 muestra un resumen de las características de cada nivel.

Por tanto, es importante seleccionar el sistema hardware necesario, en función del nivel de complejidad de procesamiento requerido. Esto permitirá explotar al máximo el rendimiento del sistema.

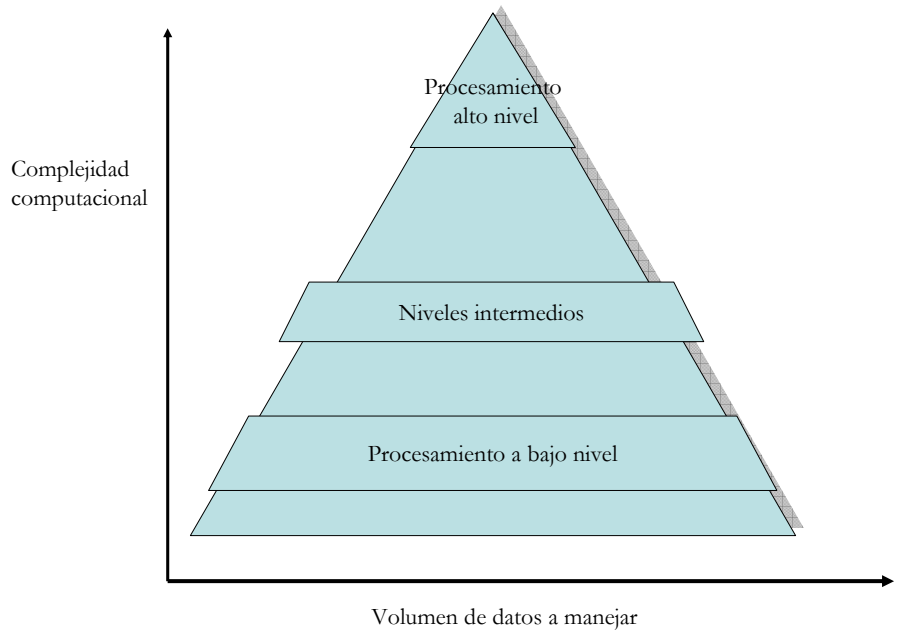


Figura 1.1. Pirámide ilustrativa del volumen de datos vs. complejidad computacional del procesamiento de imágenes [Hamid, 1994].

Tabla 1.1. Características y ejemplos de los diferentes niveles de complejidad computacional [Ratha, 1999].

Nivel	Características	Ejemplos
Bajo	Accesos a píxeles vecinos cercanos, operaciones simples, manejo de gran cantidad de datos	Detección de bordes, filtrado, operaciones morfológicas simples
Medio	Acceso a bloques de píxeles, operaciones más complejas	Transformada de Hough, conectividad.
Alto	Acceso aleatorio, operaciones complejas	Matching

El procesado digital de imágenes requiere habitualmente el manejo de un alto número de operaciones a nivel de bit que se desean ejecutar en el menor tiempo posible. Debido la arquitectura secuencial de los ordenadores convencionales, no se pueden ejecutar concurrentemente muchas operaciones. Por ello, cuando el número de datos a manejar es grande, el rendimiento del sistema es muy bajo. Sin embargo, una plataforma hardware de propósito específico (diseñada expresamente para una aplicación), puede dar excelentes resultados. Así, operaciones relativamente sencillas como filtros, descompresión de imágenes, etc., se realizan con un elevado rendimiento. Por contra, si se desea implementar algoritmos más complejos, éstos hay que reformularlos para explotar las características de la plataforma sobre la que se ejecutará.

1.2. ASPECTOS GENERALES SOBRE DETECCIÓN DE OBJETOS.

Dentro de los algoritmos de alto nivel de visión artificial, se encuentra el método PCA (*Principal Component Analysis*). Éste ha sido aplicado en diferentes campos como la identificación de elementos dentro de una base de datos [Gottumukkal, 2003] o la detección de obstáculos en movimiento [Vázquez, 2004]. Una de las principales ventajas de PCA, al igual que otras transformadas matemáticas, es la reducción de la dimensión de la información, manejando únicamente aquella más significativa y característica de un espacio de datos. En el campo de la visión artificial, el uso de PCA se justifica por la alta correlación existente entre los píxeles de una imagen. De esta forma, se puede reducir drásticamente los datos a manejar de una escena dejando únicamente aquella que contiene la mayor parte de la información.

Teniendo presente las elevadas resoluciones espaciales que demandan muchas aplicaciones prácticas, se desea priorizar la reducción de información redundante así como el procesado de la información a alta velocidad. Cada vez más, los sistemas de procesado de imágenes son más complejos y requieren mayor rapidez de cálculo, necesitando sofisticados sistemas de gestión y control. Así, surge el concepto de sistema de tiempo real como aquel capaz de cumplir con ciertos ítems temporales en una determinada aplicación [Cheng, 2002]. Esta tesis propone la aceleración del proceso de detección de objetos mediante PCA, con respecto a un sistema basado en PC, para aplicaciones en tiempo real, utilizando una plataforma hardware de propósito específico.

Evidentemente PCA no es la única solución dentro de la visión artificial para la detección de objetos, no existiendo una alternativa óptima teniendo cada una sus ventajas e inconvenientes. Dentro de todas las posibles soluciones, esta tesis doctoral se centra en la implementación del algoritmo de Análisis de Componentes Principales (PCA) dado que el grupo de investigación al que pertenece el autor ha obtenido importantes contribuciones con su aplicación. Sirva como ejemplo los resultados obtenidos en [Vázquez, 2006].

Dadas las prestaciones de la información que la visión artificial puede aportar en la detección de objetos, actualmente es una de las alternativas más atractivas. Por

ello, son numerosas las aplicaciones en las que la visión artificial se constituye bien como elemento único o como parte de un sistema multisensorial. El empleo de la visión artificial para la detección de objetos, está presente en diferentes áreas. Sirvan como ejemplo, los sistemas de monitorización de tráfico, detección de obstáculos en vías ferroviarias o detección de obstáculos en entornos interiores.

1.3. ASPECTOS GENERALES SOBRE FPGAS.

En cualquier desarrollo práctico de un sistema electrónico, existen diferentes alternativas para intentar aumentar las prestaciones sin incrementar el precio del mismo y preservando en gran medida los requisitos del diseño. Esto es particularmente aplicable a sistemas de procesado de señal ó imágenes, donde se requiere en algunos casos una alta carga computacional y al mismo tiempo mayor potencia de cálculo. Para esas aplicaciones, existen numerosas alternativas hardware/software entre las que se pueden destacar: los dispositivos DSP (*Digital Signal Processor*), ASIC (*Application-Specific Integrated Circuits*) y FPGA (*Field Programmable Gate Array*). Estas alternativas ofrecen diferentes grados de eficiencia que deben ser sopesados con respecto a factores como costes, potencia consumida o tiempo de diseño.

Otro aspecto importante que caracteriza la mejora de prestaciones que ofrecen los dispositivos es su evolución tecnológica. A día de hoy, la generación de nuevos dispositivos hardware se ajusta totalmente a la conocida ley de Moore [Moore, 1965] (Figura 1.2). Esta evolución, también es aplicable a los dispositivos FPGAs, los cuales formarán la base tecnológica de esta tesis. El número de recursos internos de éstos ha crecido notablemente en los últimos años, generándose dispositivos cada vez más preparados hacia el cómputo de imágenes en hardware (ver Tabla 1.2). Gracias a ello, es posible que estos dispositivos hayan pasado de ser usados inicialmente como elementos coprocesadores, a funcionar actualmente como verdaderos procesadores de aplicaciones de sistemas en tiempo real. Esto ha sido debido principalmente a las mejoras producidas en los procesos de fabricación de circuitos integrados y a la mejora del software de síntesis e implementación. Sirva como ejemplo el caso de los dispositivos de Xilinx, en su proceso de fabricación se ha disminuido hasta 65nm, consiguiendo un número de puertas por unidad de superficie imposible de imaginar hace 10 años, con un precio muy aceptable (dispositivos de 1 M puertas entorno a 12$) [Morris, 2003].

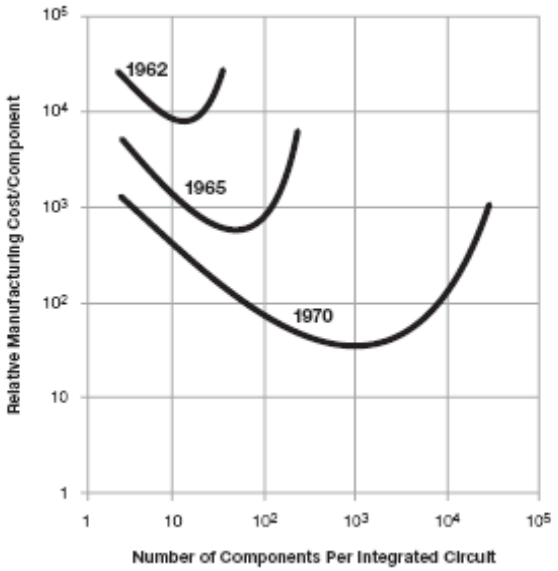


Figura 1.2. Evaluación del crecimiento de los circuitos integrados [Moore, 1965].

Tabla 1.2. Evolución y características de las distintas familias de FPGAs de Xilinx

Familia	**Año de creación**	**Número máx. de puertas**	**Novedades**
XC2000	1985	1800	Configuración SRAM
XC3000	1987	7500	Mayor número de puertas
XC4000/ Spartan	1991	180000	Memoria RAM
Virtex/ Spartan II	1998	3M2/600K	BRAMs, DLLs
Virtex II	2000	8 M	DCMs. Multiplicadores
Spartan 3	2003	5 M	Bajo Coste
Virtex II-Pro	2002	4 M	Power PC, Rocket I/O
Virtex 4	2005	8 M	Ethernet MACs, DSP Slice
Virtex 5	2006	11 M	DSP Slice Avanzada, BRAM 32KBits

La mejora en los dispositivos reconfigurables ha posibilitado la implementación de sistemas digitales de alto nivel, comúnmente denominados como SOC (*System On Chip*), con un ahorro económico y en tiempo de diseño. Económico, porque el número de circuitos integrados se reduce al poder integrar en una FPGA numerosos sistemas digitales (incluso microprocesadores de alto nivel). Si el análisis se realiza desde un punto de vista de tiempo de diseño, los tiempos de elaboración de un proyecto también se reducen al integrar todo en una FPGA.

Sin embargo, el empleo de estos dispositivos, también conlleva ciertos problemas. Uno de ellos y quizás el más importante, es el error producido por el empleo de datos habitualmente codificados en coma fija. Actualmente, pocas son las aplicaciones desarrolladas en FPGAs implementadas con datos en coma flotante.

Otro problema importante, son las herramientas de diseño y de lenguajes de programación. Éstas no permiten trabajar con lenguajes de alto nivel, lo que obliga a abordar el diseño mediante lenguajes de descripción hardware de bajo o medio nivel (por ejemplo VHDL o VERILOG).

1.4. OBJETIVOS GENERALES DE LA TESIS.

Esta tesis está enmarcada en un campo novedoso en el mundo de las FPGAs, concretamente en el de procesamiento de imágenes. La implementación de algoritmos de visión en FPGAs es todavía un área por explotar. Esto se debe fundamentalmente a la dificultad de trasladar los algoritmos convencionales (secuenciales) que se ejecutan sobre plataformas basadas en microprocesadores, a formas concurrentes que maximizan la utilización de recursos y reducen el tiempo de proceso. Hasta la fecha no existen herramientas de desarrollo de alto nivel que permitan implementar algoritmos de una manera óptima en lenguajes de alto nivel. Tampoco existen herramientas que trasladen un algoritmo puramente secuencial en un algoritmo concurrente, cuyos resultados optimicen el número de recursos internos necesarios en FPGAs.

Sin embargo, la implementación de algoritmos de alto nivel sobre FPGAs tiene ventajas frente a otras arquitecturas. A continuación se exponen alguna de ellas:

- El hecho de poder ejecutar algoritmos de forma concurrente en un dispositivo reconfigurable logra acelerar los tiempos de ejecución con respecto a los ejecutados de forma secuencial.
- Debido a la característica de reconfigurabilidad de las FPGAs, éstas permiten un alto grado de flexibilidad. Esto conlleva a que cualquier variación del diseño no tenga que llevar implícito cambio alguno en función de la plataforma empleada.
- Otra ventaja, es el precio de la plataforma dónde se desarrollará la aplicación. El coste de una plataforma basada en una FPGA es notablemente inferior al que puede tener un PC o un DSP.

Todas estas razones conducen al intento de adaptar algoritmos de alto nivel sobre dispositivos reconfigurables.

De esta forma el objetivo principal que se plantea en esta tesis es la implementación del algoritmo PCA en una FPGA aplicado a la detección de obstáculos en movimiento, lo cual conlleva la definición de otros objetivos más específicos que a continuación se detallan (ver Figura 1.3 y Figura 1.4).

- *Multiplicación de matrices en FPGAs*: Dentro del algoritmo PCA se deben realizar varias multiplicaciones de matrices para diferentes tamaños tal y como se verá en los capítulos siguientes. Esta operación aritmética necesita un análisis y estudio profundo debido a la naturaleza concurrente de una FPGA. Así, con objeto de explotar esas características se hace necesario diseñar alternativas a la multiplicación secuencial de los microprocesadores. En la década de los años ochenta se presentaron numerosos trabajos para poder realizar esta operación en dispositivos ASIC. Sin embargo la mayoría de ellos se basaban en la construcción de un sistema de multiplicación sistólico, por lo que habitualmente se necesitaban $N \times N$ multiplicadores (siendo N el tamaño de la matriz). Esos algoritmos presentan grandes dificultades para ser implementados en FPGAs, más aún cuando el tamaño de las matrices a multiplicar es elevado, por ejemplo 512x512. En este caso el número de multiplicadores que debería tener la FPGA alcanzaría un valor que no se puede encontrar en los dispositivos actuales. Por ello, se necesita emplear algoritmos que empleen un menor número de multiplicadores hardware, pero que de alguna manera permitan paralelizar el proceso de multiplicación.

- *Cálculo de autovalores y autovectores*: Dentro del algoritmo PCA, el cálculo de los autovectores es una de las partes fundamentales. Esta tarea necesita una elevada carga computacional y normalmente consume un elevado tiempo. Por esta razón, es necesario buscar alternativas que particionen las diferentes tareas de dicho cálculo, para conseguir su ejecución en paralelo con una precisión deseada.

- *Cálculo del Mapa de Distancias*: La determinación de la existencia de un objeto en movimiento dentro de una escena se realizará mediante el cálculo de la distancia euclídea entre una imagen capturada y esa imagen proyectada en un espacio transformado y posteriormente recuperada (mapa de distancias). La búsqueda de una alternativa óptima de cálculo es sin duda otro objetivo importante del algoritmo PCA.

- *Umbral Dinámico*: Una vez obtenido el mapa de distancias, se debe determinar si existe o no objeto/s en movimiento dentro de la escena. Para ello en esta tesis se propone el uso de un umbral dinámico cuyo valor se ajuste automáticamente en función de las características de la escena.

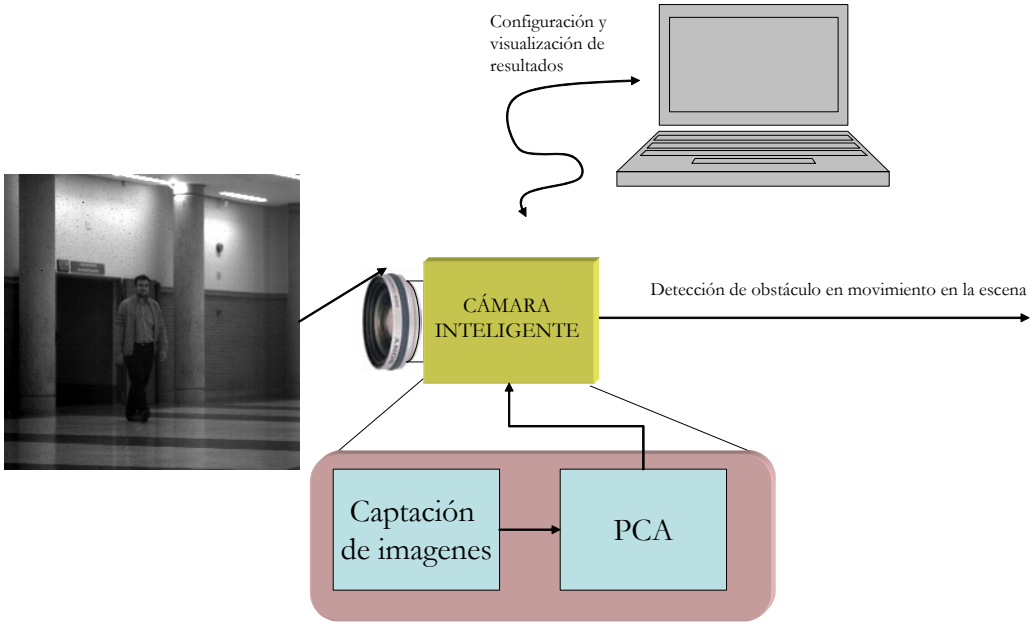


Figura 1.3. Diagrama de bloques de los diferentes elementos que componen el sistema a desarrollar en esta tesis.

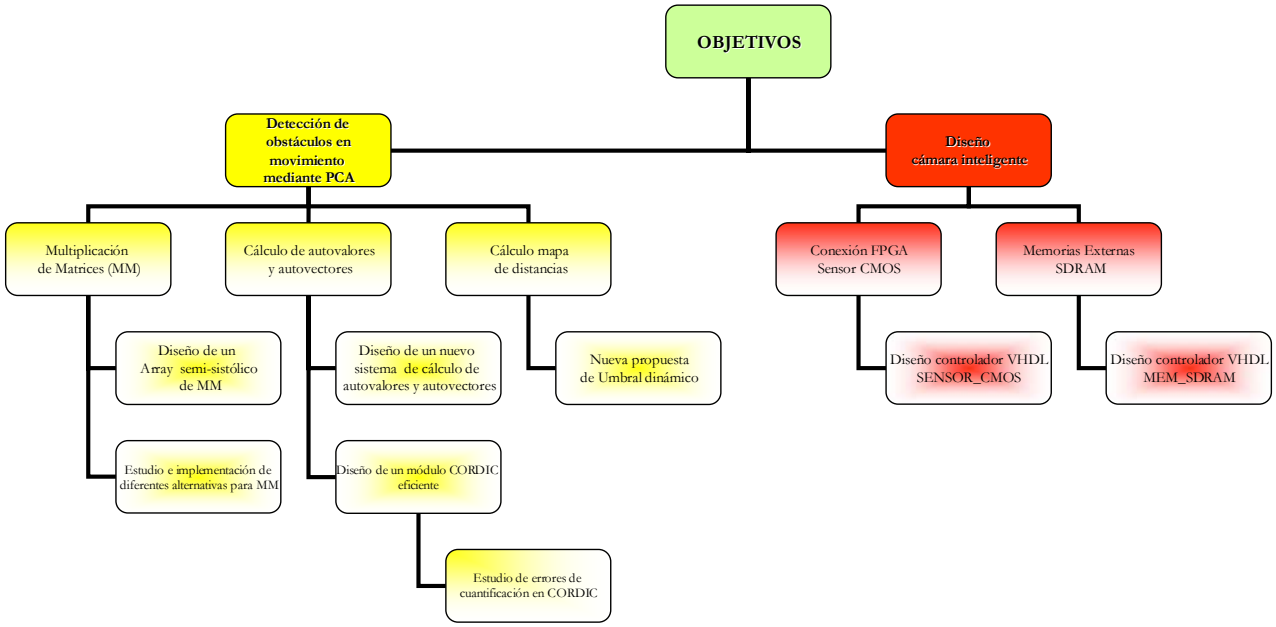


Figura 1.4. Organigrama ilustrativo con los objetivos planteados en esta tesis doctoral.

La construcción de una "cámara inteligente" es otro de los objetivos que se aborda en esta tesis. Esta cámara está básicamente compuesta por una FPGA, un sensor CMOS y memoria SDRAM externa. La FPGA ejerce como controlador y procesador del sistema proporcionando versatilidad a la hora de configurar la cámara en función de ciertas especificaciones o necesidades del usuario. En nuestro caso, se

ha optado por un sensor CMOS de alta velocidad (500 imágenes por segundo) y alta resolución (1.2 M píxeles) si bien el algoritmo PCA trabaja internamente con imágenes de 256x256 píxeles. Se emplea también un PC únicamente como elemento de configuración de parámetros y de visualización de resultados

1.5. ESTRUCTURA Y CONTENIDO DE LA MEMORIA.

Esta memoria está estructurada en un total de 8 capítulos. Tras este capítulo de introducción, el **capítulo 2** muestra el estado del arte de los temas relacionados con los objetivos de esta tesis. En él se describen los trabajos más significativos dentro del procesamiento de imágenes con FPGAs, así como de implementación del algoritmo PCA en FPGAs. También se detallan los trabajos más importantes dentro del campo de multiplicación de matrices y cálculo de autovalores y autovectores en FPGAs.

El **capítulo 3** versa sobre la propuesta concreta que se hace en esta tesis para la detección de obstáculos en movimiento sobre FPGAs. Concretamente se presentará el algoritmo PCA, desarrollando las diferentes etapas que lo forman.

Una vez presentado el sistema a desarrollar, en el **capítulo 4** se describe la implementación del algoritmo PCA sobre una FPGA. Este capítulo, mostrará en detalle cada uno de los bloques que forman el algoritmo PCA.

En el **capítulo 5**, se propone la detección de objetos en movimiento mediante la creación de un mapa de distancias euclídeas entre la imagen capturada y dicha imagen proyectada y recuperada mediante PCA, empleando un umbral dinámico.

El **capítulo 6** muestra los resultados de los ensayos realizados con el sistema completo, así como las características de la cámara inteligente construida para esta tesis.

En el **capítulo 7** se extraen las conclusiones del trabajo realizado, se detallan de forma resumida las aportaciones principales y se exponen los futuros trabajos de investigación que a opinión del autor se abren tras la realización de la presente tesis.

Por último el **capítulo 8** presenta las referencias y enlaces a páginas Web citados en esta tesis.

2. ESTADO DEL ARTE

2.1. INTRODUCCIÓN.

El procesado de imágenes en tiempo real requiere en muchas aplicaciones un rápido y continuo cálculo de operaciones a la hora de ejecutarse en un sistema determinado. Esto es debido fundamentalmente a la elevada carga computacional que lleva asociada los algoritmos de procesado de imágenes. Recientemente los dispositivos hardware reconfigurables están siendo una alternativa en el procesado de imágenes debido a su bajo precio, así como sus altas prestaciones. Sin embargo, este tipo de dispositivos presentan grandes problemas desde el punto de vista de implementación de algoritmos, ya que las herramientas para su sintetización no están desarrolladas todo lo deseable ni están estandarizadas a día de hoy.

Otras alternativas al procesado de imágenes, además del PC cuyo uso está ampliamente extendido, pueden ser el empleo de estaciones de trabajo en paralelo (procesamiento paralelo), Procesadores Digitales de Señales (DSPs) o circuitos hardware de propósito específico (ASIC).

El procesamiento paralelo, se ha planteado como una posible solución para la obtención de altas prestaciones. La mayoría de las propuestas han tendido hacia arquitecturas del tipo SIMD (*Single Instruction Multiple Data*) o más comúnmente hacia SPMD (*Single Program Multiple Data*). En esta última solución, la configuración de la imagen se divide habitualmente en varios bloques y estos se procesan de manera concurrente.

Aunque en la pasada década esta tendencia sufrió un auge importante, todavía no se ha implantado masivamente, ya que la programación de una máquina paralela entraña muchas dificultades. Otra alternativa que actualmente está empezando a explotarse es el procesamiento de imágenes mediante sistemas multiprocesador. Esta opción está proporcionando buenos resultados desde el punto de vista de ejecución. Sin embargo, existe una elevada complejidad en la sincronización entre los microprocesadores, lo que conlleva un coste elevado.

El procesamiento paralelo de imágenes es una alternativa eficaz cuando se desea realizar procesado a bajo nivel [Bouridane, 1999]. En esta situación, el particionado de una imagen y su posterior procesado en diferentes elementos no tiene asociada una problemática significativa. Sin embargo, en el caso del procesado de medio o alto nivel la partición se complica. Los algoritmos de estos niveles necesitan información de diferentes partes de una imagen, creando así una fuerte dependencia entre las diferentes particiones. Así, se necesita establecer mecanismos de comunicación entre los elementos de procesado, generando un importante trasiego de información entre ellos. Esta característica implica un aumento en el tiempo de ejecución, el cual puede ser considerable en determinadas aplicaciones.

El uso de DSPs en el procesado digital de imágenes es la alternativa más empleada. Las diferentes plataformas basadas en ellos proporcionan una buena autonomía así como una elevada tasa de operaciones por unidad de tiempo. Esto es debido a la arquitectura interna de la CPU cuyas características se adaptan perfectamente al procesado de imágenes. Otra gran ventaja de esta solución es la posibilidad de desarrollar diferentes algoritmos en lenguajes de alto nivel, dada la existencia de múltiples compiladores con elevado grado de prestaciones. Sin embargo, su naturaleza secuencial y de propósito específico no permite en muchas ocasiones alcanzar un procesamiento en tiempo real con estos dispositivos.

Otra alternativa al procesado de imágenes, es el uso de dispositivos hardware de propósito específico (ASIC). Este tipo de circuitos pueden proporcionar soluciones particulares a determinados algoritmos de procesamiento de imágenes. Sin embargo, presentan algunas desventajas que pueden resumirse en:

- Esta solución puede tener un coste elevado si el volumen de elementos necesarios no justifica los costes de fabricación.
- El diseño y desarrollo de este tipo de circuitos requiere un tiempo excesivo tanto desde el punto de vista de validación del circuito, como de fabricación. Esta causa demora notablemente los proyectos por lo que actualmente se está trabajando en el desarrollo de herramientas que realicen la implementación física automáticamente, partiendo de una *netlist* como la usada por las herramientas de emplazamiento de FPGAs.

- Si bien el rendimiento es óptimo, ya que se diseñan para una aplicación concreta, la falta de flexibilidad para modificar los algoritmos implementados constituye una desventaja.

Con objeto de evitar la rigidez de los circuitos ASIC, actualmente la alternativa hacia la que se están dirigiendo numerosas líneas de investigación es la de emplear dispositivos con mayor flexibilidad que al mismo tiempo mantengan las ventajas de los elementos ASIC en cuanto a rendimiento. Un ejemplo de estos son los dispositivos FPGA, los cuales permiten reconfigurarse o reprogramarse tantas veces como se desee y de forma sencilla. Justamente, son estos dispositivos en los que se basa todo el desarrollo de esta tesis.

En la Figura 2.1 se muestra la relación existente entre la elección de un sistema que posea ciertas facilidades para poder programarse (programabilidad), con respecto al nivel de especialización del sistema, es decir, la flexibilidad en cuanto a las posibilidades de variar la funcionalidad de un sistema. Se puede observar como existe un compromiso entre ambos parámetros. También, puede comprobarse como los circuitos de tipo ASIC poseen una gran facilidad para poder implementar sistemas con un alto grado de especialización, mientras que un procesador aporta una gran flexibilidad para poder programarse. En la Tabla 2.1 se muestra una comparativa de los diferentes sistemas para el procesamiento de imágenes tomando otros índices como coste, rendimiento, consumo y flexibilidad.

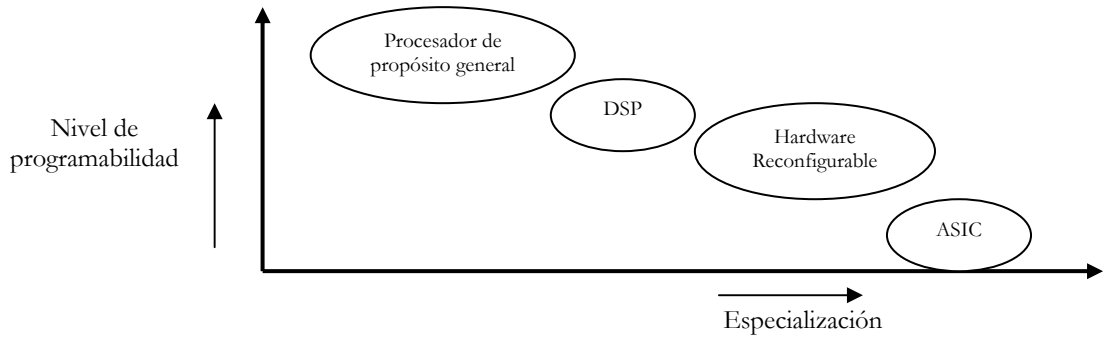


Figura 2.1. Relación entre nivel de especialización y el nivel de programabilidad, para diferentes sistemas de procesado de imágenes [Tessier, 1999].

De la información presentada en la Tabla 2.1 se comprueba como la opción de hardware reconfigurable ofrece buenos resultados para los diferentes índices evaluados. Una de las grandes características de este tipo de sistemas, es la posibilidad de ejecutar algoritmos de forma concurrente y no de manera secuencial. Esto aplicado a algoritmos de procesado de imágenes, que permitan particionar su ejecución, hace que este tipo de arquitecturas sean muy propicias para la ejecución de estas aplicaciones. A ello se le pueden añadir las tendencias actuales de introducir módulos de procesado de señal dentro de las FPGAs, por ejemplo unidades aritméticas de alta velocidad o *IPs* (*Intellectual Properties*), con lo que el empleo de esta

opción se acentúa aún más. Cabe mencionar que su flexibilidad les confiere una gran utilidad a la hora de generar un prototipo, sobre todo en I+D.

Tabla 2.1. Comparativa entre DSPs, ASICs, hardware reconfigurable y procesadores de propósito general. [Tessier, 1999].

	Rendimiento	**Coste**	**Consumo**	**Flexibilidad**	**Nivel de esfuerzo en el diseño**
ASIC	Alto	Alto	Bajo	Bajo	Alto
DSP	Medio	Medio	Medio	Medio	Medio
Procesador	Bajo	Bajo	Medio	Alto	Bajo
Hw. Reconf.	Medio	Medio	Alto	Alto	Medio

Dentro del procesamiento de imágenes con dispositivos reconfigurables no existen muchos trabajos realizados hasta el momento. Esto es debido principalmente a dos razones: por una parte, a que la aparición de estos dispositivos es reciente (aproximadamente hace veinte años), y por otra, la falta de herramientas de programación de alto nivel. Sin embargo, el hecho de que muchos algoritmos de procesado de imágenes permiten particionar su ejecución secuencial facilita el que se puedan diseñar diferentes elementos de procesado que pueden funcionar en paralelo, acelerando así su tiempo de ejecución.

Uno de los algoritmos que puede ser particionado y por lo tanto ejecutarse concurrentemente, es el de Análisis de Componentes Principales (PCA). Sin embargo, actualmente no existe ningún trabajo que desarrolle íntegramente su ejecución sobre una FPGA. Varios son los problemas asociados a este algoritmo para su desarrollo en FPGAs, como por ejemplo el cálculo de autovectores y autovalores o la multiplicación de matrices. Por esta razón, en este capítulo se presentan los trabajos más significativos relacionados al respecto, analizando sus ventajas e inconvenientes.

El resto de este capítulo se distribuye de la siguiente manera: en el siguiente apartado se realiza un recorrido por los trabajos más significativos de procesamiento de imágenes en FPGAs. A continuación, se mostrarán los trabajos más relevantes encontrados hasta el momento, sobre multiplicación de matrices en FPGAs. En el siguiente apartado se presentan diferentes trabajos asociados al cálculo de autovalores y autovectores sobre diferentes arquitecturas hardware (multiprocesadores, ASIC, FPGA) y por último se presentarán los trabajos más notables sobre aplicación del algoritmo PCA sobre FPGAs.

2.2. PROCESAMIENTO DE IMÁGENES EN FPGAS.

Desde la aparición a finales de la década de los ochenta de las FPGAs, su uso en el procesado de imágenes ha ido incrementándose continuamente. Inicialmente, los primeros trabajos de procesamiento con FPGAs usaban como plataforma una máquina genérica (*custom machine*) [Arnold, 1992] [Ziegler, 2002] [Piacentino, 1999]. De manera simultánea, otros trabajos como los expuestos en [Hamid, 1994] [Wiatr, 1998], comenzaban a explotar las características de estos dispositivos en algoritmos sencillos de preprocesamiento de imágenes. En [Guccione, 1999] se muestra un listado de todas las máquinas construidas con FPGAs desde 1994 hasta 1999.

La evolución tecnológica de las FPGAs provocó la desaparición paulatina de las *custom machines* construidas con múltiples FPGAs, a favor de sistemas con una única FPGA funcionando como coprocesador. Así, éstas coexistían con un PC o un DSP, encargándose estos últimos de manejar las operaciones computacionales complejas dejando a la FPGA el procesado de bajo nivel. Este tipo de plataformas sigue aún estando muy vigente.

El procesamiento de algoritmos de alto nivel en FPGAs, no comienza a implementarse hasta principios del año 2000 [Martín, 2000] [Bravo, 2004]. En esta fecha, aparecen las nuevas generaciones de FPGAs que incluyen nuevas prestaciones como unidades hardware de procesado digital, *cores* y bloques de memoria.

A continuación, se hace un análisis por estas tres líneas de trabajo en FPGAs en el procesado de imágenes: *custom machines*, FPGAs como coprocesadores y ejecución de algoritmos de alto nivel sobre FPGAs.

2.2.1. Custom Machines.

Con la aparición a principios de los años 90 de FPGAs de capacidad media, en cuanto al número de recursos internos y con una cierta complejidad en su estructura interna, se crearon las primeras máquinas reconfigurables (SPLASH 2, G-800, PIPS, etc.) [Bouridane, 1999].

La mayoría de estas máquinas no fueron diseñadas con un propósito específico, sino que fueron creadas con idea de ser empleadas en varias aplicaciones y áreas de trabajo. Una de estas áreas fue la de procesado de imágenes. La posibilidad de ejecutar en paralelo algoritmos de este tipo permitiría alcanzar el grado de tiempo real necesario en muchas aplicaciones, y difícil de conseguir con plataformas convencionales de propósito general.

Por esta última razón se crearon a finales de la década de los 80 las denominadas Máquinas CCM (*Custom Computing Machine*), en base a tarjetas compuestas de varias FPGAs. Estas máquinas son sistemas "embebidos" que pueden ser usados para construir arquitecturas especializadas en diferentes aplicaciones. Las FPGAs revolucionaron este campo ya que gracias a su reprogramabilidad permitían disponer de una máquina de propósito general, que se podía utilizar en diferentes

áreas sin necesidad de cambiar la plataforma hardware. En la Tabla 2.2 se presenta un resumen de las máquinas más importantes CCM.

Tabla 2.2. Resumen máquinas CCM más importantes.

Nombre	Diseñador	Configuración	Comentarios
ACME (*Adaptative Connectionist Model Emulator*)	Universidad de California	14 Xilinx 4010, 6 Xilinx 3195	Las XC3195 son usadas como interconexiones programables.
CHAMP (*Configurable Hardware Algorithm Mappable Processor*)	Lockheed Sanders, Nashua	16 Xilinx 4013, 512 K Dual Port.	Prototipo Hardware
DFFC (*Data-Flow Functional Computer*)	LIMSI-CNRS, Francia	Array de 8x8x8 FPOAs	Aplicado al procesado de imágenes en tiempo real.
DTM-1	MITRE, Corp. Virginia	16 DTM chips	Cada chip DTM es un array de 64x64 gate cells.
Enable ++	Universidad de Mannheim, Alemania	16 Xilinx 4013, 11 Xilinx 5005, 12MB de RAM	Sistema modular FPGA y multiprocesador
Marc-1	Universidad de Toronto, Canadá	25 Xilinx 4005, 6MB RAM	256Kx64 memoria de instrucciones 256Kx32 memoria de programa
PAM (*Programmable Active Memory*)	Laboratorios DEC's (Paris, Francia)	5x5 array de Xilinx 3090	Aplicado a criptografía, compresión de datos, aceleración de gráficos en 3D
VC (*Virtual Computer*)	Virtual Computer Corporation (Estados Unidos)	52 FPGAs y 24 ICUBE	Propósito académico

SPLASH 2 fue la primera máquina de propósito específico creada. Fue uno de los trabajos pioneros en la década de los 90, desarrollado por investigadores de la Universidad de Queens (Reino Unido) y de la Universidad de Michigan (Estados Unidos). En base a la plataforma SPLASH 2 se realizaron diversos trabajos desde el

punto de vista de implementación de bajo y medio nivel en algoritmia de visión [Arnold, 1992].

SPLASH 2 fue una de las máquinas más importantes que se desarrollaron con arquitectura genérica y no exclusivamente orientada al procesamiento de imágenes. Esta máquina constaba de un array de FPGAs XC4010, y surge como evolución de la *SPLASH 1*, que estaba basada en dispositivos de la familia XC3000. La Figura 2.2 muestra una vista del sistema de la arquitectura *SPLASH 2*. Ésta estaba conectada al *host* a través de un interfaz que expandía los buses. El *host* (una máquina SUN) podía leer/escribir en memoria y mapear los registros de control de la *SPLASH 2* gracias a estos buses expandidos. Cada tarjeta de procesado *SPLASH 2* tenía 16 FPGAs 4010 (X1-X16 de la Figura 2.2) que funcionan como elementos de proceso. Además, una FPGA adicional (X0) controlaba el flujo de datos dentro de la tarjeta. Cada tarjeta poseía 512 KB de memoria RAM (ver Figura 2.2).

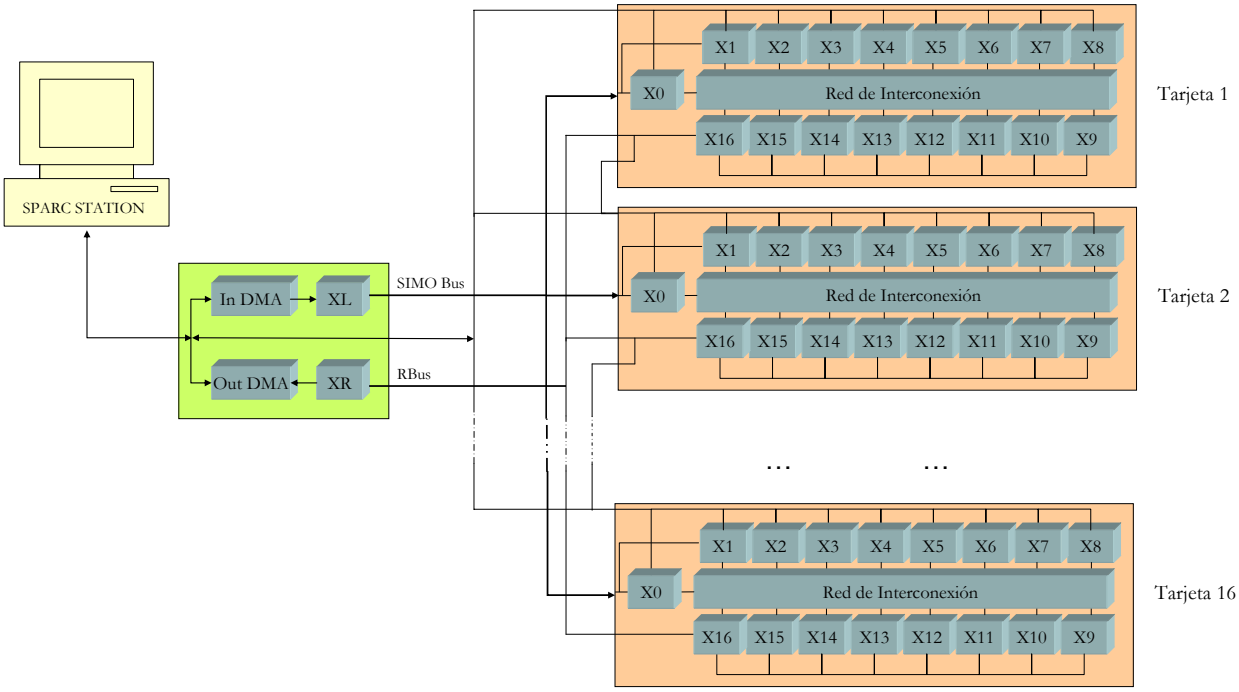


Figura 2.2. Arquitectura interna de la máquina *SPLASH 2* [Arnold, 1992].

Con esta plataforma se desarrollaron varios trabajos de procesado de imágenes, con diferente complejidad computacional. En [Ratha, 1999] se presentan algunos de ellos ordenados acorde al nivel de complejidad algorítmica. Así, dentro de los trabajos que pueden clasificarse como de bajo nivel, se implementaron operaciones de convolución y erosión/dilatación. De nivel medio, se implementó la segmentación de imágenes y de alto nivel se realizó el *matching* aplicado a detección de huellas dactilares. En la Tabla 2.3 se muestra un resumen de los tiempos consumidos para ejecutar cada algoritmo sobre la plataforma *SPLASH 2*, comparándolo con el empleado en una estación SPARC. Se observa como el tiempo de ejecución sobre la plataforma de FPGAs disminuye considerablemente, haciéndose más notable en los

algoritmos de alto nivel. En este caso el código generado no estaba optimizado ya que se utilizó un traductor de lenguaje de alto nivel a bajo nivel diseñado a medida para esta máquina. En caso de haber diseñado el algoritmo mediante lenguajes orientados al hardware los tiempos de ejecución se habrían mejorado.

Tabla 2.3. Tiempos de ejecución de los algoritmos implementados en SPLASH 2.

Nivel	Tarea	Tiempo en SUN SPARC	Tiempo en SPLASH 2
Bajo	Convolución	3.6 ms	13.89 ms
Medio	Segmentación	250 s	3.8 s
Alto	Matching de un punto	15.38 ms	9.09 μs

DFFC (Data Flow Functional Computer): Una plataforma pensada para procesado de imágenes. Fue diseñada para funcionar como plataforma de propósito general para algoritmos de visión y tenía como objetivo alcanzar tiempo real en operaciones morfológicas con imágenes [Quenot, 1994] [Boluda, 2000].

Esta plataforma estaba formada por un conjunto de FPGAs especialmente desarrolladas para este sistema, denominadas como FPOAs (*Field Programmable Operator Array*). Las FPOAs tenían una arquitectura similar a la de las FPGAs híbridas (intentan combinar los beneficios de FPGAs y PLDs) (Tabla 2.4). Estos circuitos ASIC fueron construidos con un potente bloque de interconexiones para poder asociarse en estructuras tridimensionales.

Tabla 2.4. Características de las FPOAs [Quenot, 1994].

Encapsulado	144 pines PGA
Tecnología	1 μm CMOS
Área	8.9 mm x 9.6 mm
Frecuencia de funcionamiento	25 MHz
Pines de Entrada/Salida	100 (distribuidos en 10 puertos)
Tecnología de programación	SRAM
Puertas lógicas equivalentes	33.000
Memoria interna	8.5 Kbits

La máquina DFFC estaba formada por un *array* de 8 filas x 8 columnas x 8 de tarjetas FPOAs (Figura 2.3). En ella se podían conectar directamente a la red de FPOAs las entradas y salidas de vídeo digitalizadas. Una estación de trabajo SPARC2 se encargaba de la programación de las FPOAs.

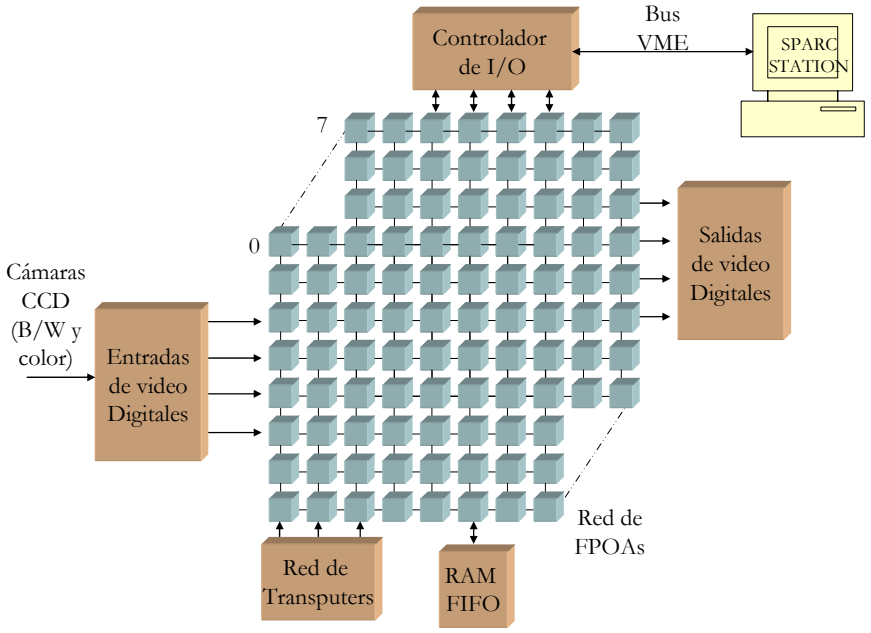


Figura 2.3. Diagrama de bloques de la arquitectura interna de la máquina DFFC [Quenot, 1994].

Para poder trabajar con la máquina DFFC se diseñaron librerías de programación de bajo y medio nivel, que posteriormente se compilaban a un formato interpretable por el hardware reconfigurable. Las librerías de bajo nivel estaban compuestas por un conjunto de primitivas o macros que contenían operadores aritmético lógicos, multiplexores, contadores, etc. Por su parte, las de medio nivel, contenían agrupaciones de primitivas que formaban elementos para el procesado con una complejidad mayor. Dentro de este grupo estaban las LUTs (*Look Up Tables*), convolucionadores o histogramas.

La versatilidad aportada con estas librerías posibilitó el desarrollo de aplicaciones de filtrado, detección de esquinas o seguimiento de un objeto de color dentro de una escena. El principal problema de esta máquina era el excesivo número de recursos hardware consumidos para poder ejecutar algoritmos de complejidad baja [Quenot, 1994].

Virtual Computer: Fue una máquina desarrollada al inicio de los años noventa por la empresa *Virtual Computer Corporation*. Esta máquina surgió como alternativa al procesado de algoritmos complejos ejecutados sobre PCs, teniendo presente la idea de que cualquier algoritmo puede alcanzar su máxima velocidad, si es implementado en hardware [Casselman, 1993].

Internamente estaba formado por FPGAs, FPID (*Field Programmable Interconnect Devices*), memoria SRAM y Dual-Port (Figura 2.4). En la Tabla 2.5 se presenta un resumen de los recursos que contenía esta máquina.

Con respecto a su utilización, inicialmente fue diseñado como una plataforma académica genérica, donde a partir de una librería codificada en C++ se proporcionaban un conjunto de funciones de medio nivel (convoluciones, operaciones morfológicas, etc.). El usuario podía trabajar en alto nivel o bajo nivel (Verilog) ya que una herramienta de síntesis/implementación de propósito específico se encargaba de generar el *bitstream* correspondiente.

Tabla 2.5. Resumen de recursos disponibles en la máquina Virtual Computer.

FPGAs	Xilinx 4010 (52 unidades)
FPIDs	I-CUBE IQ160 (24 unidades)
Puertos E/S	3 de 64 bits
SRAM	8 MB
DPORT	16K x 8K x 16 bits

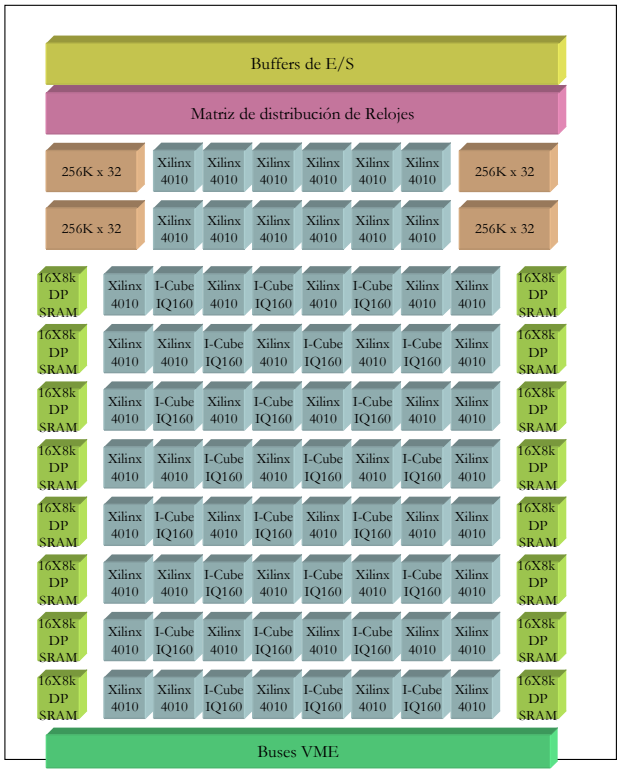


Figura 2.4. Estructura interna de la máquina superescalar Virtual Computer [Casselman, 1993].

Máquinas Monotarjeta. Posterior a la aparición de las máquinas de propósito general, las cuales contenían un gran número de dispositivos FPGAs, aparecieron nuevos trabajos donde el objetivo era la implementación de máquinas más simples

que las de propósito general. Para ello se construyeron máquinas formadas por una única tarjeta de FPGAs. De esta forma se conseguía reducir notablemente los recursos consumidos, con lo que el coste disminuía. Sin embargo, esa disminución de recursos se traducía en un potencial de cálculo inferior a las de propósito general. Por ello, su ámbito de actuación se centró en aplicaciones de procesamiento de bajo nivel. Estábamos ante las primeras tarjetas coprocesador, donde se delegaba la complejidad computacional a un *host* al que estaba conectado la tarjeta.

Ejemplos de esta línea de trabajo fueron la tarjeta WILDFORCE o SPECTRUM RC. WILDFORCE fue una tarjeta de propósito general, diseñada por la empresa Annapolis Micro Systems. Poseía un conector PCI con lo que dicha tarjeta podía ser insertada en un PC. Inicialmente el sistema fue diseñado con FPGAs de Xilinx de la familia 4000, concretamente XC4013, desarrollando algoritmos de visión de bajo nivel (filtrado y convoluciones), con la que se alcanzaron un alto grado de rendimiento. Posteriormente, se diseñó una herramienta de particionado y síntesis [Govindarajan, 1998] que permitía el diseño inicial en un lenguaje de alto nivel y su posterior traslado a la FPGA. Ello posibilitó la implementación de algunos algoritmos de complejidad media, como la compresión de imágenes mediante la técnica JPEG.

2.2.2. FPGAs como coprocesadores.

Esta nueva configuración, se caracterizaba principalmente por la coexistencia de un procesador secuencial, típicamente un PC o DSP, encargado de realizar operaciones aritméticas complejas, y de un número de FPGAs reducido (típicamente entre 1 y 3). El hecho de existir un procesador secuencial, provocaba el funcionamiento de la FPGA como un dispositivo esclavo. Así, el procesador se liberaba de carga computacional, delegando ciertas tareas a la FPGA. Es por esta razón que el nombre adoptado para estos sistemas reconfigurables sea el de tarjetas coprocesador.

Algoritmos simples, como filtrados o convoluciones, eran aplicaciones habituales de estos sistemas. Algunos ejemplos de este tipo de operaciones se muestran en [Hamid, 1994], donde se realizan con FPGAs operaciones de medias aritméticas sobre imágenes de resolución espacial de 256 x 256. Otro ejemplo, es el mostrado en [Wiatr, 1998]. En este caso, la tarjeta con FPGAs se introducía como un elemento de pre-procesado (media, convolución o filtrado) dentro de un sistema completo de adquisición y procesamiento de imágenes.

En [Jarro, 2001] se muestra el diseño de un multiplicador hardware implementado en una FPGA para realizar operaciones de convoluciones sobre imágenes. Dentro de esta línea existen numerosos trabajos como los mostrados en [Dawood, 2002] [Draper, 2003] [Kesaal, 2003], donde se realizan aplicaciones de máscaras de visión (normalmente de 3x3), sobre imágenes monocromáticas de resolución espacial 256x256.

Las tarjetas que alojan las FPGAs solían disponer de buses estándar de comunicación para conectarse con el procesador central. Esto permitía una cómoda expansión en el número de tarjetas.

La complejidad que desarrollaba el sistema compuesto por las tarjetas de FPGA y el *host* central solía ser elevada, debido a la partición de tareas entre el procesador y la FPGA. Un ejemplo de estos trabajos es el mostrado en [Knittel, 1996]. Éste constaba de una tarjeta con 4 FPGAs y un bus PCI para comunicarse con un PC. Este diseño, estaba orientado a la reconstrucción 3D de imágenes médicas.

Otras tarjetas interesantes en las que se han experimentado técnicas de pre-procesado digital son las mostradas en [Talu, 2000] [Arias, 2001] [Arias, 2001b]. Por su parte, los trabajos desarrollados en [Battle, 2002] [Meribout, 2002] [Meribout, 2002b], muestran plataformas que combinan DSPs con FPGAs. En estos últimos casos, la complejidad algorítmica que se alcanzaba era mayor que en los trabajos previos. Esto era debido al reparto de tareas hardware (FPGA)/software (DSP), y a la alta tasa de transferencia de datos alcanzada.

Varios son los problemas que este tipo de arquitecturas tenían:

a) *Separación de tareas*: La división de las tareas que debía ejecutar el *host* y el dispositivo reconfigurable era un problema importante. En función de la tarea, cada una de las plataformas es óptima para realizar una u otra operación. Ello implicaba un alto grado de sincronización entre las dos plataformas, complicando notablemente el diseño. Actualmente, la aparición de técnicas y herramientas de codiseño permiten separar de manera óptima estas facetas.

b) *Programación de dos dispositivos diferentes*: El hecho de disponer siempre de un *host* (PC, DSP, etc.) y de un dispositivo reprogramable (por ejemplo una FPGA) conllevaba la programación de dos elementos con lenguajes y métodos diferentes. Esto provoca un doble grado de especialización

2.2.3. Algoritmos de visión artificial de alta complejidad implementados sobre FPGAs.

Con la aparición de las últimas familias de FPGAs a principios del año 2000 [Xilinx, 2004] se incorporaron nuevos recursos internos orientados hacia el procesamiento de imágenes. Esto ha provocado que dentro de esta área, se haya producido un incremento en el desarrollo de algoritmos de visión de alto nivel. Sirvan como ejemplo los trabajos mostrados en [Meribout, 2002] [Meribout, 2002b]. En ellos se desarrollaban plataformas hardware orientadas a la segmentación de imágenes mediante la Transformada de Hough. Sin embargo, la ejecución de dicha transformada no era ejecutada íntegramente dentro de la FPGA, sino que también se

usaba un DSP. En [Hernández, 2004] se propone una alternativa eficaz dónde se realiza íntegramente el cálculo de esta transformada en una FPGA.

La aparición de nuevos lenguajes de programación para FPGAs con un nivel de abstracción mayor que VHDL o Verilog, como pueden ser Okapi, ImpulseC, System-C y sobre todo Handel-C, han permitido la traslación de algoritmos codificados en lenguajes de alto nivel para PC a hardware reconfigurable. En la mayoría de estos trabajos la optimización del sistema se basa en la partición del algoritmo en elementos que puedan ejecutarse en paralelo, o bien, en el diseño de arquitecturas sistólicas donde los elementos de procesado pueden trabajar en paralelo. Un ejemplo de esta readaptación puede ser el mostrado en [Martín, 2005]. En este caso, se presentaba una arquitectura basada en FPGAs orientada a la obtención de flujo óptico, alcanzando velocidades de detección de movimientos de hasta 60 frames/sg para imágenes de 256x256 píxeles de 8 bits de resolución de nivel de gris.

En el área de la navegación autónoma de robots basados en visión artificial, en [Boluda, 2000] se propone una arquitectura basada en FPGAs FLEXLM 8000 de Altera. Éstas se encargan de procesar imágenes procedentes de una cámara *Log-Polar* y de enviar la información, procedente de otros sistemas sensoriales, a un sistema de fusión de datos (Figura 2.5). Concretamente, el algoritmo implementado realiza la detección de objetos en movimiento para así poder planificar correctamente su trayectoria y evitar choques con objetos. Para ello se obtiene el flujo óptico y posteriormente se realiza una segmentación, con el objetivo de detectar objetos. El sistema completo alcanza un ratio de procesado de 100 imágenes por segundo.

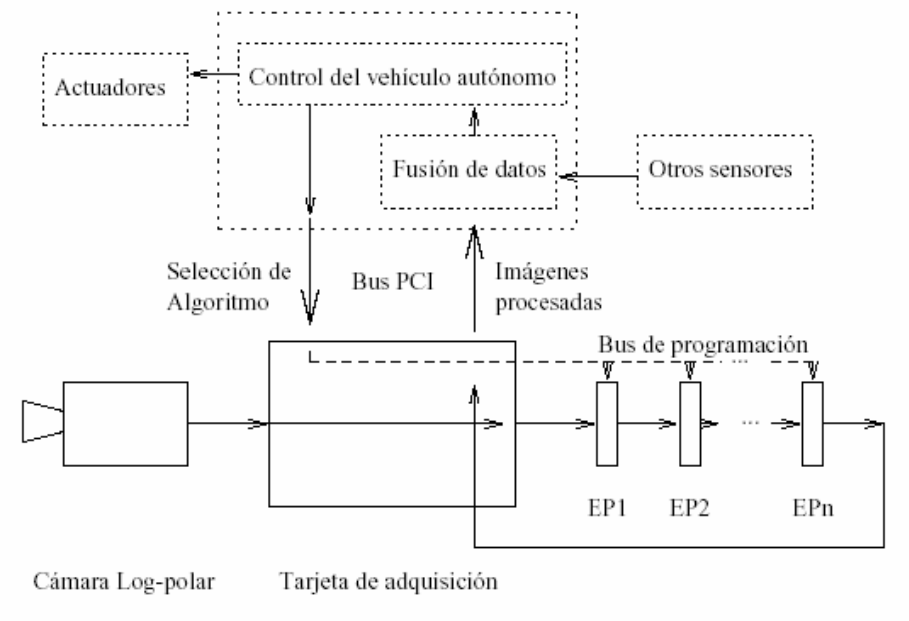


Figura 2.5. Sistema completo de navegación de un robot móvil procesando imágenes con dispositivos reconfigurables [Boluda, 2000].

Otro ejemplo de implementación de algoritmos de alto nivel sobre FPGAs es el mostrado en [Cuenca, 2002]. Este trabajo se encuentra dentro del área de

inspección visual utilizando texturas, donde se aplican diferentes técnicas basadas en distribuciones estadísticas. El sistema propuesto consta de una etapa formada por módulos reconfigurables (FPGAs), encargados de realizar preprocesamiento, y operaciones morfológicas, y un DSP encargado de gestionar el sistema total y realizar operaciones aritméticas en coma flotante.

Actualmente, otra línea de trabajo muy habitual es el uso de la herramienta Xilinx System Generator (XSG) para la implementación de algoritmos de visión artificial de alta complejidad sobre FPGAs. Este software permite traspasar, bajo ciertas restricciones, algoritmos diseñados en Matlab-Simulink a VHDL. En [Toledo, 2005] y [Bravo, 2004] se muestran dos ejemplos de empleo de XSG. El primero de ellos para la realización de operaciones de máscaras, mientras que el segundo está orientado al cálculo de autovalores y autovectores en PCA.

Por último, dentro del campo de flujo óptico, cabe destacar los trabajos desarrollados en [Díaz, 2006]. En ellos se ha desarrollado un sistema completo, de adquisición y procesado de imágenes basado en una FPGA Virtex-II de 6M de puertas. La arquitectura propuesta ha sido orientada a la detección de vehículos en el espejo retrovisor de un automóvil. Mediante este sistema se ayuda a evitar accidentes de tráfico por invasión de un carril, cuando éste está ocupado por otro vehículo. Con este sistema se consigue una alta velocidad de procesado de imágenes en un automóvil en movimiento.

2.3. MULTIPLICACIÓN DE MATRICES.

La multiplicación de matrices es una de las operaciones esenciales dentro de un amplio campo de aplicaciones. En las aplicaciones de procesado de imágenes habitualmente se trabaja con matrices de grandes dimensiones, las cuales representan las imágenes que se desean procesar. El aumento de la resolución espacial de las imágenes lleva asociado un aumento de la carga computacional a la hora de poder manejar dichas matrices. Así, cada vez se demanda mayor velocidad en los sistemas para ejecutar dichas operaciones en tiempo real.

La inclusión de nuevas prestaciones dentro de las FPGAs en los últimos años, como por ejemplo los multiplicadores hardware de propósito específico [Xilinx, 2001] [Altera, 2006], así como el incremento del número de recursos internos, hacen que estos dispositivos sean una buena alternativa para la realización de esta operación.

En este apartado se analizan en primer lugar los diferentes trabajos que existen sobre multiplicación de matrices con dispositivos hardware, y a continuación se particulariza el estudio sobre FPGAs.

2.3.1. Multiplicación de Matrices sobre dispositivos hardware.

Con idea de minimizar el tiempo de ejecución en la multiplicación de matrices, desde mediados de los años cincuenta se han estado buscando alternativas que exploten

características de paralelismo [Pan, 1984]. Para conseguir esto habitualmente las matrices usadas son cuadradas, ya que éstas permiten su descomposición en submatrices regulares, las cuales se pueden resolver de forma independiente.

Los primeros trabajos que emplearon la idea del paralelismo en la multiplicación de matrices, planteando una arquitectura basada en procesadores sistólicos, fueron los desarrollados por Kung [Kung, 1979] [Kung, 1982]. En ambos casos se buscaba la implementación de un array regular de elementos de procesado donde éstos estaban coherentemente conectados.

Tras los primeros pasos de Kung en la paralelización de la multiplicación de matrices, en [Lee, 1988] se analizaban las diferentes configuraciones de arrays sistólicos, evaluando la eficiencia y los recursos consumidos en cada alternativa. En [Lee, 1988] se describe una alternativa que fue implementada sobre dispositivos ASIC CMOS de 3 μm y 1 μm, construyendo un sistema donde los multiplicadores internos manejaban datos de 4 bits en complemento a dos y sumadores de 9 bits. Los resultados en cuanto a área y frecuencia máxima, se pueden observar en la Tabla 2.6.

Tabla 2.6. Resultados de la implementación de un multiplicador de matrices sobre el dispositivo ASIC desarrollado por [Lee, 1998].

	Tamaño máximo de matriz	**Reloj máximo** (Para tamaño máximo de matriz)	**Número de transistores** (Para tamaño máximo de matriz)
3 μm	50 x 50	1 MHz	≈ 7M
1 μm	70 x 70	1 MHz	≈ 15M

A la vista de los resultados de la Tabla 2.6 se puede comprobar como la tecnología de fabricación influye notablemente en los resultados, obteniéndose una frecuencia máxima de funcionamiento del sistema bastante baja comparándolo con las velocidades que actualmente se manejan. También se observa, como los tamaños de matrices son relativamente inferiores con respecto a los manejados habitualmente con imágenes. Otro problema importante es la fabricación de estos dispositivos mediante circuitos ASIC. El empleo de este tipo de dispositivos obliga a trabajar con una arquitectura totalmente cerrada, sin posibilidad de reutilizarse para tamaños de matrices diferentes. A partir de los datos aportados en [Lee, 1998], si actualmente se deseara implementar dichas configuraciones en FPGAs, sabiendo que en el mercado existen dispositivos de hasta 11 M puertas (aproximadamente 22 M transistores), ello sería posible reduciendo así notablemente los costes de producción y logrando aumentar la frecuencia máxima debido a sus mejores recursos de conexión. Sin embargo, el coste en cuanto al número de recursos sería excesivo.

A finales de la década de los ochenta Jagadish [Jagadish, 1989] planteó un nuevo tipo de *arrays* de procesadores sistólicos basándose en los trabajos de Kung. Esta alternativa, implementaba el algoritmo de multiplicación de matrices de

Winograd [Winograd, 1968]. De nuevo, el sistema se basa en un conjunto de procesadores, donde cada uno está formado por un multiplicador y tres sumadores. La principal aportación de estos trabajos fue la disminución del tiempo de cálculo, a costa de emplear más recursos y de aumentar la latencia inicial. La carencia de tecnología apropiada en esa época provocó que no pudiera implementarse esta nueva propuesta sobre una plataforma específica, por lo que únicamente se ofrecen resultados desde un punto de vista teórico. Si actualmente se intentara implementar este diseño toparía de nuevo con el principal inconveniente de este algoritmo: el elevado número de multiplicadores y sumadores hardware a utilizar. Además, si se quisiera manejar matrices de un orden elevado, por ejemplo 512x512, se haría muy difícil el empleo de esta alternativa, ya que el número de recursos internos consumidos alcanzaría un orden muy elevado

Simultáneamente a la evolución de las arquitecturas sistólicas basadas en dispositivos ASIC, en 1985 Xilinx introdujo en el mercado la primera familia de FPGAs. Gracias a ellas se construyeron las primeras máquinas de cómputo (*Custom Computing Machines*) basadas en FPGAs (SPLASH 2 [Buell, 1996] o PAM [Mencer, 1998]) que proporcionaron una nueva línea de plataformas. La configuración de estas nuevas máquinas con unidades de control, aritméticas y de flujo de datos totalmente flexible, permitía el empleo de éstas en numerosas aplicaciones. En los dos sistemas más representativos de esta tendencia (SPLASH 2 y PAM), se diseñaron un conjunto de librerías que permitían trabajar en alto nivel (normalmente en C++). Esto obligaba al diseño de un compilador a medida que transformaba la programación de alto nivel en una lista de conexiones (*netlist*), específica para la herramienta de implementación. En el caso de multiplicación de matrices, esta operación también podía realizarse en alto nivel abstrayéndose así del hardware sobre el que se ejecutaría la operación. Por el contrario, los retardos que se alcanzaban en este tipo de sistemas, debido a la poca optimización obtenida por parte de la herramienta de compilación, eran bastante elevados. Así, los tiempos alcanzados eran totalmente dependientes de la interpretación que realizaba el compilador. Sirvan como ejemplo los resultados expuestos en la Tabla 2.7 para la multiplicación de dos matrices de 4x4 elementos, implementados sobre la plataforma PAM-Blox 1 y 2.

Tabla 2.7. Resultados de multiplicación de dos matrices en plataforma PAM.

	Frecuencia reloj máxima (f_{CLK})	**Tiempo de ejecución**
PAM-Blox 1	33 MHz	27 T_{CLK}
PAM-Blox 2	33 MHz	19 T_{CLK}

La evolución en la tecnología de fabricación de las últimas décadas ha provocado la aparición de nuevos dispositivos y sistemas, capaces de implementar en su interior operaciones de multiplicación de matrices. Uno de estos dispositivos son los DSPs los cuales en base a una arquitectura fija permiten alcanzar elevados ratios

de rendimiento y velocidad. A la hora de desarrollar un algoritmo de multiplicación de matrices en un DSP no se plantean muchas dificultades, ya que su elevada velocidad en la ejecución de instrucciones secuenciales, permite alcanzar bajos tiempos de ejecución.

También el avance en la tecnología ha permitido dotar a dispositivos como las FPGAs de multiplicadores hardware de propósito específico. Esto ha acelerado notablemente la ejecución de multiplicaciones aritméticas en estos dispositivos, sin consumir ningún bloque lógico. Sirva como ejemplo el estudio realizado en [Scrofano, 2002] donde se muestra una comparativa del número de multiplicadores y su frecuencia máxima de funcionamiento entre una FPGA (Virtex II-Pro), un DSP (TMS320C6415) y un procesador (PXA250) (ver Tabla 2.8.)

Tabla 2.8. Comparativa de velocidades entre multiplicadores de una FPGA (Virtex II-Pro), de un DSP (TMS320C6415) y de un procesador embebido (PXA250).

Dispositivo	Tecnología	Frecuencia reloj (MHz)	Multiplicadores /precisión
Virtex II-Pro	0.13 μm	300	556 (18x18 bits)
TMS320C6415	0.12 μm	400, 500, 600	8 (8x8 bits)
PXA250	0.18 μm	400	2 (16x16 bits)

2.3.2. Multiplicación de Matrices sobre FPGAs.

Como se ha comentado anteriormente, la gran evolución tecnológica que se ha desarrollado en las últimas décadas, ha permitido la aparición de dispositivos cada vez más potentes. En el caso de las FPGAs, esto se ve acrecentado por la gran demanda que tienen. Esta situación provoca que los fabricantes de estos dispositivos saquen al mercado continuamente nuevos modelos, siendo éstos más potentes con respecto a las versiones anteriores. Debido a estas razones, el empleo de FPGAs se está expandiendo muy rápidamente a numerosas áreas, apareciendo continuamente nuevas aplicaciones.

El primer problema a resolver para la multiplicación de matrices en FPGAs, fue la implementación de multiplicadores aritméticos en FPGAs. Estos se diseñaban mediante recursos de lógica programable dentro de la FPGA [Thornton, 1999] [Li, 1996]. Ésta era la única alternativa que había, ya que hasta finales de los años 90 los principales fabricantes de FPGAs no implementaron multiplicadores hardware de propósito específico [Xilinx, 2001], [Altera, 2006]. En el caso de la multiplicación de matrices empleando FPGAs con un fin específico, los primeros trabajos relacionados datan de principios del año 2000 [Amira, 2000], [Amira, 2001], [Jang, 2002], [ElGindy, 2002].

La mayoría de los trabajos relacionados al respecto, se concentran entorno a dos grandes grupos de investigación: Universidad de Queens (Reino Unido), Universidad del Sur de California (Estados Unidos). A continuación se comentan algunos de sus principales trabajos, junto con los de otros grupos de investigación en este campo.

2.3.2.1. Universidad de Queens.

El grupo de trabajo del Departamento *Computer Science* de esta Universidad, realizó a principios del año 2000 trabajos relacionados con la multiplicación de matrices mediante arrays sistólicos de elementos de procesado [Amira, 2000]. Inicialmente, los primeros pasos fueron encaminados hacia el diseño de elementos de procesado, capaces de multiplicar datos de pequeño tamaño (máximo 4 bits). La arquitectura sistólica formada por estos elementos buscaba en todo momento alcanzar bajos tiempos en la generación de los datos de salida (*throughput*), a costa de elevar el tiempo necesario para generar el primer dato a la salida (latencia inicial). Sin embargo, los resultados obtenidos sólo permitían trabajar con matrices cuadradas de tamaño máximo de 8x8 elementos. Éstos debían estar codificados con 4 bits, alcanzándose una frecuencia de reloj máxima de 40 MHz.

Uno de los principales problemas que tuvieron que superar en este primer trabajo fue la implementación del multiplicador hardware. Inicialmente, se diseñó un multiplicador basado en el algoritmo de Bauhg-Wooley [Parhi, 1999]. Éste permitía multiplicar números en complemento a dos de una longitud reducida, debido principalmente al excesivo consumo de sumadores. Con idea de solventar este problema en [Amira, 2001] se proponía como alternativa el empleo de multiplicadores en base al algoritmo de Booth [Parhi, 1999]. El uso de este multiplicador reducía el empleo de sumadores parciales. También se modificó la metodología de los trabajos anteriores, proponiéndose una nueva línea basada en el empleo de unas novedosas unidades de multiplicación y acumulación (MAC) (ver Figura 2.6). Los resultados obtenidos ante una misma matriz con dimensiones reducidas se muestran en la Tabla 2.9. De nuevo, en este trabajo no se contemplaba el manejo de matrices de dimensiones elevadas.

Tabla 2.9. Recursos consumidos y frecuencias alcanzadas por las propuesta de [Amira, 2001] con respecto a los valores de PAM-BLox, para matrices de 4x4 y 8 bits de longitud de palabra.

Tipo de diseño	CLBs	Bloques de RAM	Frecuencia (MHz)	Ratio (Área/Velocidad)
Nueva MAC	296	4	60	4.93
MAC Clásica	270	4	46	5.87
PAM-BLox	477	--	33	14.45

Este mismo grupo de investigación también desarrolló una alternativa para aplicaciones que necesitaban trabajar con matrices de gran tamaño [Amira, 2002] [Bensaali, 2003]. En este caso la propuesta se basaba en la descomposición de la matriz en submatrices de 2x2 elementos. La principal cualidad de este sistema era el empleo de un bloque multiplicador de matrices de tamaño 2x2, donde mediante el diseño de un *pipeline* se podía alcanzar pleno rendimiento tras vencer una latencia inicial. En la Figura 2.7 se puede ver la estructura del sistema. Se aprecia un bloque encargado de fraccionar la matriz de entrada en submatrices de 2x2, un bloque multiplicador-acumulador con estructura similar a la arquitectura expuesta en la Figura 2.6, y un bloque encargado de sumar y acumular coherentemente los resultados parciales obtenidos.

El sistema implementado permitía multiplicar matrices de $N \times N$ elementos por un vector de dimensión N, alcanzando una frecuencia máxima de ejecución de aproximadamente 24MHz para $N = 1024$ con tamaños de datos de 16 bits. Para este caso se construyó un array sistólico de elementos de procesado, donde la estructura interna de cada procesador era como la mostrada en la Figura 2.7.

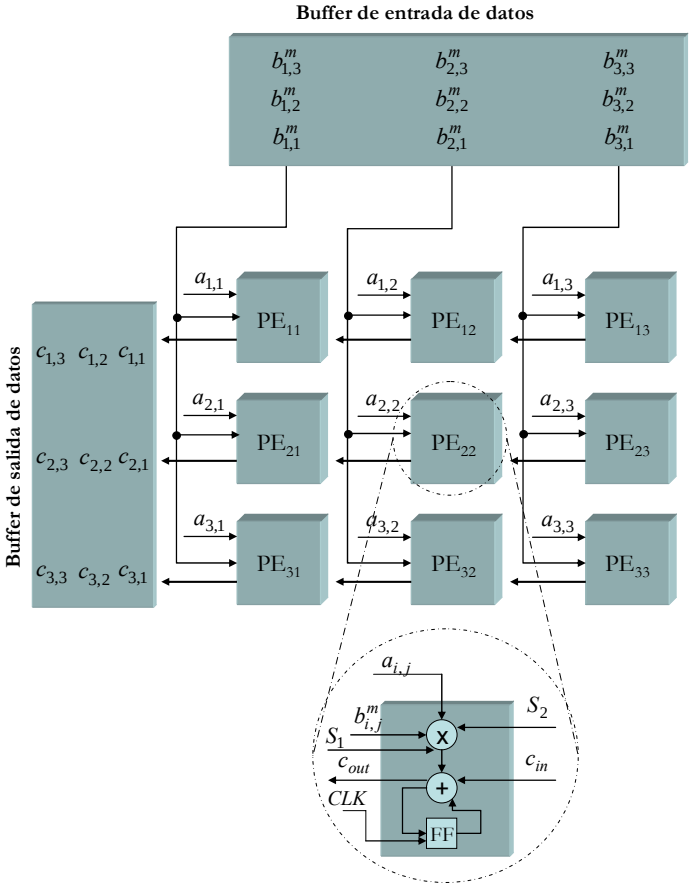


Figura 2.6. Arquitectura sistólica para la implementación del producto de matrices de 3x3 [Amira, 2001].

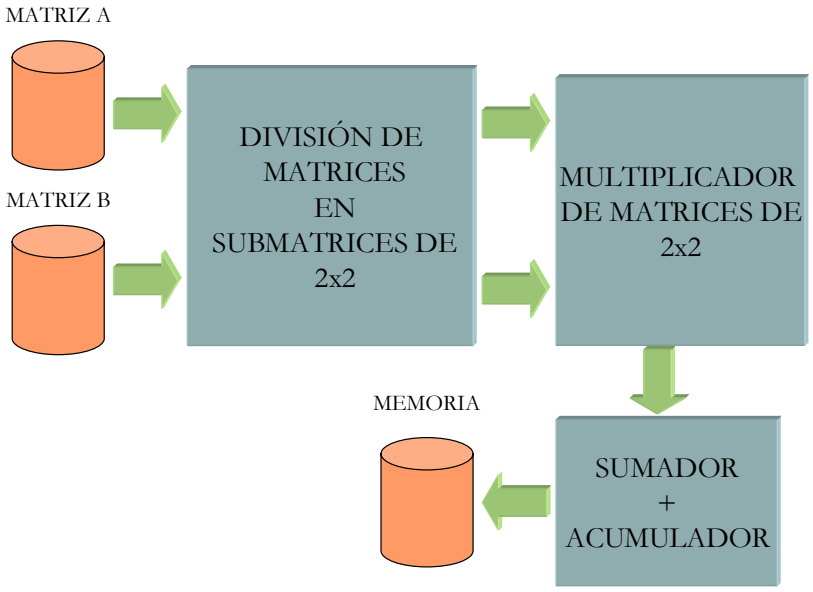


Figura 2.7. Sistema de multiplicación de matrices de gran tamaño de la Universidad de Queens.

2.3.2.2. Universidad del Sur de California.

Otro grupo de trabajo que ha obtenido interesantes resultados en la multiplicación de matrices, es el Departamento de Electricidad de la Universidad del Sur de California. Los primeros trabajos de este grupo datan del año 1991 [Prassana, 1991]. La mayoría de ellos se centraban en el estudio comparativo de la velocidad, área y potencia consumida con diferentes estrategias de multiplicación de matrices [Jang, 2002], [Jang, 2002b] [Choi, 2003].

Los trabajos mostrados en [Jang, 2002] y [Jang, 2002b] estaban basados en la disminución de la latencia inicial del sistema de multiplicación. Para ello se propuso la implementación de una arquitectura sistólica lineal (ver Figura 2.8), donde cada Elemento de Procesado (EP) empleaba una estructura formada por un multiplicador, un registro y un bloque de memoria RAM en la que almacenar datos. Esta arquitectura alcanzaba un alto rendimiento comparándola con los primeros trabajos de la Universidad de Queens [Amira, 2000]. Sin embargo el uso de esta plataforma no era apropiada para aplicaciones con matrices de gran tamaño, debido una vez más al elevado número de recursos consumidos (véase en la Figura 2.8 como cada EP posee un multiplicador, registro y un bloque de memoria).

Con idea de disminuir los recursos consumidos en las propuestas anteriormente expuestas, en [Choi, 2003] se proponía una nueva estructura de EP que consumía menor número de recursos de la FPGA, a costa de penalizar el tiempo de ejecución del algoritmo. Para ello, se proponía una factorización de las matrices de entrada mediante la descomposición LU [Choi, 1996]. Utilizando este modelo, se realizaron ensayos con matrices de entrada de 1024 x 1024 elementos con datos de

16 bits. En este caso el tiempo de ejecución asciende aproximadamente a dos segundos.

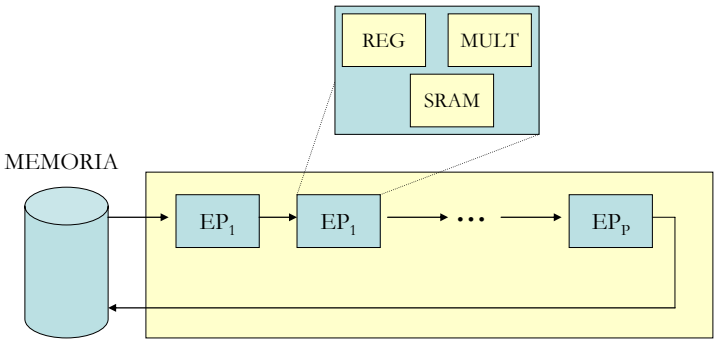


Figura 2.8. Multiplicación de matrices en un array lineal de elementos de procesado (EP).

2.3.2.3. Trabajos de otros grupos de investigación.

En [Morales, 2003] se proponía un array sistólico implementado sobre una FPGA, donde cada EP se encargaba de multiplicar cada fila de una matriz $\mathbf{A} \in \Re^{m \times k}$, por una sola columna de la matriz $\mathbf{B} \in \Re^{k \times n}$ (Figura 2.9). Por tanto, para multiplicar dos matrices de dimensiones $m \times k$ y $k \times n$ se necesitaban $k \times k$ EPs. Cada uno de estos elementos estaba formado por una unidad MAC compuesta de un sumador, un acumulador y un multiplicador. El trabajo sólo expone resultados para matrices pequeñas (hasta 7x7). Con este tipo de arquitecturas no es viable aplicaciones con matrices de gran tamaño, debido al excesivo número de recursos internos consumidos.

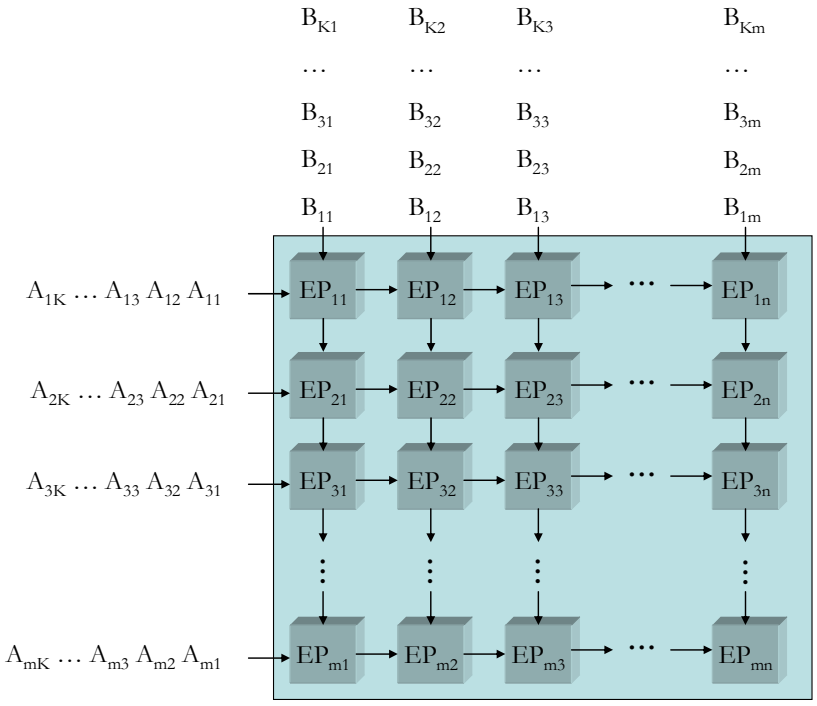


Figura 2.9. Array sistólico para realizar la multiplicación de matrices según [Morales, 2003].

En [Jianwen, 2004] se proponía otra arquitectura sistólica de EPs, que permitía trabajar con matrices de hasta 48x48 elementos. El rendimiento de esta propuesta, desde un punto de vista tiempo de ejecución, mejoraba con respecto a otros sistemas de multiplicación de matrices, a costa de consumir un número superior de recursos.

2.4. CÁLCULO DE AUTOVALORES Y AUTOVECTORES EN HARDWARE.

El cálculo de autovalores y autovectores es un problema que se presenta en numerosas aplicaciones prácticas de diferentes áreas de ingeniería (procesado de imágenes, procesado de señales, electrónica de potencia, etc.). Habitualmente el cómputo de autovalores y autovectores es realizado sobre arquitecturas basadas en procesadores secuenciales [Choi, 1997]. Sin embargo, el empleo de este tipo de arquitecturas conlleva un excesivo tiempo de cómputo debido a la elevada recurrencia que se posee. Esto justifica la continua búsqueda de alternativas para la paralelización del cálculo. Por el contrario, el empleo de este tipo de arquitecturas secuenciales garantiza una alta precisión al realizar sus operaciones aritméticas en coma flotante.

El método de Jacobi es una alternativa ampliamente empleada sobre diferentes plataformas y aplicaciones, sobre todo para aquellas que requieran un procesamiento en tiempo real [Wilkinson, 1999]. Esta propuesta, que data del año 1846, realiza la descomposición de una matriz en sus valores singulares (SVD), mediante transformaciones sobre ella. La única premisa que impone este método sobre la matriz inicial es que ésta sea cuadrada y simétrica [Brent, 1985].

Las características de este algoritmo permite que se readapte su secuencialidad a un conjunto de bloques que se ejecuten en paralelo, formando así una arquitectura sistólica [Brent, 1985]. Esta cualidad permite la implementación de dicho método para diferentes tamaños de matrices de forma cómoda, ya que la arquitectura es fácilmente ajustable o adaptable al tamaño deseado.

Desde el punto de vista de plataformas sobre las que se aplica el método de Jacobi o variantes de éste, se puede establecer una clasificación en tres grandes grupos:

- Plataformas basadas en multiprocesadores.
- Plataformas basadas en dispositivos ASIC.
- Plataformas basadas en dispositivos FPGA.

También existe un conjunto de estudios teóricos que proponen variantes y alternativas al método clásico de Jacobi, cuya aportación es muy interesante a la hora de la implementación final del sistema de cálculo de autovalores y autovectores de esta tesis.

La mayoría de los trabajos que se citan a continuación están basados en la arquitectura sistólica que Brent y Luk propusieron en [Brent, 1985]. Esta alternativa es considerada como el punto de partida de cualquier aplicación que implemente el método de Jacobi de forma concurrente.

Todos los trabajos expuestos a continuación se basan en la implementación de un array sistólico de EPs (procesadores) coherentemente ordenados, donde cada uno computa 2x2 elementos de la matriz inicial (Figura 2.10). Tras un número finito de iteraciones se consigue obtener la matriz de autovalores y autovectores.

A continuación se presentan los trabajos más destacados, tanto de propuestas de variantes del método de Jacobi, como de trabajos asociados a las diferentes plataformas sobre las que ha sido implementado el citado algoritmo.

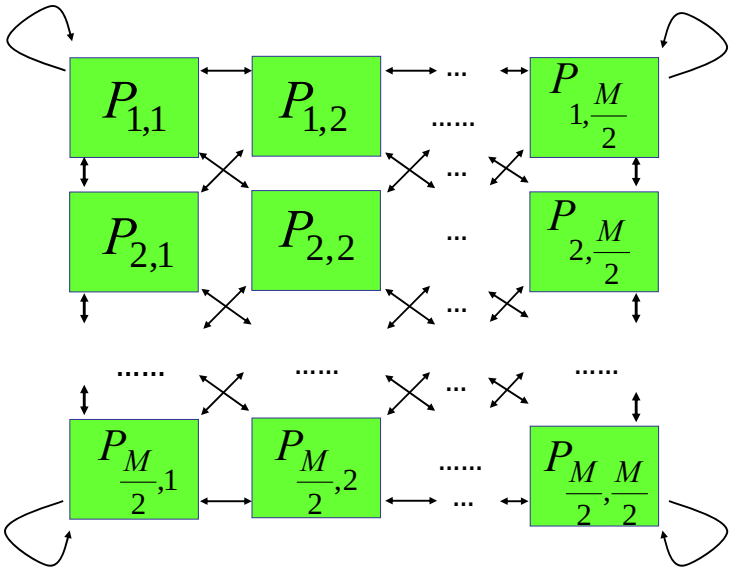


Figura 2.10. Arquitectura sistólica propuesta por Brent, compuesta por $M/2 \times M/2$ procesadores, orientada al cálculo autovalores y autovectores de una matriz $M \times M$ [Brent, 1985].

2.4.1. Plataformas basadas en multiprocesadores.

La implementación del método de Jacobi sobre una plataforma secuencial encuentra como principal hándicap la transferencia de datos entre la memoria y la CPU. Además, si bien es verdad que el tipo de operaciones aritmético-lógicas que se deben realizar en la CPU en el cálculo de autovalores es muy complejo, éste no es un gran problema. Sí lo es el volumen de datos que se debe transferir entre la memoria y la CPU, por lo que en un sistema secuencial aparece un cuello de botella en los accesos a memoria. A partir de la propuesta formulada por [Brent, 1985] en la que se descompone el algoritmo de Jacobi en módulos que se pueden ejecutar en paralelo, los trabajos expuestos a continuación fueron orientados hacia la implementación de dicha variante en varios procesadores. De esta manera se reducía notablemente el cuello de botella en los accesos a memoria, debido a la reducción del volumen de datos que manejaría cada CPU.

El primer trabajo se remonta a 1995, donde miembros de la Universidad Nacional de Australia (universidad a la que pertenecía R. Brent) implementaron el algoritmo de Jacobi sobre diferentes plataformas multiprocesador [Zhou, 1995]. La plataforma escogida fue un cluster Fujitsu VPP500, el cual estaba inicialmente configurado con 4 procesadores. Gracias a la potencia computacional que ofrecía este sistema, se realizaron numerosos tests sobre matrices de gran tamaño, cuyos resultados se muestran en la Tabla 2.10.

Tabla 2.10. Resultados de la implementación del método de Jacobi sobre la máquina Fujitsu VPP500 [Zhou, 1995].

Tamaño Matriz	Iteraciones	Tiempo (s)	Mflops
1000	13	30.73	1365
1400	13	73.02	1626
1800	13	134.68	1879
2000	13	175.2245	1983
2400	13	299.78	2006
2800	13	444.63	2151
3200	13	625.64	2285

Continuando con esta línea, un grupo de investigación perteneciente a la Universidad de Murcia (España) presentó en 1996 los resultados obtenidos para la máquina Touchstone DELTA [Giménez, 1996]. Este sistema estaba compuesto por un total de 512 procesadores i860 conectados entre sí. En la Tabla 2.11 se muestran algunos ejemplos obtenidos del método de Jacobi con dicha máquina. Cada procesador ejecutaba una tarea, donde gracias a una red de alta velocidad que los interconectaba entre sí, se transmitían entre ellos los resultados parciales que habían generado cada procesador.

Comparando los resultados obtenidos de la Tabla 2.11 con los de la Tabla 2.10, se observa como los tiempos de ejecución para la máquina Touchstone mejora en un 25% aproximadamente los tiempos de la máquina Fujitsu VPP500. Esta reducción de tiempos viene justificada por las características de esta segunda plataforma desde el punto de vista de interconexión de procesadores y número de éstos, favoreciendo así la disminución del tiempo de ejecución.

Coetáneo al trabajo presentado en [Giménez, 1996] es el mostrado en [Pan, 1996] desarrollado por investigadores pertenecientes a la Universidad de Dayton (Estados Unidos). Surge ante la necesidad de disminuir el retardo en la transferencia de datos entre los diferentes procesadores. Gracias a esto se conseguía reducir el tiempo de propagación. Para ello, se desarrolló una plataforma de multiprocesadores

conectados entre sí mediante un conjunto de buses ópticos, formando así un array lineal (ver Figura 2.11). Esta nueva red de comunicaciones aceleraba la transmisión de datos entre los diferentes procesadores. Cada procesador estaba conectado a un bus óptico por medio de un interfaz observándose en la Figura 2.11 como los interfaces eran direccionales, utilizándose cada uno de ellos para un sentido de la comunicación. Los datos que viajaban por los buses, estaban codificados de tal forma que cada procesador reconocía si él era el destinatario del mensaje enviado desde otro procesador. No se disponen de resultados prácticos obtenidos sobre esta plataforma.

Tabla 2.11. Tiempos de ejecución, en el cálculo de autovalores mediante el método de Jacobi mediante la máquina Touchstone DELTA [Giménez, 1996].

Procesadores			
Tamaño matriz	**4**	**16**	**64**
512	10.8s	4.5s	2.0s
1024	66.3s	23.5s	9.1s
1536	202.7s	64.8s	23.6s
2048	--	138.0s	47.7s

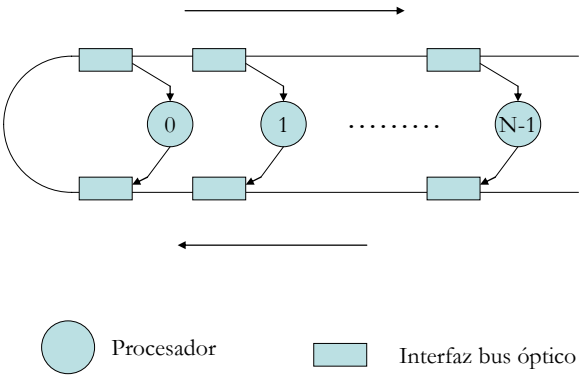


Figura 2.11. Arquitectura con múltiples procesadores unidos con bus óptico [Pan, 1996].

En los inicios del 2000, coincidiendo con la fabricación de las nuevas familias de FPGAs, aparecen los primeros trabajos basados en multiprocesadores, donde se empleaban FPGAs como elementos de coprocesado. Así, en 2001 un grupo de investigación perteneciente a la Universidad de Paderborn (Alemania) [Bobda, 2001] desarrolló un sistema híbrido en el que mediante un cluster de *Workstations* y con FPGAs, implementaron el algoritmo de Jacobi propuesto en [Brent, 1985]. Para ello se emplearon dispositivos FPGAs asociados a una CPU con idea de:

- Reducir el tiempo de cómputo en la CPU, descargando ciertas tareas sobre las FPGAs.

- Posibilitar la reconfiguración dinámica de la operatividad de las FPGAs, en función de la carga que tuviera en ese momento la CPU.
- Procesar a bajo nivel un gran volumen de datos.

Las FPGAs utilizadas eran de Xilinx y pertenecían a la familia 4000. El rendimiento de éstas dentro del algoritmo de Jacobi fue comparado con el empleado por un Pentium III a 450 MHz. A medida que el tamaño de las matrices iba aumentando se incrementaba el rendimiento de la propuesta FPGAs frente al Pentium, con lo que quedó justificado su empleo.

En la Figura 2.12 se presenta el cluster utilizado en [Bobda, 2001] para el cálculo de autovalores y autovectores mediante el método de Jacobi. Se puede observar en dicha figura como cada celda del cluster estaba compuesta por una FPGA, una CPU y una memoria RAM. Cada celda no se correspondía con una *Workstation* sino que varias conformaban una única *Workstation*.

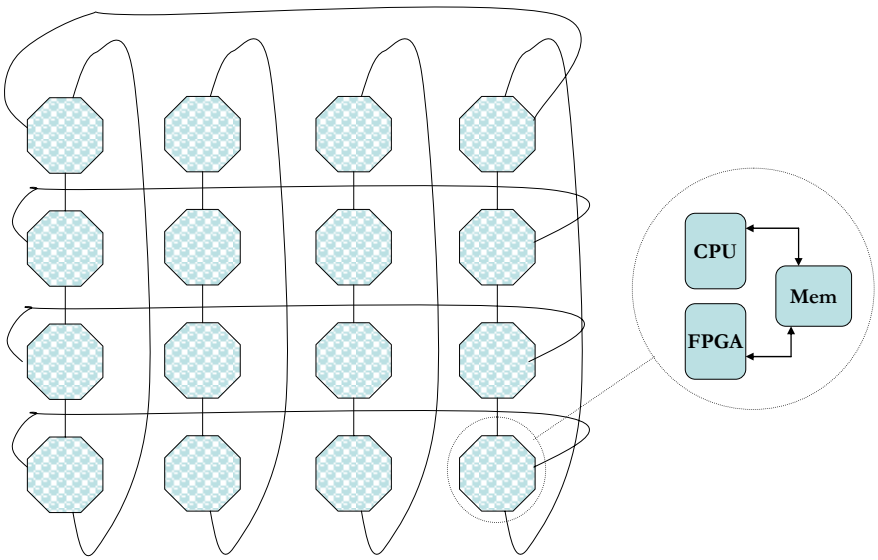


Figura 2.12. Cluster para el cálculo de autovalores mediante el método de Jacobi, empleando celdas híbridas (FPGA+*Workstation*) [Bobda, 2001].

2.4.2. Plataformas basadas en dispositivos ASIC.

El primer trabajo que implementó la propuesta mostrada en [Brent, 1985] fue desarrollado por Cavallaro y Luk en [Cavallaro, 1987] y [Cavallaro, 1988], perteneciendo ambos a la Universidad de Ithaca (Estados Unidos). La propuesta planteó el desarrollo de un sistema ASIC capaz de realizar el algoritmo de Jacobi sobre matrices de 2x2 elementos. Para matrices de entrada de tamaño superior a 2x2 elementos, éstas se dividían en submatrices de 2x2 elementos, necesitando por tanto un módulo ASIC para cada submatriz. Se justifica en [Cavallaro, 1989] que si se transmiten los resultados parciales obtenidos por cada submatriz de manera ordenada al final se consigue obtener los autovalores de la matriz total, lo cuál será empleado en la presente tesis. Por ello, el objetivo fue la implementación de una arquitectura

sistólica similar a la mostrada en la Figura 2.10 donde cada procesador era un dispositivo ASIC.

Además de la aportación de la implementación física de la propuesta de Brent, Cavallaro y Luk propusieron el uso de algoritmos CORDIC como elementos centrales de cálculo de todas las operaciones trigonométricas necesarias. El empleo de este algoritmo viene justificado desde un punto de vista hardware, por el escaso consumo de recursos, así como por el reducido tiempo de cómputo que requiere siendo empleado ampliamente con FPGAs [Andraka, 1998]. Por tanto, uno de los principales objetivos de estos trabajos fue el diseño de un módulo CORDIC óptimo. En la Figura 2.13 se muestra la arquitectura CORDIC diseñada para estos dispositivos ASIC. Su estructura, corresponde a una configuración clásica de CORDIC donde se optimizaron los tiempos de propagación y el número de sumadores y registros.

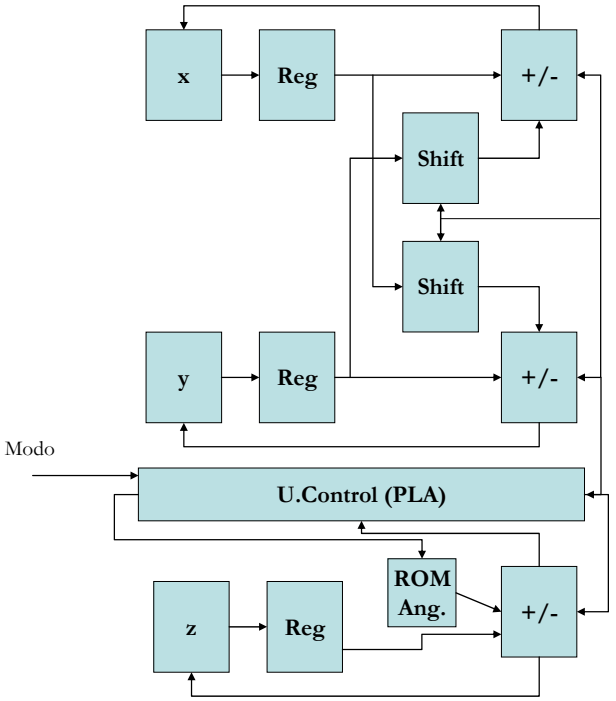


Figura 2.13. Módulo CORDIC implementado por Cavallaro y Luk para el cálculo de autovalores y autovectores [Cavallaro, 1987].

Desde un punto de vista de resultados prácticos, la tecnología de fabricación de los dispositivos ASIC limitó notablemente el funcionamiento del sistema. Así, usando un conjunto de 10x10 procesadores, acotando por tanto el tamaño máximo de la matriz de entrada a 20x20 elementos en coma fija, se empleaba un tiempo de 1 ms [Cavallaro, 1988]. También Cavallaro realizó en el año 1992 una arquitectura ASIC idéntica a la descrita anteriormente, pero para calcular autovalores complejos [Hemkumar, 1992].

Otro conjunto de trabajos destacados donde se aplicó el algoritmo de Brent en dispositivos ASIC fueron los dirigidos por el profesor Lang de la Universidad de

California (Estados Unidos). Además de implementar, casi en la misma época que Cavallaro, la arquitectura de Brent, sus trabajos fueron encaminados a la optimización de los módulos CORDIC [Lee, 1991] [Ercegovac, 1990]. De nuevo, el objetivo buscado fue la reducción del área consumida por estos módulos dentro de un dispositivo ASIC, así como la reducción del tiempo de ejecución.

2.4.3. Plataformas basadas en dispositivos FPGA.

La posibilidad de paralelizar el método de Jacobi ha permitido su implementación sobre FPGAs. El conjunto de trabajos existentes al respecto, se puede sintetizar en los realizados en la Universidad Nacional de Yokohama (Japón) y en la Universidad de Queens (Reino Unido). Ambos fueron realizados en la misma época y los dos fueron aplicados al algoritmo MUSIC (MUltiple SIgnal Classification). Este algoritmo detecta frecuencias de una señal mediante el cálculo de los autovalores sobre la matriz de covarianza de un vector de datos y mediante las muestras recibidas de una señal [Cheney, 2001].

El primer trabajo encontrado data del año 2002 y fue realizado por miembros de la Universidad Nacional de Yokohama [Kim, 2002]. En este trabajo, se presenta una propuesta de arquitectura de procesado de autovalores, siendo de nuevo la optimización de los módulos CORDIC parte fundamental del trabajo desarrollado. En la Figura 2.14 se muestra la arquitectura propuesta por este grupo, donde se observan básicamente 3 elementos:

- *Módulo R (Correlación)*: En este módulo se calculan los autovalores
- *Módulo E (Autovectores)*: Este módulo se encarga de calcular los autovectores asociados a los autovalores calculados en el Módulo R.
- *Módulo CORDIC*: Encargado de realizar rotaciones de vectores y cálculo de arco tangentes.

En el año 2003 la propuesta realizada en [Kim, 2002] es llevada a la práctica [Kim, 2003]. La estructura diseñada únicamente manejaba una matriz de 2x2 elementos. La FPGA utilizada tenía un tamaño de 600K puertas de Altera, presentándose en la Tabla 2.12 los recursos consumidos. En esta misma tabla podemos comprobar el elevado número de ciclos empleados en el cálculo de autovalores y autovectores de una matriz de 2x2 elementos.

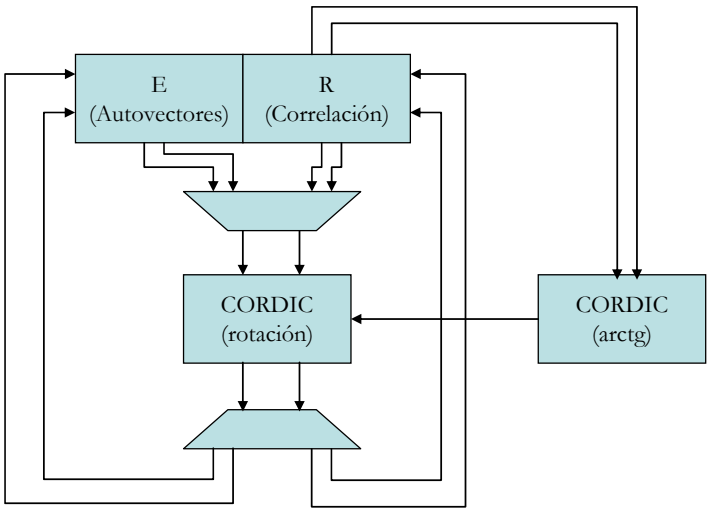


Figura 2.14. Arquitectura para el cálculo de autovalores y autovectores dentro del algoritmo MUSIC según [Kim, 2002].

Tabla 2.12. Resultados del cálculo de autovalores para la arquitectura propuesta en [Kim, 2003].

Operación realizada	Ciclos de reloj necesarios	Máxima Frecuencia	Elementos Lógicos consumidos
Cálculo Autovalores	1836	110 MHz	1836
Correlación	32	27.4 MHz	8301

El otro grupo de investigación que ha desarrollado trabajos relacionados con el método de Jacobi sobre FPGAs, pertenece a la Universidad de Queens. Los trabajos publicados hasta el momento datan del año 2004 [Ahmedsaid, 2004] [Ahmedsaid, 2004b] y como se comentó anteriormente su aplicación está también orientada hacia el algoritmo MUSIC.

La arquitectura propuesta por este grupo de trabajo se puede observar en Figura 2.15, siendo su estructura análoga a la propuesta en [Cavallaro, 1988]. En ella se pueden identificar los siguientes módulos:

- *Módulo de autovalores*: Formado por una estructura sistólica de módulos CORDIC, encargados de realizar el cálculo de autovalores.
- *Módulos interfaces de entrada y salida*: Compuestos básicamente por registros, encargados de preparar los datos de la matriz sobre la que calcular los autovalores. Esta matriz es precargada en una memoria externa.

- *Unidad de Control*: Encargada de generar los comandos necesarios para el módulo de autovalores y de manejar todas las operaciones de entrada y salida entre la FPGA, la memoria externa y ocasionalmente la comunicación con un PC.

Uno de los principales logros alcanzados en [Ahmedsaid, 2004] es la optimización de la transmisión de los resultados parciales entre los módulos CORDIC. Otra aportación de este grupo de investigación fue el incremento de la velocidad de transmisión de los resultados parciales entre los diferentes procesadores. Para ello, el sistema trabaja con dos frecuencias de reloj, una para calcular los autovalores (frecuencia interna de los procesadores) y otra para la transmisión de datos parciales. Esta segunda era más rápida que la primera, disminuyendo así la latencia existente desde que se genera un resultado parcial hasta que este es propagado al resto de procesadores.

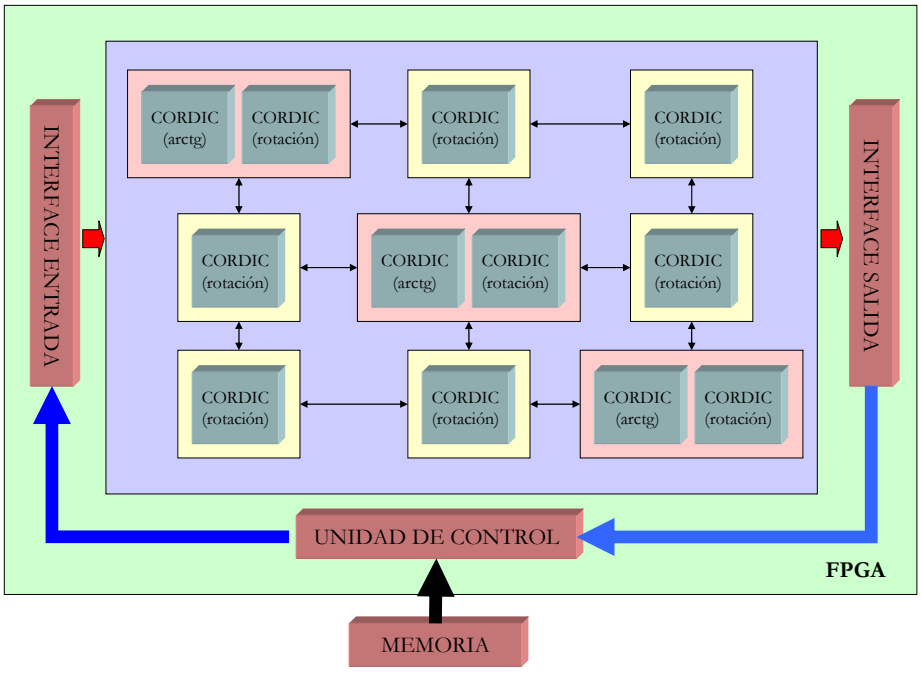


Figura 2.15. Arquitectura completa propuesta por la Universidad de Quens para el cálculo de autovalores y autovectores según [Ahmedsaid, 2004].

Estas propuestas fueron implementadas sobre una FPGA de 2M puertas de Xilinx, alcanzando los resultados expuestos en la Tabla 2.13.

Tabla 2.13. Tiempos de ejecución y recursos consumidos en el cálculo de autovalores por la propuesta desarrollada en [Ahmedsaid, 2004b].

Tamaño matriz		**4x4**	**6x6**	**8x8**
Tamaño datos				
14 bits	**Ciclos de reloj**	2652	4873	7531
	Máxima Frecuencia	87.73 MHz	96.15 MHz	85.98 MHz
	Tiempo consumido	30.22 μs	50.68 μs	87.59 μs
	Área ocupada	25 %	51 %	86 %
16 bits	**Ciclos de reloj**	2928	5379	8177
	Máxima Frecuencia	96.38 MHz	88.54 MHz	84.44 MHz
	Tiempo consumido	30.38 μs	60.75 μs	96.84 μs
	Área ocupada	28 %	58 %	99 %

2.4.4. Propuestas teóricas de mejora del algoritmo de Jacobi.

Una vez visto en el apartado anterior diferentes implementaciones del método de Jacobi, en este apartado se presentarán variantes de dicho método, desarrolladas en la década de los 90. Casi todas ellas estaban orientadas hacia la mejora del algoritmo CORDIC, ya que éste es el elemento fundamental del algoritmo de Jacobi desde un punto de vista de implementación. Todas las propuestas presentadas en este apartado no llegaron a implementarse.

Un ejemplo es la propuesta de mejora desarrollada en [Chen, 1999]. En él se presentaba una nueva forma de computar CORDIC en el cálculo de autovalores, mediante la acumulación de una serie de resultados parciales. Esta acumulación permitía disminuir el tiempo de cómputo total, ya que recurrentemente se utilizaban esos resultados. Sin embargo, esta alternativa carece de sentido si se ejecutara en una FPGA o dispositivo ASIC, ya que conllevaría el empleo de memoria externa para acumular estos resultados, así como el empleo de nuevas unidades aritméticas que permitieran implementar esos resultados parciales.

Más significativas son las propuestas realizadas por [Götze, 1993] y [Ma, 1999]. En [Götze, 1993] además de proponer una nueva variante de CORDIC, se propone una reducción de la arquitectura de Brent (ver Figura 2.10) sustituyéndola por una arquitectura triangular como la mostrada en la Figura 2.16. Debido a las

condiciones de simetría de la matriz sobre las que calcular los autovalores, se puede eliminar en el array sistólico de procesadores (los ubicados en la parte inferior de la diagonal o los ubicados en la parte superior). En esta tesis se aprovechará esta cualidad a la hora de abordar el cálculo de autovectores.

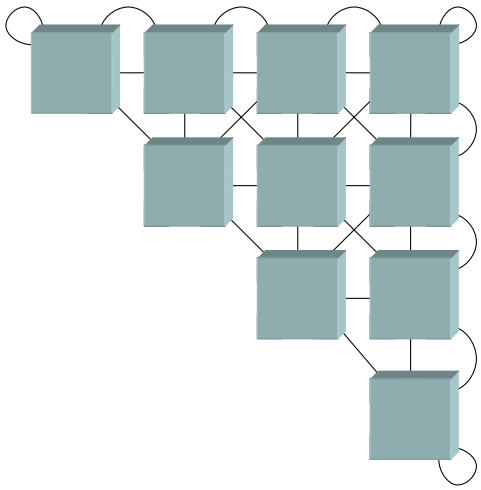


Figura 2.16. Arquitectura sistólica triangular basada en [Brent, 1985], presentada en [Götze, 1993], para el cálculo de autovalores y autovectores.

2.5. ALGORITMO PCA IMPLEMENTADO EN FPGAS.

El Análisis de Componentes Principales (PCA) es una alternativa ampliamente empleada en numerosas áreas de la ingeniería. Dentro del campo del procesado de imágenes, tal y como se presentará a continuación, existen algunos trabajos que desarrollan PCA sobre FPGAs. Sin embargo, hasta la fecha no se ha encontrado ningún trabajo relacionado con la detección de obstáculos en movimiento empleando PCA que use FPGAs como elemento de procesado.

Es importante resaltar que tampoco en ninguno de los trabajos encontrados se implementa totalmente PCA sobre FPGAs, debido fundamentalmente a la gran dependencia de datos que hay dentro de PCA. Esta situación provoca una gran dificultad en la segmentación de las diferentes partes que forma PCA. Por ello, la ejecución de PCA se reparte habitualmente entre una FPGA y un PC o microprocesador.

Las propuestas presentadas en este apartado, implementan técnicas de *pipeline* en las fases de PCA que no hay dependencia de datos, alcanzando buenos resultados desde el punto de vista de precisión y óptimos en cuanto a los tiempos de ejecución y recursos consumidos.

Básicamente todos los trabajos relacionados con PCA sobre FPGAs, se agrupan entorno a los grupos de investigación de la Universidad de Hokkaido (Japón), Universidad de Essex (Reino Unido) y Universidad de Old Dominion (Estados Unidos). A continuación se hace un análisis de los trabajos más significativos de estos tres grupos de investigación.

2.5.1. Universidad de Hokkaido.

Cronológicamente fue la primera contribución que apareció sobre la implementación de PCA sobre FPGAs [Boonkumklao, 2003] [Boonkumklao, 2001]. Data del año 2001 y en ella se mostraba la implementación de PCA sobre una plataforma genérica basada en FPGAs de Altera, orientada al procesamiento de señales e imágenes. En ella existía una "especie" de array sistólico de procesadores, donde cada uno de ellos estaba formado por: multiplexores, multiplicadores, sumadores/restadores y otros elementos lógico-aritméticos. Cada uno de los procesadores estaba convenientemente controlado por un secuenciador. Todos los procesadores estaban unidos entre sí mediante una matriz de conmutaciones, que era programada convenientemente. De esta forma, reprogramando las conexiones entre procesadores y variando la operatividad de cada procesador, la plataforma podía ser empleada en múltiples aplicaciones.

La configuración interna de cada procesador permitía una elevada potencia de cálculo, ya que podía trabajar con datos en aritmética en coma flotante. Este tipo de arquitectura se podía utilizar como un *core* IP, donde el usuario simplemente debía fijar diversos parámetros, como tamaño de los datos, tamaño de la mantisa o del exponente, etc.

En cuanto a los resultados obtenidos, en la Tabla 2.14 se presentan los tiempos de ejecución para el cálculo de los autovalores y la covarianza, para diferentes tamaños de matrices. Los datos de entrada son de dimensión n, donde n puede tomar valores de 16, 22, 32 o 64 bits.

Tabla 2.14. Tiempos consumidos por la plataforma desarrollada en [Boonkumklao, 2001] para el cálculo de autovalores, en función del tamaño de los datos (n).

Tamaño matriz	**3x3**	**4x4**	**5x5**
Ciclos de reloj consumidos en el cálculo de la covarianza	$6+2.1n$	$8+2.7n$	$16+5.1n$
Ciclos de reloj consumidos en el cálculo de los autovalores	450	600	1200

2.5.2. Universidad de Essex

El trabajo presentado en [Fleury, 2004] se centraba en la adaptación de las diferentes partes que forman PCA para implementarse en una FPGA.

Para poder adaptar el funcionamiento secuencial de PCA a una forma concurrente se proponía en [Fleury, 2004] la descomposición de la citada transformada en un total de 6 tareas:

1. Cálculo de la media.
2. Cálculo de la covarianza.

3. Normalización de la media.
4. Cálculo de autovectores.
5. Promediado de datos.
6. Proyección en el espacio transformado.

Debido a la dependencia de datos entre las diferentes tareas era imposible ejecutar todas ellas de manera concurrente. Por tanto, se proponía ejecutar algunas de ellas de manera simultánea, mediante un *pipeline* de 3 etapas (ver Figura 2.17):

- Etapa 1: En esta etapa se realizaban las tareas 1 y 2.
- Etapa 2. Ejecución de las tareas 3 y 4.
- Etapa 3: Desarrollo de las tareas 5 y 6.

Sin embargo, no todas las etapas fueron implementadas sobre FPGAs, sino que la Etapa 2, debido a su complejidad computacional y al elevado tiempo de computación que podría tener en una FPGA, se decidió realizarla en un PC.

El sistema fue implementado mediante una plataforma comercial de FPGAs [Sweeney, 2001], que poseía comunicación con el PC por bus PCI. De esta forma se conseguía sincronizar la recepción/envío de datos desde/hacia un PC, de una manera rápida y eficaz.

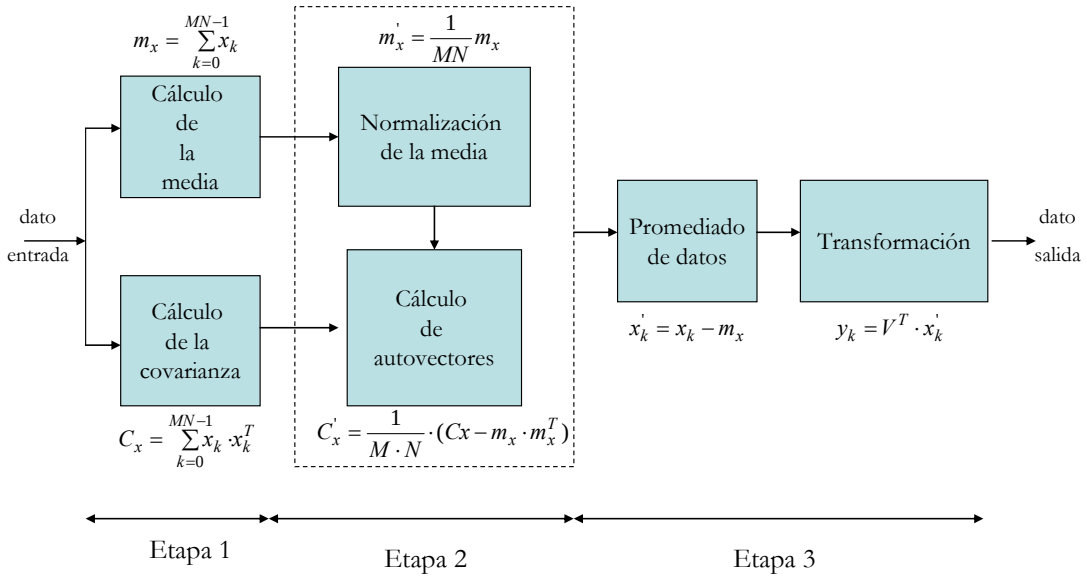


Figura 2.17. Pipeline de 3 etapas, para la ejecución en paralelo de PCA según [Fleury, 2004].

Según [Fleury, 2004] el sistema estaba teóricamente pensado para manejar imágenes de hasta 512x512 píxeles en blanco y negro, sin más que replicar la arquitectura mostrada en la Figura 2.17. Sin embargo, los únicos resultados reales que se aportaban eran para un banco de 6 imágenes de 20 píxeles cada una. La

justificación de este tamaño viene dada por las limitaciones de la plataforma comercial usada en cuanto a la capacidad de la FPGA utilizada y de la memoria externa necesaria.

El máximo tamaño que podían manejar 12 arquitecturas como la mostrada en la Figura 2.17 era de 240 píxeles, consumiendo casi el 95% de una FPGA de 1M puertas. Este tamaño es notablemente inferior a tamaños típicos de imágenes, como pueden ser 128x128 (16384 píxeles) o 512x512 (262144 píxeles). Por ello, una imagen de tamaño superior a 240 bytes debería ser dividida en varias partes y ejecutarla secuencialmente en la arquitectura desarrollada.

2.5.3. Universidad de Old Dominion

El trabajo mostrado en [Gottumukkal, 2003] presentaba una arquitectura basada en FPGAs que implementaba una variante del algoritmo PCA denominada PCA Modular [Gottumukkal, 2004]. Esta propuesta fue aplicada al reconocimiento de caras de una base de datos.

El método PCA Modular consistía en la división de una imagen inicial en subimágenes (ver Figura 2.18), y en la posterior aplicación a cada subimagen del algoritmo PCA clásico. Esta alternativa de partición de imágenes alcanzaba mejores resultados que el tratamiento de una imagen completa a medida que se aumentaba el número de subimágenes [Gottumukkal, 2003]. En la Tabla 2.15 se presentan los aciertos del método PCA y PCA modular, sobre un conjunto de imágenes, según [Gottumukkal, 2003]. Esta variante de PCA, cuando las condiciones de iluminación cambian notablemente, alcanzaba una alta efectividad de acierto.

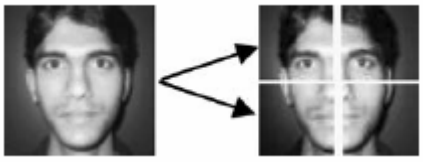


Figura 2.18. Aplicación del PCA modular sobre una imagen, generando 4 subimágenes.

Tabla 2.15. Porcentaje de acierto de PCA modular vs. PCA [Gottumukkal, 2003]

	PCA (%)	**PCA Modular**		
		4 Subimágenes (%)	**16 Subimágenes (%)**	**64 Subimágenes (%)**
Media de aciertos positivos	98,13	99,38	100	100
Media de aciertos negativos	1,88	0,63	0	0

El hecho de trabajar con subimágenes permitía reducir la complejidad computacional (ya que el número de píxeles a manejar es inferior) en ciertas tareas como el cálculo de la covarianza. Por el contrario, desde un punto de vista de cómputo, la unidad encargada de calcular la covarianza debe ser duplicada tantas veces como subimágenes se posean para poder trabajar con el máximo rendimiento.

Con respecto a la implementación de PCA modular para la detección de caras sobre una FPGA, en [Gottumukkal, 2003] sólo se presenta una arquitectura preparada para realizar parte del algoritmo PCA Modular. Dicha arquitectura se emplea para realizar la proyección y recuperación, de una cara sobre el conjunto de imágenes transformadas. Su estructura interna está básicamente formada, por un *array* sistólico lineal de elementos de procesado, un conjunto de registros y memorias, así como comparadores. Los datos necesarios para esta tarea eran calculados externamente a la FPGA, y almacenados en una memoria ROM a la que tenía acceso la FPGA. Los autores no aportan ninguna información sobre cómo se realizaba el cálculo de los autovectores y de la media aritmética de las imágenes de las caras.

Los tiempos de ejecución que se presentan en [Gottumukkal, 2003] sólo son a nivel funcional para un conjunto de imágenes de 10 personas diferentes con un tamaño de imagen de 64x64 píxeles. Realizando 256 subimágenes de las imágenes capturadas se necesitaba un tiempo total de aproximadamente 4 ms para realizar la detección.

2.6. CONCLUSIONES.

Debido a que esta tesis está enmarcada dentro del procesamiento de imágenes con FPGAs, en este capítulo se ha realizado un recorrido por los trabajos más significativos dentro del procesado de imágenes con estos dispositivos. El estudio se ha iniciado con una evaluación de los primeros sistemas basados en FPGAs, orientados hacia visión artificial y se ha finalizado, con los trabajos más destacados de implementación de PCA sobre FPGAs. Cabe destacar sobre este aspecto, como a día de hoy, la implementación de PCA sobre FPGAs es una tarea que no se ha concluido al 100%.

Dentro del análisis realizado en este capítulo, se ha evaluado el estado actual de los aspectos más significativos sobre los que está centrada esta tesis. Concretamente, se han estudiado las diferentes alternativas para realizar la multiplicación de matrices así como el cálculo de autovalores y autovectores.

Sobre los diferentes trabajos presentados, y en relación con los objetivos buscados en esta tesis (capítulo 1), se pueden sacar a modo de resumen, las siguientes conclusiones:

- Con respecto a la multiplicación de matrices (operación que se ejecuta dentro de PCA), es una tarea compleja de implementar en FPGAs debido principalmente al elevado número de recursos hardware necesarios. Casi todas las referencias encontradas apuestan por una

arquitectura de tipo sistólico. Sin embargo, cuando se trabaja con matrices de tamaño grande esta posibilidad es inviable desde un punto de vista de recursos necesarios. Por esta razón, no se han desarrollado hasta la fecha muchas variantes para solventar esta aplicación en hardware.

- El cálculo de autovalores y autovectores son algoritmos con ciertas facilidades para implementarse en hardware, gracias a los trabajos desarrollados en [Brent, 1983]. En la década de los ochenta se desarrollaron dispositivos ASIC con esta propuesta, alcanzando unos aceptables resultados desde el punto de vista de velocidad y precisión. Sin embargo, este algoritmo no ha sido aplicado a FPGAs excepto la propuesta mostrada en [Ahmedsaid, 2004b]. En este trabajo el tamaño de la matriz de entrada es relativamente pequeña, debido principalmente al elevado consumo de recursos internos de la FPGA. Este hecho también se verificó en esta tesis, tal y como se verá en el capítulo 4. Por esta razón, aunque la propuesta de Brent permita una ejecución concurrente, se hace necesario encontrar otra vía de implementación de este algoritmo modificándolo con el objetivo de no consumir demasiados recursos.

- La ejecución de PCA sobre FPGAs de una forma íntegra no se ha conseguido todavía. Además, en ninguno de los pocos trabajos encontrados se ha mostrado algún resultado en entornos exteriores.

Todas estas razones, han hecho que en esta tesis se busquen diferentes propuestas para poder solventar los problemas expuestos anteriormente.

3. ASPECTOS GENERALES DEL SISTEMA PROPUESTO

3.1. INTRODUCCIÓN.

En esta tesis se propone el diseño de un sistema autónomo de captura y procesado de imágenes basado en dispositivos FPGA. Concretamente, el sistema desarrollado permite la aplicación de la técnica PCA sobre una plataforma de propósito específico basada en FPGAs. Por tanto, los algoritmos implementados sobre las FPGAs son los encargados de realizar la captación de imágenes procedentes de un sensor CMOS, así como la ejecución de PCA.

Las características internas de las FPGAs permiten que se alcancen altas velocidades de ejecución, siempre y cuando se explote al máximo su arquitectura concurrente. Esto conlleva la división del algoritmo PCA en diversos módulos capaces de funcionar en paralelo. Con idea de alcanzar la máxima velocidad también es necesario que cada uno de los módulos que componen dicho algoritmo tenga una segmentación interna que permita también su ejecución en paralelo.

La implementación de PCA se ha particularizado para el procesado de imágenes, concretamente para la detección de objetos en movimiento. Sin embargo, el ámbito de aplicación de este método no sólo está centrado en el campo de la visión artificial, sino que también es empleado en otras aplicaciones como por ejemplo en ultrasonidos [Jiménez, 2005]. Por ello, la implementación hardware del algoritmo PCA aquí propuesto, podrá ser utilizada en otras aplicaciones sin más que

redefinir ciertos parámetros dentro del diseño realizado (ancho del bus de datos, tamaño de memoria necesaria, etc.).

3.2. CONTEXTO DE LA TESIS.

Es este apartado se muestra el contexto en el que se ha realizado la presente tesis. Ésta ha sido desarrollada íntegramente dentro del Departamento de Electrónica de la Universidad de Alcalá. Concretamente, su desarrollo se ubica en el contexto de los proyectos de investigación SILPAR I y II (SIstema de Localización y Posicionamiento Absoluto de Robots), proyecto RESELAI (Integración de redes de sensores acústicos, de visión y RFID para localización en ambientes inteligentes) y de la "Cátedra de control electrónico aplicado al transporte".

El proyecto SILPAR I (DPI 2003-05067) fue financiado por el Ministerio de Educación y Ciencia, con una duración de tres años (Septiembre 2003-2006). El objetivo principal era la localización y posicionamiento de robots dentro de un espacio inteligente con objeto de conseguir desplazamientos de éstos de una forma autónoma. Como técnicas novedosas para la localización y posicionamiento se emplearon nuevos tipos de sensores basados en infrarrojos, fibra óptica y dispositivos optoelectrónicos. Esta tesis queda enmarcada dentro del objetivo específico del proyecto: "construcción de una cámara inteligente". La cámara inteligente desarrollada en esta tesis recogerá la imagen procedente de un mazo de fibras ópticas que posteriormente se procesaba.

Como continuación de los avances obtenidos con el proyecto SILPAR I en el ámbito de los "espacios inteligentes" surgió el proyecto SILPAR II (DPI 2006-05835) financiado también por el Ministerio de Educación y Ciencia con una duración de 3 años (Octubre 2006-2009). Su objetivo principal se centra en la continuación de las investigaciones realizadas mediante técnicas de desarrollo de sistemas sensoriales optoelectrónicos y técnicas de codiseño y SoC (*System on Chip*) para el desarrollo de arquitecturas avanzadas que procesen la información de la red de sensores. Esta tesis se enmarca dentro de la integración de algoritmos de alto nivel en SoC.

El proyecto RESELAI (TIN2006-14896-C02-01) está financiado por el Ministerio de Educación y Ciencia, con una duración de tres años (Septiembre 2006-2009). El objetivo principal de este proyecto es el estudio, desarrollo e integración de redes sensoriales de tres tipos de tecnologías distintas (acústica, visión y radio frecuencia) con el fin de dotar a entornos inteligentes de funciones de detección, localización e interacción con los usuarios. Dentro de este proyecto, la presente tesis se enmarca dentro del objetivo específico "Posicionamiento y guiado de robots móviles y reconocimiento de gestos, mediante redes de cámaras".

Por su parte, la "Cátedra de control electrónico aplicado al transporte" del Departamento de Electrónica, está financiada por las empresas RENFE y Logytel S.L. Su objetivo principal es la investigación y desarrollo de sistemas electrónicos

orientados al aumento de la fiabilidad y seguridad en el transporte ferroviario. Esta tesis se enmarca dentro del campo de aplicación de algoritmos de visión artificial a la detección de obstáculos en zonas del trazado de las vías ferreas dónde exista un mayor riesgo de presencia de obstáculos (pasos elevados y bocas de túneles).

3.3. VISIÓN GENERAL DEL SISTEMA PROPUESTO.

La propuesta que se realiza en esta tesis atiende al diagrama de bloques de la Figura 3.1. El sistema está basado principalmente en un sensor CMOS de alta velocidad (hasta 500 imágenes por segundo a máxima resolución (1280x1024)) y una FPGA encargada de realizar la gestión y captación de imágenes procedentes del sensor, así como de la ejecución de PCA aplicado a la detección de objetos en movimiento.

La versatilidad es otra de las características del sistema diseñado. Así, el usuario mediante una aplicación ejecutada en un PC, puede cambiar ciertas opciones de configuración y funcionamiento. La conexión entre el PC y la FPGA se realiza mediante el puerto paralelo, esto conlleva a que la velocidad de comunicación entre el PC y la FPGA sea lenta (como máximo 2Mbytes/sg), por lo que si se deseara acelerar esta transferencia de datos se debería de implementar otro canal de comunicación en la FPGA, como bus PCI o USB. Para tareas de depuración, que es dónde fundamentalmente se ha usado, ha resultado suficiente el puerto paralelo.

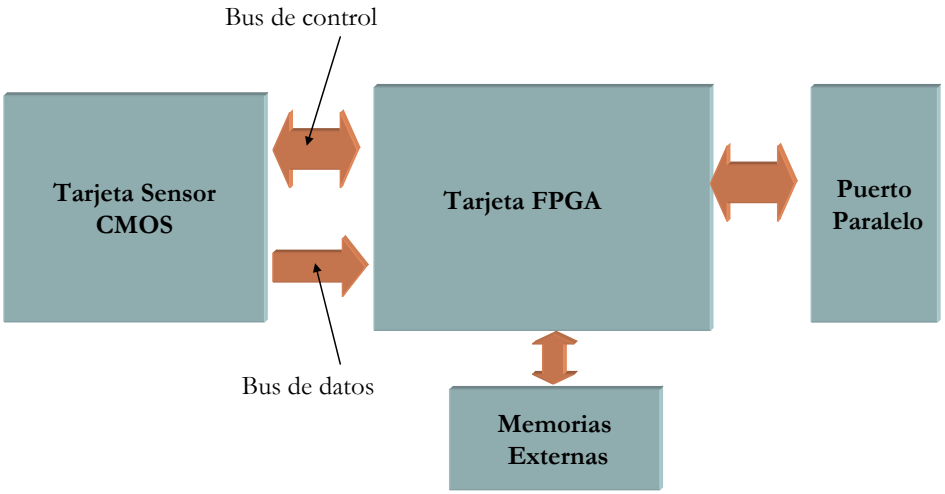


Figura 3.1. Diagrama de bloques del sistema físico implementado.

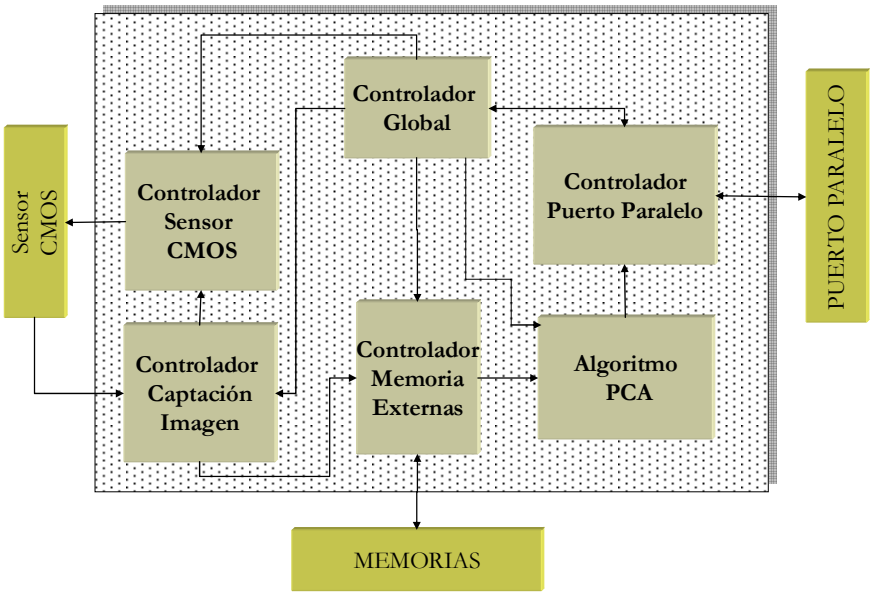


Figura 3.2. Diagrama de bloques del sistema propuesto en esta tesis.

Desde el punto de vista lógico, el sistema propuesto en esta tesis se divide en los siguientes bloques tal y como muestra la Figura 3.2:

- *Controlador del sensor CMOS*: Este bloque se encarga de realizar las peticiones de imágenes al sensor CMOS así como de parametrizar sus registros internos en función de la configuración deseada.
- *Controlador de captación de imagen*: La función principal de este bloque es el enventanado de la imagen procedente del sensor CMOS acorde al tamaño configurado por el usuario.
- *Controlador de memorias externas*: El sistema está dotado de un banco de memorias externas a la FPGA, de tipo SDRAM, con capacidad de 128 MB. En este banco se almacenan las imágenes procedentes del sensor CMOS. La alta velocidad de salida de datos desde el sensor, así como la necesidad de almacenar imágenes para el algoritmo PCA, obliga a utilizar un banco de memorias externas. El elevado tamaño de las imágenes (hasta 1280x1024 píxeles) no permite el uso de los bloques RAM de la FPGA (BRAM) como elementos de almacenamiento. Mediante el controlador de estas memorias se gestiona la comunicación desde la FPGA con el banco de memorias externo.
- *Controlador Puerto Paralelo*: Con este bloque se realiza la comunicación entre la FPGA y el puerto paralelo de un PC. Éste se emplea como vía de transmisión de comandos y resultados.

- *Controlador Global*: Este bloque es el encargado de sincronizar todo el sistema para que éste funcione de manera adecuada y a la máxima velocidad.
- *Algoritmo PCA*: Este bloque implementa el algoritmo PCA y su implementación contiene las aportaciones más significativas de esta tesis.

3.4. ALGORITMO PCA.

Según lo descrito en el Capítulo 2, el desarrollo íntegro del algoritmo PCA en FPGAs era un tema que hasta la fecha no había sido logrado. Existen trabajos que han implementado parte del algoritmo dentro de una FPGA y otra parte, sobre todo la que requiere una mayor carga computacional, realizada por un microprocesador [Fleury, 2004]. Debido a esto, una aportación importante de esta tesis es la adaptación del algoritmo PCA convencional (ideado para sistemas de procesado secuenciales) en una estructura modular concurrente que permita su implementación íntegra en una FPGA. A continuación se hace una breve descripción del algoritmo PCA con el fin de tener una visión general de las etapas que incluye.

3.4.1. Descripción del algoritmo PCA.

El Análisis de Componentes Principales (PCA), transformada de Karhunen-Loéve (TKL) o transformada de Hotelling, es un método ampliamente utilizado en diferentes campos, como la estadística, la electrónica de potencia o la visión artificial. La característica principal de PCA es la reducción de la información redundante (correlada), permaneciendo únicamente aquella que es fundamental (componentes principales). De esta forma se disminuye el volumen de información a manejar. Es decir, ante un banco de datos iniciales con muchas variables correladas entre sí, el objetivo será reducirlas a un número menor perdiendo la menor cantidad de información posible, creando así un conjunto de variables incorreladas (componentes principales). Éstas son una combinación lineal de las variables originales y además independientes entre sí.

La varianza total juega un papel importante en este método. Así, a mayor varianza presente en las características reducidas mayor información retenida de los datos originales.

El área de la visión artificial es un buen ejemplo donde la técnica de PCA se puede aplicar, ya que una imagen contiene un gran número de variables (píxeles) donde muchas de ellas están correladas entre sí. Por tanto, PCA permitirá:

a) Reducir la información redundante de las variables iniciales, generando un conjunto de componentes principales (espacio transformado) que describa la información fundamental de una imagen.

b) Determinar el grado de similitud entre una dos o más imágenes analizando únicamente sus características fundamentales.

Esta última característica es muy interesante desde el punto de vista de detección de objetos en movimiento. Así, si se captan un conjunto de imágenes de una escena sin objetos en movimiento (escena estática) y de ellas se extrae sus vectores característicos o componentes principales, se obtiene un conjunto de vectores que almacena la información fundamental de una escena. Si ahora se desea determinar si en esa misma escena existe un objeto nuevo, simplemente se debe captar una nueva imagen y comprobar si sus características fundamentales son análogas a las que describían las componentes principales de la escena estática. En caso de no ser así será indicativo de que la escena ha cambiado (nuevo objeto en la escena).

En esta tesis se propone utilizar PCA para obtener un modelo de la parte estática de una escena y a partir de este modelo, detectar la presencia de nuevos objetos que entren en la misma. La elección de PCA se debe a que la aplicación para la que se enfoca esta tesis está concebida para funcionar tanto en entornos exteriores como interiores. Esto implica que en el exterior el sistema debe funcionar correctamente en presencia de cambios de iluminación y de condiciones meteorológicas. Desde un punto de vista ideal sería aconsejable que cada nueva imagen procesada se incluyera en el modelo de la escena. Sin embargo, en el caso que la imagen procesada no aporte ninguna información característica adicional, se estaría sobrecargando innecesariamente el tiempo de ejecución de todo el sistema. Por tanto, se debe realizar un proceso de actualización del modelo de fondo de forma selectiva como se verá en capítulos posteriores.

3.4.1.1. Fundamentos matemáticos.

Partiendo de un conjunto de datos iniciales con media nula y una alta correlación entre ellos, $[x_1, x_2, ..., x_n]$, la idea que se persigue es la determinación de un nuevo grupo de variables $[y_1, y_2, ..., y_m]$ incorreladas entre sí. Cada elemento y_j, donde $j = 1,.., m$, es una combinación lineal de los datos originales $[x_1, x_2, ..., x_n]$, tal y como se muestra en (3.1), donde $m \leq n$.

$$\mathbf{y}_1^T = [y_{11}\ y_{12} \ldots y_{1m}] = [u_{11}\ u_{21}\ \ldots\ u_{n1}] \cdot \begin{bmatrix} x_{11}\ x_{12}\ \ldots x_{1j} \\ x_{21}\ x_{22}\ \ldots x_{2j} \\ \ldots\ \ldots\ \ldots\ \ldots \\ x_{n1}\ x_{n2}\ \ldots x_{nj} \end{bmatrix} = \mathbf{u}_1^T \mathbf{x} \qquad (3.1)$$

Es importante tener presente que los vectores $\mathbf{u_j}$ son ortogonales entre sí $\left(\mathbf{u}_j^T \cdot \mathbf{u}_i = 0\right)$ y ortonormales $\left(\mathbf{u}_j^T \cdot \mathbf{u}_j = 1\right)$.

Si se denomina como $\mathbf{y}_1$ a la primera componente principal, ésta corresponde con la combinación lineal de las variables originales que presenta máxima varianza. Así, nuestro objetivo es calcular los valores de $\mathbf{u}_1^T$ que obtienen dicha $\mathbf{y}_1$. Para calcular la varianza de $\mathbf{y}_1$ sujeta a la restricción de ortonormalidad anterior se emplea la expresión (3.2).

$$Var(\mathbf{y}_1) = \frac{1}{n}\mathbf{y}_1^T\mathbf{y}_1 = \frac{1}{n}\mathbf{u}_1^T \cdot \mathbf{x} \cdot \mathbf{x}^T \cdot \mathbf{u}_1 = \mathbf{u}_1^T \mathbf{C}\mathbf{u}_1 \tag{3.2}$$

donde $\mathbf{C}$ es la matriz de covarianza del vector $\mathbf{x}$, siendo su valor el mostrado en (3.3).

$$\mathbf{C} = \frac{1}{n} \cdot \sum_{i=1}^{n} x_i \cdot x_i^T \tag{3.3}$$

Para maximizar $Var(\mathbf{y}_1)$ se aplican las condiciones de ortonormalidad y ortogonolidad de $\mathbf{u}_1$ al multiplicador de Lagrange (λ_1), por lo que resolver este problema equivale a maximizar una nueva función ξ:

$$\xi = \mathbf{u}_1^T \mathbf{C}\mathbf{u}_1 - (\mathbf{u}_1^T \mathbf{u}_1 - 1) \cdot \lambda_1 \tag{3.4}$$

El valor de $\mathbf{u}_1$ que maximiza ξ se obtiene de:

$$\frac{d\xi}{du_1} = 2\mathbf{C}\mathbf{u}_1 - 2\lambda_1\mathbf{u}_1 = 0 \rightarrow \mathbf{C}\mathbf{u}_1 = \lambda_1\mathbf{u}_1 \tag{3.5}$$

Esto quiere decir que $\mathbf{u}_1$ es el autovector asociado al autovalor λ_1 de $\mathbf{C}$.

Luego para maximizar $Var(\mathbf{y}_1)$ hay que tomar para λ_1 el mayor autovalor de $\mathbf{C}$ y para $\mathbf{u}_1$ el autovector asociado a λ_1. Así, el valor de la función ξ máximo será:

$$\xi_{MAX} = \mathbf{u}_1^T \lambda_1 \mathbf{u}_1 - (\mathbf{u}_1^T \mathbf{u}_1 - 1) \cdot \lambda_1 = \lambda_1 \tag{3.6}$$

El cálculo de la segunda componente principal ($\mathbf{y}_2 = \mathbf{u}_2^T \mathbf{x}$) se obtiene de una forma análoga al de la primera teniendo presente que $\mathbf{y}_2$ tiene que estar incorrelado con $\mathbf{y}_1$. Así, para el cálculo de $\mathbf{y}_2$ se tiene que tener como objetivo que la suma de varianzas de $\mathbf{y}_1$ e $\mathbf{y}_2$ sea máxima:

$$\eta = Var(\mathbf{y}_1) + Var(\mathbf{y}_2) = \mathbf{u}_1^T \mathbf{C}\mathbf{u}_1 + \mathbf{u}_2^T \mathbf{C}\mathbf{u}_2 - \lambda_1\left(\mathbf{u}_1^T \mathbf{C}\mathbf{u}_1 - 1\right) - \lambda_2\left(\mathbf{u}_2^T \mathbf{C}\mathbf{u}_2 - 1\right) \tag{3.7}$$

Aplicando la condición de ortonormalidad, derivando e igualando a cero:

$$\frac{d\eta}{d\mathbf{u}_1} = 2\mathbf{C}\mathbf{u}_1 - 2\lambda_1\mathbf{u}_1 = 0 \Rightarrow \mathbf{C}\mathbf{u}_1 = \lambda_1\mathbf{u}_1$$
$$\frac{d\eta}{d\mathbf{u}_2} = 2\mathbf{C}\mathbf{u}_2 - 2\lambda_2\mathbf{u}_2 = 0 \Rightarrow \mathbf{C}\mathbf{u}_2 = \lambda_2\mathbf{u}_2 \qquad (3.8)$$

Por tanto $\mathbf{u}_1$ y $\mathbf{u}_2$ deben ser los autovectores de $\mathbf{C}$.

De esta forma se puede extender esta conclusión a la *j-ésima* componente de modo que para esta componente le correspondería el *j-ésimo* autovalor.

Si se define una matriz $\mathbf{U}$ formada por todos los autovectores se puede expresar la transformación lineal como:

$$\mathbf{y} = \mathbf{U}^T\mathbf{x} \qquad (3.9)$$

donde $\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_m \end{bmatrix}$, $\mathbf{U} = \begin{bmatrix} u_{1,1}\, u_{1,2} \ \dots \ u_{1,n} \\ u_{2,1}\, u_{2,2} \ \dots \ u_{2,n} \\ \dots\dots\dots\dots\dots\dots \\ u_{m,1}\, u_{m,2} \dots u_{m,n} \end{bmatrix}$, $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}$

De lo expuesto anteriormente, la varianza de $\mathbf{y}_1$ es mayor que la de $\mathbf{y}_2$, a su vez la de $\mathbf{y}_2$ es mayor que la de $\mathbf{y}_3$, etc. Por tanto, $\lambda_1 > \lambda_2 > \lambda_3 > \dots > \lambda_n$. La cuestión que se plantea ahora es la siguiente: ¿cuántos autovectores son necesarios para representar con suficiente precisión el conjunto de muestras originales? Existen diversos criterios para determinar este límite [Vázquez, 2006]:

- *Criterio de Kaiser* [Kaiser, 1960]: Propone eliminar aquellos autovalores de $\mathbf{C}$ inferiores a $\bar{\lambda}$ (media de los autovalores de $\mathbf{C}$). Con este criterio se pueden descartar autovalores pequeños pero importantes.
- *Criterio de Jolliffe* [Jolliffe, 1972] [Jolliffe, 1973]: Propone eliminar las componentes asociadas a los autovalores $\lambda_i < 0.7 \cdot \bar{\lambda}$.
- *Criterio de Cattel* [Cattel, 1966]: Sobre un gráfico *scree*, que es aquél que muestra las alturas relativas de los autovalores en orden decreciente, se debe encontrar una abscisa j tal que la línea que une λ_{j-1} con λ_j, tenga una pendiente mucho más pronunciada que la que une λ_j con λ_{j+1}, y retener j componentes. Esto equivale a descartar la parte de la curva que empieza a decrecer más lentamente.
- Seleccionar tantas componentes principales como hagan falta, para representar un porcentaje dado de la varianza total, que es la suma de los autovalores. Se suele emplear, el denominado error cuadrático medio residual normalizado (RMSE) como criterio para seleccionar los t componentes más significativas [Swets 1996] (3.10). Para la

mayoría de las aplicaciones estándar, se suelen considerar las t componentes que permiten obtener un *RMSE* menor al 5%, lo que equivaldría a un porcentaje $P = (1 - RMSE) \cdot 100\%$ [Vázquez, 2006].

$$RMSE = \frac{\sum_{i=t+1}^{n} \lambda_i}{\sum_{i=1}^{n} \lambda_i} \tag{3.10}$$

La elección de un criterio u otro, se realiza en función del comportamiento en cada aplicación.

La transformación lineal desde el espacio original hacia el transformado se realiza mediante (3.9), y la transformación inversa, es decir la recuperación desde el espacio transformado hacia el original, se realiza mediante (3.11). En la Figura 3.3 se muestra esta transformación. Es importante indicar que las variables del espacio original no poseen media nula. Según lo expuesto anteriormente, PCA parte de un conjunto de variables iniciales con media nula, por lo que se deberá calcular la media de todas las variables y restar a cada variable el valor de la media.

$$\hat{\mathbf{x}} = \mathbf{U}\mathbf{y} \tag{3.11}$$

donde $\hat{\mathbf{x}}$ representa el vector recuperado.

La justificación de la expresión (3.11) se debe realizar diferenciando las siguientes situaciones:

a) Número de componentes principales igual al de componentes iniciales ($n = t$).

b) Número de componentes principales distinto al de componentes iniciales ($n \neq t$)

En el primer caso la recuperación de los vectores proyectados en el espacio transformado es perfecta (3.12). Sin embargo en el segundo caso (3.13) aparece un error de recuperación (ε) mostrándose en (3.14) su expresión.

$$\mathbf{y} = \mathbf{U}^T\mathbf{x} \rightarrow \mathbf{U}\mathbf{y} = \mathbf{U}\left(\mathbf{U}^T\mathbf{x}\right) = \mathbf{x} \Rightarrow \begin{cases} \hat{\mathbf{x}} = \mathbf{U}\mathbf{y} \\ \mathbf{x} - \hat{\mathbf{x}} = 0 \end{cases} \tag{3.12}$$

$$\mathbf{y} = \mathbf{U}^T\mathbf{x} \rightarrow \mathbf{U}\mathbf{y} = \mathbf{U}\left(\mathbf{U}^T\mathbf{x}\right) \neq \mathbf{x} \Rightarrow \begin{cases} \hat{\mathbf{x}} = \mathbf{U}\mathbf{y} \\ \mathbf{x} - \hat{\mathbf{x}} \neq 0 \end{cases} \tag{3.13}$$

$$\varepsilon = \left\| \mathbf{x} - \hat{\mathbf{x}} \right\| \tag{3.14}$$

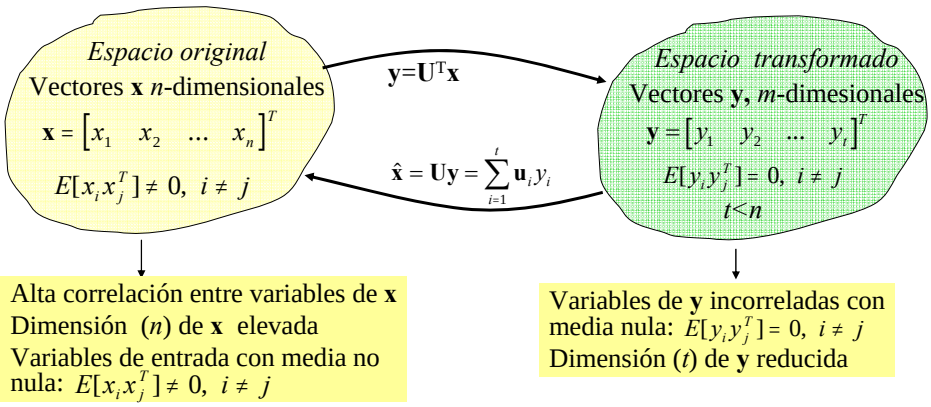


Figura 3.3. Recuperación de la transformación entre espacios en la técnica PCA.

3.4.1.2. Obtención de las componentes principales aplicado a imágenes.

En este apartado se describen las etapas que componen la obtención de la matriz de autovectores del método PCA aplicado a imágenes. En este caso, la matriz de autovectores **U** describe las componentes principales de una escena estática. Para obtener esta matriz, se deben seguir los siguientes pasos:

1. *Captura de imágenes para construir el modelo de la escena estática.* En el caso de visión artificial, las muestras iniciales para determinar las componentes principales son un conjunto de imágenes donde todas ellas describen una misma escena (aunque puede existir entre ellas variaciones de iluminación debido a cambios en las condiciones de meteorología e iluminación). Inicialmente, se parte de un total de M imágenes $\mathbf{I}_i \in \Re^{N \times N}$, donde $i = 1,...M$. Dentro de PCA todas las imágenes son tratadas como vectores de dimensiones $N^2 \times 1$.

2. *Cálculo de la media.* Acorde a la descripción de PCA del apartado anterior, los datos iniciales deben poseer media nula. Por tanto, a continuación se debe calcular el vector media de las M imágenes ($\mathbf{\Psi} \in \Re^{N^2 \times 1}$) (3.15) y restarlo a cada imagen, generando así el vector $\mathbf{\Phi}_i \in \Re^{N^2 \times 1}$ (3.16). Se identifica como $\mathbf{A} \in \Re^{N^2 \times M}$ a la matriz que contiene en sus columnas cada uno de los vectores $\mathbf{\Phi}_i$ (3.17).

$$\mathbf{\Psi} = \frac{1}{M} \cdot \sum_{i=1}^{M} \mathbf{I}_i = \begin{bmatrix} \Psi_1 \\ \Psi_2 \\ \dots \\ \Psi_{N^2} \end{bmatrix}_{N^2 \times 1} = \frac{1}{M} \cdot \begin{bmatrix} I_{1,1} + I_{2,1} + \dots + I_{M,1} \\ I_{1,2} + I_{2,2} + \dots + I_{M2} \\ \dots \\ I_{1,N^2} + I_{2,N^2} + \dots + I_{M,N^2} \end{bmatrix}_{N^2 \times 1} \qquad (3.15)$$

$$\mathbf{\Phi}_i = \mathbf{I}_i - \mathbf{\Psi} = \begin{bmatrix} I_{i,1} - \Psi_1 \\ I_{i,2} - \Psi_2 \\ \cdots\cdots\cdots\cdots \\ I_{i,N^2} - \Psi_{N^2} \end{bmatrix}_{N^2 \times 1} = \begin{bmatrix} \Phi_{i,1} \\ \Phi_{i,2} \\ \cdots\cdots \\ \Phi_{i,N^2} \end{bmatrix}_{N^2 \times 1} \quad ; \; i = 1,...,M \tag{3.16}$$

$$\mathbf{A} = \left[\mathbf{\Phi}_1 .... \mathbf{\Phi}_M\right] = \begin{bmatrix} \Phi_{1_{(1,1)}} & \Phi_{2_{(1,1)}} & .... & \Phi_{M_{(1,1)}} \\ \Phi_{1_{(2,1)}} & \Phi_{2_{(2,1)}} & .... & \Phi_{M_{(2,1)}} \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ \Phi_{1_{(N^2,1)}} & \Phi_{2_{(N^2,1)}} & .... & \Phi_{M_{(N^2,1)}} \end{bmatrix}_{N^2 \times M} \tag{3.17}$$

3. *Generación de la matriz de covarianza*. El siguiente paso es la obtención de la matriz de covarianza $(\mathbf{C} \in \Re^{N^2 \times N^2})$. Aplicando la expresión de covarianza definida en (3.3) se obtiene la expresión (3.18).

$$\mathbf{C} = \frac{1}{M} \cdot \mathbf{A} \cdot \mathbf{A}^T \tag{3.18}$$

4. *Obtención de la matriz autovectores*. Una vez obtenida la matriz de covarianza, el siguiente paso es el cálculo de la matriz de los autovectores de $\mathbf{C}$ $(\mathbf{U} \in \Re^{N^2 \times M})$. La forma más habitual de calcularlos es:

$$\mathbf{C} \cdot \mathbf{U} = \boldsymbol{\lambda} I \cdot \mathbf{U} \tag{3.19}$$

donde $\boldsymbol{\lambda} = \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \cdots \\ \lambda_M \end{bmatrix}^T$ es el vector de autovalores.

Antes de continuar es importante analizar detalladamente la expresión (3.18). Debido a que la matriz $\mathbf{A}$ es de un tamaño $N^2 \times M$, las dimensiones de la matriz $\mathbf{C}$ son $N^2 \times N^2$. Cuando el tamaño de la imagen (N^2) es grande, esto provoca un número muy elevado de operaciones aritméticas a la hora de realizar (3.18). Por esta razón y con idea de minimizar el número de operaciones a ejecutar, se debe buscar algún método alternativo que reduzca la carga computacional. Para ello, gracias a la igualdad de los autovalores de $\left(\mathbf{A}^T \cdot \mathbf{A}\right)$ y los de $\left(\mathbf{A} \cdot \mathbf{A}^T\right)$, primeramente se calculan los autovectores de $\left(\mathbf{A}^T \cdot \mathbf{A}\right)$ (ver (3.20)).

$$\mathbf{A}^T \cdot \mathbf{A} \cdot \mathbf{V} = \boldsymbol{\lambda} I \cdot \mathbf{V} \tag{3.20}$$

donde $\mathbf{V} \in \Re^{M \times M}$ es la matriz de autovectores de $\mathbf{A}^T \cdot \mathbf{A}$.

Si en la expresión (3.20) se multiplica por $\mathbf{A}$ se obtiene (3.21).

$$\left(\mathbf{A} \cdot \mathbf{A}^T\right) \cdot \mathbf{A} \cdot \mathbf{V} = \boldsymbol{\lambda} \mathbf{I} \cdot \mathbf{A} \cdot \mathbf{V} \tag{3.21}$$

Comparando la expresión (3.21) con la (3.19) se puede ver que los autovectores de $\mathbf{C}$ se obtienen como:

$$\mathbf{U} = \mathbf{A} \cdot \mathbf{V} \tag{3.22}$$

Por tanto, para conseguir la matriz de autovectores $\mathbf{U}$ se obtiene a partir de $\mathbf{V}$. En el capítulo 4 se justifica desde el punto de vista de operaciones aritméticas a realizar, que la obtención de $\mathbf{U}$ a partir de $\mathbf{V}$ consume un número inferior de operaciones.

5. *Reducción de componentes principales*. Una vez determinada la matriz de transformación $\mathbf{U}$, otra cuestión importante a resolver es determinar cuántos autovectores son necesarios. La matriz $\mathbf{U}$ con un tamaño $N^2 \times M$ posee M autovectores siendo el tamaño de cada uno N^2. De los M autovectores se deben coger aquellos que estén asociados a los t autovalores de mayor peso. Para determinar t el método empleado es el RMSE expuestos anteriormente. Ordenando los autovalores de mayor a menor ($\lambda_1 > \lambda_2 > ... > \lambda_t$), se eligen únicamente los t autovalores mayores, considerando un RMSE del P (3.23). La elección de este porcentaje es experimental, siendo el valor de P el que genere los mejores resultados prácticos.

$$RMSE = \frac{\sum_{i=t+1}^{M} \lambda_i}{\sum_{i=1}^{M} \lambda_i} < P \tag{3.23}$$

De esta forma la matriz de autovectores reducida obtenida ($\mathbf{U}_t \in \Re^{N^2 \times t}$) es la expuesta en (3.24).

$$\mathbf{U}_t = \left[\mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_t\right] \tag{3.24}$$

donde $\mathbf{u}_j = \begin{bmatrix} u_{1,j} \\ u_{2,j} \\ \cdots\cdots \\ u_{N^2,j} \end{bmatrix}$ corresponde con cada uno de los autovectores.

El proceso de cálculo de la matriz de autovectores se puede ver de forma gráfica en la Figura 3.4.

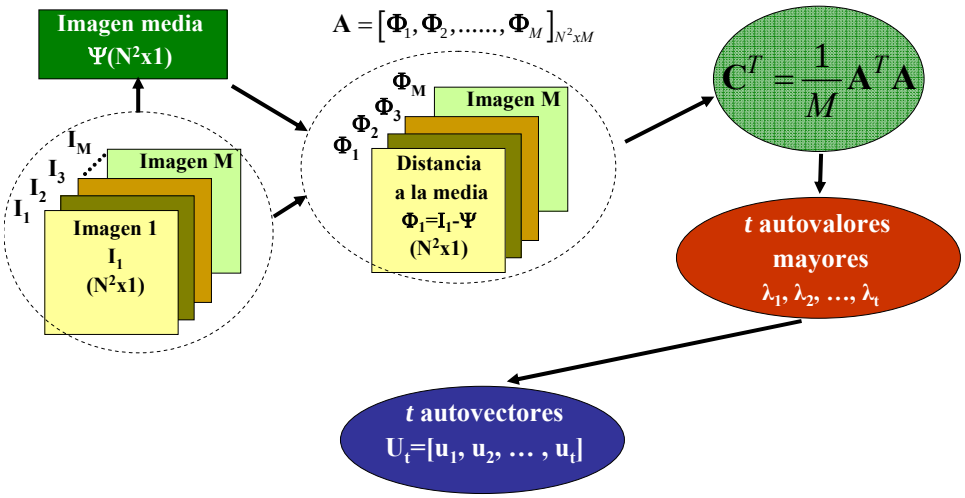


Figura 3.4. Diagrama de flujo del cálculo de la matriz de autovectores (componentes principales) aplicado al modelado de una escena estática.

Otro aspecto importante, es el número de imágenes M necesarias para definir el modelo de fondo. Es conveniente emplear un número grande de imágenes que contemple diferentes condiciones de iluminación de una misma escena. De esta forma el modelo de fondo permite describir variaciones de iluminación de la escena. Sin embargo, el empleo de un número elevado de imágenes conlleva un aumento muy significativo de carga computacional y almacenamiento de memoria. Por lo tanto, existe un compromiso a la hora de elegir el tamaño óptimo. Según los resultados mostrados en [Vázquez, 2006] para valores de M comprendidos entre 9 y 15, el error cometido es prácticamente el mismo. Sin embargo, en nuestro caso, las limitaciones hardware de la plataforma desarrollada tal y como se justifica en el capítulo 4 han condicionado la elección de $M = 8$.

Con respecto a la frecuencia de captura de las M imágenes, lo ideal es que se capten en un período de tiempo que contenga todos los posibles cambios de los objetos de la escena (nubes, árboles, etc.). Si el movimiento de esos objetos no estuviera incluido en el fondo, el algoritmo PCA detectaría un objeto en movimiento. Sin embargo, esta situación implicaría una fase inicial de calibración del sistema en la que se capturarían todas las posibilidades, demorando así el inicio de PCA. La propuesta realizada en esta tesis no contempla una fase inicial de calibrado del

sistema por lo que el modelo de fondo debe actualizarse automáticamente durante la ejecución real del mismo. El criterio empleado para insertar una imagen al fondo estático se desarrolla en el capítulo 5.

3.4.1.3. Distancia euclídea entre la imagen real y su recuperada del espacio transformado.

Una vez obtenida la matriz de transformación $\mathbf{U}_t$ a partir de imágenes de una escena sin objetos en movimiento, el siguiente paso es determinar si en una nueva imagen captada de la escena han aparecido nuevos objetos. Para ello, los pasos a seguir son los siguientes:

1. *Proyección sobre el espacio transformado.* El primer paso es la proyección sobre el espacio transformado (mediante la matriz $\mathbf{U}_t$), de la nueva imagen captada ($\mathbf{I}_j \in \Re^{N^2 \times 1}$) menos la media de las imágenes que formaban el modelo del fondo ($\mathbf{\Psi}$) (3.15). Este paso, se expone analíticamente en la expresión (3.25).

$$\mathbf{\Omega} = \mathbf{U}_t^T \cdot \mathbf{\Phi}_j = \mathbf{U}_t^T \cdot (\mathbf{I}_j - \mathbf{\Psi}) \tag{3.25}$$

   donde $\mathbf{\Omega} \in \Re^{t \times 1}$ queda caracterizado por un vector de dimensión t ($\mathbf{\Omega} = [\omega_1, \omega_2, ..., \omega_t]^T$, donde cada componente ω_i representa la contribución de cada autovector en la representación de $\mathbf{I}_j$.

2. *Recuperación de la imagen proyectada.* Una vez proyectada la imagen sobre el espacio transformado mediante (3.25), a continuación se recupera mediante (3.26).

$$\hat{\mathbf{\Phi}}_j = \mathbf{U}_t \cdot \mathbf{\Omega} \tag{3.26}$$

3. *Determinación de existencia de objeto en movimiento.* Finalmente, se compara la nueva imagen captada (tras haberle sido restada la media) con la imagen recuperada. De esa forma se puede calcular el error de recuperación (o distancia Euclídea) pudiendo determinar así si hay o no un objeto nuevo. Para ello, se compara el error de recuperación con un umbral (Th_{MD}) (3.27).

$$\begin{aligned} \left\| \mathbf{\Phi}_j - \hat{\mathbf{\Phi}}_j \right\| &\leq Th_{MD} \rightarrow \text{no hay objetos nuevos en la escena} \\ \left\| \mathbf{\Phi}_j - \hat{\mathbf{\Phi}}_j \right\| &> Th_{MD} \rightarrow \text{hay objetos nuevos en la escena} \end{aligned} \tag{3.27}$$

   El umbral Th_{MD} es un valor obtenido dinámicamente que se ajusta en función de las condiciones de la escena. En el capítulo 5 se desarrolla el método propuesto para determinar dicho valor.

4. *Localización espacial del objeto detectado*. Para detectar la presencia de nuevos objetos basta con aplicar la expresión (3.27). Sin embargo, si se desea conocer en qué parte de la escena aparecen los nuevos objetos, se debe buscar algún método de localización. Para ello, se propone la creación de un mapa de errores de recuperación, denominado como mapa de distancias, realizando la diferencia píxel a píxel entre la imagen proyectada ($\mathbf{\Phi}_j$) y la recuperada ($\hat{\mathbf{\Phi}}_j$). Con idea de minimizar el efecto del ruido, se promedia el valor de cada píxel de la imagen captada y el de la imagen recuperada, con el de sus píxeles adyacentes mediante una máscara de $q \times q$ elementos [Vázquez, 2006]. Como resultado se obtiene una matriz denominada mapa de distancias $\mathbf{MD}_{V1} \in \Re^{N^2 \times N^2}$, donde cada uno de sus elementos ($\varepsilon_{w,i}$) se corresponde con la distancia euclídea entre los correspondientes píxeles promediados de la imagen original y la recuperada (3.28). En esta tesis se emplean máscaras de 3x3 elementos tal y como se justificará en el capítulo 5.

$$\varepsilon_{w.i} = \left\| \mathbf{\Phi}_{w,i} - \hat{\mathbf{\Phi}}_{w,i} \right\| \quad w = 1,2,....N \quad i = 1,2,...., N \tag{3.28}$$

Una vez obtenido el mapa de distancias $\mathbf{MD}_{V1}$, el siguiente paso es realizar el umbralizado de dicho mapa, para así encontrar fácilmente los objetos nuevos. Para ello, se construye una nueva imagen binaria $(\mathbf{BW})$ donde cada elemento es el resultado de la comparación de cada píxel de $\mathbf{MD}_{V1}$, con un umbral Th_{MD}. Cuando la comparación de un elemento de $\mathbf{MD}_{V1}$ supere el umbral se satura a 255 y en caso contrario se satura a 0 (3.29).

$$\begin{aligned} BW_i &= 255 \text{ si } (\varepsilon_{wi}) = \left\| \mathbf{\Phi}_{wi} - \hat{\mathbf{\Phi}}_{wi} \right\| > Th_{MD} \\ BW_i &= 0 \quad \text{ si } (\varepsilon_{wi}) = \left\| \mathbf{\Phi}_{wi} - \hat{\mathbf{\Phi}}_{wi} \right\| \le Th_{MD} \end{aligned} \tag{3.29}$$

Todo el proceso de detección de objetos a partir del modelo de fondo de una escena estática, se puede ver de forma gráfica en la Figura 3.5. Por su parte la Figura 3.6 muestra el resultado final obtenido tras aplicar PCA sobre una imagen captada (Figura 3.6.a) de una escena (Figura 3.6.d).

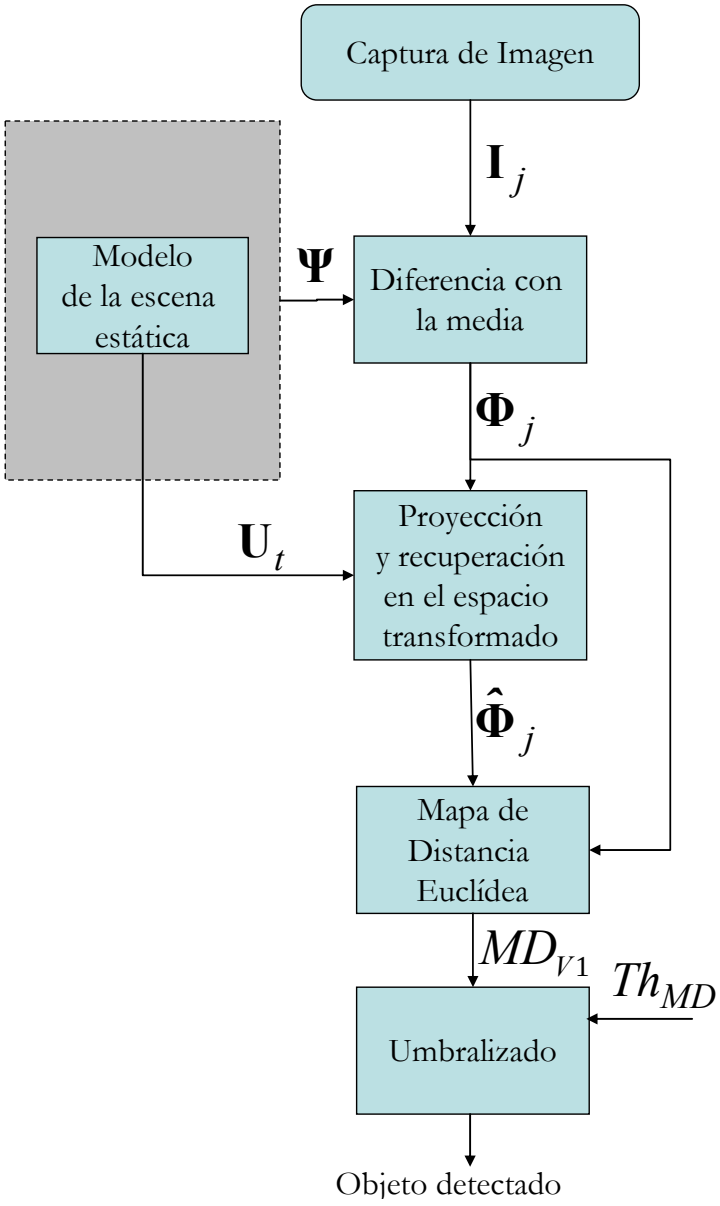
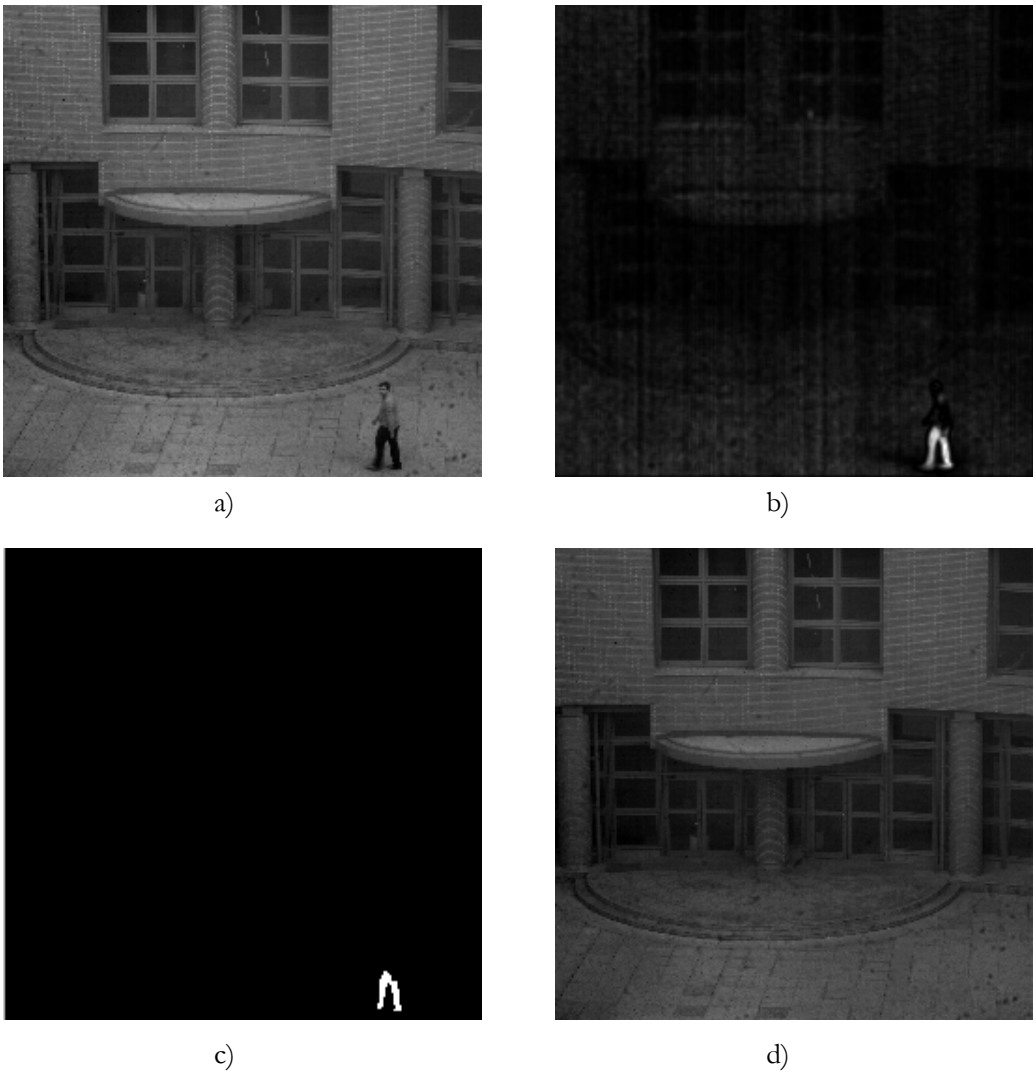


Figura 3.5. Diagrama de flujo para detectar la presencia de un objeto nuevo en una escena estática mediante PCA.



a) b)

c) d)

Figura 3.6. (a) Imagen original con objeto nuevo, (b) Mapa de distancias, (c) Imagen umbralizada, d) Fondo estático.

3.4.2. Propuesta de implementación de PCA sobre FPGAs.

Es evidente que debido al complejo cálculo matemático mostrado en el apartado anterior la inclusión del algoritmo PCA dentro de una FPGA exige una adaptación del mismo. Atendiendo a las características internas de las FPGAs se ha buscado para cada uno de los ítems que forman el algoritmo PCA alternativas que permitan una ejecución en paralelo.

En la Figura 3.7 se muestra la propuesta de implantación del algoritmo PCA aplicado a imágenes realizada en esta tesis:

- *Captación de M imágenes*. El primer paso es la adquisición de *M* imágenes desde el sensor CMOS para poder crear el modelo de una escena estática (capítulo 6). En esta tesis todas las imágenes manejadas están codificadas con 256 niveles de gris. De los 10 bits

que proporciona el sensor por cada píxel, se convierte a una resolución de 8 bits a modo y manera de cualquier cámara estándar.

- *Media aritmética* ($\boldsymbol{\Psi}$). Una vez captadas las imágenes para construir el fondo se obtiene la media aritmética de cada uno de los píxeles de las M imágenes (capítulo 4).

- *Generación Matriz* $\mathbf{A}$. En este bloque se realiza la generación de la matriz $\mathbf{A}$ a partir de las M imágenes captadas y del vector media (3.17) (capítulo 4).

- *Generación de matriz covarianza*. Para poder crear la matriz de covarianza se debe realizar la multiplicación de la matriz $\mathbf{A}^T \cdot \mathbf{A}$. También en diferentes partes del algoritmo PCA, como proyección y recuperación de una imagen, es necesario realizar la multiplicación de dos matrices. Por ello, el abordar esta operación aritmética en un dispositivo hardware como una FPGA resulta de gran interés realizar un estudio profundo de las diferentes alternativas. De las posibles soluciones, tanto de las aportadas por el autor u otras, se ha optado por la construcción de un *array* semi-sistólico que se emplea en todas las multiplicaciones de matrices (capítulo 4).

- *Cálculo de autovectores*. Mediante la variante del algoritmo de Jacobi expuesta en [Brent, 1985] se ha desarrollado un nuevo sistema de cálculo de autovectores (capítulo 4).

- *Imagen Diferencia*. Este bloque se encarga de realizar la resta de cada imagen con el vector media ($\boldsymbol{\Psi}$) (capítulo 4).

- *Recuperación y proyección*. En este bloque también se deben realizar multiplicaciones de matrices para poder obtener la imagen recuperada ($\hat{\boldsymbol{\Phi}}_j$) (ver (3.26)). En este caso, se usa el *array* semi-sistólico para multiplicar matrices (capítulo 4).

- *Cálculo de distancia*. En este bloque se ha implementado un sistema concurrente capaz de realizar la distancia euclídea entre la imagen recuperada ($\hat{\boldsymbol{\Phi}}_j$) y la original ($\boldsymbol{\Phi}_j$) (3.28). Empleando alternativas basadas en la convolución de máscaras sobre imágenes en FPGAs [Bravo, 2003], se propone un sistema basado en un *pipeline* que permite alcanzar pleno rendimiento. Una vez construido el mapa de distancias, se realiza una umbralización mediante un umbral de comparación (capítulo 5).

- *Actualización*. Para poder actualizar el espacio transformado con una nueva imagen, es necesario validarla como candidata a entrar en el

modelo de fondo. Es decir, analizar si una nueva imagen puede aportar información nueva al modelo (capítulo 5).

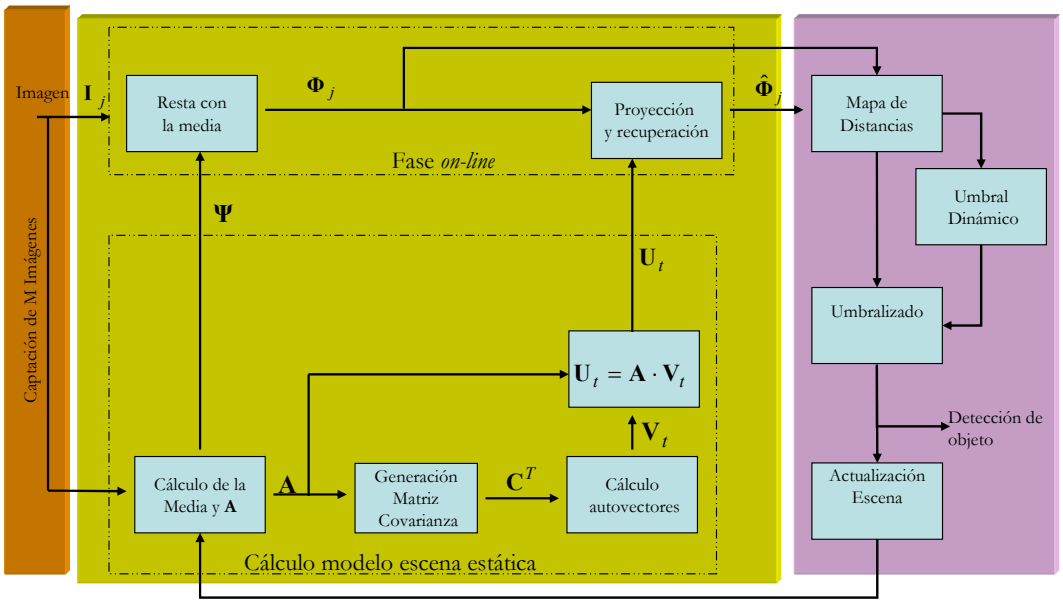


Figura 3.7. Diagrama de bloques del algoritmo PCA implementado en esta tesis en FPGA.

4. IMPLEMENTACIÓN DEL ALGORITMO PCA SOBRE FPGAs

4.1. INTRODUCCIÓN.

En este capítulo se expone la solución hardware realizada para la implementación del algoritmo PCA sobre una FPGA. La propuesta desarrollada está aplicada a visión artificial sin embargo su portabilidad hacia otras áreas es factible debido a la modularidad de la propuesta realizada.

Debido a las características de las FPGAs, en todo momento se ha buscado explotar al máximo el potencial que éstas presentan. Esto se ha concretado en la búsqueda de la mayor concurrencia posible. Para ello se ha intentado dotar al algoritmo de la mayor velocidad de ejecución, implementando numerosos subsistemas en *pipeline*. Sin embargo, esto no siempre ha sido posible ya que la dependencia de datos en diferentes fases de PCA no hace viable esta posibilidad.

Otro objetivo importante en la propuesta realizada es el consumo mínimo de recursos internos de la FPGA. Gracias a esto, todo el diseño ha sido inferido en una FPGA Virtex II-Pro XC2VP7. Esta FPGA pertenece a la gama baja-media de las diferentes familias desde el punto de vista de capacidad (entorno a 5000 *slices* y 45 multiplicadores hardware).

En cuanto a su estructura interna, en la Figura 4.1 se presenta un diagrama de bloques del algoritmo PCA implementado en la FPGA, destacando tres grandes bloques:

- *Generación de los autovectores*. Este bloque se encarga de obtener una matriz de autovectores ($\mathbf{U}_t \in \Re^{N^2 \times t}$) en base a un conjunto de M imágenes. Esta matriz servirá para formar el modelo de fondo de la escena a analizar.

- *Fase on-line*. En este bloque se proyectará y recuperará una imagen ($\mathbf{I}_i$) al espacio transformado (matriz de autovectores) con objeto de ver el grado de parecido entre la nueva imagen y el fondo en el bloque *Detección de Objetos*.

- *Detección de objetos*. A partir de la imagen actual menos la media ($\mathbf{\Phi}_j \in \Re^{N^2 \times 1}$) manejada por el bloque de la *Fase on-line*, y de la imagen proyectada y recuperada del espacio transformado ($\hat{\mathbf{\Phi}}_j \in \Re^{N^2 \times 1}$), se determinará la existencia o no de nuevos objetos en la escena bajo estudio. Este bloque se desarrollará en el capítulo 5 de esta tesis.

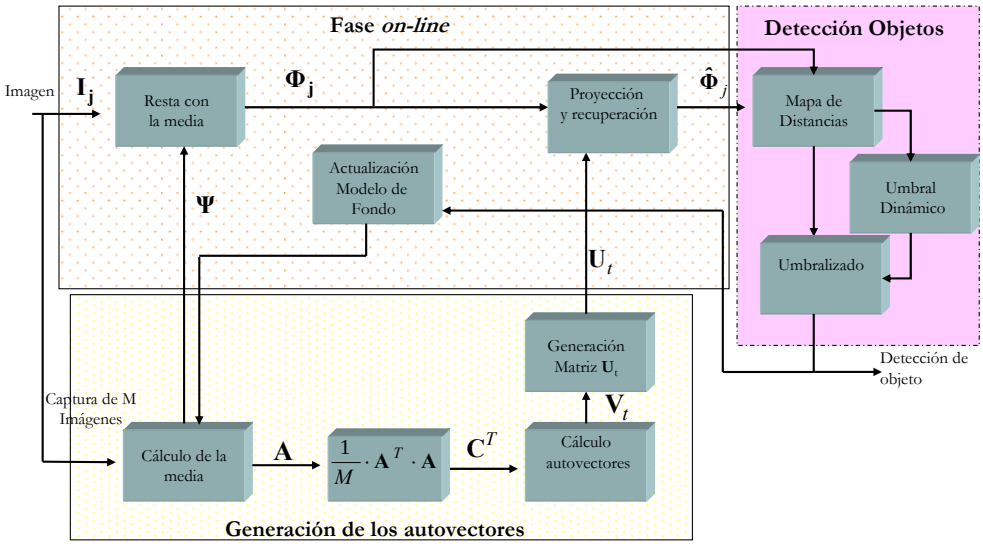


Figura 4.1. Diagrama de bloques lógicos del algoritmo PCA implementado en FPGA.

Para lograr el diagrama de bloques de la Figura 4.1 en una FPGA se han diseñado un total de 8 módulos independientes tal y como se muestra en la Figura 4.2. El módulo de Multiplicación de Matrices, es único y en él se implementan las 4 multiplicaciones de matrices que se deben desarrollar en las diferentes fases de PCA (generación de la matriz de covarianza, generación de la matriz $\mathbf{U}_t$, proyección de una imagen sobre el espacio transformado y recuperación de una imagen del espacio transformado).

Con respecto a la implementación del diseño en una FPGA, se ha de remarcar la codificación íntegra del diseño en VHDL, sin empleo de ningún *core* e

intentando en todo momento generar una estructura flexible y versátil, permitiendo así su expansión de forma cómoda y sencilla.

El presente capítulo describe las diferentes etapas que constituyen el algoritmo PCA. El proceso se divide en dos partes: primeramente, la generación del espacio transformado (matriz de autovectores) y posteriormente proyección y recuperación de una imagen sobre el espacio transformado (fase *online*). Para cada parte se presenta el diseño propuesto y realizado, así como los resultados obtenidos desde un punto de vista de exactitud, velocidad de ejecución y recursos internos de la FPGA consumidos. Todas las imágenes presentadas en este capítulo han sido captadas mediante una "cámara inteligente" desarrollada en esta tesis y que se presentará en el capítulo 6, siendo la captura y control gestionados desde la misma FPGA que implementa PCA.

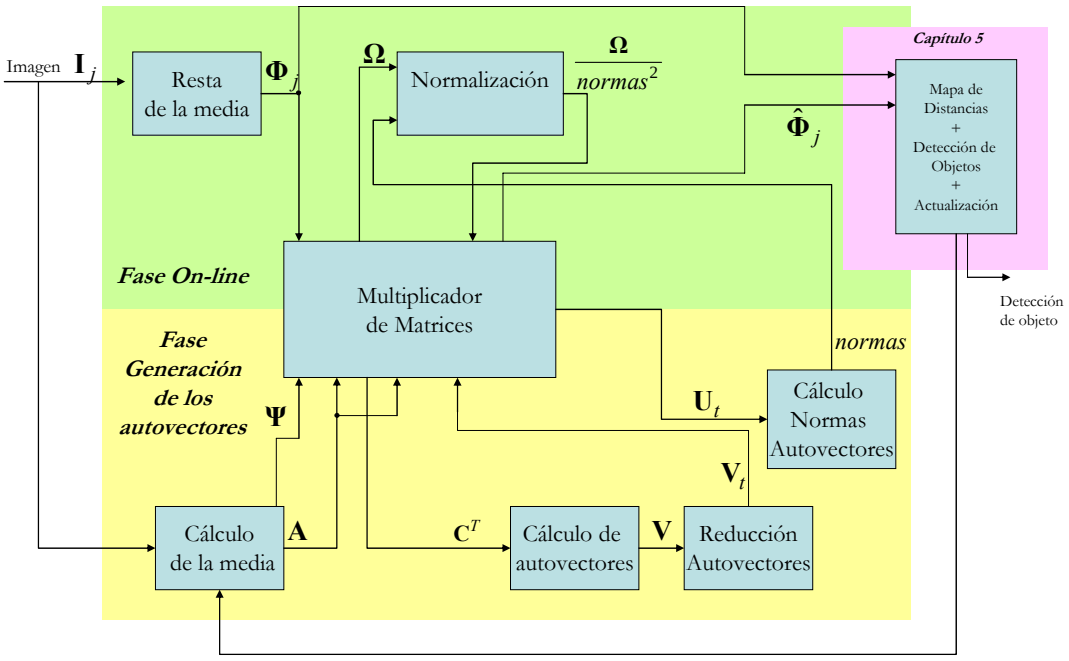


Figura 4.2. Diagrama de bloques práctico del algoritmo PCA implementado en FPGA.

4.2. CÁLCULO DE LA MATRIZ DE AUTOVECTORES ($\mathbf{U}_t$) EN FPGAs.

La primera fase del algoritmo PCA que se va a describir es el proceso de generación de la matriz de autovectores o matriz de transformación $\mathbf{U}_t$. Esta matriz se obtiene a partir de un conjunto de M imágenes captadas de una escena estática tal y como se ha indicado en el capítulo anterior. A partir de $\mathbf{U}_t$, se obtiene la expresión (3.26) que permitirá determinar en (3.27) si en una nueva imagen $\left(\mathbf{I}_j \in \Re^{N^2 \times 1}\right)$ existe un nuevo objeto (objeto en movimiento). La generación de la matriz $\mathbf{U}_t$, se divide en 5 etapas:

1. Cálculo de la media de las M imágenes $\left(\mathbf{\Psi} \in \Re^{N^2 \times 1}\right)$ (3.16) y matriz $\mathbf{A} \in \Re^{N^2 \times M}$ (3.17).
2. Obtención de la matriz de covarianza $\mathbf{C}^T \in \Re^{M \times M}$.
3. Cálculo de la matriz de autovectores $\mathbf{V} \in \Re^{M \times M}$ (3.20) y la posterior matriz de autovectores reducida $\mathbf{V}_t \in \Re^{M \times t}$.
4. Obtención de la matriz de autovectores $\mathbf{U}_t \in \Re^{N^2 \times t}$ a partir de $\mathbf{V}_t \in \Re^{M \times M}$ donde $t < M$.
5. Cálculo de las normas de la matriz de autovectores.

Para acelerar la ejecución de esta fase se ha intentado segmentar al máximo la ejecución de las etapas anteriores. Sin embargo, tanto la etapa 3 como la 4 necesitan que haya finalizado completamente su etapa previa, para poder iniciarse. Por tanto, existe una dependencia de datos que no permite paralelizar al 100% la ejecución de esta fase. En la Figura 4.3 se presenta la segmentación propuesta. A su vez, dentro de cada una de las cuatro etapas existe una segmentación interna que permite reducir el tiempo de ejecución. A lo largo de esta sección se irán describiendo y analizando las propuestas realizadas en cada una de ellas.

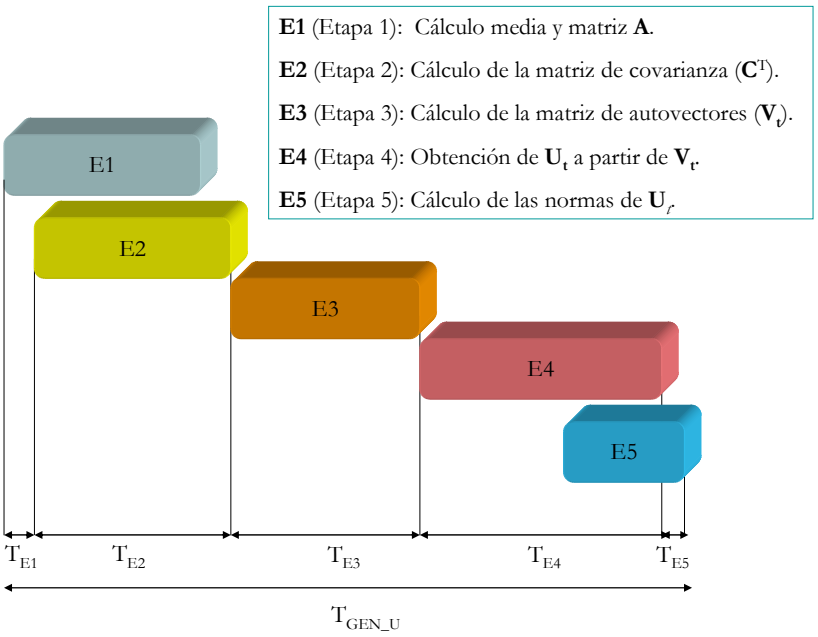


Figura 4.3. Propuesta de segmentación de las tareas para el cálculo de la matriz de autovectores (transformación).

4.2.1. Cálculo de la media aritmética de las imágenes que forman el fondo estático y de la matriz $\mathbf{A}$.

Dentro del algoritmo PCA, tal y como se describió en el apartado 3.4.1.1, los datos de partida (en nuestro caso imágenes) deben ser variables con media nula. Sin embargo las imágenes captadas no poseen media nula por lo que se debe calcular la media de todas ellas y restársela a cada una de las imágenes de entrada. Por tanto, el primer paso para determinar $\mathbf{U}_t$ es el cálculo de un vector media $(\mathbf{\Psi})$ (Figura 4.4) de las mismas dimensiones que los vectores de entrada.

Una vez calculado $\mathbf{\Psi}$, el siguiente paso es la obtención de $\mathbf{\Phi}_i$ donde $i \in [1, M]$(ver (3.16)). El conjunto de los $\mathbf{\Phi}_i$, forman la matriz $\mathbf{A} \in \Re^{N^2 \times M}$ (ver (3.17)) que se necesita para generar la matriz de covarianza.

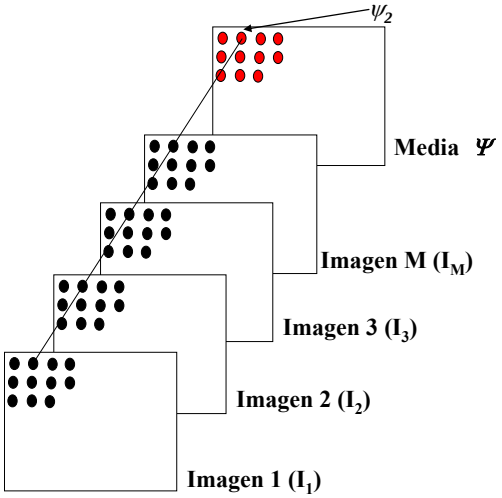


Figura 4.4. Obtención de la imagen media a partir de M imágenes captadas.

La arquitectura hardware desarrollada en esta tesis acumula las M imágenes captadas en memoria externa SDRAM. El ancho del bus de datos es de 128 bits, dividiéndose éste en dos buses de 64 bits uno para la lectura y el otro para la escritura de datos. Por tanto, si el nivel de gris de cada píxel se codifica con 8 bits se puede acceder simultáneamente a 8 píxeles. Con objeto de acelerar la ejecución, se ha fijado $M = 8$, permitiendo leer simultáneamente un byte (píxel) de cada una de las M imágenes. En base a esto, se ha construido el sistema mostrado en la Figura 4.5, donde se puede observar la existencia de los siguientes bloques:

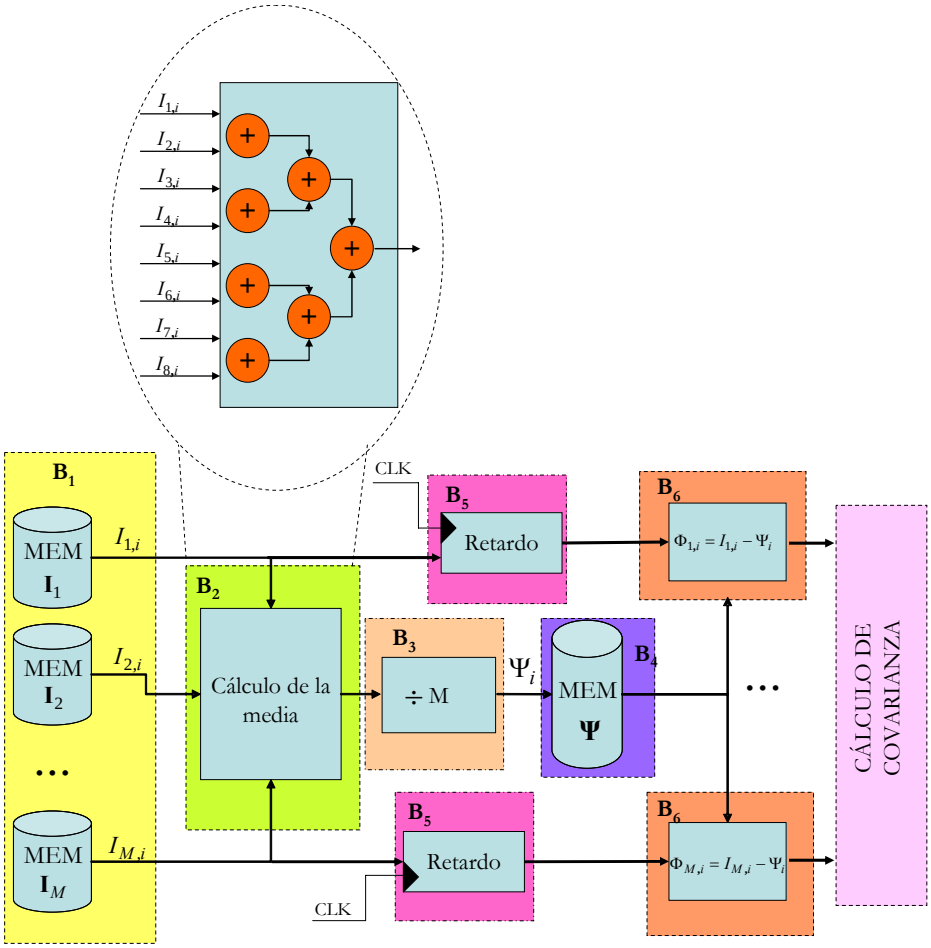


Figura 4.5. Diagrama de bloques del circuito propuesto para el cálculo de la media aritmética $(\boldsymbol{\Psi})$ de las M imágenes captadas.

- *Memorias de las imágenes (B_1).* Las M imágenes necesarias son almacenadas en memoria externa de tal forma que se permita la lectura de hasta 8 píxeles dentro de un mismo ciclo de reloj. En nuestro caso, como $M = 8$ se accede simultáneamente a un píxel de cada imagen.

- *Circuito de cálculo de media (B_2).* Al fijar $M = 8$ hacen falta tres niveles de sumadores de dos entradas (ver Figura 4.5) para poder calcular la suma de los 8 píxeles correspondientes.

- *Registro de desplazamiento (B_3).* Para poder finalizar el cálculo de la media, hace falta dividir el resultado de cada elemento entre el número de imágenes empleado. Como M es una potencia de dos, para realizar la división simplemente hay que hacer desplazamientos.

- *Memoria de la media (B_4).* El resultado del punto anterior es almacenado en memoria interna de la FPGA para posteriormente ser usado en la

fase *on-line*. El tamaño de los datos almacenados en la memoria es de 9 bits, ya que debido a la operación resta que se realiza se incrementa en 1 bit el tamaño del dato original.

- *Circuito retardador (B_5)*. Con idea de reducir accesos a la memoria externa, los datos leídos de cada imagen se almacenan en unos registros. El tiempo que están almacenados es de 3 ciclos de reloj ($\log_2(M)$), siendo este tiempo el que se emplea en el cálculo de la media.

- *Circuito restador (B_6)*. Se encarga de hacer la diferencia entre los valores de los píxeles imagen y la media asociada. El resultado es la matriz **A** la cual se envía al bloque de cálculo de la matriz de covarianza.

El funcionamiento del sistema expuesto en la Figura 4.5 es el siguiente:

- Inicialmente, se accede a un píxel de cada una de las 8 imágenes empleando la misma dirección para todas las imágenes (B_1 de la Figura 4.5).

- A continuación, estos píxeles son sumados entre sí para poder calcular la media (B_2 de la Figura 4.5). Posteriormente, se hace un triple desplazamiento a derecha para realizar la división final de la suma entre 8. Todo este proceso necesita tres ciclos de reloj ya que los sumadores empleados son síncronos con objeto de acelerar la frecuencia máxima de reloj.

- En paralelo con la fase anterior, los píxeles leídos son introducidos en bloques retardadores (B_5 de la Figura 4.5) compuestos internamente por 3 registros. De esta forma se establece un retardo de 3 ciclos de reloj que es el tiempo que tarda en realizarse la suma y desplazamiento.

- El último paso de este bloque es la resta de cada píxel con su correspondiente de la imagen media calculada anteriormente (B_6 de la Figura 4.5). El resultado es enviado a la etapa de cálculo de la matriz de covarianza. Debido a que en cada ciclo de reloj se generan 8 datos, esto permite que la etapa de cálculo de la covarianza se inicie en el momento que el circuito de la Figura 4.5 genere un dato.

Con respecto al tiempo de ejecución, en la Figura 4.6 se presenta la segmentación realizada en esta primera etapa. Se observa cómo la ejecución de ésta (color marrón) se ha dividido en 3 subetapas o *pipes*, los cuales se solapan con la siguiente etapa (generación de la matriz de covarianza). El tiempo asociado a cada una de las 3 subetapas es el mismo, siendo su valor el dado en (4.1) particularizado para los valores elegidos en esta tesis ($N = 256$ y $M = 8$).

$$T_{RD\_MEM} = T_{\Psi} = T_{GEN\_A} = \frac{N^2 \times M \times 8bits}{BW_{MEM\_EXT}} T_{CLK} = \\ = \frac{65536 \times 8 \times 8bits}{64bits} T_{CLK} = 65536 T_{CLK} \quad (4.1)$$

Por otra parte, las latencias mostradas en la Figura 4.6 (L_1, L_2, L_3) vienen derivadas de la arquitectura diseñada en la Figura 4.5. Así, L_1 es de un ciclo de reloj siendo ésta el tiempo empleado desde que se solicita un dato a memoria externa hasta que éste está disponible a la entrada de cada sumador de la Figura 4.5. Por su parte, L_2 corresponde al tiempo de espera empleado por los circuitos retardadores de la Figura 4.5, siendo por tanto su valor $\log_2(M)$. Por último, L_3 es el tiempo que se tarda en tener disponibles los primeros elementos de la matriz $\mathbf{A}$ para que la etapa de generación de la matriz de covarianza comience su cálculo. Esta latencia tiene también un valor de un ciclo de reloj.

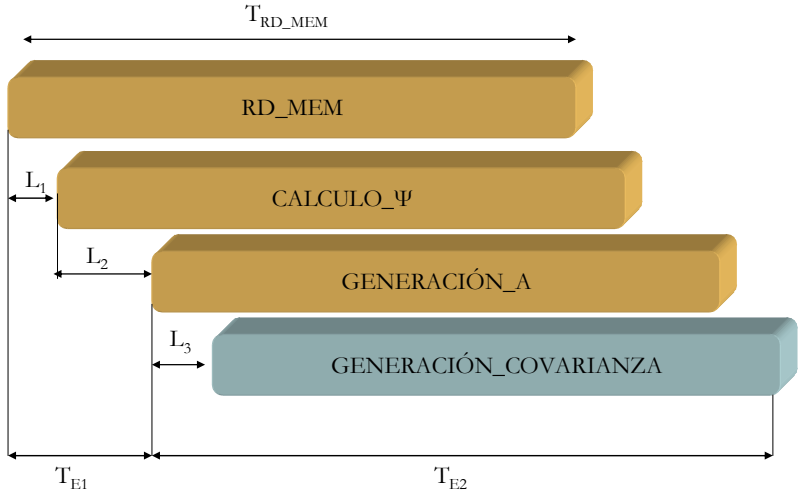


Figura 4.6. Segmentación del cálculo de la media $(\boldsymbol{\Psi})$ y generación de la matriz $\mathbf{A}$.

Para poder determinar el tiempo consumido por esta primera etapa debido a la paralelización del inicio de la etapa 2 (generación de matriz de covarianza) con la etapa 1, el tiempo a tener en cuenta para el cómputo total de T_{GEN_U} (ver Figura 4.3) es el denominado como T_{E1} mostrado tanto en la Figura 4.6 así como en la Figura 4.3. Acorde con lo expuesto anteriormente su valor se muestra en (4.2).

$$T_{E1} = L_1 + L_2 = T_{CLK} + (\log_2(M)) \cdot T_{CLK} = T_{CLK} + 3T_{CLK} = 4T_{CLK} \quad (4.2)$$

En cuanto al número de recursos consumidos por esta etapa, en la Tabla 4.1 se muestran los resultados obtenidos para la FPGA empleada en esta tesis (XCV2P-7). Se observa cómo debido a la simplicidad del diseño realizado, el número de *slices* ocupados es relativamente bajo (2%). Sin embargo, como $\boldsymbol{\Psi}$ se almacena en BRAM el número de éstos se eleva hasta el 75% de los bloques disponibles.

Tabla 4.1. Recursos consumidos por la etapa de cálculo de media y matriz $\mathbf{A}$.

Área (*Slices*)	BRAM	Multiplicadores	Máxima frecuencia
117 (2,5%)	33 (75%)	0	165 MHz

4.2.2. Cálculo de la matriz de covarianza en FPGAs.

Dentro del cálculo de la matriz de autovectores $\mathbf{U}_t$, el primer paso es la obtención de la matriz de covarianza $\mathbf{C}^T$ tal y como se justificó en el apartado 3.4.1.2. Para generar $\mathbf{C}^T$ se realiza el producto $\frac{1}{M}\mathbf{A}^T \cdot \mathbf{A}$. Por tanto, el siguiente aspecto a resolver es la multiplicación de dos matrices dentro de una FPGA.

La implementación de la Multiplicación de Matrices (Mult. Mat.) dentro de una FPGA es una labor compleja. Además, dependiendo de la naturaleza de las matrices a manejar, la solución implementada para un tipo de matrices puede que no sea válida para otros. En el caso de PCA, son cuatro las operaciones de Mult. Mat. que se deben realizar para: generar la matriz de covarianza, generar la matriz de autovectores ($\mathbf{U}_t$), proyección de una imagen sobre la matriz de autovectores y recuperación de la imagen proyecta. En estas cuatro multiplicaciones aparecen matrices no cuadradas y de muy diferentes tamaños. Por tanto, con idea de minimizar el número de recursos internos consumidos, la solución desarrollada debe servir para los cuatro tipos de multiplicaciones a realizar.

Lo anteriormente expuesto, obliga a realizar un estudio exhaustivo de las diferentes alternativas de Mult. Mat. para FPGAs. Por tanto, inicialmente en esta sección se comienza con un análisis de distintas variantes para la Mult. Mat. En FPGAs. Una vez justificada la variante empleada en esta tesis para la Mult. Mat., se describirá el proceso de generación de la matriz de covarianza y su implementación hardware.

4.2.2.1. Paralelización de los algoritmos de multiplicación de matrices.

Cuando se desea realizar el producto de dos matrices en el menor tiempo posible haciendo uso de una FPGA, la alternativa óptima de implementación es mediante arquitecturas sistólicas. Sin embargo el máximo número de multiplicadores hardware disponibles en las nuevas familias de FPGAs (hasta 400) [Altera, 2006] [Xilinx, 2001] obliga a manejar tamaños de matrices muy pequeños. La estructura interna de una arquitectura sistólica clásica se muestra en la Figura 4.7. En ésta se dispone de un interfaz para gestionar los datos de una matriz de entrada $\mathbf{X} \in \Re^{n \times i}$ (interfaz filas) y de una matriz de entrada $\mathbf{B} \in \Re^{i \times n}$ (interfaz columnas), multiplicadores, sumadores y una memoria para almacenar el resultado final.

En el caso de Mult. Mat. en visión artificial, el tamaño habitual de imágenes (256x256, 512x512 ó 1024x1024 píxeles) hace inviable el empleo de arquitecturas

sistólicas en dispositivos reconfigurables al no disponer del número necesario de multiplicadores. Esto hace que nuestro objetivo sea la búsqueda de un algoritmo de Mult. Mat. donde el número de recursos internos requeridos en la FPGA sea suficiente y donde la ejecución de dicho algoritmo explote al máximo las posibilidades de concurrencia de las FPGAs.

La Mult. Mat. tiene características muy específicas en lo que se refiere al diseño e implementación de un algoritmo paralelo:

- La generación de cada elemento $g_{i,j}$ (ver Figura 4.7) de la matriz resultado ($\mathbf{G} \in \Re^{n \times n}$) es independiente del resto de elementos de $\mathbf{G}$. Esto es sumamente útil ya que permite un amplio grado de flexibilidad en lo referente a la paralelización.
- Regularidad de la organización de los datos y de las operaciones que se realizan sobre éstos: los datos están organizados en estructuras bidimensionales (las matrices mismas) y las operaciones son básicas (multiplicación y suma).

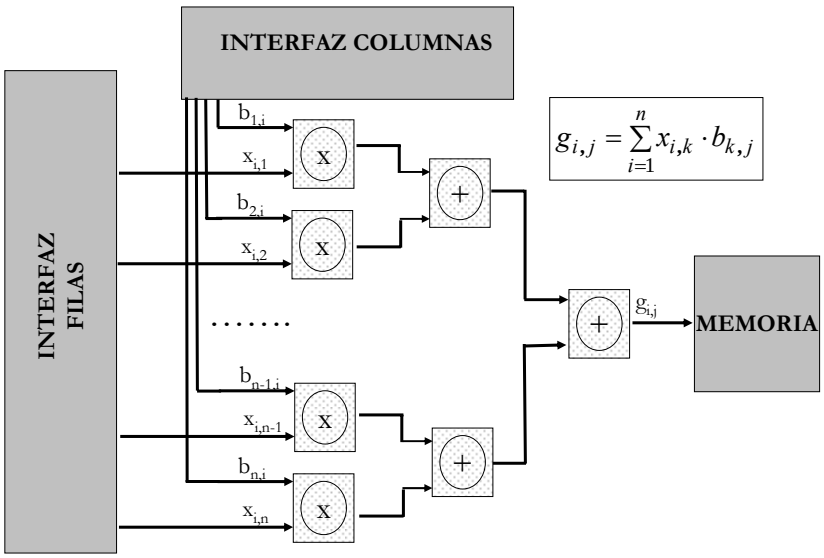


Figura 4.7. *Array* sistólico clásico para la Mult. Mat. cuadradas.

La primera característica hace que la Mult. Mat. sea especialmente apropiada para sistemas multiprocesadores, donde un conjunto de elementos de procesamiento comparten una misma memoria. Los algoritmos paralelos para multiprocesadores suelen seguir las ideas básicas de descomposición o división de los datos a calcular.

La segunda característica hace que los algoritmos propuestos para la Mult. Mat. en paralelo sigan el modelo de cómputo SPMD, en el que un mismo programa se ejecuta en cada procesador de manera independiente y eventualmente se sincroniza y/o comunica con los demás procesadores.

En general los algoritmos se pueden adaptar de una manera más o menos compleja a cada una de las arquitecturas de procesamiento paralelo disponibles. Por

tanto, en función de la plataforma hardware donde se implemente el sistema de Mult. Mat. es necesario implementar un algoritmo diferente.

Existen numerosas alternativas para explotar las características anteriores mostrándose en la Tabla 4.2 un resumen de ellas. Atendiendo a las posibilidades de paralelización de cada algoritmo de la Tabla 4.2, en esta tesis se han desarrollado los sistemas correspondientes a los algoritmos de Winograd, Strassen y particionado de matrices para FPGAs [Bravo, 2006c].

Las tres alternativas han sido implementadas en VHDL sobre una FPGA 2V500fg256-5 y se han simulado temporalmente. En todos los casos las matrices que se han probado son matrices cuadradas ya que ello permite evaluar todos los algoritmos. Para validar éstos se han realizado pruebas con distintos tamaños de matrices (8x8, 32x32, 64x64, 128x128, 256x256 y 512x512), frecuencia de reloj de la FPGA de 50MHz y con un tamaño constante de datos de 16 bits. Los tiempos consumidos por los tres algoritmos evaluados para los diferentes tamaños de matrices, se pueden observar en la Figura 4.8. Si estos resultados se comparan con los generados por el denominado algoritmo clásico (4.3) (ver Figura 4.10), se comprueba que en todos los casos cualquiera de los tres algoritmos, es más rápido que el clásico. Esto se acentúa aún más, cuando el tamaño de las matrices aumenta. Si se analizan los resultados de los tres algoritmos entre sí, se verifica que la alternativa de división o partición de las matrices es la más rápida y la que mejores resultados presenta desde el punto de vista de recursos consumidos. A medida que el tamaño de las matrices de entrada aumenta, este algoritmo mejora aún más sus resultados con respecto a los otros. El tiempo empleado por el algoritmo de Strassen y el de particionado es el mismo, sin embargo, el número de recursos consumidos por este último se reduce a la mitad (ver Figura 4.9).

$$\mathbf{G}_{m\times n} = \mathbf{X}_{m\times i} \times \mathbf{B}_{l\times n} \quad g_{i,j} = \sum_{k=1}^{l} x_{i,k} \cdot b_{k,j} \quad \begin{matrix} i = 1...n \\ j = 1...l \end{matrix} \tag{4.3}$$

Como conclusiones de los algoritmos evaluados se puede decir que los tiempos de ejecución alcanzados son superiores a los que se emplearían en arquitecturas sistólicas. Sin embargo, estas arquitecturas no se pueden implementar en las FPGAs actuales cuando el tamaño de las matrices de entrada es superior a 20x20 debido al número máximo de multiplicadores internos disponibles. Por tanto, si se desean manejar matrices de tamaños superiores se deben emplear otro tipo de algoritmos no tan concurrentes como los sistólicos. Esta ha sido la razón principal por la que se han evaluado diferentes algoritmos de multiplicación no sistólicos [Bravo, 2006c].

Si se aplican estos algoritmos a PCA, atendiendo a la naturaleza de las matrices a multiplicar se observa que el propuesto por Winograd no es útil ya que sólo es válido para matrices cuadradas. Con respecto a los otros dos (Strassen y particionado), debido a los diferentes tipos de matrices que se deben manejar, su

arquitectura es bastante compleja. Además, los tiempos de ejecución para realizar las Mult. Mat. de PCA serían muy elevados. Por esta razón, se ha desarrollado una nueva alternativa en la Mult. Mat. basada en el diseño de un *array* semi-sistólico. Esta solución ofrece unos resultados de tiempo de ejecución mejores que los algoritmos evaluados anteriormente, a costa de consumir mayor número de recursos.

Tabla 4.2. Resumen de algoritmos para la Mult. Mat. en paralelo.

Método de multiplicación de matrices	**Características más importantes**
Particionado directo	Varias unidades de procesado, donde cada unidad calcula una parte de la matriz resultado. Problemas con los accesos a memoria compartida.
Divide and Conquer recursivo	Multiplicación mediante submatrices. Llamadas recursivas para la obtención de resultados. Aumento del tamaño de memoria requerido.
*Array*s sistólicos	*Array* de procesadores de dos dimensiones. Distribución simple de los cálculos dentro del *array*. Requerimientos de memoria iguales para todos los procesadores.
Cannon	Arquitectura sistólica realimentada. Distribución inicial de elementos de entrada. Minimización de los tiempos de ejecución.
Fox	Similar al de Cannon. Mecanismos de *broadcast* por filas. Mayor requerimiento de memoria
Winograd	Disminuye el número de operaciones aritméticas con respecto al método clásico. Difícil paralelización.
Strassen	Su funcionamiento se basa en la reutilización de resultados parciales. Sus tiempos de ejecución son bajos.
Particionado de matrices	Diseño de una unidad de Mult. Mat. de 2x2 elementos. Excelente relación tiempo ejecución vs. recursos consumidos.

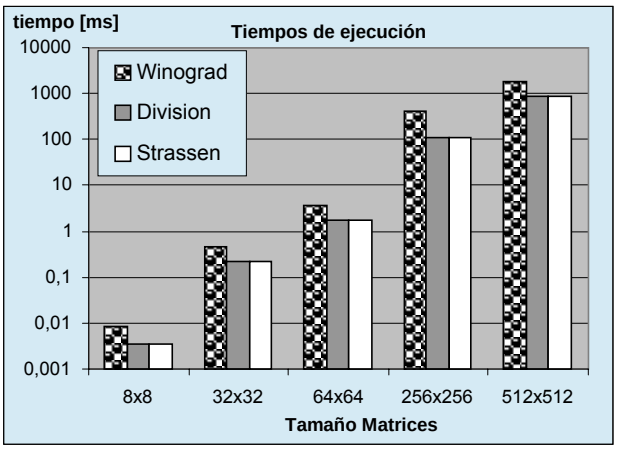


Figura 4.8. Tiempo de ejecución de la FPGA para los algoritmos evaluados y con diferentes tamaños de matrices.

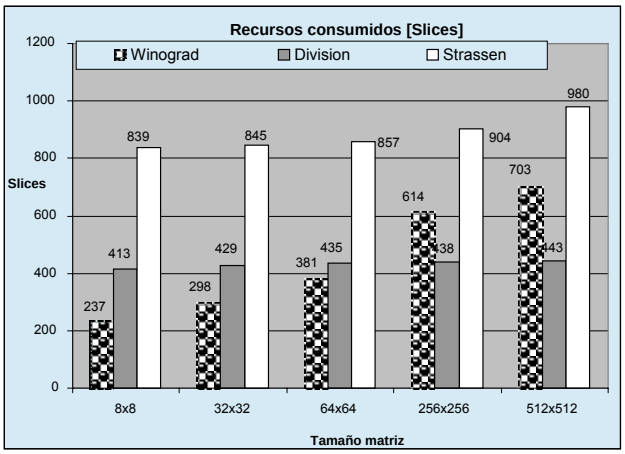


Figura 4.9. Recursos internos de la FPGA consumidos por los algoritmos evaluados y para diferentes tamaños de matrices.

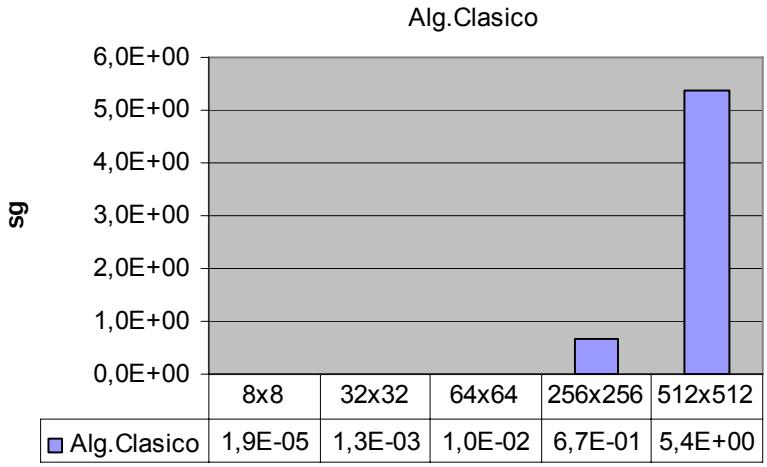


Figura 4.10. Tiempos de ejecución del algoritmo clásico para diferentes tamaños de matrices.

4.2.2.2. *Array* semi-sistólico para la multiplicación de matrices en FPGAs.

Analizando la naturaleza de las diferentes matrices a multiplicar en PCA se ha optado por el diseño de un *array* semi-sistólico que es empleado para las cuatro multiplicaciones de matrices de PCA. La elección de este tipo de *array* tiene como principal ventaja el ahorro de recursos internos con respecto a un sistólico, la posibilidad de manejar matrices cuadradas y no cuadradas, así como su reutilización en las 4 operaciones de multiplicación de matrices de PCA. Por el contrario, el tiempo de ejecución se ve incrementado con respecto a un *array* sistólico.

El diseño realizado ha sido construido para $M = 8$, pudiéndose observar en la Figura 4.11 su estructura interna. En dicha figura se identifican claramente las celdas encargadas de realizar la multiplicación, diferenciándose 3 tipos: 1x, 2x, 3x:

- La celda denominada como 1x se utiliza en las cuatro multiplicaciones de matrices de PCA. Internamente está compuesta por dos multiplexores y una unidad de multiplicación y acumulación denominada MAC2.
- Por otra parte, la celda 2x se emplea sólo en la segunda multiplicación (generación de matriz $\mathbf{U}_t$). Dado que únicamente se emplea en una multiplicación su estructura interna es básica, estando formada por un multiplicador hardware.
- Por último, las celdas 3x, localizadas en todas las filas del *array*, están divididas en dos tipos: las ubicadas entre la primera y tercera fila, y el resto. Las primeras se emplean para generar la matriz de covarianza y $\mathbf{U}_t$, mientras que el resto únicamente se emplea en la generación de la matriz de covarianza. Su estructura interna está compuesta por un multiplexor y una unidad de multiplicación y acumulación (MAC1).

El circuito de la Figura 4.11 además posee los siguientes elementos:

- *Buses de salida*: se han implementado un total de 5 buses de salida, estando algunos de ellos compartidos por varias operaciones de multiplicación de matrices de PCA.
- *Bloques de registros y sumadores*: estos 3 bloques distribuidos en las 3 primeras filas del *array* están diseñados para generar correctamente los datos de salida pertenecientes a la matriz $\mathbf{U}_t$ y $\hat{\mathbf{\Phi}}$, tal y como se justificará posteriormente.

Con respecto al tamaño de los datos de entrada y salida, el *array* diseñado admite diferentes valores, tanto de entrada como de salida, para así poder adaptarse a las diferentes necesidades de Mult. Mat. que se deben desarrollar.

Todas las celdas de multiplicación, así como los buses de salida, están gestionadas por una máquina de estados que se encarga de activar adecuadamente cada señal de control.

Una vez descrita la funcionalidad del *array*, en lo que sigue se analizará de forma más específica, el comportamiento interno del *array* en cada una de las 4 multiplicaciones de PCA.

Con respecto a los recursos consumidos por este bloque, se puede observar en la Tabla 4.3 como el número de éstos es bastante elevado. Sin embargo, debido a la importancia que tiene este bloque dentro de PCA, así como su uso en diferentes fases de este algoritmo, es asumible un número tan elevado de recursos.

Tabla 4.3. Recursos consumidos y máxima frecuencia de reloj, del *array* semi-sistólico implementado.

Área (*Slices*)	Multiplicadores	BRAM	Máxima frecuencia
1642 (33%)	40 (91%)	0	129,53MHz

4.2.2.3. Cálculo de la matriz de covarianza en el *array* semi-sistólico.

Una vez descrito el *array* sistólico y antes de analizar su funcionamiento en la obtención de la matriz de covarianza, se va a justificar matemáticamente la alternativa propuesta en esta tesis para el cálculo de los autovectores de dicha matriz. Una vez argumentada la opción escogida se describirá el funcionamiento del *array* para la creación de esta matriz.

4.2.2.3.1. Aspectos generales de la matriz de covarianza.

Según lo expuesto en el capítulo 3 la matriz $\mathbf{C} \in \Re^{N^2 \times N^2}$ viene dada por:

$$\mathbf{C} = \frac{1}{M}\mathbf{A}^T \cdot \mathbf{A} \tag{4.4}$$

La obtención de los autovectores de la matriz $\mathbf{C}$ conlleva un número muy elevado de operaciones aritméticas debido al elevado tamaño de $\mathbf{A} \in \Re^{N^2 \times M}$. Para reducir el número de éstas, una posible alternativa es a partir de los autovalores de $\mathbf{C}^T$ (4.5) ya que gracias a éstos se pueden obtener sus autovectores asociados, tal y como se justifica posteriormente.

$$\mathbf{C}^T = \frac{1}{M}\mathbf{A}^T \cdot \mathbf{A} \tag{4.5}$$

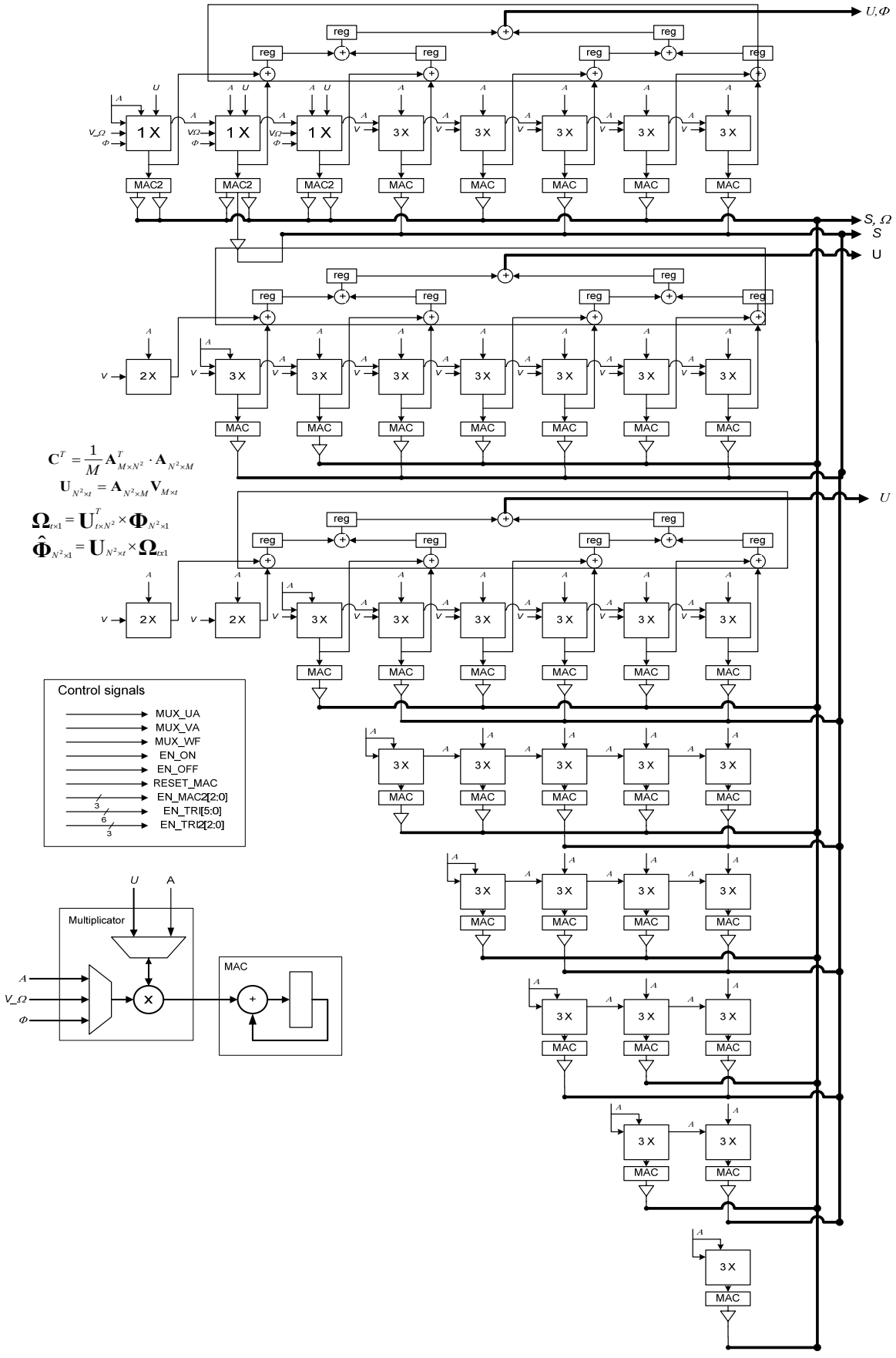


Figura 4.11. Diagrama de bloques del *array* semi-sistólico para implementar en una FPGA todas las multiplicaciones de matrices que demanda PCA.

Debido a que la matriz $\mathbf{C}$ es de dimensión $N^2 x N^2$ y $\mathbf{C}^T$ de MxM, resulta más sencilla la obtención de los autovalores y autovectores de $\mathbf{C}^T$ que de $\mathbf{C}$. Lógicamente una vez obtenidos los autovectores de $\mathbf{C}^T$ hay que obtener los de $\mathbf{C}$. Para establecer la relación entre los autovectores de $\mathbf{C}^T$ y $\mathbf{C}$ se puede escribir:

$$\mathbf{CU} = \boldsymbol{\lambda}\mathbf{IU} \tag{4.6}$$

$$\mathbf{C}^T\mathbf{V} = \boldsymbol{\lambda}'\mathbf{IV} \tag{4.7}$$

donde $\mathbf{U}$ y $\mathbf{V}$ son las matrices de autovectores de $\mathbf{C}$ y $\mathbf{C}^T$ respectivamente y $\boldsymbol{\lambda}$ y $\boldsymbol{\lambda}'$ los vectores de autovalores de $\mathbf{C}$ y $\mathbf{C}^T$.

Si se multiplica por la izquierda la expresión (4.7) por $\mathbf{A}$ se obtiene:

$$\frac{1}{M}\mathbf{A}\mathbf{A}^T(\mathbf{AV}) = \boldsymbol{\lambda}'\mathbf{I}(\mathbf{AV}) \tag{4.8}$$

Por tanto si $\boldsymbol{\lambda} = \boldsymbol{\lambda}'$ (condición que se cumple si $\mathbf{C}$ es simétrica), la relación entre $\mathbf{U}$ y $\mathbf{V}$ es:

$$\mathbf{U} = \mathbf{AV} \tag{4.9}$$

Para evaluar la reducción de carga computacional que supone la obtención de $\mathbf{U}$ utilizando $\mathbf{C}^T$ se supone:

a) Las imágenes de entrada que forman $\mathbf{A}$ siempre son cuadradas y de tamaño $N \times N$.

b) Los datos con los que se trabajan en todo momento son números reales. Así, todas las operaciones aritméticas a realizar se pueden descomponer en sumas y multiplicaciones reales. Estas dos operaciones se emplean como índices de comparación a la hora de justificar el método idóneo desde un punto de vista de carga computacional.

c) El número de imágenes captadas para obtener $\mathbf{C}$ es M.

Bajo estas condiciones el tamaño de la matriz $\mathbf{A}$ es $N^2 \times M$ y por tanto la obtención de $\mathbf{U}$ directamente a partir de $\mathbf{C}$ supone: $N^4 \cdot M$ operaciones de multiplicación y $N^4 \cdot (M-1)$ operaciones suma para obtener $\mathbf{C}$.

Si por el contrario se obtiene $\mathbf{U}$ a partir de $\mathbf{C}^T$ el número de operaciones para obtener $\mathbf{C}^T$ sería: $M^2 \cdot N^2$ operaciones de multiplicación y $M^2 \cdot (N^2 - 1)$ operaciones suma. A este número de operaciones hay que añadir ahora las que se necesitan para obtener $\mathbf{U}$ a partir de $\mathbf{V}$. En este caso se emplean $N^2 \cdot M^2$

operaciones multiplicación y $N^2M \cdot (M-1)$ operaciones suma. Las operaciones de normalización de $\mathbf{U}$, tal y como se describe en los apartados 4.2.5 y 4.3.2, se solapan temporalmente con otras operaciones, por lo que no se tienen en cuenta.

Llamando L_{C_p} y $L_{C_p^T}$ al número total de multiplicaciones utilizando $\mathbf{C}$ y $\mathbf{C}^T$ respectivamente, se tiene:

$$L_{C_p} = N^4 M \tag{4.10}$$

$$L_{C_p^T} = 2M^2N^2 \tag{4.11}$$

$$\frac{L_{C_p}}{L_{C_p^T}} = \frac{N^2}{2M} \tag{4.12}$$

Para el número de sumas aritméticas, se denomina como L_{C_s} y $L_{C_s^T}$ al número total de sumas utilizando $\mathbf{C}$ y $\mathbf{C}^T$ respectivamente, se obtiene:

$$L_{C_s} = N^4(M-1) \tag{4.13}$$

$$L_{C_s^T} = N^2\left(2M^2 - M - \frac{M^2}{N^2}\right) \tag{4.14}$$

$$\frac{L_{C_s}}{L_{C_s^T}} = \frac{N^2(M-1)}{\left(2M^2 - M - \frac{M^2}{N^2}\right)} \tag{4.15}$$

En los casos prácticos el valor de $M << N^2$ por lo que (4.15) se puede expresar como:

$$\frac{L_{C_s}}{L_{C_s^T}} \approx \frac{N^2(M-1)}{M(2M-1)} \tag{4.16}$$

Para evaluar (4.12) y (4.16) se presenta en la Figura 4.12 los ratios de las operaciones de multiplicación (4.12) y de suma (4.16) para el caso práctico de $N^2 = 256^2$ para un rango de $M = [6,16]$. Se observa como el número de operaciones necesarias en el cálculo de autovalores y autovectores es siempre mayor cuando se realiza a través de $\mathbf{C}$ en vez de $\mathbf{C}^T$, alcanzándose los valores máximos cuando $M << N$. Con ello se justifica que si $M << N$, situación que ocurre en esta tesis, el cálculo de autovalores y autovectores mediante $\mathbf{C}^T$ necesita muchas menos operaciones que a través de $\mathbf{C}$.

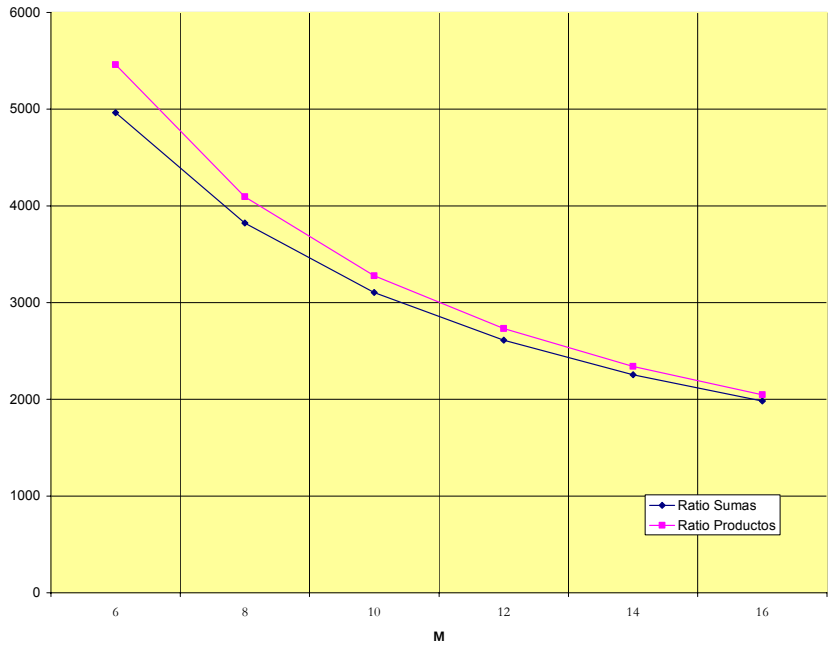


Figura 4.12. Ratios de operaciones suma y multiplicación para el cálculo de la matriz de covarianza particularizado para N=256 y M variable.

4.2.2.3.2. Funcionamiento del array semi-sistólico en la generación de la matriz de covarianza.

Según lo expuesto anteriormente, para la obtención de la matriz de autovectores $\mathbf{U}$ se debe calcular previamente la matriz $\mathbf{C}^T$. Para ello, se hace uso del *array* semi-sistólico mostrado en la Figura 4.11 utilizando las celdas de multiplicación 1x y 3x de dicha figura. Así, en la obtención de $\mathbf{C}^T$ únicamente se necesitan como datos de entrada $\mathbf{A}$ y $\mathbf{A}^T$. Una vez obtenido el producto $\mathbf{A}^T \cdot \mathbf{A}$, éste se dividirá entre M para obtener $\mathbf{C}^T$ (4.5).Con respecto al tamaño de los datos de entrada en este caso son de 9 bits, ya que todos los elementos de la matriz $\mathbf{A}$ poseen este tamaño. Los datos se salida son acotados a 18 bits ya que tal y como se verá posteriormente, es el tamaño escogido como patrón en esta tesis.

El *array* comienza a funcionar en el momento en el que el bloque de cálculo del vector media (apartado 4.2.1) genera los primeros datos de salida, correspondiendo éstos con la primera fila de la matriz $\mathbf{A}$ o primera columna de $\mathbf{A}^T$ (ver Figura 4.13). Debido a la naturaleza de las matrices a multiplicar, cada fila de $\mathbf{A}$ está asociada a la columna correspondiente de $\mathbf{A}^T$. Así, si se denomina como $a_{i,j}$ al elemento perteneciente a la fila i, columna j de la matriz $\mathbf{A}$, los primeros datos de entrada son $a_{1,1}, a_{1,2}, ..., a_{1,8}$ los cuales corresponderán con la primera fila de $\mathbf{A}$ y la primera columna de $\mathbf{A}^T$. Debido a la forma en que llegan los datos de $\mathbf{A}$ no se puede generar ningún dato de salida de la matriz de covarianza hasta que no llegue al *array* la última fila de $\mathbf{A}$.

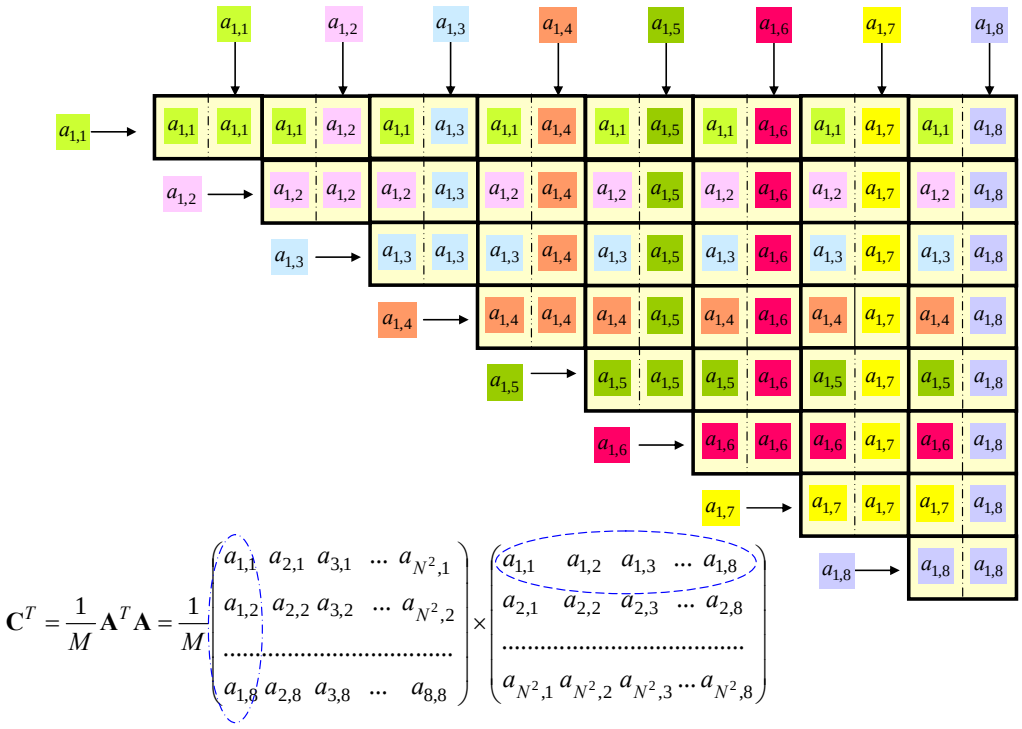


Figura 4.13. Secuencia de entrada de la primera fila de datos de $\mathbf{A}$ en el *array* semi-sistólico, para la generación de $\mathbf{C}^T$.

Si se numeran las celdas de multiplicación del *array* de la Figura 4.11 en filas y columnas siendo la celda (1,1) la ubicada en la esquina superior izquierda, para realizar la multiplicación $\mathbf{A}^T \cdot \mathbf{A}$ se introduce cada uno de los datos de entrada en las celdas asociadas a la *j-ésima* columna, así como a las asociadas a la *j-ésima* fila (Figura 4.13). Así por ejemplo, el dato $a_{1,4}$ se introduce en todas las celdas de la fila 4 y en todas las celdas de la columna 4. Con estos primeros datos de entrada, cada celda dispone de dos datos que deben multiplicarse entre sí y almacenarse internamente en las unidades MAC de cada celda de multiplicación (ver Figura 4.13).

Una vez finalizada la multiplicación de la primera columna de $\mathbf{A}^T$ por la primera fila de $\mathbf{A}$, de nuevo el *array* está preparado para recibir la siguiente fila de $\mathbf{A}$. Estos datos son distribuidos y multiplicados en cada celda de manera idéntica a lo expuesto anteriormente. Los resultados de esta segunda multiplicación son sumados a los de la primera, almacenándose en cada celda la suma acumulada en la unidad MAC.

Repitiendo el proceso de multiplicación de forma iterativa, cuando llegue la última fila de datos de $\mathbf{A}$ se realiza, en cada celda, la última multiplicación y acumulación. En este momento el contenido de cada unidad MAC asociada a cada celda es uno de los 36 elementos que forman la matriz $\mathbf{C}^T$. El tamaño de la matriz $\mathbf{C}^T$ para $M = 8$ es 8×8, con lo que $\mathbf{C}^T$ esta compuesta por 64 elementos. Sin embargo, aprovechando la propiedad de simetría de $\mathbf{C}^T$, únicamente se trabajará con

la matriz triangular superior. Gracias a esto, se reduce notablemente el número de operaciones aritméticas a realizar, tanto en el proceso de multiplicación de matrices como en el del cálculo de autovalores y autovectores. En la Figura 4.14 se muestra la segmentación realizada en esta multiplicación de matrices.

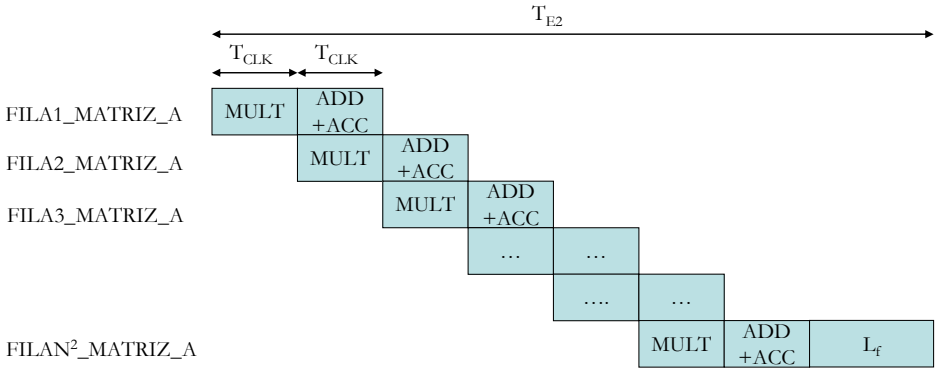


Figura 4.14. Temporización de la generación de la matriz de covarianza $\mathbf{C}^T$ en el *array* semi-sistólico.

Una vez finalizada la multiplicación de las dos matrices $\mathbf{A}^T$ y $\mathbf{A}$, se debe extraer de cada unidad MAC su contenido, dividirlo entre M (4.5) y enviarlo hacia el bloque de cálculo de autovalores y autovectores (ver Figura 4.2). Debido a que $M = 8$ la división se transforma en un desplazamiento que es realizado en el *array*.

El interfaz de comunicación entre el array y la etapa de autovectores es a través de una memoria *Dual Port* contenida en la etapa de autovectores. Esta memoria funciona al doble de velocidad que el *array*, con lo que se pueden escribir simultáneamente 4 datos en un ciclo de reloj. Debido a esto, se utilizan dos buses de salida extrayéndose los datos de las unidades MAC de forma ordenada para su posterior manejo en el cálculo de autovectores. El tiempo empleado por esta última fase es la tarea L_f de la Figura 4.14 (ver (4.17)). En (4.18) se presenta el número total de ciclos de reloj empleados en la multiplicación de matrices $\frac{1}{M}\mathbf{A}^T \cdot \mathbf{A}$, suponiendo $N = 256$ que es el tamaño de las imágenes que se manejan en esta tesis.

$$Lf = \frac{36 datos}{2 datos / ciclo\_reloj} T_{CLK2X} = \frac{36}{4} T_{CLK} = 9 T_{CLK} \tag{4.17}$$

$$T_{E2} = (N^2 + 9) T_{CLK} = 65545 T_{CLK} \tag{4.18}$$

Con respecto al número de bits empleados en la codificación de los datos de entrada (elementos de $\mathbf{A}$), éste es de 9 bits. En el *array* diseñado para realizar esta primera Mult. Mat. con ese tamaño de datos de entrada, teóricamente cada dato de salida de la matriz $\mathbf{C}^T$ puede alcanzar hasta 34 bits (4.19) empleando una codificación de 8 bits para los píxeles captadas por el sensor CMOS ($p = 8$), $M = 8$ y

$N = 256$. Sin embargo, se justifica empíricamente que el tamaño máximo de bits que se alcanza en los datos de salida para esta primera multiplicación es de 30 bits, siendo este caso muy puntual y estando asociada a imágenes captadas con ciertos defectos (saturación de la imagen, movimiento de la imagen, etc.). En la Figura 4.15 se muestran los tamaños máximos y mínimos de los datos de $\mathbf{C}^T$ producidos en el *array* semi-sistólico para un conjunto de 1000 imágenes.

$$n^o\_bits\_\mathrm{cov} = \log_2\left(\sum_{i=1}^{N^2}\left(2_i^{p+1}\times 2_i^{p+1}\right)\right) = \log_2\left(\sum_{i=1}^{256^2} 2_i^{18}\right) = 34 \tag{4.19}$$

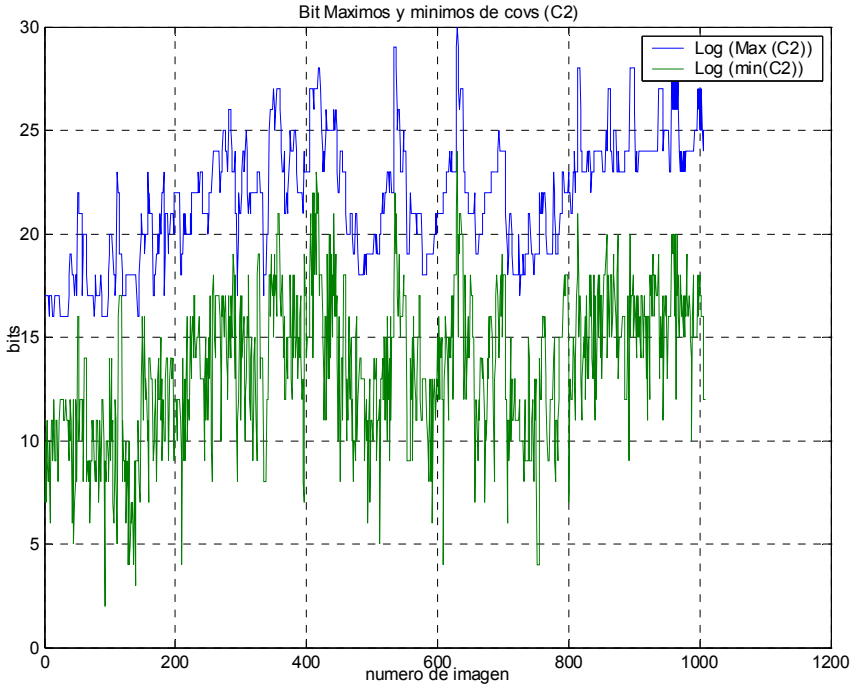


Figura 4.15. Tamaño máximo de bits alcanzado por la matriz de covarianza sobre un conjunto de 1000 imágenes.

Tras analizar los resultados de la Figura 4.15, se observa que:

- El valor máximo medio de bits de las matrices de covarianza generadas con un conjunto de 1000 imágenes de diferentes escenas, sin tener en cuneta las imágenes con situaciones anómalas, se establece entorno a 24 bits.
- Se debe realizar un truncamiento en los datos de las matrices de covarianza a 18 bits debido a que el módulo de cálculo de autovalores y autovectores, tal y como se expondrá en el apartado 4.2.3.9, posee este ancho de palabra.
- Debido a la estructura interna del módulo de autovectores diseñado, existen operaciones aritméticas internas que producen *overflow*. Para evitar esta situación, se deben establecer bits de salvaguarda. Posteriormente se justificará en el apartado 4.2.3.10.2, que tras

analizar los resultados prácticos ofrecidos por el módulo de cálculo de autovectores, se debe establecer un valor de 2 bits de salvaguarda. Por esta razón, el valor máximo que deben alcanzar los datos de la covarianza si se desea un ancho de palabra de 18 bits ($n = 18$) es: $2^{n-bits\_salvag} = 2^{18-2} = 2^{16}$. Debido a esto, los datos de la matriz de covarianza generados se truncan hasta tener 16 bits. Este truncamiento provoca un error de precisión en los datos de salida. Para ver la influencia que tienen estos truncamientos en el funcionamiento de PCA, en el apartado 4.4, se realiza un estudio de la influencia de este truncamiento sobre el error final de recuperación.

4.2.3. Cálculo de autovalores y autovectores.

Dentro del algoritmo PCA el cálculo de autovalores y autovectores es una de las operaciones que requiere mayor carga computacional. Esto representa un gran hándicap dentro de las FPGAs debido a su naturaleza interna. Por esta razón, se deben crear soluciones que se ejecuten concurrentemente dentro de estos dispositivos. Este apartado presenta una nueva propuesta para el cálculo de autovalores y autovectores en dispositivos FPGA mejorando el trabajo propuesto en [Brent, 1985]. Inicialmente se implementó en esta tesis la propuesta de [Brent, 1985] y [Cavallaro, 1988] en FPGAs, tal y como se expone en 4.2.3.7. Sin embargo, esta opción consume masivamente recursos internos de la FPGA por lo que fue descartada. Así, surgió una nueva propuesta que en lugar de utilizar un sistema sistólico de procesadores tal y como se propone en [Brent, 1985] y [Cavallaro, 1988], emplea un sistema concurrente que permite ahorrar hasta un 90% el número de recursos internos de la FPGA con respecto al método anterior. Al igual que en [Cavallaro, 1988] el funcionamiento del sistema propuesto y desarrollado en esta tesis está basado en módulos CORDIC (*COordinate Rotation DIgital Computer*) que se encargan de ejecutar ciertas operaciones trigonométricas, que posteriormente se justifican. Además, en la solución desarrollada en esta tesis se aprovecha la simetría de la matriz de entrada (matriz de covarianza) generada por el *array* semi-sistólico de Mult. Mat., reduciendo así el número de operaciones aritméticas dentro del cálculo de autovalores y autovectores.

4.2.3.1. Métodos de cálculo de autovectores y autovalores

El cálculo de autovalores y autovectores es un problema que se presenta en numerosas aplicaciones prácticas, entre las que se encuentran aquellas que utilizan técnicas de PCA en visión artificial [Qiu, 2003].

En cualquier caso, la solución para la obtención de autovalores y autovectores que se propone en esta tesis puede ser útil en otros campos de PCA, como puede ser el reconocimiento de patrones [Jiménez, 2005] [Yang, 2000] [Haiqing, 2000]. Para que esta propuesta sea aplicable a otras áreas, la única condición que se debe cumplir es que la matriz sobre la que se calculen sus autovectores y autovalores, sea real y

simétrica. En lo que se refiere a la dimensión de las matrices, la solución desarrollada permite una fácil expansión a diferentes tamaños. Sin embargo en nuestro trabajo se ha particularizado a matrices cuadradas de 8x8 $(M=8)$. La elección de este tamaño viene justificada básicamente por tres razones:

a) Atendiendo a los algoritmos de visión: Según el estudio experimental realizado en [Vázquez, 2006] para PCA aplicado a visión artificial, el número óptimo de imágenes para construir el modelo de fondo de PCA en la detección de objetos, se establece entorno a 10 imágenes $(M=10)$. Sin embargo, los resultados obtenidos para $M=8$ también se consideran adecuados.

b) El número de recursos consumidos dentro de la FPGA empleada: La FPGA empleada en esta tesis doctoral (XC2VP7) no posee un excesivo número de recursos internos (aproximadamente 5000 *slices*), por lo que el tamaño de $M=8$ ha permitido un consumo moderado de recursos (entorno al 25%) dejando capacidad suficiente para el resto de bloques de PCA.

c) Bus de datos de memoria externa: En nuestro caso la memoria externa almacena las imágenes necesarias para construir la matriz $\mathbf{A}$. El ancho del bus de datos de memoria externa es de 128 bits. Sin embargo, con objeto de evitar accesos simultáneos a memoria, se ha limitado el ancho a 64 bits de escritura y 64 de lectura. Debido a esto y a que los niveles de gris de cada píxel se codifica con 8 bits (inicialmente el sensor CMOS los envía con 10 bits) en un solo ciclo de reloj se pueden leer y escribir a la vez hasta 8 píxeles.

Buscando la estabilidad del sistema, la arquitectura diseñada permite una fácil expansión. Empleando una FPGA con capacidad adecuada se puede extender la arquitectura propuesta a diferentes tamaños.

Con respecto al cálculo de autovalores y autovectores, habitualmente éste se desarrolla en sistemas basados en procesadores digitales [Minami, 1992] [Choi, 1997]. Debido a las características internas de sus arquitecturas, el proceso de cálculo es eminentemente secuencial, empleando un elevado tiempo de cómputo en ello. Con el fin de mejorar la velocidad del sistema, en algunas arquitecturas se guardan en tablas de memoria los valores trigonométricos necesarios para realizar el cálculo de autovalores y autovectores [Herron, 2003]. La gran ventaja de este tipo de sistemas es la precisión alcanzada, debido principalmente a la existencia de unidades aritméticas de coma flotante.

En cuanto a los métodos matemáticos para calcular los autovalores y autovectores de una matriz inicial $\mathbf{C}^T$, sus autovalores asociados ($\boldsymbol{\lambda}$) y su matriz de autovectores ($\mathbf{V}$), todas las alternativas se basan en la expresión (4.20)

$$\mathbf{C}^T \cdot \mathbf{V} = \boldsymbol{\lambda}\mathbf{I} \cdot \mathbf{V} \tag{4.20}$$

En este caso los autovalores son reales debido a que las matrices de entrada son simétricas.

Para la obtención de los autovalores de una matriz utilizando hardware específico se han propuesto diferentes técnicas todas ellas basadas en métodos recurrentes que buscan la diagonalización de la matriz. Una vez diagonalizada ésta, los autovalores coinciden con los valores de la diagonal. Existen diferentes alternativas en el cálculo de autovalores que se pueden clasificar en aquellas que utilizan transformaciones más iteraciones (QR, QZ, Muller, etc.) [Parlett, 2000] [Francis, 1961] [Rutishauser, 1977] y las que utilizan únicamente iteraciones (Jacobi, Givens, etc.) [Givens, 1953] [Sleijpen, 2000]. De estas dos últimas, la basada en el método propuesto por Jacobi [Wilkinson, 1999] es la más interesante, ya que facilita la implementación de estructuras hardware de procesamiento paralelo.

La solución propuesta en esta tesis está basada en el método de Jacobi, pero a diferencia de otros trabajos presenta una arquitectura diferente, intentando en todo momento acelerar el tiempo de ejecución y reduciendo el número de recursos internos consumidos de la FPGA. Para ello, se emplean módulos que se ejecutan en paralelo, no existiendo un alto grado de dependencia de datos entre los distintos módulos, pudiendo así trabajar con elevada velocidad en el cómputo de los autovalores y autovectores. Otra gran ventaja de la solución propuesta, es la posibilidad de extender la arquitectura para diferentes tamaños de la matriz de entrada, ya que la propuesta realizada utiliza módulos con estructuras iguales y por ello se puede ampliar sin necesidad de grandes cambios en la configuración de la arquitectura global. El tipo de datos que se maneja está codificado en coma fija, permitiéndose variar su tamaño cómodamente.

A continuación se describe el algoritmo de Jacobi, así como la arquitectura clásica asociada a este algoritmo.

4.2.3.2. Algoritmo de Jacobi.

El método de Jacobi fue desarrollado en 1846 y se basa en la diagonalización de una matriz original mediante una secuencia de rotaciones [Wilkinson, 1999]. Una vez obtenida la matriz diagonal, $\mathbf{D}$, los valores que aparecen en su diagonal coinciden con los valores singulares (SVD) de la matriz original, y dado que ésta es una matriz Hermítica, los valores singulares coinciden a su vez con sus autovalores. Esto es debido a que dada una matriz inicial $\mathbf{C}^T$ que se denomina de aquí en adelante como $\mathbf{S} \in \Re^{M \times M}$ para simplificar su notación, sus autovalores asociados ($\boldsymbol{\lambda}$) y su matriz de autovectores ($\mathbf{V} \in \Re^{M \times M}$) se obtienen a partir de la expresión (4.20). Si en (4.20) se multiplica por la izquierda por $\mathbf{V}^{-1}$, se obtiene:

$$\mathbf{V}^{-1}\mathbf{S}\mathbf{V} = \mathbf{D} \tag{4.21}$$

donde la matriz $\mathbf{D} \in \Re^{M \times M}$ corresponde con una matriz diagonalizada que contiene los autovalores de $\mathbf{S}$. Al multiplicar por sí misma la expresión (4.21) se obtiene la expresión (4.22)

$$\begin{aligned} &\mathbf{V}^{-1}\mathbf{S}\mathbf{V}\mathbf{V}^{-1}\mathbf{S}\mathbf{V} = \mathbf{D}\mathbf{D} \\ &\mathbf{V}^{-1}\mathbf{S}^2\mathbf{V} = \mathbf{D}^2 \end{aligned} \tag{4.22}$$

Por tanto, la expresión (4.22) generalizada para cualquier potencia a se puede expresar como:

$$\mathbf{V}^{-1}\mathbf{S}^a\mathbf{V} = \mathbf{D}^a \tag{4.23}$$

La matriz $\mathbf{V}$ es unitaria y ortogonal por lo que su inversa es igual a su transpuesta ($\mathbf{V}^T = \mathbf{V}^{-1}$). Así, se puede reescribir la expresión (4.23) como:

$$\mathbf{V}^{\mathrm{T}}\mathbf{S}^a\mathbf{V} = \mathbf{D}^a \tag{4.24}$$

Los valores singulares de una matriz $\mathbf{S}$, son la raíz cuadrada de los autovalores de $\mathbf{S}^T \cdot \mathbf{S}$ [Weisstein, 1999]. Como el término $\mathbf{S}^T \cdot \mathbf{S}$ equivale a $\mathbf{S}^2$, los valores singulares de $\mathbf{S}^2$ corresponden con los autovalores de $\mathbf{S}$ según (4.24). Por tanto, el método de Jacobi se puede aplicar a la obtención de los autovalores y autovectores de matrices Hermíticas. En el caso de PCA esta cualidad se cumple, ya que la matriz sobre la que se determinan sus autovalores y autovectores, es una matriz de covarianza la cual siempre es Hermítica.

El algoritmo de Jacobi es un método iterativo donde teóricamente se necesitarían un número infinito de iteraciones. En la práctica, el número de iteraciones necesarias, se fija teniendo en cuenta que el elemento de fuera de la diagonal con mayor valor absoluto sea inferior a una tolerancia prefijada. En ese momento se habrá conseguido diagonalizar la matriz original ($\mathbf{S}$) y por lo tanto se habrá obtenido $\mathbf{D}$. Así, particularizando (4.24) para $a = 1$ expresada en función de la iteración actual (k), se obtiene:

$$\mathbf{S}^{(k+1)} = \mathbf{V}^{(k)^T} \cdot \mathbf{S}^{(k)} \cdot \mathbf{V}^{(k)} \tag{4.25}$$

donde k es el número de iteración y su valor va desde 0 hasta teóricamente infinito. Para $k = 0$, $\mathbf{S}^{(0)} = \mathbf{S}$. Nótese que la matriz $\mathbf{D}$ ha sido sustituida por $\mathbf{S}^{(k+1)}$. Esta matriz representa el resultado de realizar el producto de la matriz $\mathbf{V}^{(k)}$ y su transpuesta por la matriz $\mathbf{S}$ de la iteración anterior ($\mathbf{S}^{(k)}$). Suponiendo que en la

iteración x se alcanza la tolerancia prefijada en la diagonalización, en ese caso $\mathbf{S}^{(x)} = \mathbf{D}$.

Por tanto, el objetivo de este método es encontrar una matriz $\mathbf{V}^{(x)} = \mathbf{V}$, de tal forma que al realizarse la doble multiplicación de la expresión (4.24), resulten prácticamente nulos todos los coeficientes de $\mathbf{D}$ excepto los de la diagonal. Justamente es en esta característica en la que se fundamenta la propuesta de Jacobi: la búsqueda de una matriz final $\mathbf{D}$ haciendo paulatinamente cero los elementos no pertenecientes a la diagonal. Para conseguir eliminar coeficientes progresivamente, el método de Jacobi emplea las matrices de rotación que posteriormente se denominaron como de Givens [Givens, 1953] [Golub, 1989]. Este tipo de matrices son de tipo unitario y permiten la eliminación de un determinado elemento (i, j) de la matriz $\mathbf{S}^{(k)}$, mediante la multiplicación de la matriz original por la matriz de Givens o matriz de rotación ($\mathbf{R}\left(\alpha^{(k)}\right) \in \Re^{M \times M}$). Ésta es igual a la matriz identidad, a excepción de los elementos $r_{i,i} = r_{j,j} = \cos$ y $r_{i,j} = -r_{j,i} = \sin$ (ver (4.26)), donde i corresponde a la fila y j a la columna del elemento de fuera de la diagonal de $\mathbf{S}^{(k)}$ que se desea eliminar. Así, el método de Jacobi emplea como matriz $\mathbf{V}^{(k)}$ de la expresión (4.25) la matriz de Givens ($\mathbf{R}\left(\alpha^{(k)}\right) \in \Re^{M \times M}$) (4.26), transformándose (4.25) en (4.27).

$$\mathbf{R}\left(\alpha^{(k)}\right) = \begin{pmatrix} 1 & . & 0 & . & 0 & . & 0 \\ . & & . & & . & & . \\ . & & . & & . & & . \\ 0 & & \cos & & \sin & & 0 \\ 0 & & -\sin & & \cos & & 0 \\ . & & . & & . & & . \\ . & & . & & . & & . \\ 0 & & 0 & & 0 & & 1 \end{pmatrix} \tag{4.26}$$

donde $\cos = \cos(\alpha^{(k)})$ y $\sin = \sin(\alpha^{(k)})$. El valor del ángulo $\alpha^{(k)}$ depende del elemento (i, j) de la matriz $\mathbf{S}^{(k)}$ tal y como se justificará posteriormente.

$$\mathbf{S}^{(k+1)} = R\left(\alpha^{(k)}\right)^T \cdot \mathbf{S}^{(k)} \cdot R\left(\alpha^{(k)}\right) \tag{4.27}$$

Dos son las variantes que este método posee [García, 2005]:

- *Jacobi clásico*: en cada iteración se hace cero el mayor elemento en valor absoluto de fuera de la diagonal.
- *Jacobi cíclico*: en cada iteración se realiza un barrido por filas, para anular sucesivamente todos los elementos de fuera de la diagonal.

En la matriz de Givens el módulo de cada una de sus columnas vale uno y al multiplicarse por la matriz $\mathbf{S}^{(k)}$ permite ir eliminando elementos que no pertenezcan a la diagonal. Como la matriz $\mathbf{R}\left(\alpha^{(k)}\right)$ sólo elimina un elemento (dos si la matriz es simétrica) se necesita aplicar recursivamente diferentes matrices $\mathbf{R}\left(\alpha^{(k)}\right)$, de tal forma que los factores trigonométricos de estas matrices se vayan desplazando por las posiciones de los elementos a eliminar. La forma de ir recorriendo los elementos a eliminar puede ser mediante el método clásico o mediante el cíclico. Esto implica que en cada iteración se debe crear una nueva matriz $\mathbf{R}\left(\alpha^{(k)}\right)$ en función del elemento a eliminar. Cuando se realice la última iteración se habrá obtenido una matriz diagonal ($\mathbf{D}$) donde los elementos de la diagonal se corresponden con los autovalores y donde la matriz $\mathbf{R}\left(\alpha^{(k)}\right)$, contendrá en cada columna los autovectores asociados. Para ilustrar el funcionamiento de este algoritmo se supone una matriz $\mathbf{S}^{(0)}$ simétrica de tamaño $M \times M$, donde en cada iteración se tiene que calcular el valor de la matriz $\mathbf{R}\left(\alpha^{(k)}\right)$ aplicando el método de Jacobi cíclico. Así, supóngase que el primer valor a eliminar es el s_{12}, posteriormente el s_{13}, etc. La expresión (4.28) presenta el valor de las matrices $\mathbf{S}^{(k)} \in \Re^{M\times M}$ y $\mathbf{R}\left(\alpha^{(k)}\right) \in \Re^{M\times M}$ para la primera iteración $(k=0)$. En dicha expresión se debe determinar el valor de $\alpha^{(0)}$ para conseguir eliminar el elemento $s_{1,2}^{(1)}$ y por el hecho de ser $\mathbf{S}^{(0)}$ simétrica, también el $s_{2,1}^{(1)}$.

$$\mathbf{S}^{(1)} = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0.....0 \\ -\sin(\alpha) & \cos(\alpha) & 0.....0 \\ 0 & 0 & 1.....0 \\ ..... & & \\ 0 & 0 & 0.....1 \end{pmatrix}^{T^{(0)}} \cdot \begin{pmatrix} s_{11}s_{12}.....s_{1M} \\ s_{21}s_{22}.....s_{2M} \\ ..... \\ ..... \\ s_{M1}s_{M2}.....s_{MM} \end{pmatrix}^{(0)} \cdot \begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0.....0 \\ -\sin(\alpha) & \cos(\alpha) & 0.....0 \\ 0 & 0 & 1.....0 \\ ..... & & \\ 0 & 0 & 0.....1 \end{pmatrix}^{(0)} \tag{4.28}$$

Si el objetivo es que $s_{1,2}^{(1)} = 0$ desarrollando su valor desde (4.28) se obtiene (4.29).

$$\begin{aligned} s_{1,2}^{(1)} = s_{1,2}^{(0)}\cos^2(\alpha^{(0)}) + s_{1,1}^{(0)}\sin(\alpha^{(0)})\cos(\alpha^{(0)}) - \\ s_{2,2}^{(0)}\cos(\alpha^{(0)})\sin(\alpha^{(0)}) - s_{2,1}^{(0)}\sin^2(\alpha^{(0)}) = 0 \end{aligned} \tag{4.29}$$

Como $2\sin(\alpha^{(0)})\cos(\alpha^{(0)}) = \sin(2\alpha^{(0)})$, $\cos^2(\alpha^{(0)}) - \sin^2(\alpha^{(0)}) = \cos(2\alpha^{(0)})$ y además $s_{1,2}^{(0)} = s_{2,1}^{(0)}$, la expresión (4.29) se simplifica a (4.30) extrayendo de esta última el valor del ángulo para la primera iteración ($\alpha^{(0)}$) (ver (4.31)).

$$\left(s_{1,1}^{(0)} - s_{2,2}^{(0)}\right)\cdot \sin\left(2\alpha^{(0)}\right) + 2s_{1,2}^{(0)}\cos(2\alpha^{(0)}) = 0 \tag{4.30}$$

$$\tan\left(2\alpha^{(0)}\right)=\frac{2s_{1,2}^{(0)}}{s_{2,2}^{(0)}-s_{1,1}^{(0)}};$$
$$\alpha^{(0)}=\frac{1}{2}\tan^{-1}\left(\frac{2s_{1,2}^{(0)}}{s_{2,2}^{(0)}-s_{1,1}^{(0)}}\right) \quad (4.31)$$

Si ahora se sustituye en (4.28) el valor de $\alpha^{(0)}$ de (4.31), al realizar las dos multiplicaciones de matrices se consigue hacer cero $s_{1,2}^{(1)}$ y $s_{2,1}^{(1)}$ en el caso de matrices simétricas. Además, al realizar la doble multiplicación de (4.28) provoca que el resto de los elementos de la matriz $\mathbf{S}^{(1)}\in\Re^{M\times M}$ también vean modificado su valor siendo en posteriores iteraciones cuando se anularán. Tras este proceso, la matriz $\mathbf{S}^{(1)}$ posee dos elementos de fuera de la diagonal nulos y habría finalizado la primera iteración. Los valores anulados ya no se verán modificados en posteriores iteraciones, ya que la rotación de un valor nulo es cero.

En la segunda iteración el elemento a eliminar es $s_{1,3}^{(2)}$ y debido a la simetría de $\mathbf{S}^{(2)}$ también $s_{3,1}^{(2)}$. Para ello, se fija en la matriz $\mathbf{R}\left(\alpha^{(k)}\right)$ los índices i,j en $1,3$. De forma análoga a lo expuesto en (4.31) se busca el valor de $\alpha^{(1)}$ que permita eliminar $s_{1,3}^{(2)}$. Este proceso se repite continuamente hasta que los elementos de fuera de la diagonal sean cero o próximos a éste, situación en la cual se obtendrá la matriz diagonalizada $\mathbf{D}$.

En cuanto a los autovectores, la matriz final de autovectores $\mathbf{V}=\mathbf{V}^{(x)}$ es el producto final de las matrices de rotación empleadas en cada iteración, donde cada columna $v_j^{(x)}$de $\mathbf{V}^{(x)}$ se corresponde con el autovector asociado al autovalor $\lambda_{j,j}^{(x)}$ de $\mathbf{S}^{(x)}\in\Re^{M\times M}$. La expresión (4.32) presenta esta condición donde el superíndice x hace alusión a la iteración en la cual el sistema finaliza.

$$\mathbf{V}=\prod_{k=0}^{x}\mathbf{R}\left(\alpha^{(k)}\right) \quad (4.32)$$

4.2.3.3. Cálculo de autovalores con el método de Jacobi, mediante descomposición en matrices de 2x2 elementos.

Para conseguir los autovalores de una matriz Hermítica, el método de Jacobi desarrollado en [Forsythe, 1960] [Brent, 1985] propone la descomposición de la matriz original $\mathbf{S}\in\Re^{M\times M}$ (4.33) en submatrices de 2×2 elementos ($\mathbf{S}_{i,j}\in\Re^{2\times 2}$) (4.34)(4.35). Esta división de las matrices, entre otras ventajas, permite la diagonalización final en un número menor de iteraciones con respecto a matrices enteras, acorde a lo presentado en el apartado anterior. Según el trabajo desarrollado

en [Brent, 1983], la solución se alcanza en aproximadamente $M \cdot \log(M)$ iteraciones, siendo $M \times M$ el tamaño de la matriz inicial. Esto es debido a la posibilidad de hacer nulo más de un elemento en una misma iteración. Además, tal y como se expone posteriormente, está división permite el diseño de módulos idénticos para el procesado de las submatrices, lo que supone una ventaja a la hora de realizar un diseño hardware de estructura regular.

$$\mathbf{S}^{(k)} = \begin{pmatrix} s_{11}^{(k)} s_{12}^{(k)} \dots\dots\dots s_{1(M\text{-}1)}^{(k)} s_{1M}^{(k)} \\ s_{21}^{(k)} s_{22}^{(k)} \dots\dots\dots s_{2(M\text{-}1)}^{(k)} s_{2M}^{(k)} \\ \dots\dots\dots\dots\dots\dots\dots\dots \\ s_{M1}^{(k)} s_{M2}^{(k)} \dots\dots\dots s_{M(M\text{-}1)}^{(k)} s_{MM}^{(k)} \end{pmatrix} \tag{4.33}$$

$$\mathbf{S}^{(k)} = \begin{bmatrix} \begin{pmatrix} s_{1,1}^{(k)} & s_{1,2}^{(k)} \\ & \mathbf{S}_{1,1}^{(k)} \\ s_{2,1}^{(k)} & s_{2,2}^{(k)} \end{pmatrix} & \begin{pmatrix} s_{1,3}^{(k)} & s_{1,4}^{(k)} \\ & \mathbf{S}_{1,2}^{(k)} \\ s_{2,3}^{(k)} & s_{2,4}^{(k)} \end{pmatrix} & \begin{pmatrix} \dots\dots \\ \dots\dots \\ \dots\dots \\ \dots\dots \end{pmatrix} & \begin{pmatrix} s_{1,M-1}^{(k)} & s_{1,M}^{(k)} \\ & \mathbf{S}_{1,\frac{M}{2}}^{(k)} \\ s_{2,M-1}^{(k)} & s_{2,M}^{(k)} \end{pmatrix} \\ \begin{pmatrix} s_{3,1}^{(k)} & s_{3,2}^{(k)} \\ & \mathbf{S}_{2,1}^{(k)} \\ s_{4,1}^{(k)} & s_{4,2}^{(k)} \end{pmatrix} & \begin{pmatrix} s_{3,3}^{(k)} & s_{3,4}^{(k)} \\ & \mathbf{S}_{2,2}^{(k)} \\ s_{4,3}^{(k)} & s_{4,4}^{(k)} \end{pmatrix} & \begin{pmatrix} \dots\dots \\ \dots\dots \\ \dots\dots \\ \dots\dots \end{pmatrix} & \begin{pmatrix} s_{3,M-1}^{(k)} & s_{3,M}^{(k)} \\ & \mathbf{S}_{2,\frac{M}{2}}^{(k)} \\ s_{4,M-1}^{(k)} & s_{4,M}^{(k)} \end{pmatrix} \\ \dots\dots & \dots\dots & \dots\dots & \dots\dots \\ \begin{pmatrix} s_{M-1,1}^{(k)} & s_{M-1,2}^{(k)} \\ & \mathbf{S}_{\frac{M}{2},1}^{k} \\ s_{M,1}^{(k)} & s_{M,2}^{(k)} \end{pmatrix} & \begin{pmatrix} s_{M-1,3}^{(k)} & s_{M-1,4}^{(k)} \\ & \mathbf{S}_{\frac{M}{2},2}^{(k)} \\ s_{M,3}^{(k)} & s_{M,4}^{(k)} \end{pmatrix} & \begin{pmatrix} \dots\dots \\ \dots\dots \\ \dots\dots \\ \dots\dots \end{pmatrix} & \begin{pmatrix} s_{M-1,M-1}^{(k)} & s_{M-1,M}^{(k)} \\ & \mathbf{S}_{\frac{M}{2},\frac{M}{2}}^{(k)} \\ s_{M,M-1}^{(k)} & s_{M,M}^{(k)} \end{pmatrix} \end{bmatrix} \tag{4.34}$$

$$\mathbf{S}_{ij}^{(k)} = \begin{bmatrix} s_{2l-1,2m-1}^{(k)} s_{2l-1,2m}^{(k)} \\ s_{2l,2m-1}^{(k)} \; s_{2l,2m}^{(k)} \end{bmatrix} i,j,=1,\dots,\frac{M}{2}; l=i; m=j \tag{4.35}$$

donde el superíndice k hace alusión a la iteración correspondiente.

En el conjunto total de submatrices de 2×2 se establece una clasificación de éstas, acorde a la posición que ocupan dentro de la matriz $\mathbf{S}^{(k)}$ completa: se denomina como submatrices diagonales, aquellas en las que $i = j$ y, como no diagonales a las que $i \neq j$. Esta distinción es importante, ya que solamente las submatrices diagonales son simétricas, por lo que únicamente en éstas se podrá conseguir $s_{2l-1,2m}^{(k)} = s_{2l,2m-1}^{(k)} = 0$ en cada iteración.

Ahora cada submatriz de 2×2 elementos, debe ejecutar la multiplicación por las matrices de rotación (4.36).

$$\mathbf{S}_{i,j}^{(k+1)} = \mathbf{R}\left(\alpha_{i,i}^{(k)}\right)^{T} \cdot \mathbf{S}_{i,j}^{(k)} \cdot \mathbf{R}\left(\alpha_{j,j}^{(k)}\right) \quad i,j = 1,2,...,\frac{M}{2} \qquad k = 1,...,M\log M \tag{4.36}$$

donde $\mathbf{R}\left(\alpha^{(k)}\right) \in \Re^{2\times 2}$ es, de nuevo, la matriz de rotación siendo su valor el mostrado en (4.37).

$$\mathbf{R}\left(\alpha^{(k)}\right) = \begin{pmatrix} \cos(\alpha^{(k)}) & \sin(\alpha^{(k)}) \\ -\sin(\alpha^{(k)}) & \cos(\alpha^{(k)}) \end{pmatrix} \tag{4.37}$$

Se observa en (4.36), como la matriz $\mathbf{R}$ posee diferentes ángulos de rotación $\left(\alpha_{i,i}, \alpha_{j,j}\right)$, diferenciándose así esta expresión de la expuesta en (4.28). Esto es debido a que ahora, al tener dos tipos de submatrices, simétricas (diagonal) y no simétricas (no diagonal), en función del tipo de submatriz el ángulo de $\mathbf{R}$ variará. Concretamente, si la submatriz es diagonal ($i = j$), con idea de eliminar los elementos de fuera de su diagonal, entonces $\alpha_{i,i} = \alpha_{j,j}$. Por tanto, para lograr esta última condición, el valor de $\alpha_{i,i}$ necesario debe ser calculado previamente por cada submatriz diagonal acorde a la expresión (4.38).

$$tg(2\alpha_{i,i}^{(k)}) = \frac{2s_{2l-1,2m}^{(k)}}{s_{2l,2m}^{(k)} - s_{2l-1,2m-1}^{(k)}} \tag{4.38}$$

Tal y como se ha expuesto anteriormente, las submatrices no diagonales no poseen la condición de simetría, así independientemente del valor de ángulo que se tuviera, no serían capaces de eliminar ningún término. Por esa razón, son las submatrices diagonales las encargadas de realizar la anulación de términos. Los ángulos $\alpha_{i,i}^{(k)}$ y $\alpha_{j,j}^{(k)}$ que necesitan las submatrices no diagonales son calculados por cada submatriz diagonal adyacente. Se entiende como adyacente a la submatriz diagonal asociada a la misma fila (i) y a la misma columna (j), que la submatriz no diagonal. Por tanto, cada submatriz no diagonal posee dos matrices adyacentes.

Una vez que cada submatriz ha realizado la doble rotación o multiplicación de (4.36) con sus ángulos correspondientes, los resultados parciales obtenidos en cada submatriz se reordenan internamente y se transfirieren adecuadamente entre las distintas submatrices con el objetivo de situar en las submatrices diagonales los elementos a anular. Debido a que las submatrices diagonales son las encargadas de anular elementos, las matrices no diagonales deben transferir a éstas sus resultados parciales. La nueva matriz obtenida sigue teniendo propiedades de simetría. Todo este proceso se resume de forma gráfica en la Figura 4.16. Tras h repeticiones de todo este proceso, donde $h = M\log M$, el resultado final es una matriz diagonalizada

[Brent, 1985]. El valor de h se obtiene de forma empírica para una arquitectura sistólica y con datos en coma flotante, considerando un error en los elementos de fuera de la diagonal de $\mathbf{S}^{(h)} \in \Re^{M\times M}$ inferior a 10^{-12} [Brent, 1985].

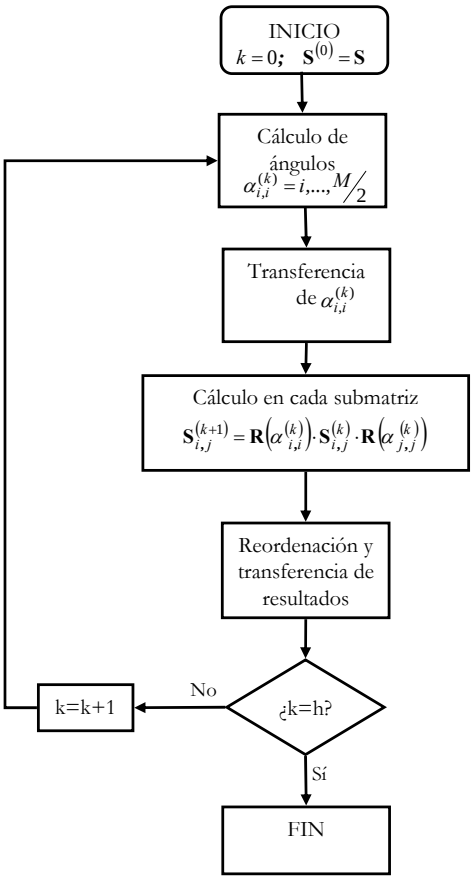


Figura 4.16. Diagrama de flujo del funcionamiento para el cálculo de autovalores mediante submatrices de 2x2 elementos basado en el método de Jacobi.

4.2.3.4. Cálculo de autovectores con el método de Jacobi, mediante descomposición en matrices de 2x2 elementos.

Acorde a la descomposición en submatrices de 2×2 elementos que se ha realizado en el cálculo de autovalores, la forma de determinar ahora los autovectores también difiere de la presentada en el apartado 4.2.3.2 para matrices completas. La obtención de la matriz de autovectores ($\mathbf{V}\in\Re^{M\times M}$) asociada a los autovalores de una matriz $\mathbf{S}\in\Re^{M\times M}$, también se basa en la descomposición de una matriz completa en submatrices de 2×2 elementos ($\mathbf{V}_{i,j}\in\Re^{2\times 2}$) (4.39). En este caso, la matriz inicial a dividir en submatrices es una matriz identidad ($\mathbf{V}^{(0)}=\mathbf{I}\in\Re^{M\times M}$), la cual también se descompone en submatrices de 2×2 elementos $\mathbf{V}_{i,j}^{(k)}\in\Re^{2\times 2}$ donde en cada iteración se actualiza su valor según la expresión (4.40) [Brent, 1985]. Como la matriz de autovectores es unitaria se parte de una matriz de este tipo como es el caso de la

matriz identidad, y sobre ésta se aplican sucesivas rotaciones hasta encontrar los autovectores asociados. En este caso no hay distinción entre submatrices diagonales y no diagonales, ya que una vez realizada la primera rotación ninguna submatriz es simétrica, por lo que sólo se establece un tipo de submatriz.

$$\mathbf{V}^{(k)} = \begin{bmatrix} \underset{\mathbf{V}_{1,1}^{(k)}}{\begin{pmatrix} v_{1,1}^{(k)} & v_{1,2}^{(k)} \\ v_{2,1}^{(k)} & v_{2,2}^{(k)} \end{pmatrix}} & \underset{\mathbf{V}_{1,2}^{(k)}}{\begin{pmatrix} v_{1,3}^{(k)} & v_{1,4}^{(k)} \\ v_{2,3}^{(k)} & v_{2,4}^{(k)} \end{pmatrix}} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \underset{\mathbf{V}_{1,\frac{M}{2}}^{(k)}}{\begin{pmatrix} v_{1,M-1}^{(k)} & v_{1,M}^{(k)} \\ v_{2,M-1}^{(k)} & v_{2,M}^{(k)} \end{pmatrix}} \\ \underset{\mathbf{V}_{2,1}^{(k)}}{\begin{pmatrix} v_{3,1}^{(k)} & v_{3,2}^{(k)} \\ v_{4,1}^{(k)} & v_{4,2}^{(k)} \end{pmatrix}} & \underset{\mathbf{V}_{2,2}^{(k)}}{\begin{pmatrix} v_{3,3}^{(k)} & v_{3,4}^{(k)} \\ v_{4,3}^{(k)} & v_{4,4}^{(k)} \end{pmatrix}} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \underset{\mathbf{V}_{2,\frac{M}{2}}^{(k)}}{\begin{pmatrix} v_{3,M-1}^{(k)} & v_{3,M}^{(k)} \\ v_{4,M-1}^{(k)} & v_{4,M}^{(k)} \end{pmatrix}} \\ \cdots & \cdots & \cdots & \cdots \\ \underset{\mathbf{V}_{\frac{M}{2},1}^{k}}{\begin{pmatrix} v_{M-1,1}^{(k)} & v_{M-1,2}^{(k)} \\ v_{M,1}^{(k)} & v_{M,2}^{(k)} \end{pmatrix}} & \underset{\mathbf{V}_{\frac{M}{2},2}^{(k)}}{\begin{pmatrix} v_{M-1,3}^{(k)} & v_{M-1,4}^{(k)} \\ v_{M,3}^{(k)} & v_{M,4}^{(k)} \end{pmatrix}} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \underset{\mathbf{V}_{\frac{M}{2},\frac{M}{2}}^{(k)}}{\begin{pmatrix} v_{M-1,M-1}^{(k)} & v_{M-1,M}^{(k)} \\ v_{M,M-1}^{(k)} & v_{M,M}^{(k)} \end{pmatrix}} \end{bmatrix} \quad (4.39)$$

$$\mathbf{V}_{i,j}^{(k+1)} = \mathbf{V}_{i,j}^{(k)} \cdot \mathbf{R}\left(\alpha_{j,j}^{(k)}\right) \qquad i,j = 1,2,\ldots,\frac{M}{2} \qquad k = 1,\ldots,M\log M \qquad (4.40)$$

El ángulo de rotación necesario para las submatrices de autovectores ($\alpha_{j,j}^{(k)}$) es generado en la etapa de autovalores, concretamente por las submatrices de tipo diagonal $j = i$. Es decir, cada columna de submatrices de autovectores posee el mismo ángulo, siendo éste generado por la submatriz diagonal correspondiente en la etapa de autovalores.

La ventaja del cálculo de autovectores a través de Jacobi es la obtención simultánea de autovalores y autovectores, una vez calculados los ángulos de rotación.

Cuando se dispone del $\alpha_{j,j}^{(k)}$ correspondiente en cada submatriz de autovectores la metodología de funcionamiento es análoga a la presentada en los autovalores. Es decir el siguiente paso es el cálculo de (4.40). Una vez finalizado este cálculo se deben reordenar internamente los coeficientes de cada submatriz con el mismo criterio que los autovalores y a continuación, cada submatriz transfiere sus resultados parciales al resto. Trascurrido el mismo número de iteraciones que en los autovalores, los elementos que forman la matriz de autovectores están contenidos en cada submatriz de 2×2 tal y como se expone en la Figura 4.21. Todo este proceso se muestra de forma resumida en la Figura 4.17.

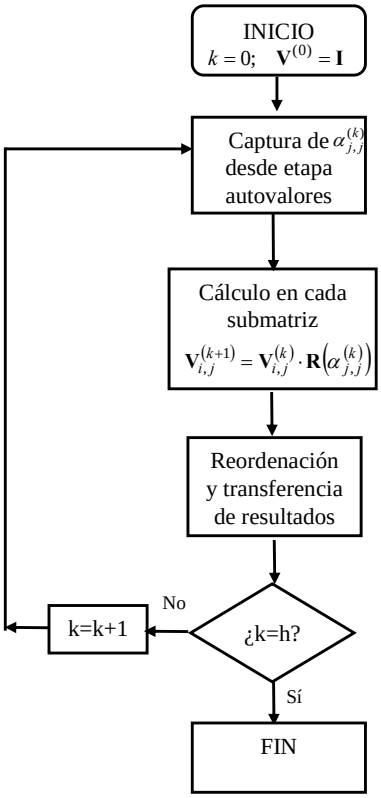


Figura 4.17. Diagrama de flujo del funcionamiento para el cálculo de autovectores mediante submatrices de 2x2 elementos basado en el método de Jacobi.

4.2.3.5. Arquitectura clásica asociada al método de Jacobi.

Según la exposición realizada anteriormente, el método de Jacobi tiene una estructura modular que hace muy recomendable su implementación en FPGAs. La estructura y características internas de éstas permiten la ejecución en paralelo de cada submatriz, por lo que éstas son idóneas para la implementación de este algoritmo.

Se presentarán dos arquitecturas diferentes, una para el cálculo de autovalores y otra para autovectores, y aunque existe dependencia de datos entre ambas se pueden realizar numerosas tareas en paralelo acelerando considerablemente el tiempo de ejecución con respecto a arquitecturas secuenciales.

4.2.3.5.1. Arquitectura para autovalores.

A la hora de implementar el flujograma expuesto en la Figura 4.16 se puede observar que hay básicamente cuatro operaciones, independientemente de la plataforma hardware escogida para ejecutar el algoritmo:

- Cálculo de ángulos de rotación (expresión (4.38)).
- Transferencia de ángulos desde las submatrices diagonales a las no diagonales.
- Operación de doble multiplicación o rotación en cada submatriz (expresión (4.36)).

- Reordenamiento de datos parciales en cada submatriz y transferencia de estos a las submatrices contiguas.

Desde el punto de vista de los bloques necesarios para realizar la implementación del algoritmo, hay dos tipos: los que están asociados a las submatrices diagonales y los asociados a las no diagonales. Los primeros realizan los cuatro ítems expuestos anteriormente y los segundos sólo los dos últimos. Existen un total de $M/2$ elementos encargados de realizar operaciones de submatrices diagonales y el resto $\left(\left(M/2\right)^2 - M/2\right)$ para no diagonales, siendo $M \times M$ el tamaño de la matriz.

Los trabajos desarrollados por Brent y Luk [Brent, 1985], [Brent, 1983] muestran una arquitectura de $M/2 \times M/2$ procesadores, donde cada uno trabaja con matrices de tamaño 2×2 ($\mathbf{S}_{i,j}^{(k)} \in \Re^{2\times 2}$), por lo que se hace viable la utilización de las propuestas de los citados trabajos en una arquitectura sistólica basada en FPGAs. Existen básicamente dos elementos en esta arquitectura: Procesadores Diagonales (PDs) (manejan las submatrices $\mathbf{S}_{i,i}^{(k)}$) y Procesadores No Diagonales (PNDs) (manejan las submatrices $\mathbf{S}_{i,j}^{(k)}$). Por ello, se denota como $P_{i,j}$ al procesador encargado de gestionar la submatriz $\mathbf{S}_{i,j}^{(k)}$. En la Figura 4.18 se representa, de forma genérica, la arquitectura sistólica necesaria para realizar el cálculo de autovalores de una matriz de tamaño $M \times M$. Se observa en la citada figura, como se deben realizar las transferencias de resultados parciales entre los distintos procesadores ($P_{i,j}$). Cabe destacar en los procesadores ubicados en las cuatro esquinas, la existencia de una auto-realimentación de resultados parciales. Posteriormente se detallará el mecanismo de transferencia de datos (apartado 4.2.3.5.3).

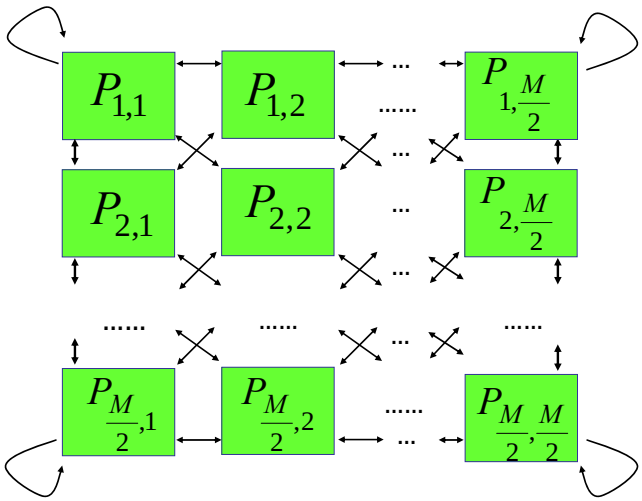


Figura 4.18. Arquitectura de $M/2 \times M/2$ elementos de procesado para la obtención de autovalores.

A continuación se describen las diferentes tareas que ambos procesadores realizan:

1. Inicialmente, cada procesador $P_{i,j}$ debe ser cargado con los 4 elementos de su $\mathbf{S}_{i,j}^{(k)} \in \Re^{2\times 2}$ correspondiente según (4.34).

2. Una vez que cada procesador dispone de sus datos iniciales, el siguiente paso se realiza únicamente en los PDs. Concretamente cada uno de ellos calcula el ángulo de rotación $\alpha_{i,i}^{(k)}$ que él mismo necesita, así como los PNDs pertenecientes a su misma fila y su misma columna, y que debe ser propagado posteriormente (4.38). Para el cálculo del ángulo $\alpha_{i,i}^{(k)}$ en cada PD ($P_{i,i}$) se utilizan los elementos de la matriz de 2×2 que maneja ($\mathbf{S}_{i,i}^{(k)} \in \Re^{2\times 2}$) (4.34). Este cálculo se realiza de manera independiente y paralela en los $M/2$ PDs. En [Cavallaro, 1988] se propone el empleo del algoritmo CORDIC (*Coordinate Rotation Digital Computer*) [Volder, 1959] [Walther, 1971] para resolver el ángulo de rotación de (4.38) en cada PD. En el apartado 4.2.3.6 se justifica el empleo de CORDIC para realizar el cálculo del ángulo de rotación.

3. Una vez calculado el ángulo correspondiente en cada PD, estos ángulos se propagan hacia el resto de PNDs asociados a su misma fila y su misma columna. Tal y como se expone en (4.36), cada procesador necesita dos ángulos ($\alpha_{i,i}, \alpha_{j,j}$). En el caso de los PDs, el ángulo para realizar las rotaciones es el mismo ($\alpha_{i,i} = \alpha_{j,j}$) siendo este valor el ángulo que el propio procesador ha calculado previamente. Por su parte, en los PNDs el ángulo $\alpha_{i,i}$ es enviado desde el procesador diagonal asociado a su fila y el ángulo $\alpha_{j,j}$, desde el procesador diagonal asociado a su columna. En la Figura 4.19 se muestra la forma de propagación de los ángulos desde los PDs a los PNDs correspondientes.

4. Tras las fases anteriores, todos los procesadores poseen los ángulos de rotación necesarios para poder realizar en paralelo la doble multiplicación (rotación) expuesta en (4.36). De nuevo se ha optado por la implementación de arquitecturas basadas en CORDIC para realizar esta operación [Cavallaro, 1988]. También en el apartado 4.2.3.6 se justifica matemáticamente la idoneidad de esta alternativa.

5. Debido a que los PDs manejan siempre matrices simétricas, éstos eliminan los valores de fuera de su diagonal. Como en cada PNDs se implementa la expresión (4.36), sus valores son modificados pero no se consigue anular ningún elemento tras la doble rotación. Por ello, se realiza un trasiego o propagación de resultados entre los distintos procesadores según se muestra en la Figura 4.19, donde los elementos a,b,c,d de cada procesador se

corresponden con los elementos $s^{(k)}_{2l-1,2m-1}, s^{(k)}_{2l-1,2m}, s^{(k)}_{2l,2m-1}$ $s^{(k)}_{2l,2m}$ de (4.35). En el apartado 4.2.3.5.3 se expone la forma de propagar los datos obtenidos en cada iteración. Estos nuevos datos son los que cada procesador debe manejar con idea de empezar una nueva iteración, con lo que en este punto se puede decir que ha finalizado la primera iteración. De nuevo, los PDs vuelven a calcular los ángulos y después de ello los propagan al resto de procesadores, se realiza una rotación en todos los procesadores y queda finalizada la segunda iteración.

6. Para conseguir diagonalizar la matriz $\mathbf{S}$, según [Brent, 1985] son necesarias aproximadamente h iteraciones. Este valor es puramente empírico y, en función del tipo de datos y tamaño de matrices, en ocasiones es necesario aumentar el valor estimado en [Brent, 1985] para conseguir la convergencia.

Una vez finalizadas todas las iteraciones, las diagonales de cada una de las submatrices que contienen los PDs, contendrán cada uno de los autovalores. El resto de elementos, tanto de los PDs como de los PNDs, serán cero o próximos a cero.

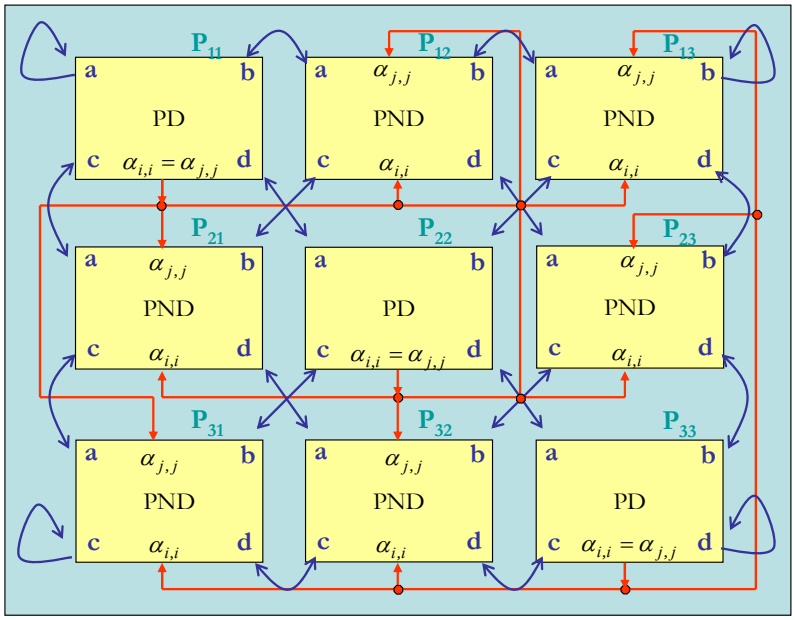


Figura 4.19. *Array* de procesadores para el cálculo de autovalores, detallada la propagación de ángulos y datos particularizado para una matriz inicial de 6x6.

4.2.3.5.2. Arquitectura para autovectores.

Como se comentó anteriormente, la fase de cálculo de autovectores puede ejecutarse de manera simultánea a la de autovalores. La única dependencia entre ambas, es el ángulo $\alpha_{i,i}$ que los autovectores deben recibir de los PDs.

A la hora de implementar esta nueva etapa, se propone un nuevo *array* de procesadores de tamaño $M/2 \times M/2$ [Cavallaro, 1988] [Cavallaro, 1986]. En esta arquitectura únicamente hay un tipo de procesadores que se denomina como PVs, capaces de realizar la rotación expuesta en (4.40).

La metodología de funcionamiento vuelve a ser análoga a la explicada en el caso de los autovalores, utilizando el mismo método de iteraciones para conseguir los autovectores asociados a los autovalores. La principal diferencia con los autovalores es que aquí solamente se realiza una rotación (rotación simple) y que el ángulo de rotación es el creado por los PDs.

El funcionamiento resumido de esta etapa es:

1. Cada PV es cargado inicialmente con una matriz de 2×2 elementos $\mathbf{V}_{i,j} \in \Re^{2\times2}$, siendo en este caso la matriz inicial una matriz identidad de tamaño igual al de la matriz inicial de partida $\mathbf{S} \in \Re^{M\times M}$.
2. El ángulo que cada PV necesita es enviado desde los PDs de la arquitectura de autovalores. Concretamente cada PV maneja el ángulo generado en el procesador de diagonal asociado a su misma columna. Por lo que el siguiente paso es la recepción en cada procesador de autovectores del ángulo de rotación ($\alpha_{j,j}$).
3. Una vez recibido el ángulo en todos los PVs, el siguiente paso es la ejecución de la rotación expuesta en (4.40). También esta rotación se puede ejecutar en base al algoritmo CORDIC, por lo que se implementa un módulo para solventar dicha rotación.
4. Una vez realizada la rotación simple por esta etapa, el siguiente paso es la propagación de los resultados parciales entre los diferentes PVs, de manera idéntica a la de autovalores (ver Figura 4.20).
5. El número de iteraciones necesarias para alcanzar la convergencia es el mismo que en los autovalores.

En la Figura 4.21 se muestra la distribución de cada uno de los autovectores finales, en cada PV una vez que el módulo encargado de obtener los autovectores ha finalizado. Además, cada uno de estos autovectores está ubicado en la misma posición que su autovalor correspondiente. Es decir, si el autovalor λ_i está situado en la posición *i-ésima* de la diagonal de autovalores, su autovector asociado ($\vec{\mathbf{v}}_i$) está en la *i-ésima* columna de la matriz de autovectores.

A la vista del funcionamiento de esta etapa, una vez recibido desde la fase de autovalores el ángulo necesario para la rotación, la arquitectura de autovectores actúa de manera independiente a la de autovalores, por lo que se puede paralelizar su funcionamiento. También, debido al hecho de que las operaciones que realizan son más sencillas, su tiempo de ejecución es inferior a la de autovalores, por lo que no demora al sistema total.

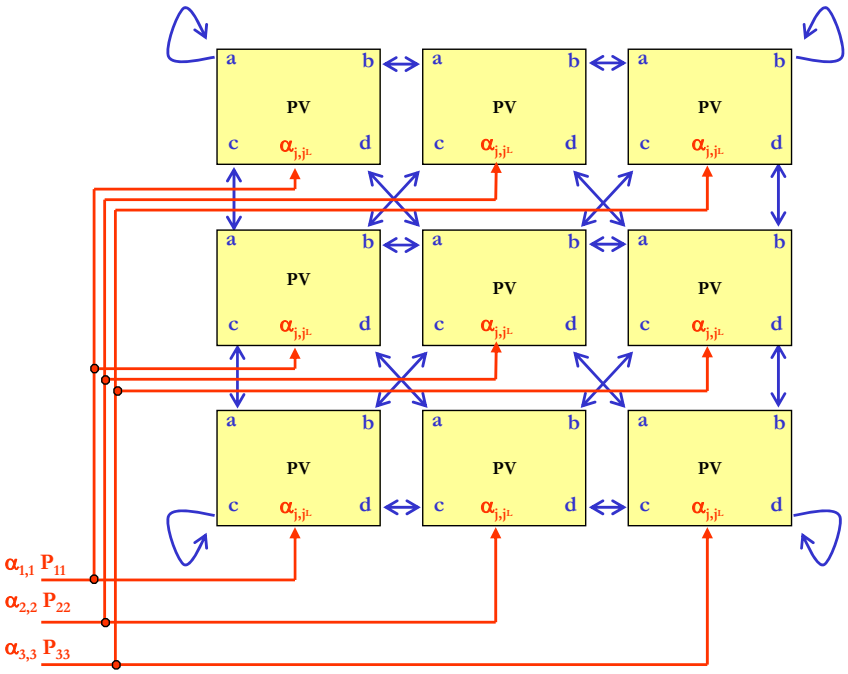


Figura 4.20. *Array* de procesadores para el cálculo de autovectores, detallada la propagación de ángulos y datos particularizado para una matriz inicial de 6x6.

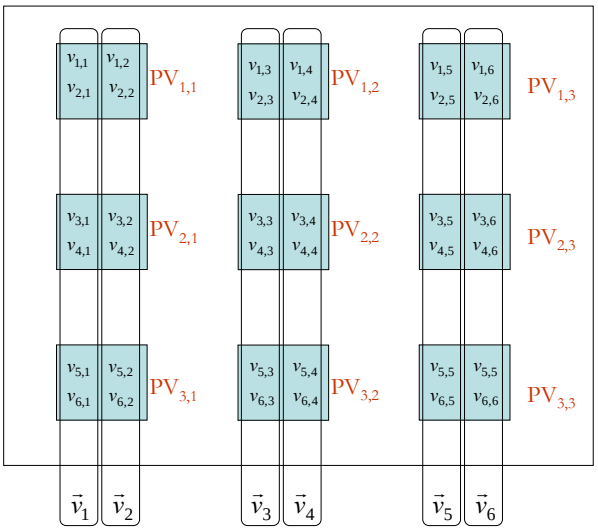


Figura 4.21. Autovectores finales de una matriz de 6x6, generados utilizando el método de Jacobi mediante descomposición de matrices de 2x2.

4.2.3.5.3. Propagación de datos y ángulos, entre procesadores.

En este apartado se presenta la forma en la cual se propagan en cada iteración los ángulos de rotación y los resultados temporales de cada procesador hacia sus procesadores contiguos, tanto para los autovalores como autovectores (ver Figura 4.19 y Figura 4.20). Esta manera de reordenar y propagar los resultados, fue propuesta en [Brent, 1983] como adaptación del método de Jacobi al procesamiento en un *array* sistólico de matrices de 2×2 elementos.

Comenzando con la propagación de ángulos, cada PD debe enviar el ángulo que calcula en cada iteración a:

- Los PNDs que estén en su misma fila y en su misma columna (ver Figura 4.22).
- Los PVs que ocupen la misma columna que ellos (ver Figura 4.23).

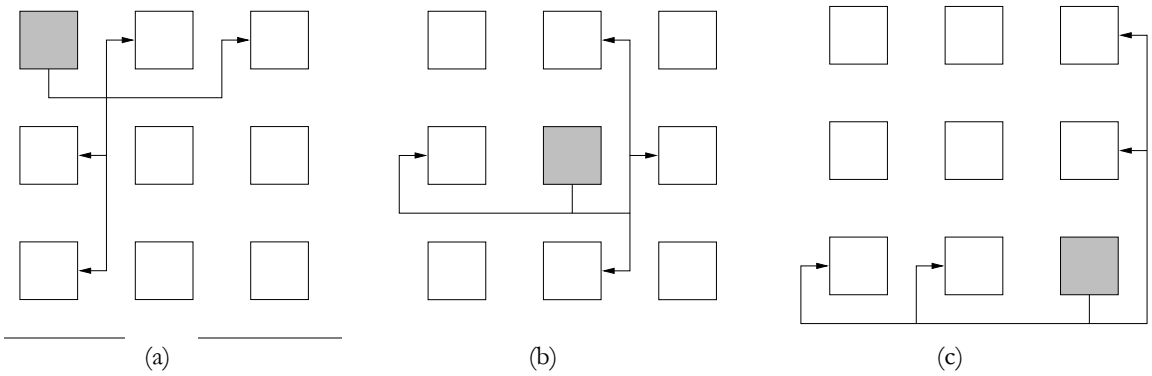


Figura 4.22. Propagación de los ángulos generados en los PDs de una matriz de 6x6 elementos, a los PNDs pertenecientes a su misma fila y columna, (a) Procesador (1,1), (b) Procesador (2,2) y (c) Procesador (3,3).

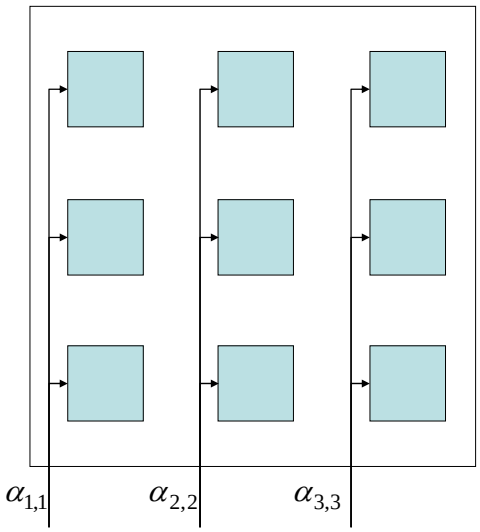


Figura 4.23. Ángulos recibidos por cada PV desde los PDs del módulo de autovalores para una matriz de 6x6 elementos.

Una vez enviado el ángulo correspondiente a cada procesador, tanto el módulo de autovectores como el de autovalores realizan las rotaciones pertinentes, acorde a (4.40) (autovectores) y (4.36) (autovalores). En este momento y como paso previo a la propagación de datos a los procesadores contiguos, cada procesador debe reordenar la nueva matriz de 2×2 elementos obtenida tras la rotación [Brent, 1983]. La Figura 4.24.a. muestra el método de esta primera reordenación interna de datos que debe realizar cada procesador. Esta recolocación es idéntica para todos los procesadores, pudiéndose distinguir cuatro tipos de reordenaciones:

- Primer procesador (esquina superior izquierda): este procesador no tiene que recolocar internamente sus datos.
- Procesadores pertenecientes a la primera fila: los cuatro elementos que forman la matriz de 2×2 deben intercambiarse sus valores horizontalmente.
- Procesadores pertenecientes a la primera columna: en este caso, el intercambio de valores es verticalmente.
- Resto de procesadores: intercambian internamente sus datos diagonalmente.

Este reordenamiento interno junto con la propagación de datos entre procesadores viene justificado por:

- La anulación de los datos de fuera de la diagonal en el caso de los autovalores. Los PNDs deben enviar a los PDs sus datos ordenadamente para que éstos se encarguen de converger sus datos a cero (anulación). Por su parte los PDs también deben reordenar sus datos para propagar los elementos anulados a los PNDs.
- En el caso de autovectores, mediante la reordenación interna y propagación de datos, además de obtener los autovectores correspondientes se consigue posicionar cada autovector en la misma posición que su autovalor correspondiente.

Con respecto a la propagación de datos entre procesadores, tras haber sido reordenados internamente, en la Figura 4.24.b se muestra cómo deben intercambiarse los datos. Cabe en este caso destacar cuatro tipos diferentes de propagación:

- Procesadores ubicados en las esquinas: debido a su posición, cada uno de estos cuatro procesadores poseen un dato que no se puede compartir con otro procesador, por lo que ese dato nunca se transmite.
- Procesadores ubicados en la primera y última fila.
- Procesadores ubicados en la primera y última columna.
- Resto de procesadores.

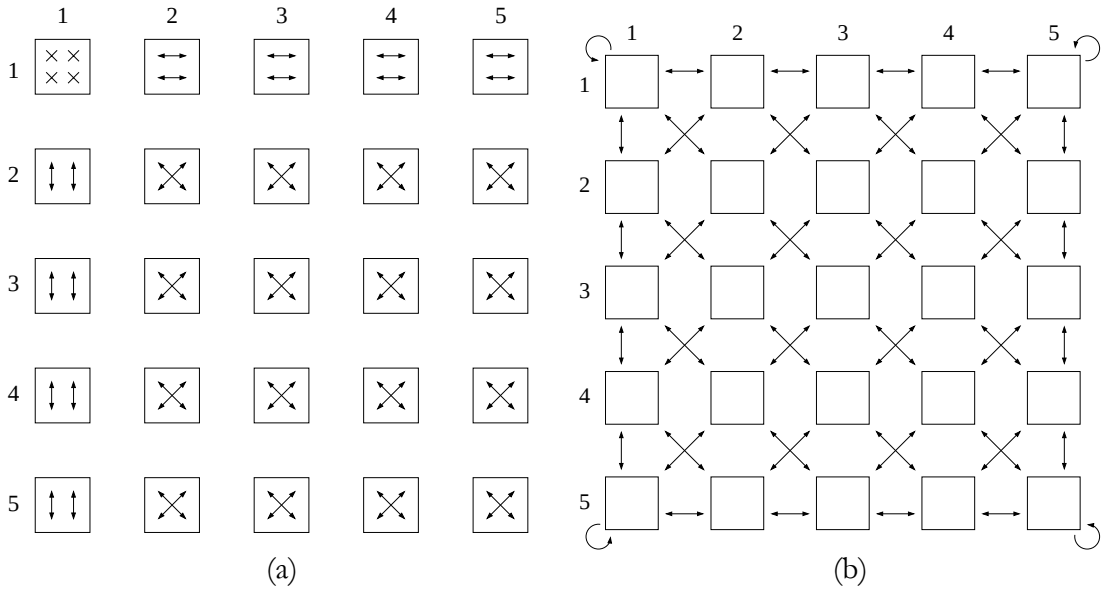


Figura 4.24. Recolocación interna de datos dentro de cada procesador (a) e intercambio de datos entre procesadores (b), para una matriz de 10x10 elementos.

4.2.3.6. Aplicación de CORDIC al cálculo de autovalores y autovectores mediante el método de Jacobi.

El algoritmo CORDIC basa su funcionamiento en la rotación de un vector de dos dimensiones en un sistema de coordenadas circular, lineal o hiperbólico [Walther, 1971]. Esta propiedad es la que justifica el empleo de este algoritmo dentro del método de Jacobi. En este punto se analiza el empleo de CORDIC en el cálculo de los ángulos de rotación (expresión (4.38)) y en el de la doble rotación, en el caso de autovalores (expresión (4.36)), y el de la rotación simple en el caso de autovectores (expresión (4.40)).

4.2.3.6.1. Cálculo de ángulo de rotación.

Como se ha explicado en apartados anteriores, cada PD de la etapa de autovalores debe calcular en cada iteración en base a los elementos que forman su matriz de 2×2 elementos, el ángulo necesario para conseguir diagonalizar dicha matriz (ver (4.38)). Por tanto, para el cálculo de dicho ángulo ($\alpha_{i,i}^{(k)}$) es necesario realizar una operación arco tangente. El algoritmo CORDIC resuelve esta operación funcionando en modo vectorización y con coordenadas circulares. En definitiva, el cálculo de $\alpha_{i,i}^{(k)}$ no es más que el ángulo resultante de la rotación de un vector con coordenadas $\left[s_{2l,2m}^{(k)} - s_{2l-1,2m-1}^{(k)}, 2s_{2l-1,2m}^{(k)}\right]$ sobre el eje de abcisas (ver Figura 4.25). En base a estos datos de partida, en la Figura 4.26 se propone un módulo hardware que permite resolver esta operación.

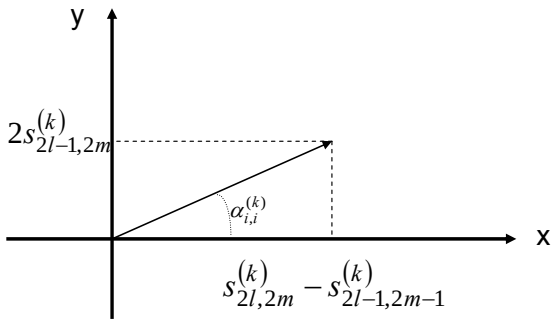


Figura 4.25. Rotación del vector $\left[s_{2l,2m}^{(k)} - s_{2l-1,2m-1}^{(k)}, 2s_{2l-1,2m}^{(k)}\right]$, para obtener el ángulo de rotación en cada PD.

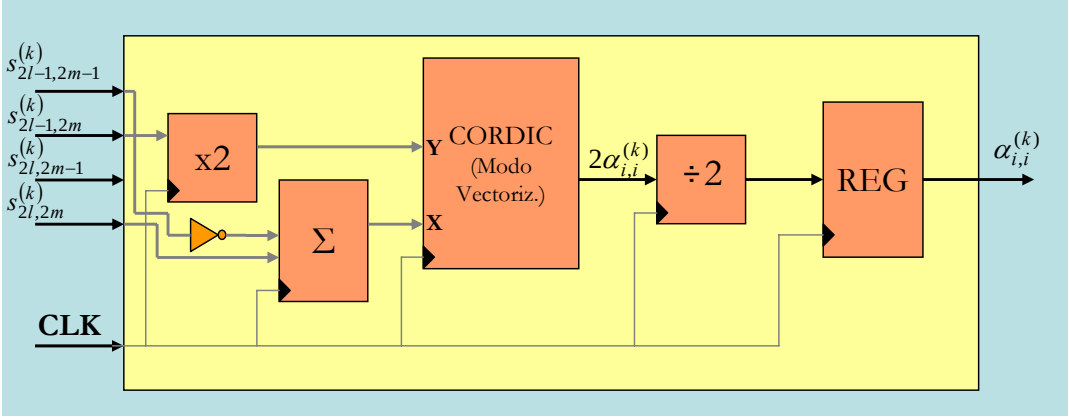


Figura 4.26. Bloque de cálculo del ángulo de rotación para el módulo de autovalores mediante el método de Jacobi.

4.2.3.6.2. Doble rotación en el cálculo de autovalores.

Con respecto a la doble rotación que deben realizar por todos los procesadores del módulo de autovalores (ver (4.36)), también mediante el uso de CORDIC se pueden ejecutar dichas rotaciones. Para justificar esto, en (4.41) se muestra la expresión (4.36) en función de los elementos de un procesador cualquiera y en función de sus ángulos de rotación $\alpha_{i,i}, \alpha_{j,j}$.

$$\mathbf{S}_{i,j}^{(k+1)} = \mathbf{R}\left(\alpha_{i,i}^{(k)}\right)^T \cdot \mathbf{S}_{i,j}^{(k)} \cdot \mathbf{R}\left(\alpha_{j,j}^{(k)}\right)$$

$$\begin{bmatrix} s_{2l-1,2m-1}^{(k+1)} & s_{2l-1,2m}^{(k+1)} \\ s_{2l,2m-1}^{(k+1)} & s_{2l,2m}^{(k+1)} \end{bmatrix} = \begin{bmatrix} \cos(\alpha_{i,i}^{(k)}) & \sin(\alpha_{i,i}^{(k)}) \\ -\sin(\alpha_{i,i}^{(k)}) & \cos(\alpha_{i,i}^{(k)}) \end{bmatrix}^T \cdot \begin{bmatrix} s_{2l-1,2m-1}^{(k)} & s_{2l-1,2m}^{(k)} \\ s_{2l,2m-1}^{(k)} & s_{2l,2m}^{(k)} \end{bmatrix} \cdot \begin{bmatrix} \cos(\alpha_{j,j}^{(k)}) & \sin(\alpha_{j,j}^{(k)}) \\ -\sin(\alpha_{j,j}^{(k)}) & \cos(\alpha_{j,j}^{(k)}) \end{bmatrix} \quad (4.41)$$

Si al primer producto de la expresión (4.41) se denomina como $\mathbf{Q}_{i,j}^{(k)}$ (ver (4.42)), ahora se puede expresar (4.41) tal y como se muestra en (4.43).

$$\mathbf{Q}_{ij}^{(k)} = \begin{bmatrix} \cos(\alpha_{i,i}^{(k)}) \cdot s_{2l-1,2m-1}^{(k)} - \sin(\alpha_{i,i}^{(k)}) \cdot s_{2l,2m-1}^{(k)} & \cos(\alpha_{i,i}^{(k)}) \cdot s_{2l-1,2m}^{(k)} - \sin(\alpha_{i,i}^{(k)}) \cdot s_{2l,2m}^{(k)} \\ \sin(\alpha_{i,i}^{(k)}) \cdot s_{2l-1,2m-1}^{(k)} + \cos(\alpha_{i,i}^{(k)}) \cdot s_{2l,2m-1}^{(k)} & \sin(\alpha_{i,i}^{(k)}) \cdot s_{2l-1,2m}^{(k)} + \cos(\alpha_{i,i}^{(k)}) \cdot s_{2l,2m}^{(k)} \end{bmatrix} \quad (4.42)$$

$$\mathbf{S}_{ij}^{(k+1)} = \mathbf{Q}_{i,j}^{(k)} \cdot \begin{bmatrix} \cos(\alpha_{j,j}^{(k)}) \sin(\alpha_{j,j}^{(k)}) \\ -\sin(\alpha_{j,j}^{(k)}) \cos(\alpha_{j,j}^{(k)}) \end{bmatrix} \quad (4.43)$$

Analizando (4.42), se observa que dicha expresión equivale a la rotación del vector de coordenadas $\left[s_{2l-1,2m-1}^{(k)}, s_{2l,2m-1}^{(k)}\right]$ un ángulo $\alpha_{i,i}$ y a la rotación del vector $\left[s_{2l-1,2m}^{(k)}, s_{2l,2m}^{(k)}\right]$ también un ángulo $\alpha_{i,i}$. Por tanto, para realizar la rotación de cada vector se puede utilizar un módulo CORDIC con coordenadas circulares trabajando en modo rotación. Para ello, una posible arquitectura se muestra en la Figura 4.27, donde $s'^{(k)}_{2l-1,2m-1}$ y $s'^{(k)}_{2l,2m-1}$ son las componentes del vector $\left[s_{2l-1,2m-1}^{(k)}, s_{2l,2m-1}^{(k)}\right]$ rotado mientras que $s'^{(k)}_{2l-1,2m}$ y $s'^{(k)}_{2l,2m}$ son las del vector $\left[s_{2l-1,2m}^{(k)}, s_{2l,2m}^{(k)}\right]$ rotado. De esta forma, se completa la primera rotación.

Para realizar la segunda rotación de nuevo se emplea CORDIC. Así, si a cada uno de los elementos de $\mathbf{Q}_{i,j}^{(k)}$ (ver (4.42)) se le denomina como $\begin{bmatrix} s'^{(k)}_{2l-1,2m-1} & s'^{(k)}_{2l-1,2m} \\ s'^{(k)}_{2l,2m-1} & s'^{(k)}_{2l,2m} \end{bmatrix}$, y sobre ellos se aplica la segunda rotación con un ángulo $\alpha_{j,j}$, se obtiene la expresión (4.44).

$$\begin{aligned} \mathbf{S}_{ij}^{(k+1)} &= \begin{bmatrix} s_{2l-1,2m-1}^{(k+1)} s_{2l-1,2m}^{(k+1)} \\ s_{2l,2m-1}^{(k+1)} \quad s_{2l,2m}^{(k+1)} \end{bmatrix} = \begin{bmatrix} s'^{(k)}_{2l-1,2m-1} & s'^{(k)}_{2l-1,2m} \\ s'^{(k)}_{2l,2m-1} & s'^{(k)}_{2l,2m} \end{bmatrix} \cdot \begin{bmatrix} \cos(\alpha_{i,i}^{(k)}) \ \sin(\alpha_{i,i}^{(k)}) \\ -\sin(\alpha_{i,i}^{(k)}) \cos(\alpha_{i,i}^{(k)}) \end{bmatrix} = \\ &= \begin{bmatrix} s'^{(k)}_{2l-1,2m-1} \cos(\alpha_{i,i}^{(k)}) - s'^{(k)}_{2l-1,2m} \sin(\alpha_{i,i}^{(k)}) & s'^{(k)}_{2l-1,2m-1} \sin(\alpha_{i,i}^{(k)}) + s'^{(k)}_{2l-1,2m} \cos(\alpha_{i,i}^{(k)}) \\ s'^{(k)}_{2l,2m-1} \cos(\alpha_{i,i}^{(k)}) \ - s'^{(k)}_{2l,2m} \sin(\alpha_{i,i}^{(k)}) & s'^{(k)}_{2l,2m-1} \sin(\alpha_{i,i}^{(k)}) + s'^{(k)}_{2l,2m} \cos(\alpha_{i,i}^{(k)}) \end{bmatrix} \end{aligned} \quad (4.44)$$

En esta última expresión se puede de nuevo comprobar, como los resultados obtenidos equivalen de nuevo a una rotación de dos vectores. Así por una parte, se tiene el vector de coordenadas $\left[s'^{(k)}_{2l-1,2m-1}, s'^{(k)}_{2l-1,2m}\right]$ y por otra el de coordenadas $\left[s'^{(k)}_{2l,2m-1}, s'^{(k)}_{2l,2m}\right]$, rotando ambos con el mismo ángulo $\alpha_{j,j}$. Gracias a esto, de nuevo se pueden emplear dos módulos CORDIC de coordenadas circulares funcionando en modo rotación para conseguir dichas rotaciones. Así, en la Figura 4.28 se presenta el diseño de esta segunda rotación, junto con el empleado en la primera rotación de $\alpha_{i,i}$.

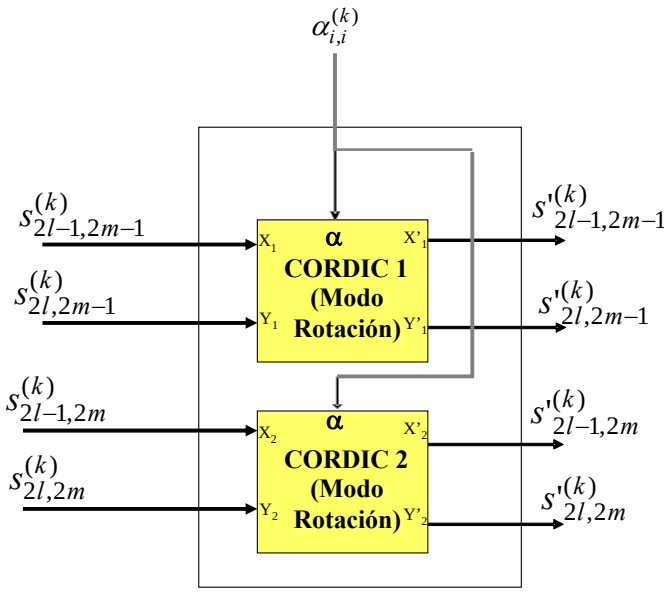


Figura 4.27. Arquitectura para realizar la primera rotación en cada procesador del módulo de autovalores, empleando módulos CORDIC.

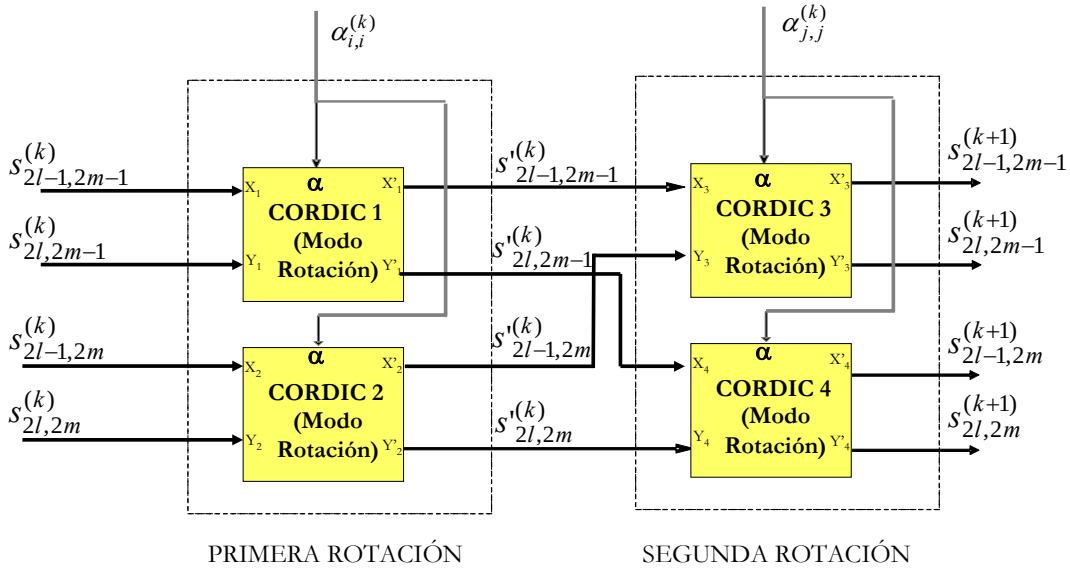


Figura 4.28. Sistema completo para realizar las dos rotaciones del módulo de autovalores, basados en CORDIC.

La arquitectura mostrada en la Figura 4.28, desde un punto de vista de velocidad, es la óptima. Sin embargo, la duplicidad de bloques CORDIC eleva en exceso el número de recursos internos. Una posible alternativa, que reduce a casi la mitad el número de recursos consumidos, es la mostrada en la Figura 4.29. En este caso, debido al empleo de multiplexores el sistema reduce levemente su velocidad de ejecución.

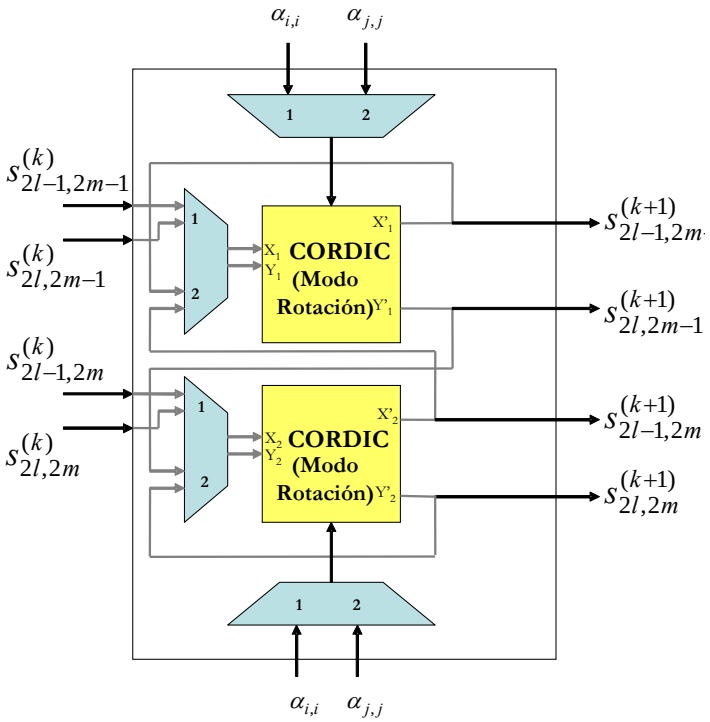


Figura 4.29. Sistema para realizar la doble rotación de autovalores, empleando únicamente dos módulos CORDIC.

4.2.3.6.3. Rotación simple en el cálculo de autovectores.

En el caso del cálculo de autovectores, todos los PVs realizan la misma operación (4.40). En este caso, se corresponde con la rotación de dos vectores un ángulo determinado $(\alpha_{j,j})$. Así, desarrollando la expresión (4.40) se obtiene (4.45).

$$\begin{bmatrix} v^{(k+1)}_{2l-1,2m-1} & v^{(k+1)}_{2l-1,2m} \\ v^{(k+1)}_{2l,2m-1} & v^{(k+1)}_{2l,2m} \end{bmatrix} = \begin{bmatrix} v^{(k)}_{2l-1,2m-1} & v^{(k)}_{2l-1,2m} \\ v^{(k)}_{2l,2m-1} & v^{(k)}_{2l,2m} \end{bmatrix} \cdot \begin{bmatrix} \cos(\alpha^{(k)}_{j,j}) & \sin(\alpha^{(k)}_{j,j}) \\ -\sin(\alpha^{(k)}_{j,j}) & \cos(\alpha^{(k)}_{j,j}) \end{bmatrix} \tag{4.45}$$

De nuevo, se observa como dicha expresión se corresponde con la rotación de un ángulo $\alpha_{j,j}$ sobre los vectores $\left[v^{(k)}_{2l-1,2m-1}, v^{(k)}_{2l-1,2m}\right]$ y $\left[v^{(k)}_{2l,2m-1}, v^{(k)}_{2l,2m}\right]$. Por tanto, análogamente al caso de las rotaciones de autovalores, una posible solución para realizar (4.45) es empleando dos CORDIC de coordenadas circulares en modo rotación, con una arquitectura como la mostrada en la Figura 4.27.

4.2.3.7. Arquitecturas propuestas basadas en CORDIC, para el cálculo de autovalores y autovectores.

Una vez justificado el uso de CORDIC para la obtención de los autovalores y autovectores de una matriz Hermítica, en esta tesis doctoral se proponen dos arquitecturas, ambas basadas en CORDIC y en los trabajos desarrollados en [Brent, 1983] [Brent, 1985] [Cavallaro, 1988]. La primera propuesta está compuesta por una arquitectura sistólica clásica de procesadores, una para el cálculo de autovalores y otra

para el de autovectores. Esta alternativa es sencilla de implementar en hardware, ya que únicamente se deben desarrollar tres modelos de procesadores (PD, PND, PV) y replicar estas unidades acorde a la propuesta de [Brent, 1983]. Sin embargo, esta solución emplea un número muy elevado de recursos por lo que su uso no está recomendado. La otra propuesta se basa en el funcionamiento de la arquitectura sistólica de procesadores empleando únicamente dos módulos CORDIC. Esta opción consume prácticamente el mismo tiempo de cómputo que la anterior y además reduce drásticamente el número de recursos internos de la FPGA consumidos.

4.2.3.8. Arquitectura sistólica de procesadores para autovalores y autovectores.

El primer diseño propuesto para el cálculo de los autovectores y autovalores, es la implementación de dos arquitecturas sistólicas, una para autovalores y autovectores, acorde a la Figura 4.19 y Figura 4.20. El diseño realizado es válido para cualquier matriz cuadrada de dimensión par. Cada PD emplea un módulo CORDIC para calcular los ángulos de rotación, y otros dos CORDIC para la doble rotación. Por su parte, los PNDs y PVs usan dos módulos CORDIC para realizar la doble rotación y simple rotación respectivamente.

En este último caso, los módulos CORDIC empleados han sido generados a partir de la herramienta *Core Generator* de Xilinx [Xilinx, 2004]. Se ha optado por la utilización de los *cores* CORDIC de Xilinx, ya que aportan la flexibilidad y eficiencia necesarias. El resto del diseño ha sido codificado en VHDL, permitiendo modificaciones del tamaño y precisión de los datos de entrada en tiempo de compilación.

Estos *cores* permiten trabajar en coordenadas circulares o hiperbólicas, para modo rotación o vectorización, permitiendo seleccionar: sistema de coordenadas (circulares o hiperbólicas), estructura interna (serie o paralelo), tamaño de datos de entrada (desde 0 a 48 bits), modo de funcionamiento (rotación, vectorización), compensación factor de escala e iteraciones del algoritmo CORDIC.

Como premisas de la implementación del sistema de cálculo de autovalores y autovectores, se adoptan las siguientes:

- Matrices de entradas simétricas y cuadradas.
- Posibilidad de configurar el sistema para matrices de cualquier dimensión par.
- Posibilidad de modificar el tamaño de los datos de entrada.

Todo el diseño realizado posee una estructura modular jerárquica que permite su reutilización, así como la utilización de diferentes criterios de síntesis para cada uno de los bloques desarrollados.

En cuanto a la representación de los datos, se ha optado por la representación en coma fija *1Qn* para las componentes x, y de los vectores de entrada (ver (4.46)) y *2Qn* para los ángulos (ver (4.47)). El formato *1Qn* utiliza 1 bit de signo, 1 bit de parte entera y *n* bits de parte fraccionaria. Por su parte, la *2Qn* utiliza 1 bit de signo, 2 bit de parte entera y *n* bits de parte fraccionaria.

$$X_{1Qn} = -2 \cdot x_{n+1} + \sum_{i=0}^{n} \frac{2^i \cdot x_i}{2^n} \tag{4.46}$$

$$X_{2Qn} = -2^2 \cdot x_{n+2} + \sum_{i=0}^{n+1} \frac{2^i \cdot x_i}{2^n} \tag{4.47}$$

donde *n* es el número de bits de la parte fraccionaria y x_i representa cada uno de los bits del número en coma fija, siendo x_0 el bit menos significativo. Por tanto, la longitud de un número *1Qn* es $n+2$ y para un número *2Qn* es de $n+3$.

Este tipo de representación en coma fija ha sido elegida de esta forma ya que:

a) En el caso de los ángulos, su valor está dentro del intervalo $\pm\frac{\pi}{2} = \pm 1.5707$. Este número codificado en binario necesita un bit de signo y un bit más de parte entera y dependiendo de la precisión deseada *n* bits de parte fraccionaria (codificación 1*Qn*). Sin embargo como CORDIC es un método iterativo que ejecuta diversas operaciones aritméticas, para evitar problemas de *overflow* se ha añadido un bit más de parte entera, empleando por tanto formato *2Qn*.

b) Con respecto a los vectores de entrada, en las versiones iniciales del *core* CORDIC de Xilinx era obligatorio que los datos de las coordenadas x, y estuvieran normalizados. Las nuevas versiones del *core* permiten introducir datos, siempre en coma fija, fijándose por el usuario el tipo de representación. En este caso se optó por normalizar los datos x, y para poder utilizar este diseño con las versiones antiguas de *cores* de Xilinx. En el desarrollo del sistema aparecieron diversas situaciones de *overflow* que obligaron a aumentar el tamaño de la coordenada x. Concretamente, se estableció un bit más de salvaguarda en la parte entera empleando así una codificación *1Qn*.

El diseño jerárquico superior se puede observar en la Figura 4.30. Este bloque combina el cálculo de autovalores y autovectores en un sistema único. La integración de estos dos elementos provee al usuario de un sistema completo de cálculo con una interfaz cómoda desde el punto de vista de reutilización de este módulo en otros diseños.

El tamaño de datos está inicialmente fijado en 16 bits. Este tamaño estándar en numerosas aplicaciones permite tener un punto de partida a la hora de evaluar la precisión y número de recursos internos de la FPGA consumidos. Más adelante se justifica que incluso con este tamaño el sistema es inviable desde un punto de vista de recursos consumidos. Por esta razón, no se ha extendido el estudio a otros tamaños.

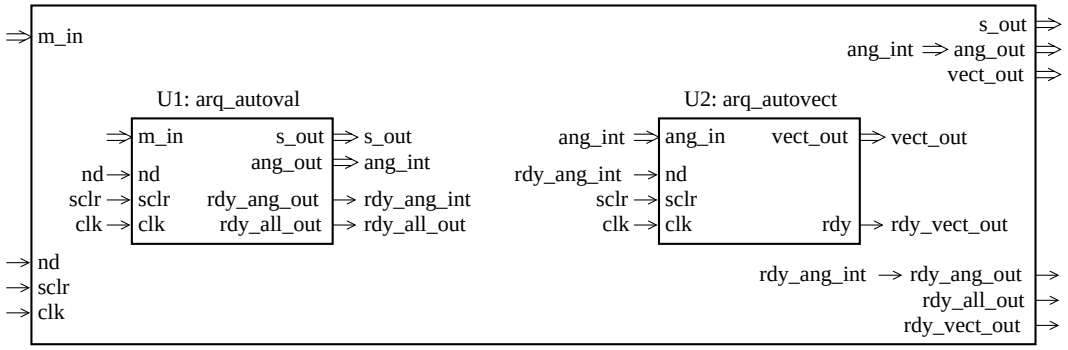


Figura 4.30. Arquitectura sistólica propuesta, basada en [Brent, 1985] empleando *cores* de CORDIC de Xilinx.

A continuación se describen los aspectos más importantes del bloque sistólico de cálculo de autovalores y autovectores de la Figura 4.30 propuestos en esta tesis.

4.2.3.8.1. Bloque Autovalores (arq_autoval).

Tal y como se describió anteriormente, este bloque está formado por una matriz de procesadores, siendo su estructura interna idéntica a la mostrada en la Figura 4.19. Adicionalmente los ángulos de rotación generados por este módulo son utilizados en el cálculo de los autovectores. El diseño es totalmente flexible, ya que se han empleado sentencias de VHDL *generate*, que condicionan el valor de los parámetros del paquete de datos de configuración. De esta forma se puede definir teóricamente cualquier tamaño de matrices de entrada. En la práctica, el número de recursos consumidos delimita el tamaño máximo de la matriz de entrada. La precisión de los datos y el número de iteraciones del algoritmo de Jacobi también se pueden modificar de forma sencilla a través de un paquete de configuración.

El proceso realizado por este bloque consta, a grandes rasgos, de 5 etapas:

- Cálculo de los ángulos de rotación por parte de los procesadores diagonales.
- Propagación de los ángulos al resto de procesadores.
- Cálculo de las rotaciones.
- Recolocación de los datos dentro de cada procesador.
- Propagación de datos entre procesadores.

Estos dos últimos procesos de recolocación e intercambio de datos se realizan conjuntamente. Para realizar estas fases simultáneamente se ha analizado el

trasiego de datos expuesto en [Brent, 1985], realizándose una clasificación de procesadores en función de la forma en que reordenan internamente sus datos y cómo los propagan al resto. Así, se establece una clasificación de grupos de procesadores, llegándose a encontrar hasta 16 grupos diferentes para matrices mayores o iguales a 10×10. Para matrices de tamaño inferior se eliminan algunas zonas y por tanto el número de grupos disminuye. En la Figura 4.32, marcados por líneas discontinuas, se muestran los 16 tipos diferentes para una matriz de entrada de 10×10. Los datos de un procesador que pertenece a un grupo, se mueven exactamente igual que los del resto de procesadores de ese grupo. Así, la codificación de trasiego de datos es única para cada uno de los grupos y en el caso peor, se deben realizar 16 métodos distintos de recolocación, independientemente del tamaño de la matriz de entrada.

Por otra parte en la Figura 4.31 se muestra el diagrama de bloques de la arquitectura sistólica de cálculo de autovalores, particularizado para una matriz de entrada de 6×6 elementos. En esta figura, se puede observar fuera de la estructura sistólica, un bloque de control (*ctrl_autoval*) encargado de sincronizar el funcionamiento de esta fase de cálculo.

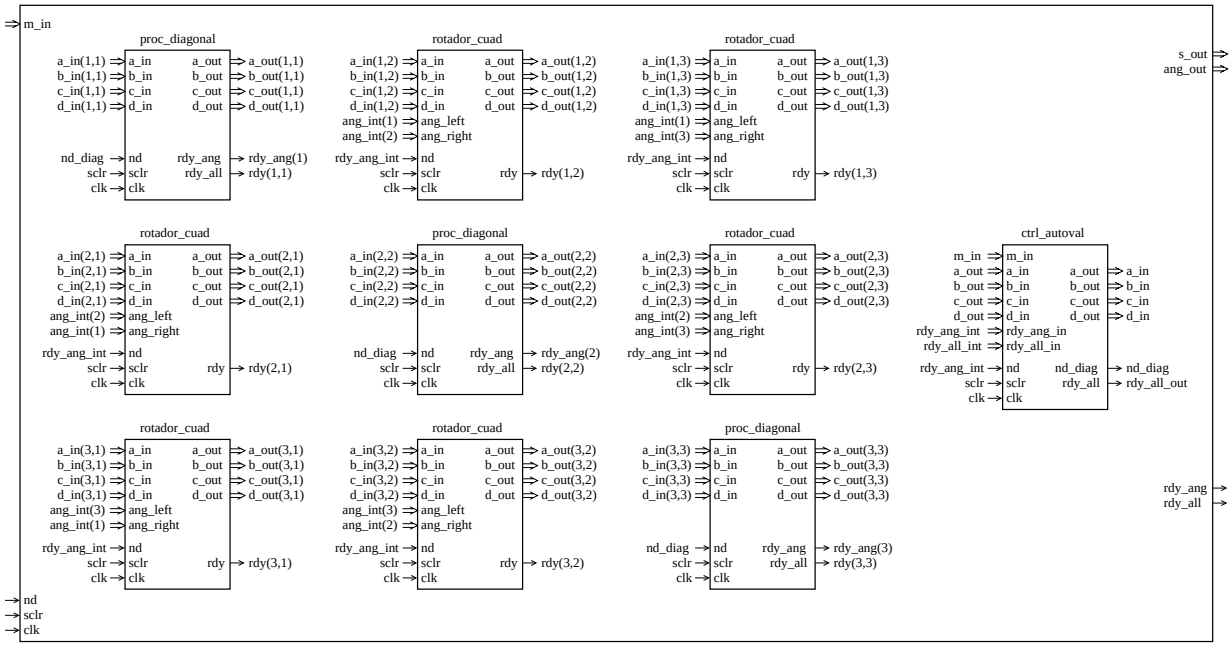


Figura 4.31. Diagrama de bloques de la arquitectura sistólica para el cálculo de autovalores para una matriz de entrada de 6x6 elementos.

Dentro de la Figura 4.31 se observan dos tipos de bloques: *proc_diagonal* y *rotador_cuad*. El primero se corresponde con los PDs descritos anteriormente y el segundo con los PNDs.

El bloque de PD está internamente compuesto por dos bloques: *rotador_cuad* y *vector_cuad*. El primero de ellos se encarga de realizar la doble rotación que deben

ejecutar los PDs y el segundo del cálculo del ángulo de rotación correspondiente. Por tanto, este bloque debe cumplir dos tareas:

- Calcular el ángulo de rotación que utilizan todos los PNDs de su fila y de su columna, para poder realizar sus rotaciones (*vector_cuad*).
- Realizar la doble rotación de su submatriz de 2×2 (*rotador_cuad*).

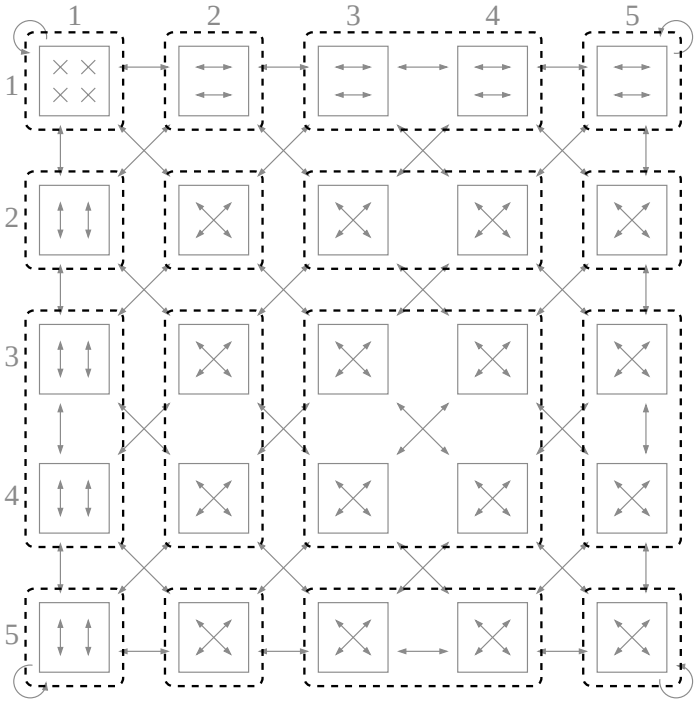


Figura 4.32. Clasificación de las diferentes zonas de trasiego de autovalores según su intercambio de datos.

4.2.3.8.2. Bloque autovectores (arq_autovec).

La actividad de este bloque se inicia una vez recibidos los ángulos de rotación. Su funcionamiento se ejecuta en paralelo con la de autovalores.

También está formado por un *array* sistólico de procesadores, donde cada uno realiza una rotación de una submatriz de 2×2 elementos. La Figura 4.33 presenta una arquitectura de cálculo de autovectores para una matriz de 6×6 elementos. Se observa como cada uno de los PNDs (bloque *rotador_right* de la Figura 4.33), recibe cuatro datos (elementos de la matriz de 2×2 elementos) y genera otros cuatro elementos. En este caso, únicamente hay un ángulo de entrada ya que en los PNDs sólo se realiza una rotación.

Al igual que en el bloque de autovalores, existe un elemento fuera del *array*, en este caso denominado como *ctrl_autovect*, encargado de gestionar el trasiego de datos entre los procesadores de la matriz. Este trasiego de datos es idéntico al realizado en el caso de los autovalores.

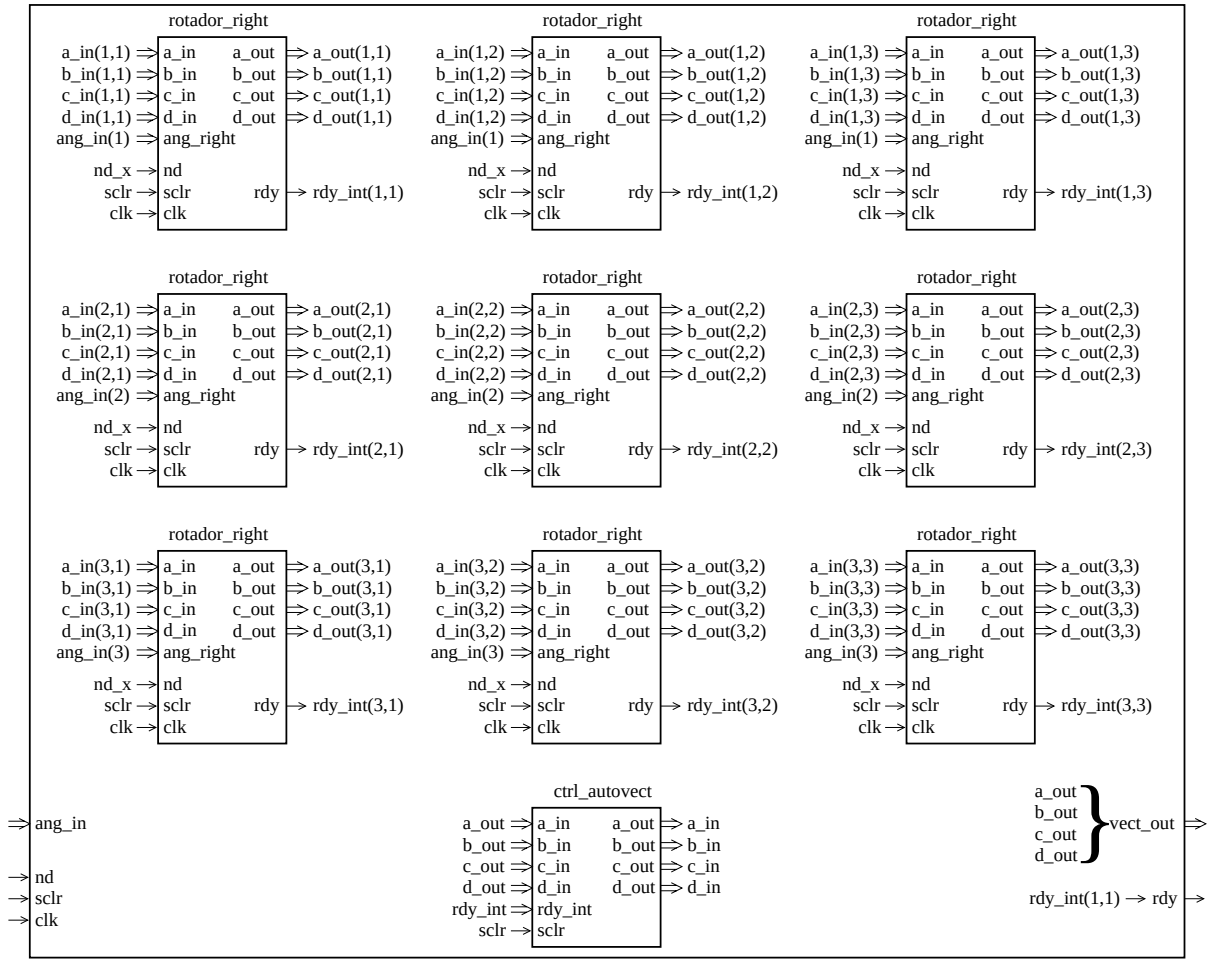


Figura 4.33. Diagrama de bloques de la arquitectura sistólica para el cálculo de autovectores para una matriz de entrada de 6x6 elementos.

4.2.3.8.3. Determinación del tamaño de datos.

La elección del tamaño óptimo de los datos de entrada es fundamental desde dos puntos de vista: exactitud del sistema y recursos consumidos. Desde el punto de vista de exactitud, a mayor número de bits, teóricamente se mejora la precisión. Sin embargo, el aumento del tamaño de bits implica un aumento notable de los recursos internos consumidos.

Para determinar el valor óptimo del tamaño en bits de los datos, en esta propuesta no se ha realizado un estudio exhaustivo del tamaño ideal en VHDL, sino que mediante la herramienta gráfica Xilinx System Generator (XSG) [XSG, 2003], se ha diseñado el mismo sistema que el expuesto en el apartado anterior, y se han testado diversos tamaños [Bravo, 2004]. La Figura 4.34 presenta la pantalla principal de la aplicación desarrollada para una matriz de entrada de 6x6 elementos.

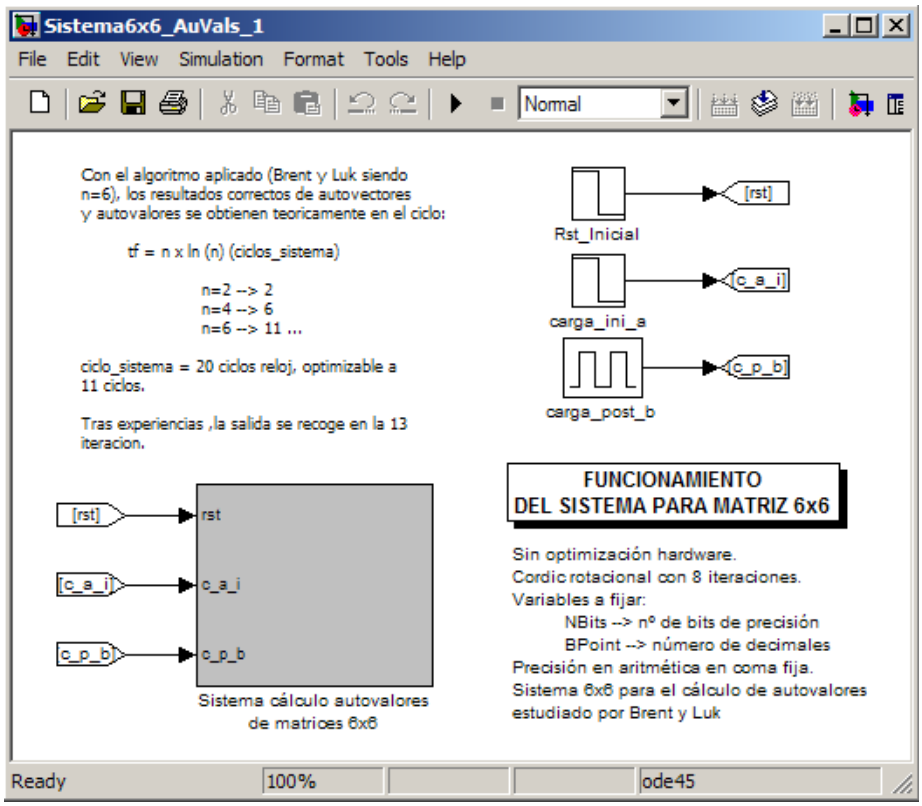


Figura 4.34. Pantalla principal entorno XSG para el cálculo de autovalores y autovectores para una matriz de entrada de 6x6 elementos.

Los tamaños elegidos para realizar el análisis han sido: 16 bits, 20 bits y 50 bits. En todo momento se ha intentado dotar de la máxima flexibilidad a este sistema, permitiendo el cambio del tamaño de datos y precisión de forma cómoda. Las conclusiones de los resultados obtenidos se pueden resumir en:

- El tamaño de 16 bits es el que ofrece los peores resultados de los tres. Sin embargo, su error disminuye si los datos de entrada se normalizan a formato *1Qn*.
- El tamaño de 50 bits presenta, desde el punto de vista de exactitud, los mejores resultados. Sin embargo, es la opción que consume mayor número de recursos (330.000 *slices* con respecto a los 24.000 que se utilizan con 16 bits).
- El número de recursos consumidos por 20 bits es el doble que con 16 bits.
- Para esta arquitectura el número de iteraciones del método de Jacobi, debe estar en 13 en vez de las 11 ($6\log(6)$) que se propone en [Brent, 1983]. Este valor se obtuvo empíricamente.

Debido a razones de recursos consumidos se ha empleado como tamaño de datos 16 bits. La exactitud alcanzada en el cálculo de autovalores y autovectores con esta propuesta, se presenta en el siguiente apartado.

4.2.3.8.4. Resultados de la arquitectura sistólica.

En este apartado se realiza un análisis del error de precisión obtenido, así como del número de recursos consumidos por las arquitecturas sistólicas diseñadas. Como entrada al algoritmo se ha utilizado una matriz de covarianza de 6×6 elementos. Sobre esta matriz se ha realizado el cálculo de sus autovalores y autovectores mediante el sistema propuesto, empleándose una codificación de 16 bits en *1Qn*. Los resultados generados han sido comparados con los ofrecidos por MATLAB. Concretamente, se han utilizado los resultados que ofrece la función *eigs* de MATLAB la cual opera y genera datos en coma flotante.

Primeramente, se muestra el error cometido entre la arquitectura propuesta y los resultados ofrecidos por MATLAB (4.48). Así, en la gráfica superior de la Figura 4.35 se presenta el porcentaje de error cometido para los autovalores, siendo en este caso el error máximo obtenido (4.50) de $5.36\cdot10^{-5}\%$. Por su parte, el error cuadrático medio (4.49) alcanza el valor de $3.28\cdot10^{-5}\%$. En ambos casos se puede afirmar que el error cometido es realmente bajo. Con respecto a los autovectores, en la gráfica inferior de la Figura 4.35 se presenta el error cometido por cada autovector con respecto al valor ideal (4.48). El error máximo cometido en dicha gráfica es de $7.95\cdot10^{-4}\%$, mientras que el cuadrático medio de cada autovector, se muestra en la Tabla 4.4. Si ahora calculamos el error cuadrático medio total, el valor obtenido es de $3.32\cdot10^{-4}\%$. En ambos casos el error obtenido también es bajo, debido a la propagación y amplificación del error que se realiza en el diseño tras realizar cada iteración. Se puede comprobar como en el caso de autovectores el error es superior al de autovalores, debido fundamentalmente a que el valor de los autovectores siempre está entre 0 y 1, por lo que el error relativo es mayor. Cabe destacar en este análisis que a los errores mostrados, habría que añadirles el cometido en la normalización a formato *1Qn* de los datos de la matriz de covarianza de entrada.

$$error_i(\%) = \frac{\left|V_i\right|_{VHDL} - V_i\big|_{MATLAB}\right|}{V_i\big|_{MATLAB}} \times 100 \tag{4.48}$$

$$e_{SQRT}(\%) = \sqrt{\frac{\sum_{i=0}^{k}\left(error_i(\%)\right)^2}{k}} \tag{4.49}$$

$$e_{MAX}(\%) = \max(error_i(\%)) \tag{4.50}$$

Tabla 4.4. Error cuadrático medio de cada uno de los autovectores.

autovector 1	autovector 2	autovector 3	autovector 4	autovector 5	autovector 6
0.3178E-3	0.4002E-3	0.4971E-3	0.3289E-3	0.2101E-3	0.0631E-3

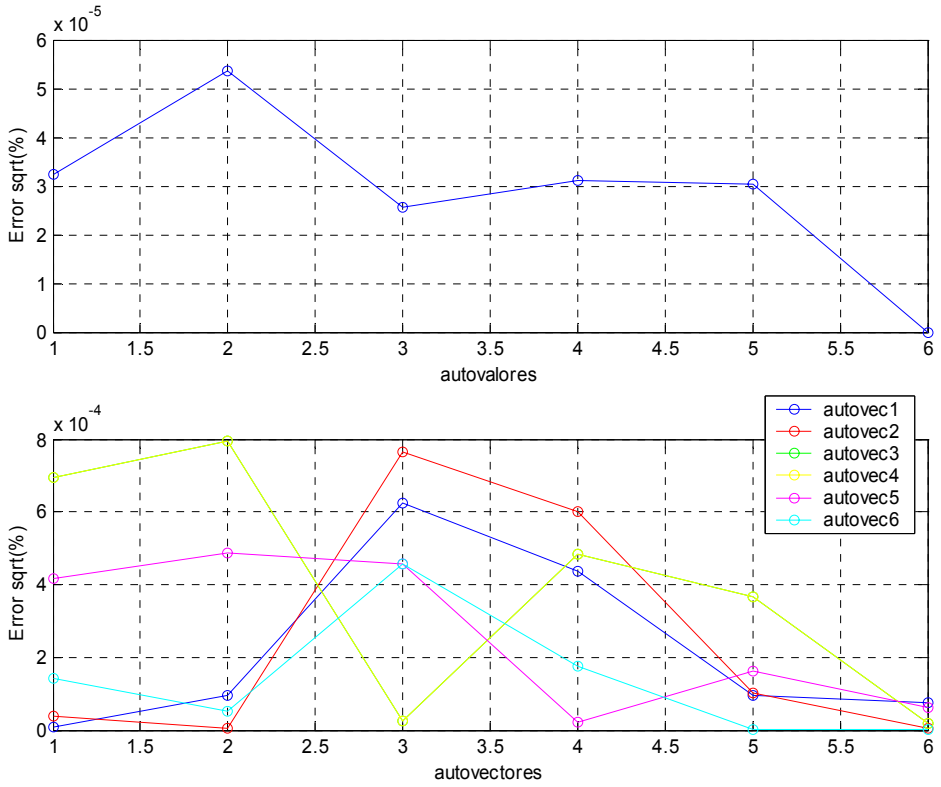


Figura 4.35. Error numérico de autovalores y de autovectores con respecto a los resultados de MATLAB para la primera arquitectura propuesta.

Sin embargo y aunque a la vista de la exactitud obtenida anteriormente pudiera parecer lo contrario, la alternativa planteada es inviable. Ello se debe a que desde un punto de vista de recursos consumidos, el número es muy elevado, haciéndose necesario emplear casi al 100% una FPGA de grandes prestaciones y por tanto de elevado precio.

El diseño ha sido sintetizado e implementado sobre una FPGA XC2VP50 de Xilinx, la cual posee un total de 23616 *slices*. Se ha elegido esta familia de FPGAs para poder comparar los resultados obtenidos, con los generados desde la otra propuesta desarrollada en esta tesis (apartado 4.2.3.9), la cual se implementa en una XC2VP7. La FPGA XC2VP50 es la FPGA más pequeña dentro de la familia V2P que se puede emplear para este diseño. Esto es así, ya que el diseño realizado ocupa un total de 23503 *slices*, es decir el 99% del total de una XC2VP50.

Por tanto, se puede concluir que esta propuesta, desde el punto de vista de tamaño necesario de FPGA, se hace inviable, al ocupar un excesivo número de recursos consumidos. Una alternativa para disminuir el número de *slices* es el diseño de módulos CORDIC específicos en VHDL o Verilog. Con esta opción se reducen notablemente los recursos consumidos.

4.2.3.9. Nueva arquitectura segmentada basada en CORDIC para el cálculo de autovalores y autovectores.

Debido al elevado número de recursos consumidos por la alternativa descrita en el apartado anterior surge esta nueva propuesta. Para el desarrollo de este nuevo diseño se ha analizado en profundidad el funcionamiento del sistema mostrado en [Brent, 1983], y se ha intentado optimizar al máximo el número de recursos consumidos, intentando dotar al mismo de la máxima velocidad de ejecución. Para ello se ha creado un *pipeline* que se ejecuta a pleno rendimiento.

El funcionamiento del nuevo diseño se basa en lo descrito en el apartado anterior, así como en el apartado 4.2.3.5. Tras la carga inicial de datos en cada procesador, la secuencia de funcionamiento en una iteración dentro del algoritmo de Jacobi, adaptado a matrices de $2x2$ elementos, es la siguiente:

- Cálculo de ángulos de rotación en los PDs.
- Doble rotación en los procesadores de autovalores y rotación simple en los de autovectores.
- Reordenación interna de resultados en cada procesador y transmisión a los procesadores adyacentes.

Debido a la simetría de las matrices con las que se trabaja, la estructura sistólica clásica de Brent (ver Figura 4.18) puede ser reducida a una estructura sistólica triangular, como la expuesta en la Figura 4.36, eliminando así los PNDs correspondientes a la matriz triangular inferior. Esta arquitectura es solamente aplicable al cálculo de autovalores, ya que en el caso de los autovectores no existe dicha simetría.

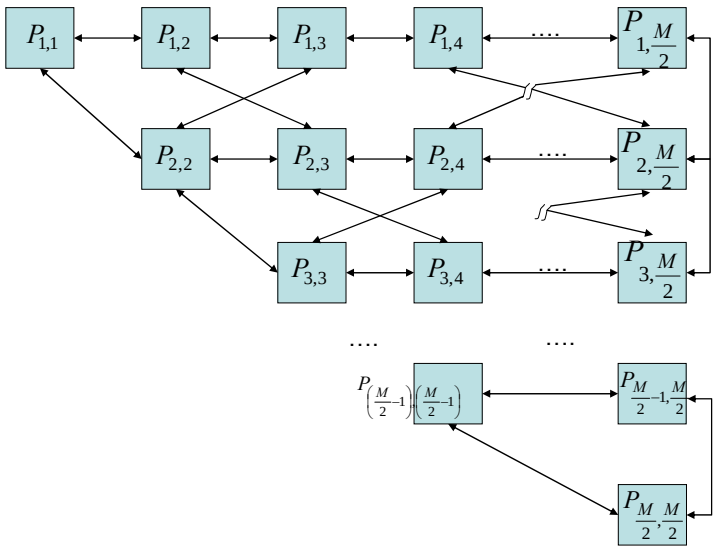


Figura 4.36. Arquitectura sistólica de Brent simplificada para matrices de entrada simétricas.

Mediante esta simplificación, si el tamaño de la matriz de entrada es $M \times M$, en (4.51) se presenta el número de PNDs que se reducen frente a los $M/2 \times M/2$ que necesita la propuesta de [Brent, 1983].

$$reducción\_PNDs = \frac{M \cdot (M-2)}{8} \tag{4.51}$$

Para tener una idea del número de recursos que permite ahorrar esta nueva estructura, se va a analizar desde el punto de vista de *slices* consumidos. Así, si se utilizara el módulo CORDIC rotación de la herramienta *Xilinx Core Generator*, el número de *slices* que éste ocuparía bajo las mismas condiciones del apartado 4.2.3.8, estaría entorno a 405. Si cada PNDs emplea dos unidades CORDIC rotación, cada procesador ocuparía junto a la lógica adicional, aproximadamente 825 *slices*. En el caso de una matriz de entrada de 6x6 elementos el número de PNDs suprimidos sería tres, por lo que el número de *slices* ahorrados sería 2475. Para poder valorar este resultado, supongamos que la FPGA escogida es la misma que en el apartado anterior (XC2VP50), el porcentaje de ahorro de recursos consumidos alcanzaría el 28.5%. Este porcentaje aumenta notablemente a medida que aumenta el tamaño de la matriz de entrada (para $M = 12$ el porcentaje de ahorro sería del 35.7%). Por tanto, la propuesta a desarrollar se beneficiará de esta posibilidad de reducción.

Basándose en la reducción del número de datos a operar (Figura 4.36), la nueva propuesta que se realiza en tesis se muestra en la Figura 4.38. Su funcionamiento interno ejecuta el método de Jacobi aplicado a matrices de $2x2$ elementos (apartado 4.2.3.2) sin necesidad de implementar ningún tipo de procesador de matrices de $2x2$ elementos. Así, su estructura interna está compuesta únicamente de dos módulos CORDIC, memorias, registros y multiplexores. Esta propuesta de implementación tiene como características destacables:

- Ejecución del algoritmo de cálculo de autovalores y autovectores de forma íntegra en una FPGA.
- Cálculo de autovalores y autovectores de $\mathbf{S} \in \Re^{M \times M}$ a partir de su matriz triangular superior ($\mathbf{S}_T \in \Re^{M \times M}$) gracias a la condición de simetría de $\mathbf{S}$.
- La arquitectura física propuesta no responde a la de un *array* sistólico, pero desde el punto de vista de procesamiento, las características son similares a las que se obtendrían con un array sistólico.
- Alta velocidad de ejecución del cálculo de autovalores y autovectores con respecto a plataformas basadas en sistemas secuenciales.
- Bajo número de recursos de la FPGA consumidos comparado con estructuras sistólicas para el cálculo de autovalores y autovectores (apartado 4.2.3.8) e independiente del tamaño de las matrices de entrada.

- Empleo de módulos CORDIC trabajando en coordenadas circulares, para resolver las multiplicaciones de matrices de (4.41) y (4.45).
- Flexibilidad para trabajar con diferentes tamaño de matrices (M) y tamaños de datos (n)
- Arquitectura interna concurrente.
- Cálculo de autovalores y autovectores realizado mediante una misma arquitectura con una estructura interna en *pipeline*.

El algoritmo propuesto es un proceso iterativo, siendo su funcionamiento el siguiente:

1.- El primer paso para el cálculo de autovalores es el almacenamiento de la matriz $\mathbf{S}_T$ en una memoria. A la hora de manejar los elementos de $\mathbf{S}_T$ estos deben ser agrupados en submatrices de 2×2 elementos ($\mathbf{S}_{Ti,j} \in \Re^{2\times 2}$), donde se realiza la clasificación entre submatrices diagonales ($\mathbf{S}_{Ti,i}$) y no diagonales ($\mathbf{S}_{Ti,j},\ i \neq j$).

2.- Con respecto al cálculo de autovectores, la matriz inicial que se necesita ($\mathbf{V}^{(0)} \in \Re^{M\times M}$) es una matriz identidad $\mathbf{I} \in \Re^{M\times M}$. Esta matriz se almacena en la misma memoria donde está $\mathbf{S}_T$.

3.- Una vez que la memoria contiene todos los datos iniciales, se calculan los ángulos de rotación $\alpha_{i,i}^{(k)}$ (4.38) a partir de las submatrices diagonales ($\mathbf{S}_{Ti,i} \in \Re^{2\times 2}$). El método elegido para resolver (4.38) ha sido CORDIC ya que la obtención de ángulos es una de las operaciones que este algoritmo resuelve cuando se trabaja en modo vectorización con coordenadas circulares [Volder, 1959] [Walter, 1971]. El algoritmo CORDIC es una alternativa muy apropiada para resolver estas operaciones trigonométricas en hardware, ya que su estructura interna es sencilla de implementar en una FPGA [Andraka, 1998]. Los ángulos generados deben ser almacenados ya que serán utilizados tanto por el proceso de cálculo de autovalores (4.41) como por el de autovectores (4.45).

4.- Una vez que se dispone de los ángulos de rotación, la fase de autovalores es la primera que se ejecuta ya que de esta forma, tal y como se justifica posteriormente, se optimiza el tiempo de ejecución de todo el algoritmo. Para ello, cada $\mathbf{S}_{Ti,j} \in \Re^{2\times 2}$ debe realizar la operación indicada en (4.41). Esta operación incluye dos multiplicaciones de submatrices de 2×2 elementos, las cuales se resuelven mediante el algoritmo CORDIC en modo rotación para coordenadas circulares, ya que estas multiplicaciones corresponden a la rotación de dos vectores un ángulo $\alpha_{i,i}^{(k)}$. Para resolver (4.41) mediante CORDIC, tal y como se ha justificado en el apartado 4.2.3.6, primeramente

se debe realizar el producto: $\mathbf{R}\left(\alpha_{i,i}^{(k)}\right)^T \cdot \mathbf{S}_{T\,i,j}^{(k)} = \mathbf{Q}_{i,j}^{(k)}$, donde $\mathbf{Q}_{i,j}^{(k)} \in \Re^{2\times 2}$, y una vez obtenido éste se calcula $\mathbf{Q}_{i,j}^{(k)} \cdot \mathbf{R}(\alpha_{j,j}^{(k)}) = \mathbf{S}_{i,j}^{(k+1)}$. Si el número de etapas del módulo CORDIC que coincide con el número de bits de los datos (n) es tal que $n > M$, en la generación de $\mathbf{Q}_{i,j}^{(k)}$ utilizando CORDIC se debe tener presente que éste posee una latencia inicial proporcional al número de etapas de la arquitectura interna (n). Con idea de optimizar el uso de los módulos CORDIC y disminuir el tiempo de ejecución del cálculo de autovalores y autovectores, en aplicaciones prácticas en las que $n > M$, se puede iniciar la fase de cálculo de autovectores (4.45) en el mismo módulo CORDIC durante la latencia de generación del primer elemento de $\mathbf{Q}_{i,j}^{(k)}$. Esto es posible ya que la multiplicación de matrices a realizar en (4.45) posee características análogas a (4.41) por lo que de nuevo se emplea CORDIC en modo rotación con coordenadas circulares. La fase de cálculo de autovectores se interrumpe cuando se supera esta latencia y se reanuda cuando finaliza la operación $\mathbf{Q}_{i,j}^{(k)} \cdot \mathbf{R}(\alpha_{j,j}^{(k)}) = \mathbf{S}_{i,j}^{(k+1)}$.

5.- Todos los resultados de $\mathbf{S}_{T_{i,j}}^{(k)} \in \Re^{2\times 2}$ y de $\mathbf{V}_{i,j}^{(k)} \in \Re^{2\times 2}$, deben ser almacenados en la memoria donde se habían guardado las matrices iniciales. La forma de almacenar cada $\mathbf{S}_{T_{i,j}}^{(k)}$ y cada $\mathbf{V}_{i,j}^{(k)}$ debe responder al proceso de reordenación y transferencia de resultados mostrados en la Figura 4.24.

Una vez realizado esto, habrá finalizado la primera iteración y el sistema repetirá los pasos 3, 4 y 5, otras h iteraciones más. En la Figura 4.37 se presenta el diagrama de flujo del algoritmo propuesto en esta tesis.

Para la implementación del algoritmo expuesto en la Figura 4.37 se propone la arquitectura mostrada en la Figura 4.38. Como se puede comprobar esta nueva alternativa está compuesta únicamente por dos módulos CORDIC (para realizar el cálculo de los ángulos de rotación (4.38) así como las multiplicaciones de matrices dadas por (4.41) y (4.45)), memorias internas para almacenar los resultados temporales y finales, registros y multiplexores. Como se pondrá de manifiesto más adelante, la arquitectura propuesta presenta unas características de tiempo de procesamiento muy similares a las de un array sistólico siendo el número de recursos empleados muy bajo sobre todo si se compara con otras alternativas como la presentada en [Ahmedsaid, 2004b]. Otro aspecto muy importante de la arquitectura desarrollada es el mínimo incremento del número de recursos internos cuando se varían las dimensiones de la matriz de entrada ya que sólo se debe modificar el tamaño de la memoria DP y de la ROM. A continuación se describen los elementos que forman la arquitectura mostrada en la Figura 4.38.

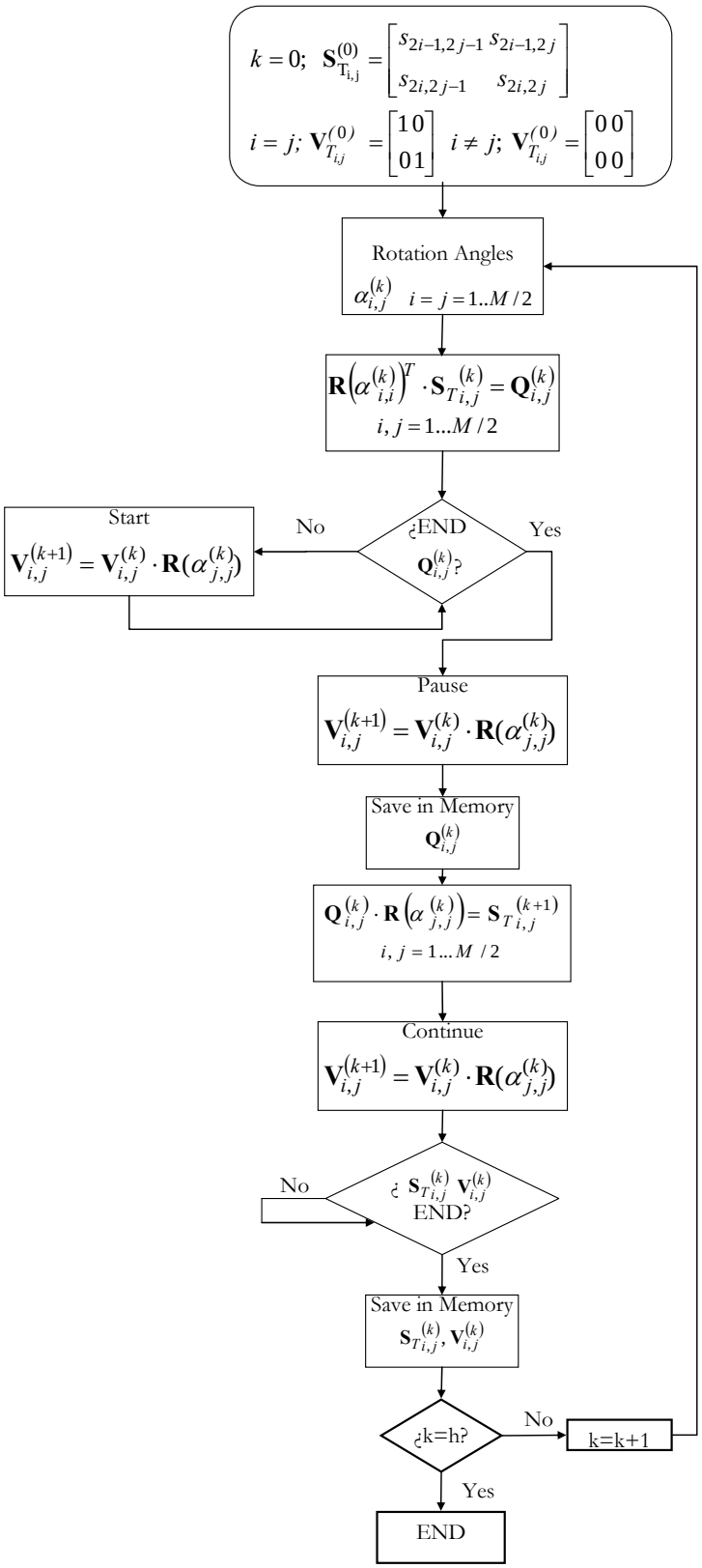


Figura 4.37. Diagrama de flujo del funcionamiento del algoritmo propuesto para el cálculo de autovalores y autovectores.

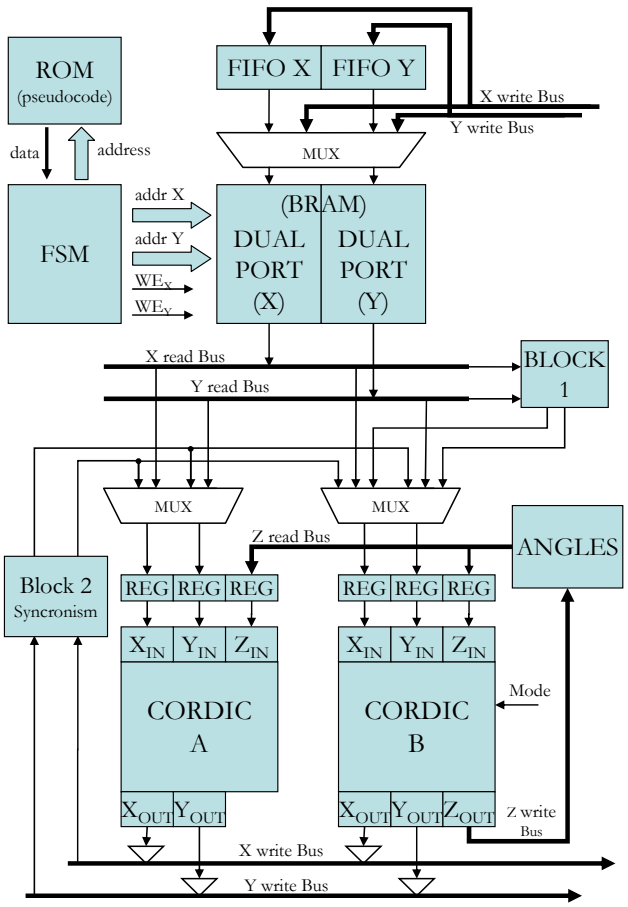


Figura 4.38. Nueva arquitectura paralela desarrollada para el cálculo de autovectores y autovalores.

- *Memoria ROM*. Almacena un conjunto ordenado de pseudocódigos que son diseccionados y decodificados por la máquina de estados (FSM). Estos pseudocódigos se corresponden con la activación/desactivación de cada una de las señales de control de la Figura 4.38. En la Figura 4.39 se muestra el contenido de cada una de las celdas de la memoria ROM. Cabe destacar en dicha figura una zona que estará ocupada si $n>F/4$, donde n es el número de etapas del módulo CORDIC que coincide con el número de bits de los datos de $\mathbf{S}_T \in \Re^{M\times M}$ y F el número de datos de la matriz $\mathbf{S}_T$ (4.52). Los pseudocódigos asociados a esta zona indican el inicio del proceso de cálculo de autovectores mientras se están generando los resultados de la primera rotación de autovalores.

$$F = \frac{M\cdot(M+1)}{2} \qquad (4.52)$$

  La modificación de n y M sólo se puede realizar en tiempo de compilación, ya que en función del tamaño de estas variables provoca una modificación en el contenido de la ROM. Por esta razón éstos únicamente pueden ser variados en tiempo de compilación.

Figura 4.39. Pseudocódigos de la memoria ROM de la Figura 4.38.

- *Memoria Dual-Port (DP).* En ella se almacenan los datos de $\mathbf{S}_T$ y la matriz identidad $\mathbf{I} \in \Re^{M \times M}$. Por tanto su tamaño será de $\frac{\left(F + M^2\right)}{2}$ datos de $2n$ bits cada uno. También esta memoria se utiliza para almacenar los datos temporales de cada iteración y los autovectores y autovalores finales. Se ha escogido este tipo de memoria para poder realizar lecturas/escrituras de forma simultánea de la coordenada x e y, de cada uno de los vectores que manejan los módulos CORDIC.

- *Máquina de estados (FSM).* Se encarga de decodificar la información de la memoria ROM y de activar acorde a esa información las señales de control necesarias así como las direcciones de la memoria DP.

- *Memorias FIFO.* Almacenan temporalmente los autovectores en cada iteración cuando el bus de datos de la DP está ocupado por alguna fase del cálculo de autovalores.

- *Módulos CORDIC.* Se han empleado dos módulos CORDIC-A y CORDIC-B para coordenadas circulares, cuya estructura interna es idéntica. La diferencia entre ellos es que el módulo A funciona sólo en modo rotación, mientras que el B lo hace en rotación y vectorización. Esto es así, ya que si $n > M/2$ con un único módulo trabajando en modo vectorización es suficiente para la obtención de los $M/2$ ángulos de rotación $(\alpha_{i,i}^{(k)})$ con la máxima eficiencia. La estructura interna del

módulo CORDIC desarrollado corresponde a una arquitectura paralela de n etapas coincidiendo n con el tamaño de los datos de entrada tal y como se justificará posteriormente al analizar la convergencia del módulo CORDIC en función del número de bits usado en los datos de entrada. La Figura 4.40 presenta la arquitectura interna tanto del CORDIC A como del B. Se puede comprobar como cada etapa genera sobre la siguiente el modo de funcionamiento y las entradas x, y, z. Esta configuración permite un funcionamiento simultáneo en modo vectorización y rotación sobre un mismo módulo CORDIC. Sin embargo, esta situación no llega a ocurrir dentro del método de Jacobi ya que no se solapa en ningún momento el cálculo de ángulos (modo vectorización) con la rotación de vectores (modo rotación).

El tamaño máximo de los datos de entrada, n, no está acotado. Así, teóricamente será el número de recursos internos disponibles de la FPGA lo que limitará dicho tamaño. Sin embargo, debido a la existencia en la etapa de salida de cada módulo CORDIC de multiplicadores hardware encargados de aplicar el factor de corrección del algoritmo CORDIC (K_m) cuyo tamaño máximo a su entrada es de 18 bits, el valor de n se acotará a 18 bits. El sistema sólo posee dos módulos ya que debido al número de etapas que tienen los módulos CORDIC (n), el ancho de la memoria DP ($2n$) y el tamaño de la matriz de entrada (M^2), con dos módulos se consigue el máximo rendimiento.

- *Bloque 1*. Este bloque está compuesto por registros, multiplexores y un sumador/restador. Su función es la de generar correctamente, a partir de los datos procedentes de otros bloques, las entradas del CORDIC B, en el cálculo del ángulo de rotación atendiendo a (4.38).

- *Bloque 2*. Implementa la realimentación e intercambio de valores entre la primera y la segunda rotación de la etapa de autovalores, acorde a la expresión (4.44). Este bloque evita el paso por la DP en la transición entre la primera y segunda rotación, por lo que no limita el ancho de banda de la DP.

- *Memoria de ángulos (ANGLES)*. En esta memoria son guardados los ángulos de rotación generados en el CORDIC B. En función de la fase de cálculo en que esté operando el sistema la FSM direcciona un ángulo u otro.

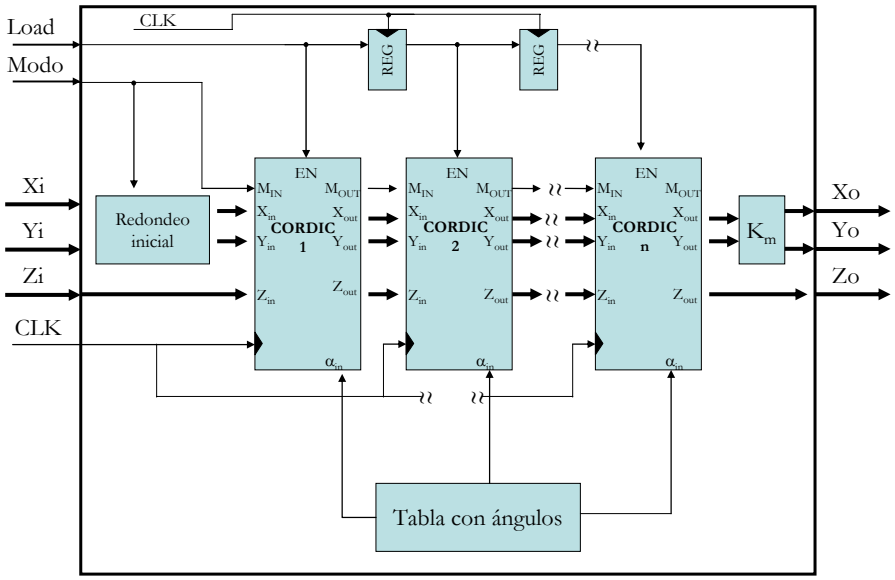


Figura 4.40. Estructura interna del módulo CORDIC vectorización y rotación desarrollado en esta tesis.

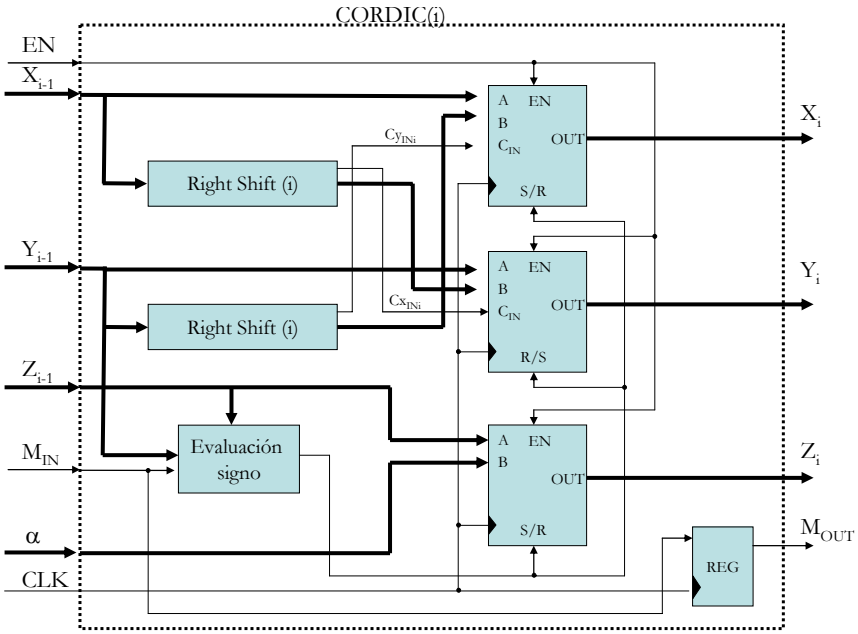


Figura 4.41. Arquitectura interna del módulo CORDIC(i) base de la Figura 4.40.

En cuanto al funcionamiento del sistema, la memoria ROM de la Figura 4.38 contiene la secuencia ordenada de cada iteración: cálculo de ángulos, doble rotación para autovalores y rotación para autovectores (ver Figura 4.39). El funcionamiento detallado de esta nueva propuesta se expone a continuación mostrándose también en la Figura 4.42 su secuencia temporal:

1.- *Cálculo de ángulos*: Inicialmente, tras almacenarse en la memoria DP los valores de la matriz sobre la que determinar los autovalores y la matriz identidad para los autovectores, el primer paso a ejecutar es el cálculo del ángulo de rotación (4.38)

(T_{VECT} de la Figura 4.42). Para ello, si la matriz de entrada posee *MxM* elementos en cada iteración se calculan $M/2$ ángulos de rotación diferentes. La FSM de la Figura 4.38 se encarga de decodificar los primeros pseudocódigos de la ROM de dicha figura, donde éstos contienen la información asociada a las direcciones de la DP que contienen los operandos necesarios para el cálculo de ángulos. Observando la expresión (4.38), se comprueba como en el denominador de ésta aparece una resta. Para realizar esta operación se emplea el Bloque 1, donde su resultado es enviado a los registros de entrada de las coordenadas x, y del CORDIC_B, que es la unidad que opera en esta fase. Para el cálculo de cada ángulo de rotación $\alpha_{i,i}^{(k)}$, según (4.38), hacen falta los datos: $s_{2i-1,2j}, s_{2i,2j}, s_{2i-1,2j-1}$. Por tanto el número total de elementos necesarios para la obtención de todos los ángulos en una iteración es $(3\times M/2)$. El proceso de cálculo de ángulos es secuencial, es decir, primero se toman desde la DP los 3 operandos necesarios para el primer ángulo, a continuación los asociados al segundo ángulo, y así sucesivamente. Cuando los datos del último ángulo de rotación a calcular son introducidos en CORDIC, se interrumpe la entrada nuevos datos hasta que se supere la latencia del CORDIC. En este tiempo no se inicia ningún otro cálculo ya que tanto la fase de cálculo de autovalores como la de autovectores dependen de estos ángulos. Cuando los ángulos son generados se almacenan en la memoria ANGLES de la Figura 4.38 desde dónde posteriormente se leerán en la fase de autovalores y autovectores.

Debido al ancho del bus de datos de la DP, y a que $n > M/2$ $(n = 18; M = 8)$, únicamente se emplea un módulo en este cálculo, en este caso el B, ya que el tiempo de cómputo con una unidad es el mismo que las dos funcionando en paralelo.

2.- <u>*Primera rotación de autovalores*</u> *(Autoval_1):* Una vez generado el último ángulo de rotación, comienza la fase de cálculo de autovalores. Esta fase se divide en dos etapas: primera rotación $\mathbf{R}\left(\alpha_{i,i}^{(k)}\right)^T \cdot \mathbf{S}_{T\,i,j}^{(k)} = \mathbf{Q}_{i,j}^{(k)}$ (T_{ROT1} de la Figura 4.42) y segunda rotación $\mathbf{Q}_{i,j}^{(k)} \cdot \mathbf{R}\left(\alpha_{j,j}^{(k)}\right) = \mathbf{S}_{T\,i,j}^{(k+1)}$ (T_{ROT2} de la Figura 4.42). Esta división es necesaria ya que la segunda rotación necesita los resultados de la primera rotación para poder ejecutarse.

En ambas rotaciones, los datos $\mathbf{S}_{T\,i,j}^{(k)}$ necesarios están almacenados en la DP. Esta memoria posee una frecuencia de reloj (T_{CLK2X}) el doble que la del resto de elementos secuenciales de la propuesta (T_{CLK}), por lo que se pueden extraer de la DP en un T_{CLK} cuatro datos simultáneamente (dos por cada puerto) permitiendo así introducir un vector de datos (x, y) para cada CORDIC en el mismo ciclo de reloj.

Para la primera iteración se necesita la matriz de entrada $\mathbf{S}_T^{(0)} = \mathbf{S}_T$ para autovalores y una matriz identidad $\mathbf{I} \in \Re^{M\times M}$ para autovectores. En el resto de iteraciones se emplean los resultados obtenidos en la iteración previa como datos de

entrada. En ambos casos, tanto la matriz inicial como los resultados de las distintas iteraciones, los datos son almacenados en la DP. Por tanto, es fundamental almacenar en esta memoria de forma ordenada acorde al criterio de "reordenación y transferencia" expuesto en el apartado 4.2.3.5.3.

Los pseudocódigos necesarios para esta primera rotación de autovalores están ubicados en la memoria ROM justo a continuación de los asociados al cálculo de ángulos.

3.- *Inicio rotación autovectores (Autovec_1)*: Coincidiendo con la introducción de los últimos datos de la primera rotación del proceso de cálculo de autovalores, considerando el caso práctico $n > F/4$ y debido a la latencia de los módulos CORDIC, en la salida aun no se han obtenido los primeros elementos de $\mathbf{Q}_{i,j}^{(k)}$. Por tanto, en ese intervalo de espera se introducen los primeros datos asociados al cálculo de autovectores. Concretamente, el número de vectores $m=(x,y)$ que se puede introducir en cada módulo CORDIC en este intervalo viene dado por (4.53). El tiempo consumido en esta fase se denomina como $T_{AUTOVEC}$ en la Figura 4.42.

$$m = n - \frac{F/2}{nº\_módulos\_CORDIC} - 1 = n - F/4 - 1 \qquad (4.53)$$

4.- *Segunda rotación autovalores (Autoval_2):* A partir del instante en el que se generan los primeros resultados de la primera rotación de los autovalores, se interrumpe la entrada de datos de autovectores y se inicia la segunda rotación de autovalores (T_{ROT2} de la Figura 4.42). Los resultados de la primera rotación tras su paso por el Bloque 2 de la Figura 4.38, van siendo introducidos en los módulos CORDIC según se van generando. Para que se pueda iniciar la segunda rotación, de nuevo se debe direccionar la memoria de ángulos para introducir en los dos módulos CORDIC los ángulos necesarios para esta segunda rotación.

La codificación empleada en la propuesta realizada no permite, para el caso de $n \leq F/4$, que se introduzca ningún dato asociado a los autovectores en un módulo CORDIC. En este caso, la fase de obtención de los autovectores tendría que esperar a la finalización de la segunda rotación de autovalores.

Mientras que se introducen los nuevos datos para la segunda rotación, a la salida de los módulos CORDIC se van obteniendo los resultados de los autovectores introducidos previamente. Estos resultados son almacenados en unas memorias FIFO intermedias que evitan el acceso simultáneo a la DP, ya que en este instante están accediendo los datos de autovalores de la segunda rotación. En el momento que la DP se queda libre se vuelca el contenido de estas memorias a las posiciones correspondientes de la DP.

5.- *Fin rotación autovectores (Autovec_1_fin)*: Cuando se terminan de introducir los datos de autovalores a la entrada de los CORDIC, se extraen desde la DP los

datos restantes de autovectores y se empiezan a insertar en los CORDIC ($T_{AUTOVEC2}$ de la Figura 4.42). Los resultados de autovectores de la fase *Autovec_1* que se siguen generando son almacenados en las memorias FIFO y serán volcados desde éstas a la DP cuándo ésta esté libre.

Cuando se generan los primeros datos de autovalores de la fase *Autoval_2*, éstos se acumulan ordenadamente en la DP ($T_{WR\_AUTOVAL}$). En este momento todavía la iteración no ha finalizado ya que quedan resultados de autovectores que todavía no han sido generados (fase *Autovec_1_fin*). Sin embargo, para acelerar el funcionamiento del sistema es justamente en este momento cuando se inicia la siguiente iteración. Es decir, el denominado $T_{WR\_AUTOVEC}$ de la Figura 4.42 es solapado con el T_{VEC} de la siguiente iteración. Para ello, de nuevo, es necesario acumular en las memorias FIFO los resultados de autovectores, volcándose su contenido a la DP en el momento que ésta quede libre.

Tras un determinado número de iteraciones, h, el proceso de cálculo de autovalores y autovectores se puede dar por finalizado.

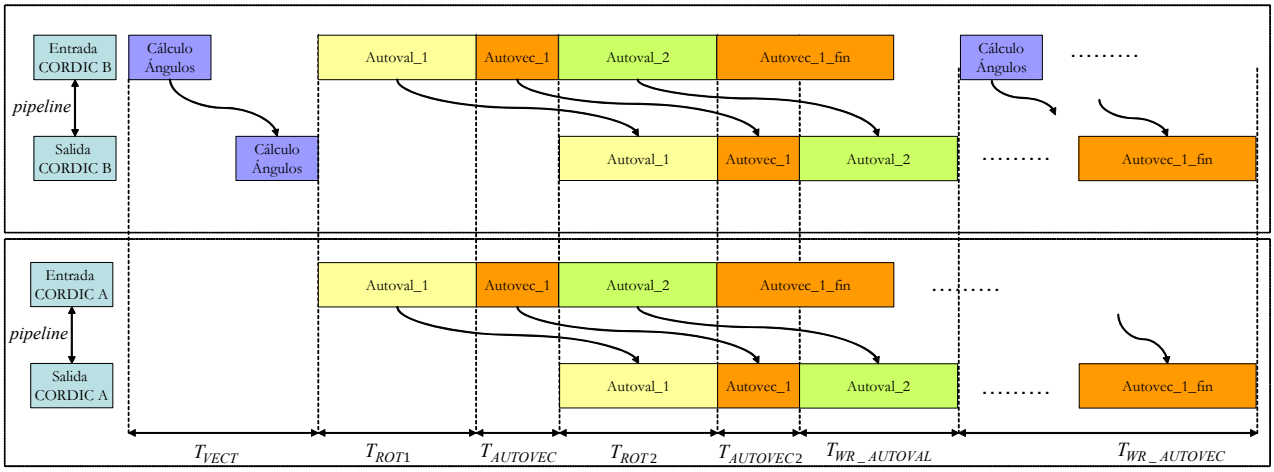


Figura 4.42. Secuencia de funcionamiento de los módulos CORDIC en una iteración.

4.2.3.10. Características del módulo CORDIC diseñado.

Con idea de minimizar el número de recursos consumidos en la FPGA se ha diseñado un bloque CORDIC de propósito específico. La celda básica de los módulos CORDIC A y B (Figura 4.41), puede funcionar tanto en modo rotación como en vectorización para coordenadas circulares. Los bloques principales de la citada figura son:

- *Tres sumadores/restadores de coma fija.* Cada uno de ellos está asociado a una coordenada. En función del resultado de la evaluación del signo se decide si se debe realizar una suma o resta en cada elemento.

- *Circuito evaluador del signo de comparación*. En función del modo de funcionamiento y del resultado de la iteración anterior, se decide si se deben hacer sumas o restas en las coordenadas x, y, z.

- *Registros de desplazamiento*. Al tratarse de una arquitectura CORDIC paralela, el desplazamiento que debe hacerse en cada celda es constante. Esta característica elimina los *barrel shifter* típicos de CORDIC serie, permitiendo así realizar en menos de un ciclo de reloj dicho desplazamiento. En función de la posición en la que esté ubicada cada celda se hará un desplazamiento fijo.

- *Registro del modo*. Con idea de poder trabajar simultáneamente dentro del CORDIC general, en modo rotación y vectorización, se transmite a la etapa posterior el modo de funcionamiento.

Con respecto a las características que permite este módulo, cabe destacar:

- *Tamaño de los datos*: El diseño realizado está abierto a cualquier tamaño de datos. Sin embargo, el tamaño de éstos no puede cambiarse dinámicamente. También es importante hacer notar que el tamaño de datos impone el número de celdas básicas.

- *Tipo de datos*: Los datos de entrada están codificados en coma fija. Sin embargo éstos no están normalizados, situación habitual de la mayoría de los módulos CORDIC convencionales [Antelo, 1995]. Por el contrario, los valores que maneja la coordenada z sí están codificados en *2QN*, ya que el ángulo sólo oscila entre $-\pi$ y π radianes.

- *Rango de ángulos*: El módulo diseñado permite trabajar con rotaciones entre $\pm\pi$ radianes.

- *Cuadrantes de funcionamiento*: El módulo funciona correctamente en los cuatros cuadrantes. Sin embargo, en la práctica sólo opera en el primer y cuarto. Si el vector a rotar estuviera ubicado en el segundo o tercer cuadrante se realizaría un cambio de coordenadas (signo) para obtener su equivalente en el primer o cuarto cuadrante.

- *Número de etapas*: El número de etapas serie que posee este módulo, coincide con el tamaño de los datos de entrada seleccionado (n). Este número de datos es el óptimo desde un punto de vista de exactitud en relación al hardware empleado. Para llegar a esta afirmación se han realizado dos simulaciones: en la Figura 4.43 se presenta el error cuadrático medio (e_{SQRT}) (4.50), del error cometido en modo vectorización para un ancho de palabra constante (18 bits). En este caso, el resultado del error máximo normalizado (e_{MAX}) (4.49) es similar al e_{SQRT}. Por su parte, la Figura 4.44 presenta el e_{MAX} y e_{SQRT} para modo

rotación. El cálculo de e_{MAX} así como del e_{SQRT}, en los análisis realizados en este apartado, son el resultado de comparar el valor obtenido del módulo diseñado en VHDL (coma fija), con los resultados obtenidos en MATLAB, mediante la simulación en coma flotante de un módulo CORDIC con comportamiento idéntico al de VHDL. En ambos errores se ha realizado una normalización con respecto al rango dinámico del ángulo (4.48). Observando los resultados de la Figura 4.43 y Figura 4.44 se confirma que para el caso de 18 bits el e_{SQRT} mínimo se alcanza en 18 iteraciones para modo rotación, mientras que en vectorización el error se hace mínimo en 17 iteraciones. Esto permite disponer de los datos de salida en este último modo un ciclo de reloj antes que en rotación. Esta situación se repite para otros tamaños de palabras, es decir, para un tamaño n, el número de iteraciones óptimo sería n. En este trabajo, tal y como se justifica posteriormente, el tamaño de datos de entrada se establece en 18 bits.

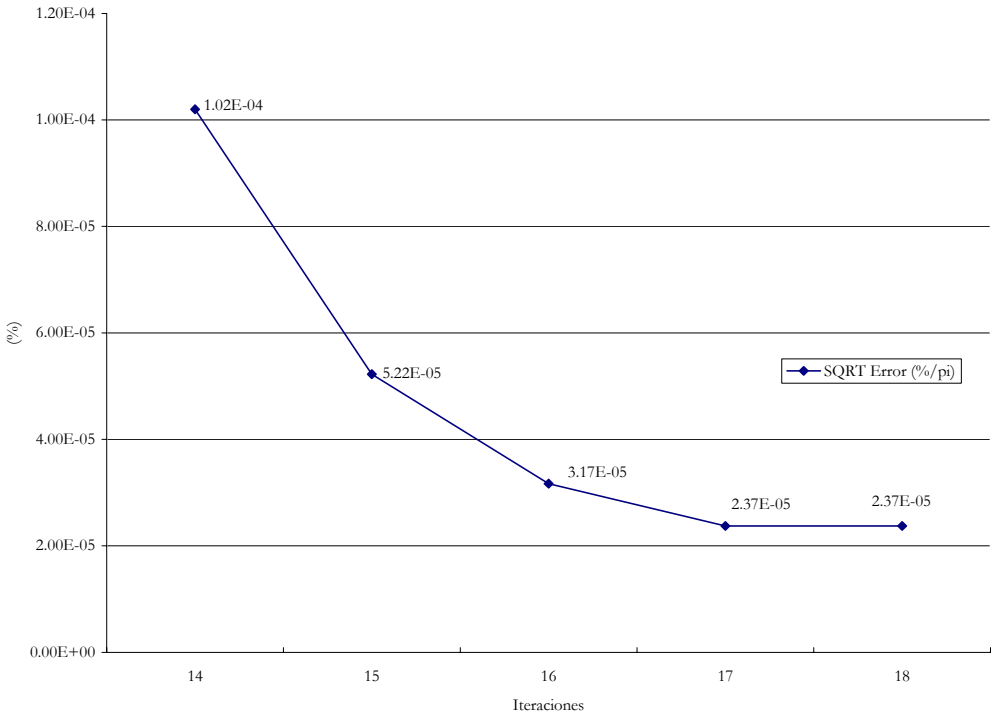


Figura 4.43. Error cuadrático medio para datos de entrada de 18 bits en función del número de iteraciones del módulo CORDIC en modo vectorización.

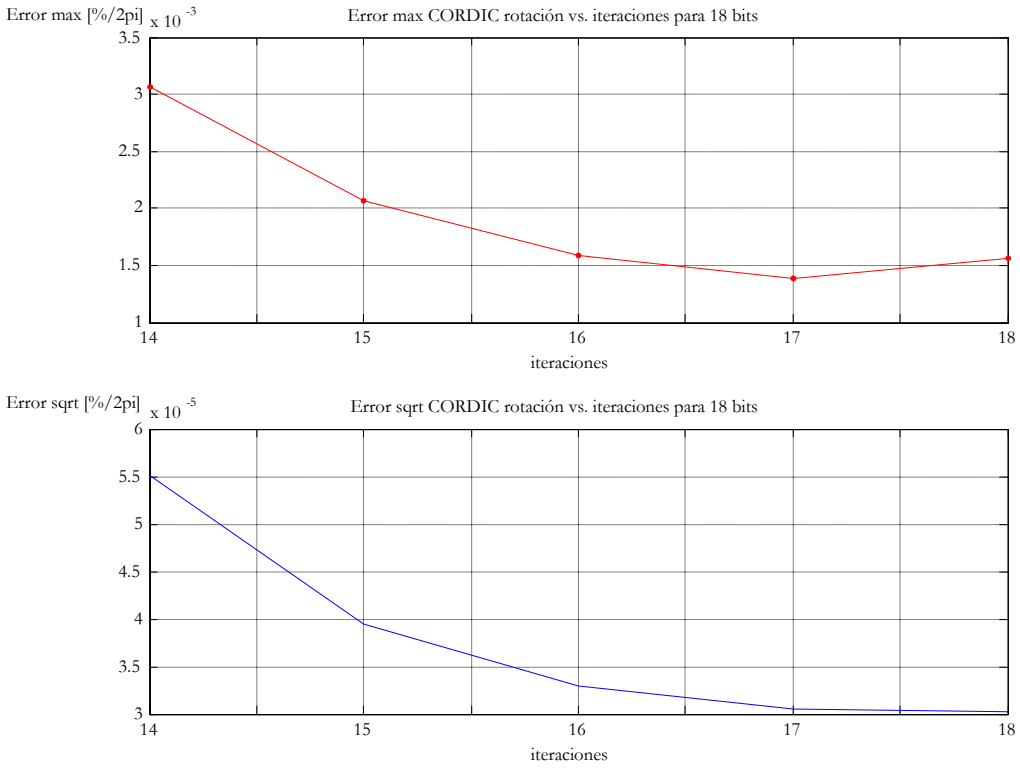


Figura 4.44. Error cuadrático medio y máximo para datos de entrada de 18 bits en función del número de iteraciones del módulo CORDIC en modo rotación.

4.2.3.10.1. Tiempos de ejecución.

Con respecto al tiempo consumido por los módulos CORDIC cabe, de nuevo, distinguir entre el caso de vectorización y el de rotación. En vectorización, el tiempo de ejecución es inferior al de rotación, ya que el resultado final en rotación se obtiene en la iteración n, mientras que en vectorización se alcanza en la $n-1$. Esto es debido a que en modo vectorización, la coordenada z (cálculo del ángulo) converge al resultado final una etapa antes que la x e y. Según se observa en la Figura 4.43, el error en la penúltima iteración coincide con el de la última, por lo que el resultado final (ángulo) se puede obtener en la iteración $n-1$. Cada una de las n etapas que forman el módulo CORDIC necesitan un ciclo de reloj para poder realizar las operaciones aritméticas descritas en CORDIC (ver Figura 4.41). Por tanto, el tiempo total que necesita el módulo CORDIC en modo vectorización (T_{MV}) es:

$$T_{MV} = L_i + (n-1)\cdot T_{CLK} \qquad (4.54)$$

donde L_i es el valor de la latencia inicial que tarda el sistema en introducir los datos al módulo CORDIC, n es el número de etapas internas del CORDC y T_{CLK} la frecuencia de reloj a la que trabaja el módulo CORDIC.

Con respecto al modo rotación, el resultado final se obtiene en la última iteración. Esto se puede comprobar en la Figura 4.44, dónde se observa que en la

última iteración el e_{SQRT} es el menor. Una vez finalizada la última etapa del módulo CORDIC, el resultado de las coordenadas x, y debe ser multiplicado por el factor de corrección (K_m). Por tanto, se debe añadir un tiempo adicional (T_{MULT}) para poder realizar esta operación. La expresión (4.55) presenta el tiempo total consumido en el modo rotación (T_{MR}), en función del número de bits de los datos de entrada (n).

$$T_{MR} = L_i + n \cdot T_{CLK} + T_{MULT} \tag{4.55}$$

donde T_{MULT} es el tiempo que se tarda en realizar la multiplicación del factor de corrección.

4.2.3.10.2. Elección del tamaño de datos de entrada.

La elección del tamaño de datos es fundamental desde dos puntos de vista: exactitud del sistema y recursos consumidos. Este apartado se centra en el estudio de la exactitud del sistema, obteniéndose un número de bits óptimo. Posteriormente, se presentarán los recursos consumidos para diferentes tamaños de bits. Analizando ambos resultados se podrá elegir un tamaño de datos ideal.

La búsqueda del tamaño óptimo, desde un punto de vista de exactitud, se determina evaluando el e_{SQRT} y el e_{MAX} obtenido en: la cuantificación de los datos de entrada, en la variación del ángulo a rotar, en la codificación de un dato de entrada en diferentes tamaños, y en la aplicación de redondeo en diferentes partes del CORDIC.

Otro aspecto a tener en cuenta en la elección del tamaño óptimo de datos es la eliminación de cualquier situación de *overflow* en las coordenadas x, y. Este *overflow* surge ante las diferentes sumas que se realizan en el interior del módulo CORDIC. Para eliminarlo se acota el valor máximo que puede tomar el módulo de las coordenadas x, y. Tal y como se comentó en apartados anteriores, según [Kota, 1993] y [Walther, 1971], es necesario un número mínimo de bits de salvaguarda. Esta salvaguarda se traduce en una limitación del valor máximo del módulo en función del número de bits. Tal y como se verá posteriormente (ver Figura 4.55), tras realizar diferentes simulaciones del módulo CORDIC diseñado, se llega a la conclusión que para éste el número de bits de salvaguarda debe ser 2. Es decir, el valor máximo del módulo que se puede aplicar al módulo CORDIC diseñado no debe exceder 2^{n-2}, siendo n el número de bits de los datos de entrada.

La elección del tipo de representación de los datos condiciona la exactitud del sistema. La forma más habitual empleada en aplicaciones con CORDIC para FPGAs, ha sido aritmética en coma fija. Sin embargo, actualmente se está comenzando a emplear representación en coma flotante [Zhuo, 2004] [Paschalakis, 2003]. La elección de este último tipo de representación aporta una mayor exactitud a costa de incrementar la dificultad de la arquitectura. En esta tesis se ha optado por coma fija ya que el diseño de una arquitectura en coma flotante, implica una alta complejidad

que no se justifica en la mejora de prestaciones, ya que tal y como se expondrá posteriormente el error cometido en coma fija con respecto a coma flotante (simulaciones realizadas en MATLAB) es relativamente pequeño. A continuación se analizan diferentes errores asociados al módulo CORDIC. De su estudio se podrá postular un tamaño de datos ideal.

4.2.3.10.3. Error de cuantificación de los datos de entrada.

El primer error que se va a analizar, denominado habitualmente como de cuantificación, es el cometido al convertir los datos de entrada representados en coma flotante. También se incluye dentro de este tipo de error, al producido por el truncamiento de los datos de entrada codificados en coma fija con un tamaño superior al manejado. Sin embargo, este error en función de la aplicación, puede ser considerado o no. En nuestra aplicación final los datos iniciales son generados desde el sensor CMOS en forma digital. Por tanto, el error de cuantificación está asociado al sensor y no será considerado en las diferentes etapas de PCA incluida la de CORDIC, ya que en todo momento trabaja con los datos cuantificados por el sensor. Sin embargo, se va a estudiar la influencia de este tipo de error en el módulo CORDIC diseñado para analizar su comportamiento en un hipotético caso de uso de datos cuantificados de coma flotante a fija. De esta forma queda modelado dicho error por si se reutilizara en otras aplicaciones en las que sí se tuviera que tener en cuenta éste. Los datos de entrada han sido cuantificados en diferentes tamaños e introducidos al módulo CORDIC. Así, se han obtenido una serie de resultados que han sido comparados con los obtenidos en MATLAB en coma flotante, para un módulo CORDIC de características idénticas al diseñado en VHDL.

Concretamente, se han realizado dos ensayos para analizar dicho efecto: por una parte, se ha simulado la situación en la que los datos de entrada son truncados a 18 bits (Figura 4.45), situación que denominaremos *sin redondeo*. Mientras que por otra, se ha simulado el uso de *redondeo* en vez de truncamiento sobre los datos de entrada (Figura 4.46). En ambos casos, el error se presenta en tanto por ciento para datos de entrada de 18 bits introducidos al módulo CORDIC en modo vectorización. También se ha simulado esta situación para modo rotación, obteniéndose unos resultados análogos a los mostrados en vectorización. Con respecto al valor del módulo de los datos de entrada, éste es constante siendo su valor 2^{16}. Con este valor en 18 bits no se produce *overflow* en ningún momento. Los datos de entrada han sido sometidos a un barrido de ángulos entre $\pm\pi$ radianes.

Las formas de onda de dichas gráficas son muy similares, y en ambas se puede observar como el error máximo es bastante bajo (inferior a $\pm 2 \cdot 10^{-3}\%$). Si analizamos la Figura 4.46 de manera detallada (Figura 4.47), se comprueba como la forma de onda resultante corresponde a un error típico de cuantificación, existiendo básicamente un error continuo y constante a excepción de determinados puntos donde el error se duplica. Estos puntos son aquellos en los que la cuantificación de la entrada coincide con un valor múltiplo del escalón de cuantificación.

Como conclusión de este primer error estudiado, podemos decir que la influencia del error de cuantificación en el módulo CORDIC diseñado, introduciría un error inferior a $\pm 2 \cdot 10^{-3}\%$. La aplicación de redondeo en los datos de entrada en vez de truncamiento, no aporta mucha mejora ya que el error cometido es semejante. Esto se debe fundamentalmente al hecho de trabajar con datos de un tamaño elevado, donde el efecto de truncamiento o redondeo en el último bit apenas varía el valor final. Por tanto, no compensa aplicar redondeo ya que la lógica adicional que llevaría asociada incrementa los recursos hardware consumidos en la FPGA.

En el análisis realizado se ha tomado 18 bits como tamaño patrón. Si se utilizara otro tamaño los resultados generados son equivalentes. Sirva como ejemplo la Figura 4.48. En esta figura, se presenta el error cometido para 20 bits sin aplicar redondeo. La forma de onda y los valores del error son parecidos a los mostrados para 18 bits bajo las mismas situaciones. En este caso el error disminuye con respecto a 18 bits, al tener sobre el valor final menor importancia el redondeo o truncamiento en el bit de menor peso. Por tanto, este error aumenta a medida que se disminuye el tamaño de la palabra.

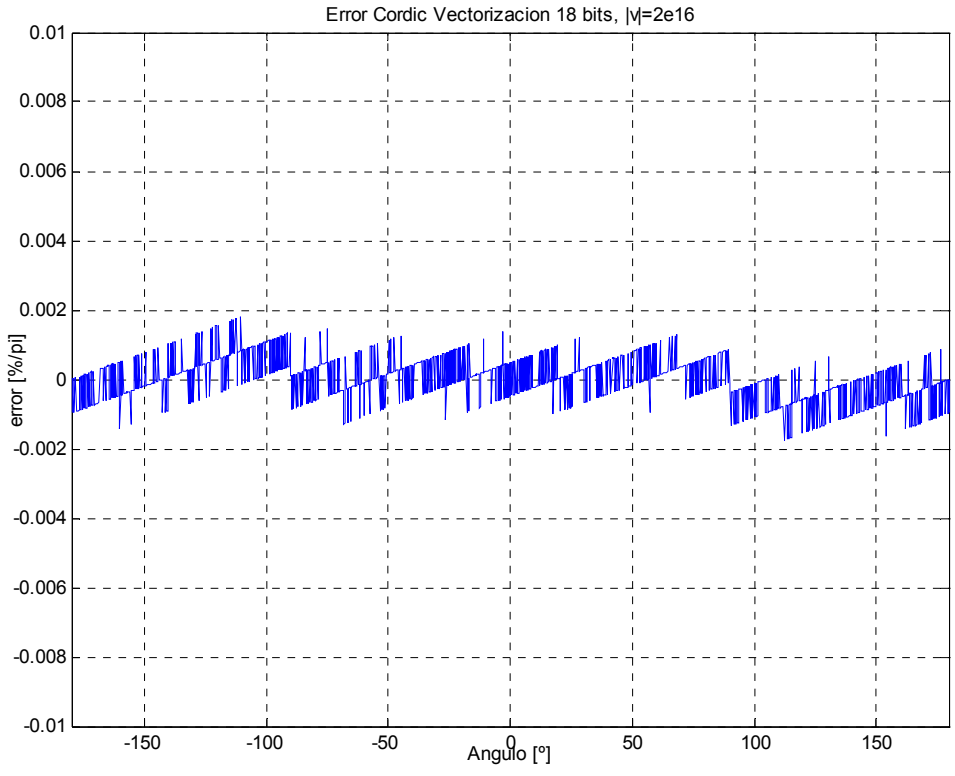


Figura 4.45. Error de ángulo normalizado en módulo CORDIC para 18 bits en modo vectorización en función de una distribución aleatoria de ángulos entre $\pm\pi$ radianes sin aplicar redondeo.

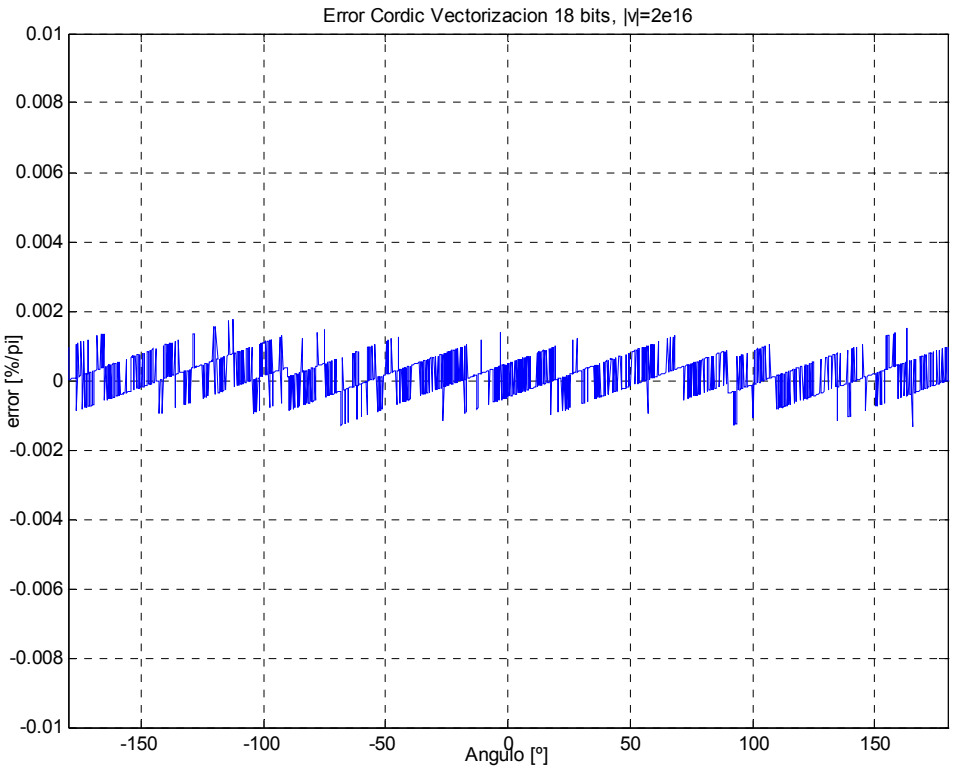


Figura 4.46. Error de ángulo normalizado en módulo CORDIC para 18 bits en modo vectorización en función de una distribución aleatoria de ángulos entre $\pm\pi$ radianes aplicando redondeo.

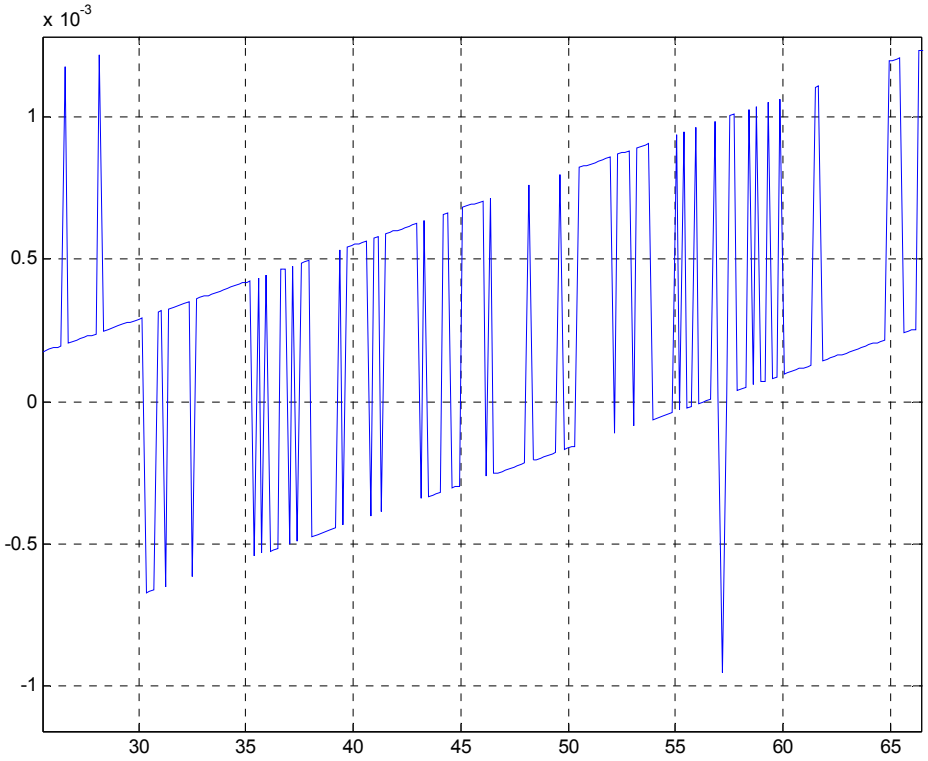


Figura 4.47. Detalle del error de cuantificación de la Figura 4.46.

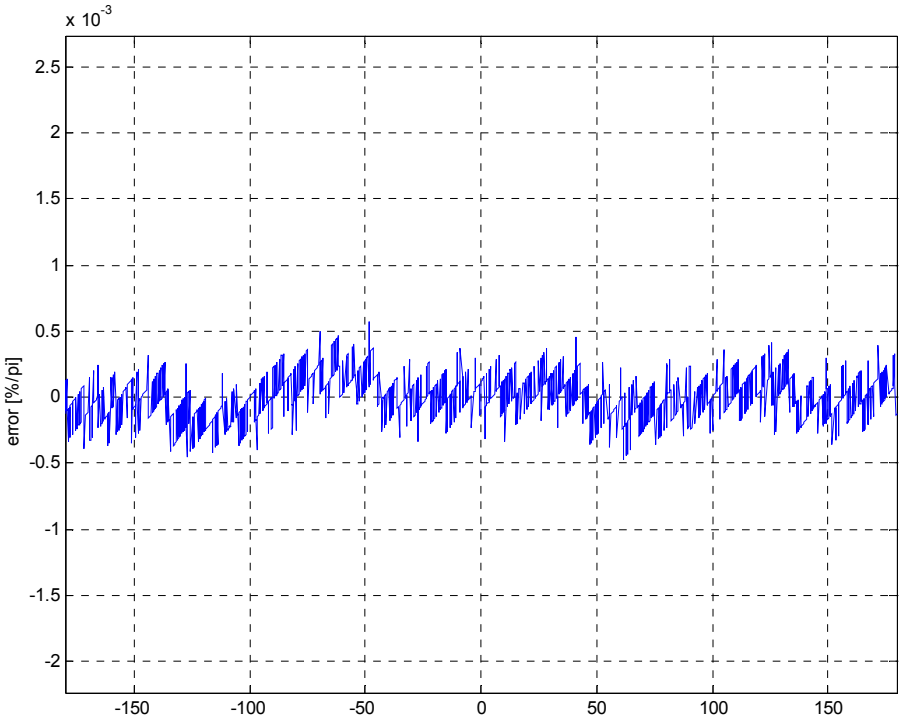


Figura 4.48. Error de ángulo normalizado en módulo CORDIC para 20 bits en modo vectorización en función de una distribución aleatoria de ángulos entre $\pm\pi$ radianes sin aplicar redondeo.

4.2.3.10.4. Error debido a la variación del ángulo a rotar.

Otra fuente de error que recogen diversos trabajos [Hampson, 2002] [Hu, 1992] [Antelo, 1997], es el error que se comete al aplicar al CORDIC diferentes ángulos iniciales así como diferentes ángulos de rotación. Así, tal y como se muestra en la Figura 4.49 y en la Figura 4.52, se ha aplicado un barrido de ángulos iniciales al módulo CORDIC desarrollado en modo rotación y vectorización, fijando el número interno de etapas al tamaño de los datos de entrada, con objeto de analizar su respuesta ante cualquier ángulo de entrada. Se puede observar, en ambas figuras, como el error aparece de forma homogénea para todo el barrido de ángulos, siendo además de un valor muy reducido. Este error es debido a la exactitud con la que CORDIC converge hacia el valor final. En función de los datos de entrada y de los ángulos, se converge hacia el valor final con un cierto error. Este error, no se puede eliminar ni reducir. Sin embargo, los resultados mostrados de la Figura 4.50 y de la Figura 4.51 donde se presenta un resumen del error cometido para diferentes tamaños de bits bajo las mismas condiciones de análisis anteriores, se deduce que al aumentar el tamaño de bits el error máximo y cuadrático medio disminuye. Por tanto, a mayor tamaño de datos menor error de este tipo se comete.

4.2.3.10.5. Error debido a diferentes tamaños de datos.

Una manera rápida de determinar el tamaño óptimo de los datos de entrada es analizar el comportamiento tanto en modo rotación como en vectorización, de un dato de entrada sometido a un barrido exhaustivo de ángulos de rotación. Para ello, se han simulado en los dos modos de CORDIC varios datos de entrada codificados

en diferentes tamaños de bits, a los que se les ha aplicado un barrido de ángulos iniciales y de rotación. El criterio empleado para elegir el valor del dato de entrada ha sido determinado de forma empírica. Concretamente, se ha buscado el valor máximo de entrada que no produjera *overflow*. De esta forma se analizaron e_{SQRT} y e_{MAX} para tamaños de 14, 15, 16, 17, 18, 19 y 20 bits, diferenciando entre el modo rotación y vectorización.

En el caso de rotación, los valores de entrada han sido analizados sobre diferentes ángulos iniciales de entrada y con una distribución aleatoria de ángulos de rotación entre $\pm\pi$ radianes. Para ver el error generado en VHDL, particularizado para una de estas condiciones con respecto al valor obtenido en MATLAB, se presenta la Figura 4.49. En ella se puede observar el error cometido al someter a un vector de entrada de módulo constante 2^{18}, con un ángulo inicial comprendido entre $\pm\pi$ radianes, a un ángulo de rotación aleatorio, comprendido en el rango de $\pm\pi/2$ radianes. Se puede comprobar, como también en este caso el error obtenido es bastante pequeño alcanzando un valor máximo de $8\cdot10^{-4}\%$.

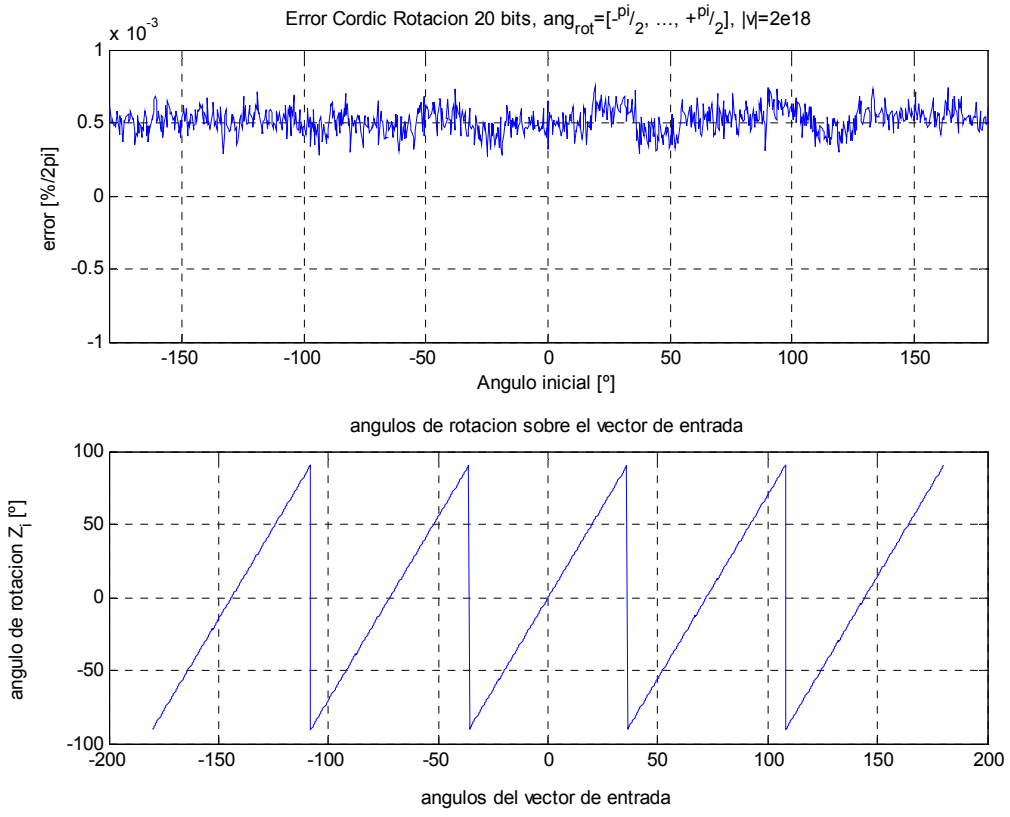


Figura 4.49. Error de CORDIC en modo rotación para un vector de entrada constante con diferentes ángulos de entrada sometido a una distribución aleatoria de ángulos de rotación para 20 bits.

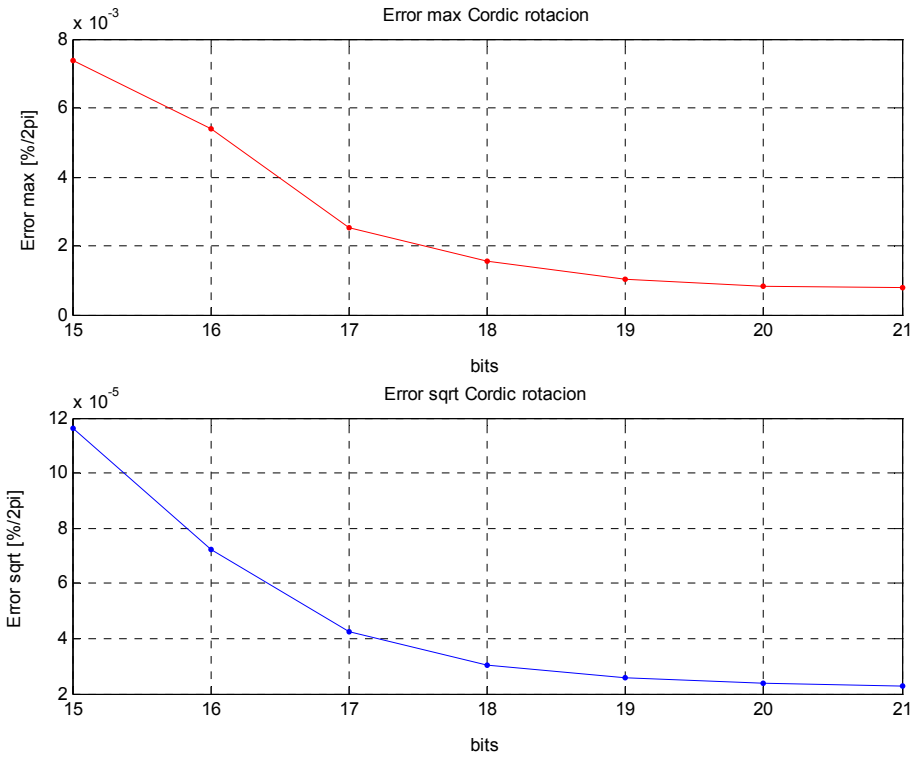


Figura 4.50. Error máximo y cuadrático medio, de rotación para diferentes tamaños de bits, con vectores de entrada de valor máximo permitido en cada caso, para diferentes ángulos iniciales y sometidos a una distribución aleatoria de ángulos de rotación.

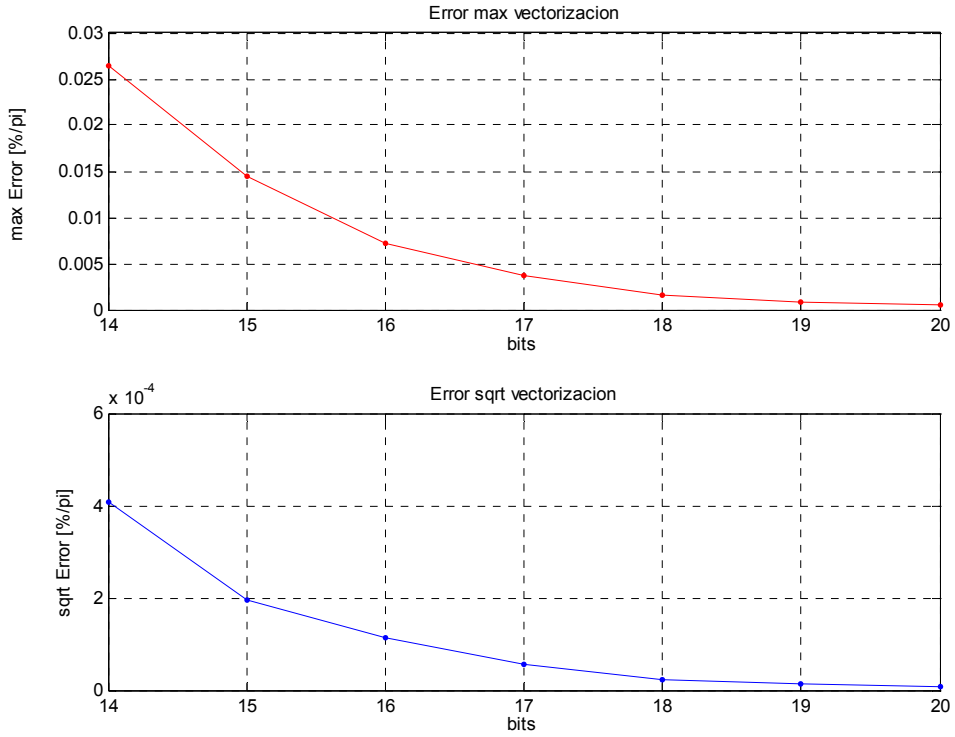


Figura 4.51. Error máximo y cuadrático medio, de vectorización para diferentes tamaños de bits, con vectores de entrada de valor máximo permitido en cada caso, para diferentes ángulos iniciales y sometidos a una distribución aleatoria de ángulos de rotación.

La Figura 4.50 presenta un resumen del e_{SQRT} y e_{MAX} para todos los tamaños de bits expuestos anteriormente, donde el valor del módulo de los datos de entrada es en cada caso el máximo permitido (2^{n-2}), y donde la variación del ángulo inicial y de rotación es idéntica en cada caso al expuesto en la Figura 4.49. Se puede observar como a medida que aumenta el tamaño de bits, tanto el e_{SQRT} como el e_{MAX} disminuyen casi de forma exponencial. Se puede comprobar como en la Figura 4.50 a partir de 18 bits ambos errores poseen valores muy bajos (inferiores a $15 \cdot 10^{-4}\%$) no produciéndose variaciones importantes de 18 a 21 bits.

Con respecto al modo vectorización, se han realizado diferentes simulaciones para los mismos tamaños de bits y para los mismos valores del módulo de entrada. En este caso, a los vectores de entrada se les ha aplicado un ángulo inicial que abarca $\pm \pi/2$ radianes. A modo de ejemplo se puede ver el error cometido en el caso de 20 bits en la Figura 4.52. En este caso, el error no alcanza el $6 \cdot 10^{-4}\%$ siendo por tanto inferior al generado para el mismo tamaño en modo rotación. Como resumen de todos los ensayos se presenta la Figura 4.51. Ésta muestra el e_{SQRT} y e_{MAX} para los diferentes tamaños de bits. En este caso, también a medida que aumenta el tamaño de bits, disminuye de una forma casi exponencial el error cometido. También, se puede observar como a partir de 18 bits el error es muy bajo, apenas variando entre 18, 19 y 20 bits.

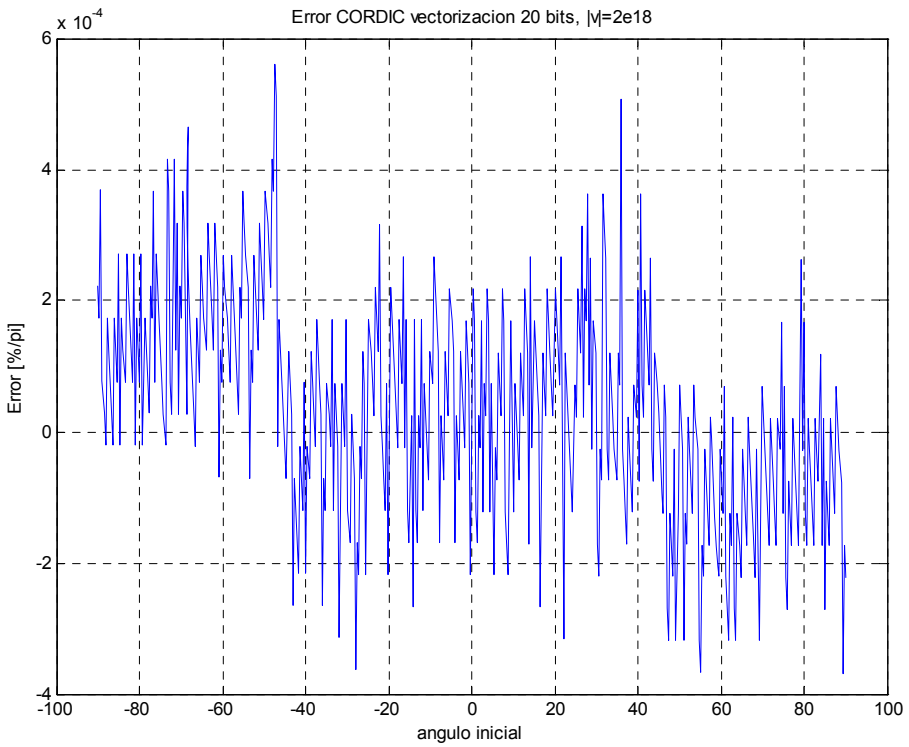


Figura 4.52. Error de vectorización para 20 bits y módulo de entrada 2^{18} para diferentes ángulos iniciales.

4.2.3.10.6. Error cometido con datos mayores de 18 bits.

En modo rotación se debe realizar el denominado factor de escalado [Antelo, 1997]. Esto implica el uso de multiplicadores que se encarguen de realizar dicha operación. En nuestro caso, se han empleado multiplicadores hardware de Xilinx cuyo tamaño máximo de datos de entrada es de 18 bits. Si los datos de entrada al CORDIC poseen un tamaño superior a 18 bits, implica que a la entrada del multiplicador se debe realizar un truncamiento o redondeo de los datos.

La aplicación de un redondeo o truncamiento implica la pérdida de información. Por esta razón, se ha realizado un estudio para el caso de manejar datos mayores de 18 bits, para comparar los resultados producidos mediante el empleo de multiplicadores hardware de 18x18 bits para las coordenadas x, y, aplicando el correspondiente truncamiento o redondeo, con respecto a un bloque de multiplicación que permite trabajar con datos mayores de 18 bits, sin necesidad de redondear o truncar sus entradas. Este bloque está compuesto por un multiplicador de 18x18 bits para la parte alta, otro para la parte baja y posteriormente un sumador, que junto con la lógica adecuada, ofrece el resultado correcto.

La Figura 4.51 presenta los resultados generados mediante el empleo de un multiplicador de 18x18 bits para las coordenadas x, y, truncando los datos a la entrada de los multiplicadores. Por su parte, la Figura 4.53 presenta los resultados para el caso de emplear un bloque de multiplicación para datos mayores de 18 bits, sin aplicar truncamiento o redondeo.

Comparando las dos gráficas se construye la Figura 4.54. En ella se puede comprobar que aunque existe una leve mejora en el e_{MAX} empleando multiplicadores superiores a 18x18 bits, en el e_{SQRT} apenas existe mejora. Esta es la razón por la que se deshecha el empleo de un bloque multiplicador, para tamaños de datos superiores a 18 bits.

4.2.3.10.7. Error de redondeo de datos intermedios.

Debido a las diferentes operaciones aritméticas que se realizan en el módulo CORDIC, ciertos resultados intermedios son superiores al tamaño escogido, provocando truncamientos de datos. Estos truncamientos también producen un error sobre el resultado final. Por ello, se evalúa el error introducido por esos truncamientos. Una forma de disminuir el error asociado a los truncamientos, es la aplicación de redondeo en vez de truncamiento. Sin embargo, la aplicación de redondeo conlleva el empleo de más recursos internos y la disminución de la frecuencia máxima de reloj. Por tanto, se debe sopesar la ventaja/desventaja del empleo de redondeo en vez de truncamiento.

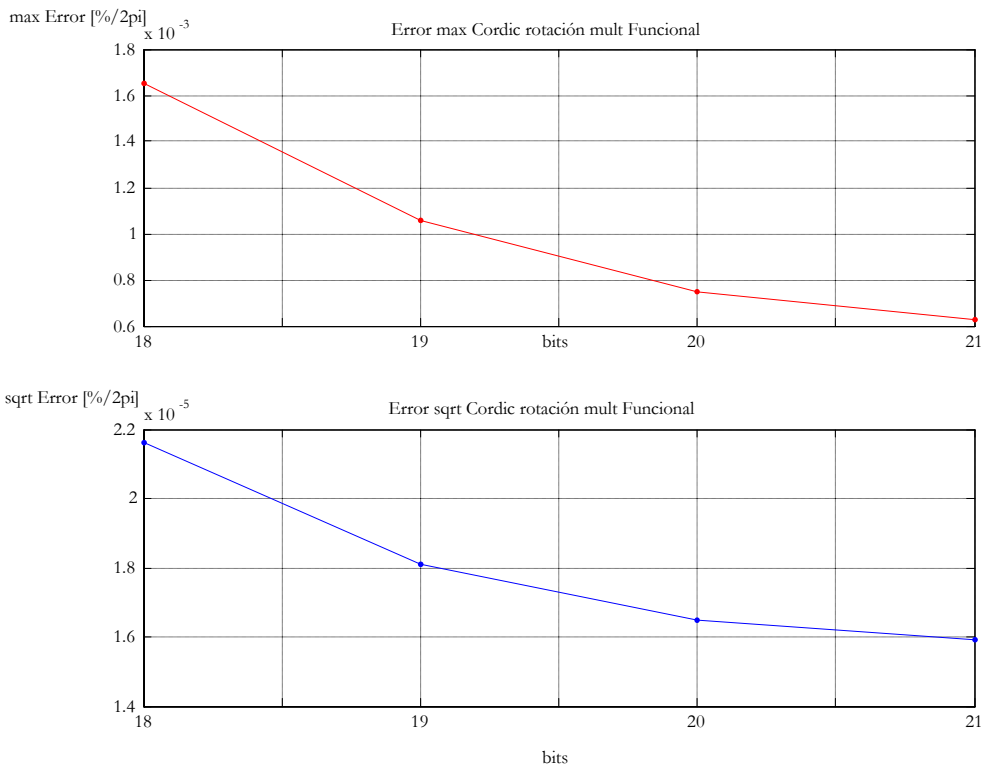


Figura 4.53. Error en CORDIC rotación empleando una bloque de multiplicación, para datos de entrada superiores a 18 bits.

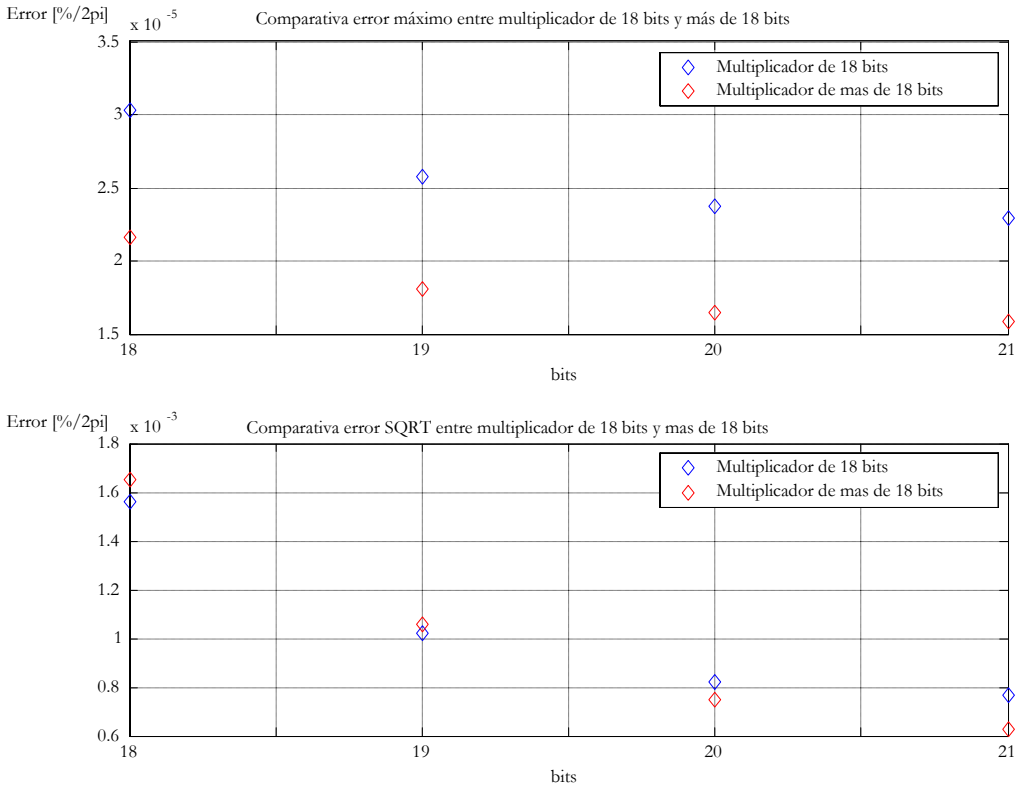


Figura 4.54. Comparativa del error cuadrático medio y máximo en función del número de bits de un CORDIC con multiplicadores de 18x18 bits y otro con tamaño superior.

Dentro del módulo CORDIC diseñado, existen tres situaciones en las que se debe realizar dicho truncamiento:

- *Desplazamientos internos de las coordenadas* x, y. Empleando aritmética en coma fija, un módulo CORDIC genera un error de truncamiento adicional cuando su coordenada x o y posee componentes negativas [Hampson, 2002]. Esto es debido al desplazamiento a derecha sobre los datos de entrada que el módulo CORDIC realiza internamente en cada celda. Para ver el efecto que produce este desplazamiento, supongamos que sobre un dato positivo se aplicara un desplazamiento continuo; teóricamente el resultado final debería tender a cero. Si el número es positivo realmente tiende hacia cero. Sin embargo, si el número es negativo al realizar un desplazamiento iterativo, éste tendería hacia el valor $0xFF$, que equivale a -1. Esto es un problema en CORDIC, ya que al ser un proceso iterativo el error se iría acumulando. Una alternativa para eliminar ese error, pasa por redondear el valor desplazado en vez de truncarlo, por ejemplo, añadiendo el bit eliminado en el desplazamiento al valor desplazado. Esta solución no requiere de un hardware adicional muy costoso, ya que en los sumadores internos del CORDIC simplemente hay que dotarlos de una entrada de acarreo. A este redondeo se le denominará en lo que sigue como *redondeo1*.

- *Entrada del multiplicador del factor de corrección.* La salida de las coordenadas x, y de la última celda CORDIC, se debe acotar a un tamaño máximo de 18 bits que es el máximo que admiten los multiplicadores hardware de las FPGAs de Xilinx. A este redondeo, se le denominará posteriormente como *redondeo2*.

- *Salida del multiplicador del factor de corrección.* La salida del multiplicador será como máximo de 36 bits. Sin embargo, si se está trabajando con datos de entrada de n bits la salida se debe reducir también a ese valor. Este último redondeo será denominado como *redondeo3*.

Todos estos errores aumentan cuando se hace un empleo recursivo de los módulos CORDIC. Por esta razón no se presentan resultados del efecto de dichos redondeos sobre una celda CORDIC, presentándose en la sección de resultados de la arquitectura propuesta basada en CORDIC para el cálculo de autovalores y autovectores, el efecto de dichos redondeos sobre los autovalores y autovectores finales.

4.2.3.10.8. Error debido al tamaño del módulo de los datos de entrada

Anteriormente se ha expuesto que el tamaño máximo del módulo de vector de entrada no debe superar $2^{(n-2)}$, ya que si no se produciría *overflow*. Sin embargo, la pregunta que ahora se plantea es: ¿cómo se comporta CORDIC para datos con módulo inferior a $2^{(n-2)}$? Para dar respuesta a esta cuestión se han realizado diferentes simulaciones. Concretamente, en la Figura 4.55 se presenta el resultado del error cometido por el módulo CORDIC en modo vectorización con 18 bits, para tamaños de módulo de entrada que oscilan entre un valor mínimo 2^7 y un máximo de 2^{16}. Por su parte, la Figura 4.56 presenta el resultado en modo rotación bajo las mismas condiciones que en vectorización. Se observa como a medida que aumenta el valor del módulo, disminuye el error, siendo prácticamente nulo a partir de 14 bits.

Como conclusión se puede decir que la elección del tamaño de bits también debe ajustarse en función del rango de valores a manejar, intentando en todo momento que el valor del módulo manejado sea lo más próximo a 2^{n-2}.

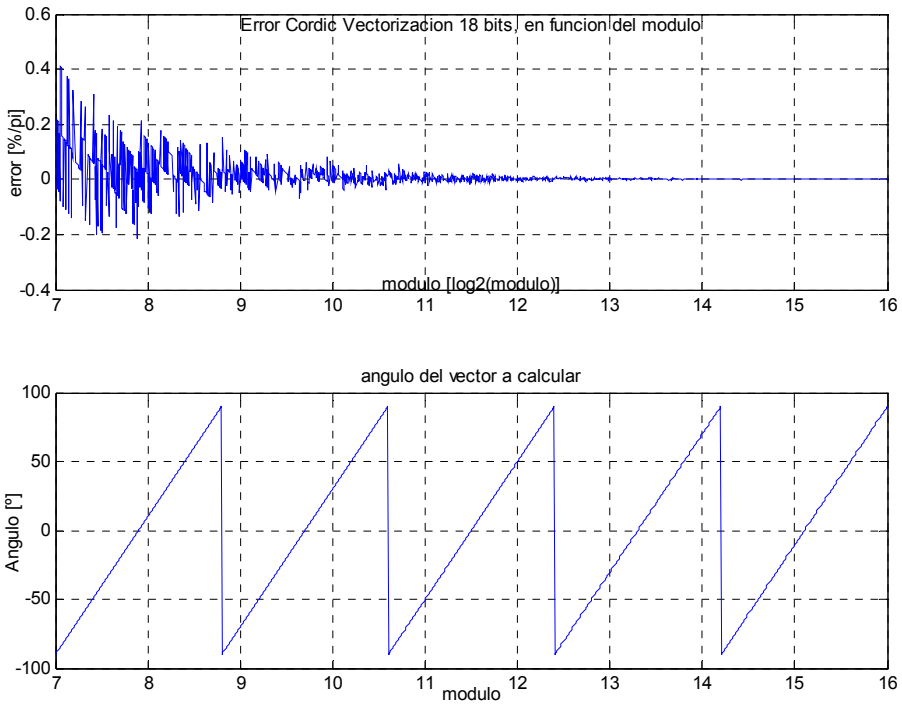


Figura 4.55. Error del módulo de entrada, para datos de 18 bits en modo vectorización evaluado para una distribución aleatoria de ángulos entre ±90º.

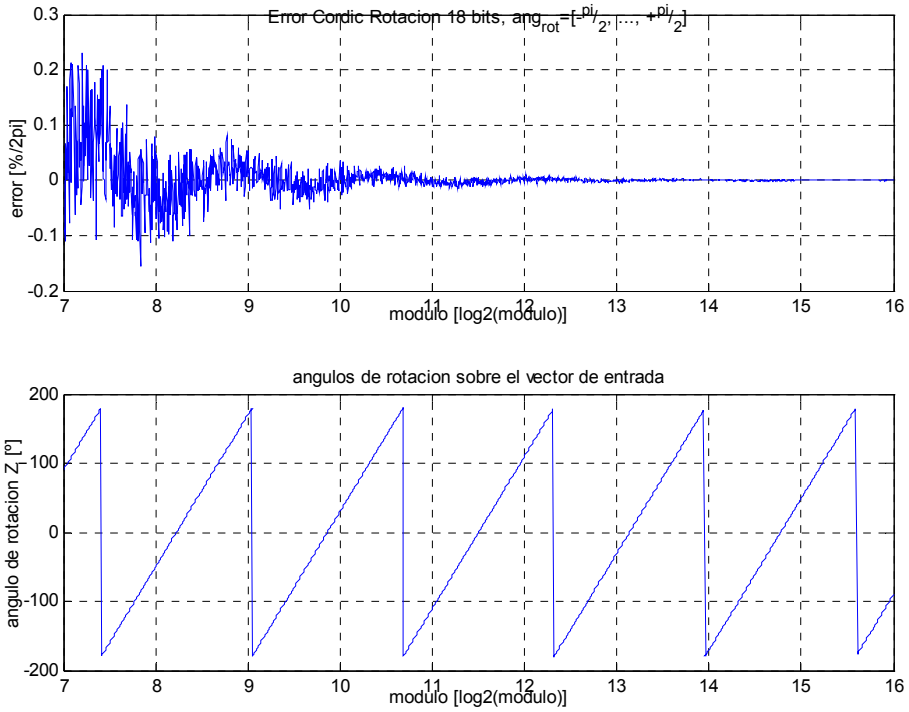


Figura 4.56. Error del módulo de entrada para datos de 18 bits, en modo rotación evaluado para una distribución aleatoria de ángulos entre $\pm \pi/2$ radianes.

4.2.3.10.9. Recursos consumidos por el módulo CORDIC.

A continuación se va a evaluar el número de recursos consumidos por el módulo CORDIC diseñado sobre una FPGA XC2VP7. Este estudio pretende analizar los recursos consumidos en una FPGA, en función del tipo de redondeo aplicado así como el tamaño de los datos de entrada. Por último, se analiza el porcentaje de ocupación de una XC2VP7 si fijando un tamaño de bits se decide variar el número de iteraciones. Únicamente se analiza el número de *slices* consumidos ya que es el único parámetro que varía en función del tamaño de bits y del redondeo aplicado. El número de multiplicadores hardware usados, en todo momento será 2, correspondiendo éstos con el factor de escalado de las coordenadas x, y.

En la Figura 4.57 se presenta el número de *slices* consumidos por el módulo CORDIC diseñado en función del número de bits de los datos de entrada, cuando no se aplica ningún tipo de redondeo intermedio y cuando se aplican los tres tipos de redondeo explicados anteriormente en la sección 4.2.3.10.7. Tras la comparación de estos dos casos, se puede afirmar que el incremento del número de recursos, independientemente del número de bits se establece entorno a 7.5%.

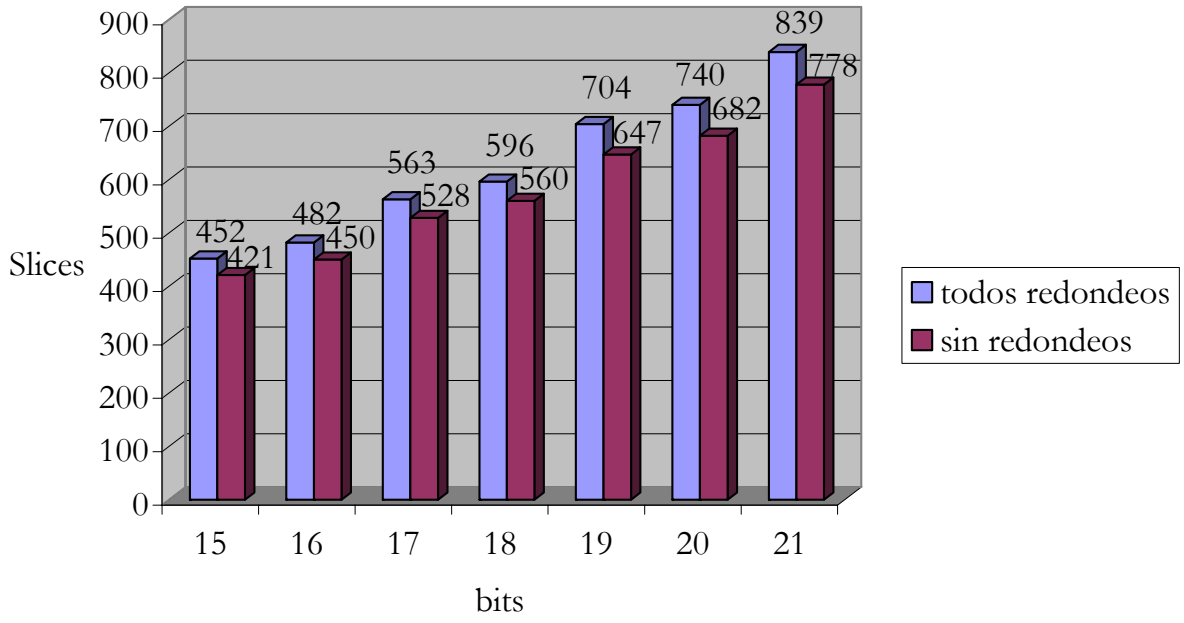


Figura 4.57. Recursos consumidos por el módulo CORDIC diseñado, en función del número de bits de los datos de entrada, aplicando todos y ningún redondeo.

A continuación se evalúa el incremento que lleva asociado cada tipo de redondeo. Así, en la Figura 4.58 se presenta el número de *slices* consumidos por cada redondeo, incluso combinaciones de ellos, para cada tamaño de bits. A la vista de dicha figura se pueden sacar varias conclusiones:

- Las diferencias entre los diversos redondeos, se traduce en un incremento máximo del 7.5% de recursos usados con respecto a la ausencia de redondeo.
- El redondeo que menos recursos incrementa es el relativo a la entrada del multiplicador del factor de escalado (*redondeo 2*). Esto es así, ya que para datos de tamaño menor o igual a 18 bits no es necesario realizar este redondeo. Mientras que para datos mayores la lógica adicional es mínima.
- El resto de redondeos incrementan el mismo porcentaje de recursos. Esto es debido a que la lógica empleada en ambos redondeos es similar.

Con respecto a la ocupación asociada a las iteraciones de CORDIC, en la Figura 4.59 se presenta el estudio detallado de los recursos consumidos para el redondeo 1 ó 3 (en ambos casos coincide), sin ningún tipo de redondeo y con el redondeo 1+3. En este caso se ha particularizado para 18 bits, por lo que no se ha representado el redondeo 2, ya que para este tamaño de datos no existe tal redondeo. Obviamente, a medida que el número de iteraciones se incrementa el número de recursos aumenta linealmente.

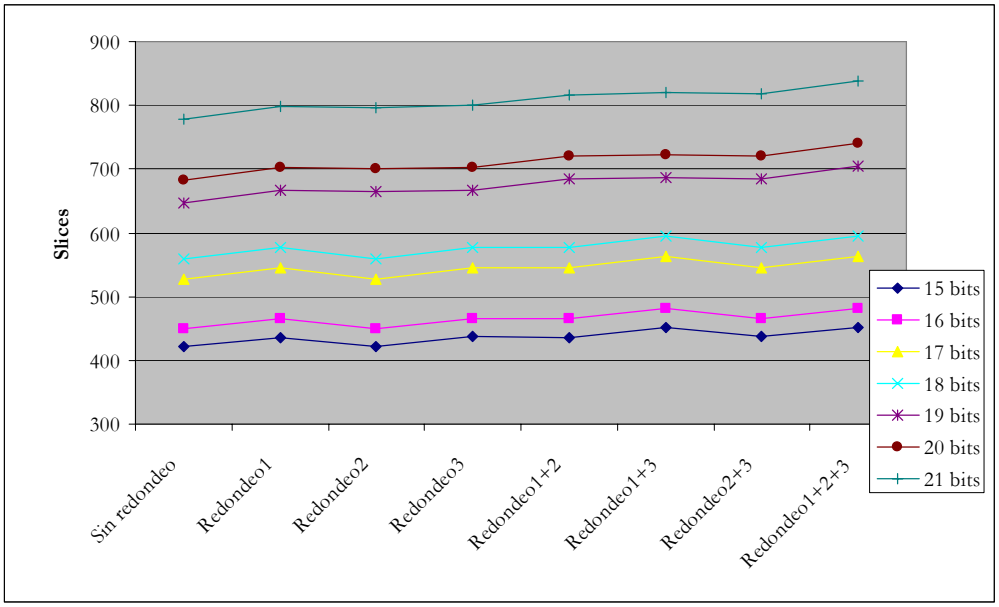


Figura 4.58. Recursos consumidos por el módulo CORDIC diseñado, en función del tipo de redondeo aplicado, para datos desde 15 bits a 21 bits.

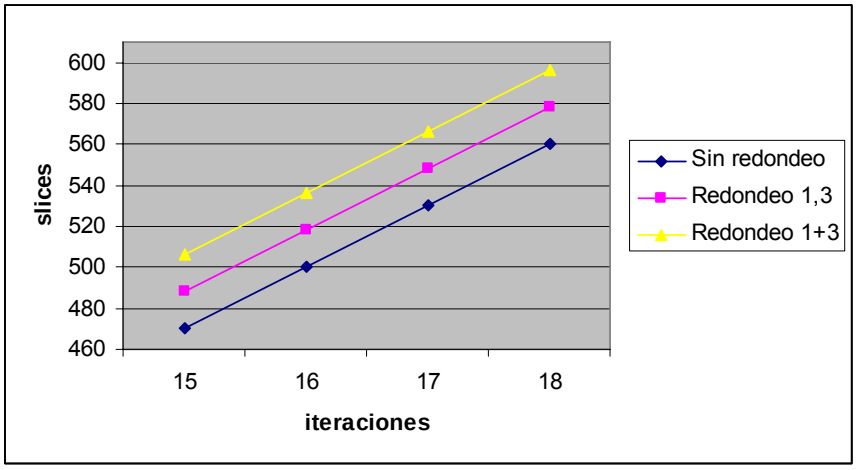


Figura 4.59. Recursos consumidos por cada tipo de redondeo en el módulo CORDIC para 18 bits, con número de iteraciones variables.

4.2.3.10.10. Conclusiones sobre el módulo CORDIC diseñado.

Con objeto de reducir al máximo el número de recursos consumidos, disminuir al máximo el tiempo de ejecución y alcanzar una precisión óptima, se ha descrito y analizado en esta sección el módulo CORDIC de coordenadas circulares, propuesto en esta tesis, funcionando en los dos modos que admite CORDIC. Dada la estructura modular de la propuesta realizada su uso no sólo se limita al cálculo de autovalores y autovectores, sino que puede ser usado en cualquier otra aplicación.

Una vez analizado el número de recursos consumidos y los diferentes tipos de errores que se generan en los módulos CORDIC diseñados, se puede comprobar como existe un compromiso entre número de recursos consumidos y exactitud del sistema. A mayor número de bits en el tamaño de los datos de entrada, mayor precisión se alcanza, a costa de consumir más recursos internos. Sin embargo, a partir

de 18 bits la disminución del error no es tan significativa. Además, la elección de este tamaño no hace necesario el redondeo 2, ya que éste se comienza a aplicar para datos superiores a 18 bits.

Desde un punto de vista de recursos internos el empleo de datos de 18 bits implica un ahorro frente a tamaños de 19 bits, 20 bits o 21 bits, de un 15%, 20% y 40% respectivamente. Por esta razón, junto con la precisión obtenida para este tamaño, se justifica el tamaño de 18 bits empleando 18 iteraciones. En cuanto al tipo de redondeo escogido, la exactitud que cada redondeo tiene asociado se presenta en los resultados de la aplicación final (apartado 4.2.3.12). De esta forma, debido a la naturaleza recursiva del método de Jacobi se puede observar mejor el efecto de dichos redondeos sobre los resultados finales. Sí que es importante destacar que el empleo de todos los tipos de redondeo implica un incremento de un 7.5% en los recursos consumidos, siendo la aportación tanto del redondeo 1 como del 3, de aproximadamente del 3,6%.

4.2.3.11. Reducción del número de autovectores.

En las aplicaciones de PCA se utilizan únicamente los autovectores asociados a los autovalores más significativos, ya que estos autovectores son los que poseen mayor información característica. De esta forma se reduce el volumen de información a tratar.

Para realizar la reducción de autovectores se buscan los t autovalores de mayor peso y posteriormente se seleccionan sus autovectores asociados. Para determinar t, se ha fijado el porcentaje sobre la suma de autovalores obtenidos que se desee tener. Es decir, si por ejemplo se desea encontrar los t autovalores cuya suma alcance el 95% de la suma de todos ellos, se tendría la expresión (4.56).

$$\frac{\sum_{i=1}^{t} \lambda_i}{\sum_{i=1}^{M} \lambda_i} \leq 0.95 \qquad (4.56)$$

A diferencia de otras alternativas, el método de Jacobi no genera ni los autovalores ni los autovectores ordenados de mayor a menor o viceversa. Por tanto, el primer paso para encontrar t es la ordenación de los autovalores en orden ascendente o descendente, así como sus autovectores asociados.

En esta tesis se ha empleado como algoritmo de ordenación el denominado *método de la burbuja* [Wikipedia, 2006]. Este método, ampliamente usado en algoritmos de bases de datos, permite ordenar de forma iterativa una secuencia de valores. Aunque en sistemas basados en microprocesador, debido a su recurrencia, es un sistema lento, en dispositivos hardware su ejecución e implementación es óptima. Para explicar su funcionamiento, apliquemos el siguiente ejemplo: si $M = 8$ y sabiendo que el autovalor λ_7 es el mayor de todos y λ_0 el menor, supongamos que

los autovalores generados mediante Jacobi, se ubican en las posiciones de una memoria (Figura 4.60).

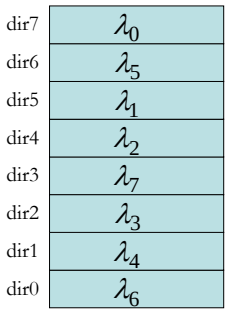


Figura 4.60. Posiciones y contenidos de la memoria con los autovalores obtenidos mediante Jacobi para un ejemplo ilustrativo.

En este caso se va a realizar una ordenación descendente, quedando el autovalor más significativo (λ_7) en la dirección 0, y el menos significativo (λ_0) en la dirección 7. El método de la burbuja se basa en sucesivas comparaciones entre dos datos permutándose sus posiciones en función del resultado de la comparación. Tras hacer todas las comparaciones el dato mayor se almacena en la dirección base de cada iteración.

En la primera iteración, la dirección base (dirección en la que se almacena el dato mayor) es la 0. La primera comparación es realizada entre el dato asociado a la dirección 0 (λ_6) con el de la dirección 1. En el ejemplo de la Figura 4.60 $\lambda_6 > \lambda_4$, por lo que cada dato no varía su posición. La segunda comparación se realiza entre λ_6 y λ_3. Al igual que en la primera comparación ningún dato cambia de posición. Es en la tercera comparación donde se realiza el primer cambio. En este caso, se compara el dato de la dirección 3 (λ_7) y el dato de la dirección 0 (λ_6). Como $\lambda_7 > \lambda_6$ en la dirección base (dirección 0) se almacena λ_7 y en la dirección 3 λ_6. En la cuarta comparación los datos a evaluar son λ_7 y λ_2. Como $\lambda_7 > \lambda_2$ no se realiza ninguna permuta de posiciones. En el ejemplo aplicado, no hay ningún cambio más de posiciones en el resto de comparaciones, por lo que al finalizar la séptima comparación en la dirección 0 queda el dato mayor (λ_7).

Tras finalizar cada iteración, la dirección base aumenta en uno. Así, el objetivo de la segunda iteración es la búsqueda del dato mayor de los seis restantes (los comprendidos desde la dirección 1 a la 7). Tras realizar las comparaciones de forma análoga a la explicada en la iteración 1, en la última comparación, en la dirección 1 (dirección base de la segunda iteración) se almacena el segundo dato mayor (λ_6).

Al igual que en las dos iteraciones anteriores, el proceso se repetiría durante otras 5 iteraciones. El número total de comparaciones que se realiza para ordenar los 8 valores de la Figura 4.60, asciende a 28.

Para aplicar el método de la burbuja a la ordenación de autovalores en una FPGA, se ha implementado el circuito mostrado en la Figura 4.61. Asimismo, el propio circuito finaliza la evaluación/ordenación en el momento que se tengan los t autovalores mayores cuya suma sea superior a un % del total (P de la Figura 4.61). Se denominará como $\mathbf{V}_t \in \Re^{M\times t}$ a la matriz de autovectores $\mathbf{V} \in \Re^{M\times M}$ reducida, es decir, la matriz $\mathbf{V}$ con los t autovectores de mayor peso.

Dentro de dicho circuito cabe destacar los siguientes bloques:

- *Memoria*: En cada celda de la memoria, se almacena un autovalor (bus de datos de tamaño n bits), así como m bits de dirección. Estos m bits sirven para conocer qué autovector tiene asociado cada autovalor.
- *Contador*: Se encarga de direccionar la memoria tanto en una primera fase de almacenamiento de los autovalores de la memoria, como en cada una de las iteraciones que forman el algoritmo de la burbuja.
- *Bloque 1 (Block 1)*: Este bloque tiene implementada toda la lógica necesaria para poder realizar las comparaciones según el método de la burbuja expuesto anteriormente.
- *Bloque 2 (Block 2)*: La función de este segundo bloque es la de encontrar los t autovalores más significativos acorde a la expresión (4.56). Fundamentalmente, está compuesto por una unidad de multiplicación y acumulación (MAC) y un comparador (COMP2), que evalúa la suma parcial de autovalores con una constante P que indica el porcentaje de la suma de los t autovalores.

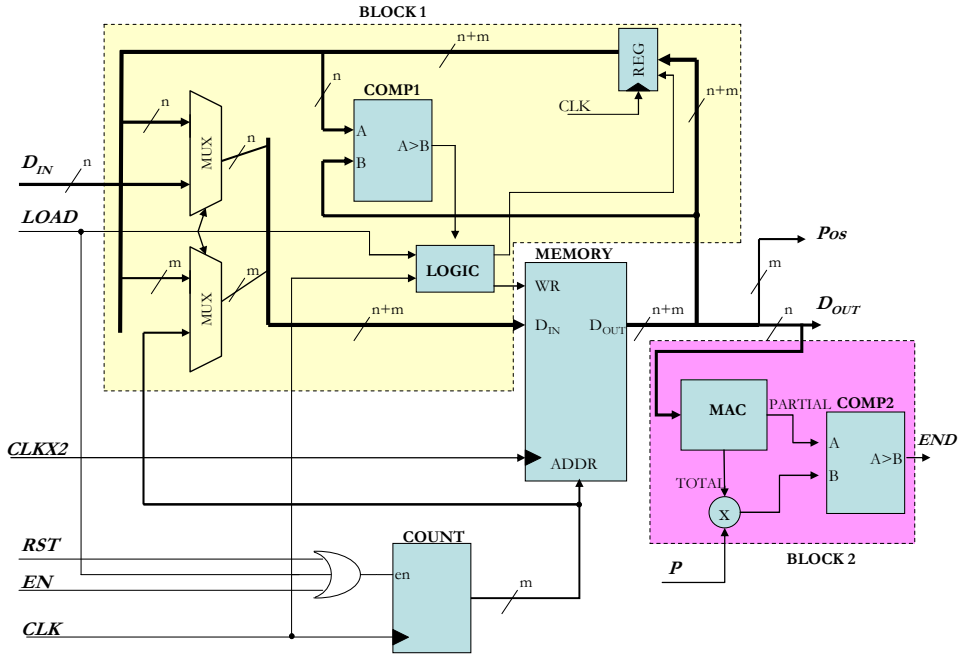


Figura 4.61. Diagrama de bloques del sistema de ordenación y reducción de autovalores y autovectores.

El sistema expuesto en la Figura 4.61 posee como salidas las señales: *END* (indica cuando se ha alcanzado la suma de autovalores deseada), D_{OUT} (contiene el valor del autovalor de salida) y *Pos* (indica la dirección de cada autovalor que sale por D_{OUT}). Este módulo comienza a funcionar en el momento que finalice el cálculo de autovalores y autovectores.

Con respecto al funcionamiento de la arquitectura mostrada en Figura 4.61, ésta se puede dividir en cuatro fases, tal y como se presenta en Figura 4.62:

1. El primer paso que se realiza es la carga de todos los autovalores obtenidos previamente en la memoria interna del sistema de la Figura 4.61.
2. Tras vencer una latencia inicial ($L_i = T_{CLK}$), en paralelo con la carga de autovalores se va obteniendo el valor de la suma de todos los autovalores. El número de ciclos de reloj consumidos por esta fase ($T_{SUM\_AUTOVAL}$) es el igual al de la carga de autovalores, siendo su valor igual al número de autovalores obtenidos (M).
3. Una vez ya almacenados los autovalores en memoria se procede a la ordenación de éstos aplicando el método de la burbuja. El sistema mostrado en la Figura 4.61 emplea en esta fase para el peor de los casos un total de $84T_{CLK}$ (4.57), necesitando 3 T_{CLK} en cada comparación ($J = 3$).
4. Por último, se realiza la reducción del número de autovalores significativos en función del porcentaje deseado (P de la Figura 4.61). Este tiempo es variable, ya que es función del porcentaje deseado y del peso de cada autovalor. En (4.58) se muestra el valor obtenido para el peor de los casos (ordenación de todos los autovalores).

$$T_{ORD} = \left(J \cdot T_{CLK/Comparacion} \times n\_comparaciones\right) = \left(3 \times \sum_{i=1}^{M-1} i\right) \cdot T_{CLK} = 84T_{CLK} \qquad (4.57)$$

$$T_{RED} = (M-1) \cdot T_{CLK} = 7T_{CLK} \qquad (4.58)$$

Por tanto, el tiempo total, en el peor de los casos, consumido en esta fase para M autovalores es el mostrado en (4.59).

$$T_{ORDENAR\_REDUCIR} = L_i + T_{SUM\_AUTOVAL} + T_{ORD} + T_{RED} = 100T_{CLK} \qquad (4.59)$$

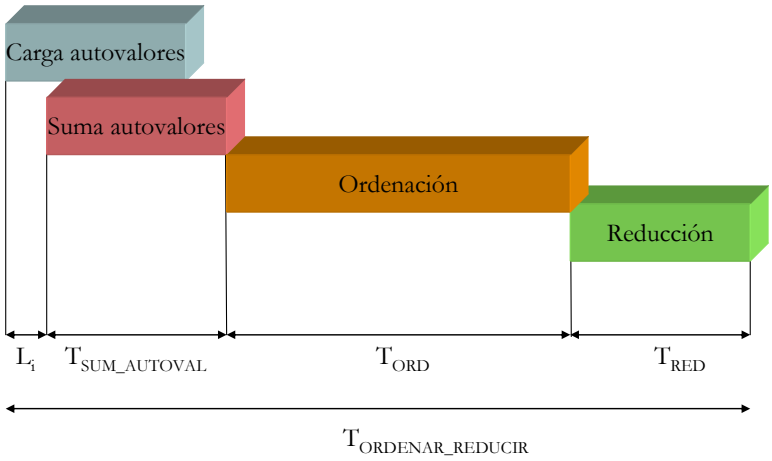


Figura 4.62. Segmentación y temporización de la etapa de ordenación y búsqueda de autovalores más significativos.

4.2.3.12. Resultados de la nueva arquitectura propuesta para el cálculo de autovalores y autovectores.

A continuación se presentan los resultados temporales, errores y recursos consumidos dentro de la FPGA, obtenidos en el cálculo de autovalores y autovectores para diferentes matrices de covarianza de $M \times M$ elementos particularizado para $M = 8$, todas ellas procedentes de las imágenes captadas con la plataforma hardware desarrollada en esta tesis. La elección de este tamaño viene justificada principalmente por el ancho de banda de la memoria externa y los resultados expuestos en [Vázquez, 2006] tal y como se justificó en el apartado 4.2.1.

Con idea de obtener la máxima precisión posible en los autovalores y autovectores, es importante fijar el número óptimo de iteraciones n y h necesarias. En este punto es importante recordar que el número de iteraciones óptimas para el módulo CORDIC fue justificado en el apartado 4.2.3.10, siendo éste n para el modo rotación, y $(n-1)$ para vectorización. Por otra parte, tal como ya se ha indicado anteriormente, en [Brent, 1985] se establece de forma empírica que $h = M \log(M)$ puede ser una posible solución; sin embargo dependiendo de la precisión que se quiere obtener en la obtención de los autovalores y en consecuencia autovectores, es necesario buscar el valor óptimo de h. En nuestro caso se han simulado diferentes valores de h para tamaños de n entre 17 y 21 bits y con $M = 8$. La elección de estos valores de n y M viene justificada por el tipo de datos a manejar dentro de todo el algoritmo PCA. Para encontrar el valor de h óptimo se deben tener en cuenta las siguientes premisas del sistema:

- *Multiplicadores hardware*: Se emplean los multiplicadores hardware de Xilinx de 18x18 bits para realizar el factor de conversión.

- *Redondeo*: Del análisis de los tres tipos de redondeo expuestos previamente y comparando los errores de estos tres tipos de redondeo con respecto a la ausencia de redondeo, se estima que el incremento de recursos hardware es pequeño en relación a los beneficios en cuanto a mayor precisión del sistema.
- *Datos de entrada*: Los datos que forman la matriz de covarianza de entrada llegan secuencialmente y se van cargando en la memoria DP de la Figura 4.38. Por su parte, la matriz identidad necesaria para el cálculo de autovectores se almacenará en la misma DP después de la de covarianza.
- *Datos de salida*: Según van generándose los datos correspondientes a la última iteración, estos también son almacenados en la DP.

Comenzando con el análisis, se han simulado diferentes tamaños de n, concretamente entre 17 y 21 bits, con un número de iteraciones h variable. La elección de este rango de bits, viene justificada por el tipo de datos a manejar dentro de todo el algoritmo PCA. Debido a que los elementos de la matriz de covarianza proceden de la multiplicación $\frac{1}{M}\mathbf{A}^T \cdot \mathbf{A}$ y teniendo presente que cada uno de los datos de la matriz $\mathbf{A}$ está codificado con 9 bits, para determinar posibles errores de *overflow* dentro de Jacobi, se ha simulado un conjunto de 10 bancos de imágenes con unas 100 imágenes para cada uno que contemplan diferentes escenas. Para este conjunto de imágenes, el tamaño de n óptimo está entre 17 y 21 bits. Así, en la Figura 4.63 se presenta el e_{MAX} y e_{SQRT} normalizado con respecto al valor máximo para autovalores, sin aplicar ningún tipo de redondeo para los diferentes tamaños de n con h variable. El mismo caso pero para autovectores se presenta en la Figura 4.64. Si el estudio se centra en los autovectores que es el objetivo final de toda esta etapa, se puede observar como el e_{SQRT} mejora a partir de $h = 19$ para todos los tamaños de bits, siendo mínimo entorno a la iteración 20, 21. Este valor de iteraciones para las cuales el error es mínimo, difiere de la expuesta en [Brent, 1985], donde para una matriz de covarianza de 8×8 estaría en 17 ($8\log(8)$). Por tanto, como primera conclusión desde un punto de vista de manejo de datos internos sin ningún tipo de redondeo, se puede decir que para esta propuesta el número óptimo de iteraciones de Jacobi estará en la iteración 20 ó 21.

Para ver la mejora que los redondeos producen sobre los valores finales, se presentan en la Figura 4.65 (autovalores) y en la Figura 4.66 (autovectores) el e_{MAX} y e_{SQRT} normalizado en el caso de aplicar la suma de los tres redondeos. Si de nuevo se centra el análisis en la iteración 17 (iteración propuesta para $M = 8$ según [Brent, 1983]), se comprueba como el error no es mínimo en dicha iteración sino que disminuye a partir de la 19, siendo mínimo aproximadamente en la 23. Con respecto a la mejora que aporta el redondeo en la Tabla 4.5 se presenta un resumen del e_{SQRT}

de autovectores para la iteración 17, con y sin redondeo, así como la iteración óptima para cada tamaño de bits. Se puede observar como el redondeo mejora el porcentaje de error para cualquier tamaño de bits. También en dicha tabla se puede verificar como el aumento del número de iteraciones con respecto a la propuesta de [Brent, 1985], proporciona una notable mejora del error. Por esta razón, se emplearán redondeos.

Tras varias pruebas de simulación para diferentes tamaños de M se ha llegado a la conclusión que debido a la estructura interna de la arquitectura propuesta en este trabajo el valor de h óptimo es: $h = 6 + M\log(M)$. El aumento de 6 iteraciones incrementa el tiempo de ejecución pero no así el número de recursos internos consumidos de la FPGA.

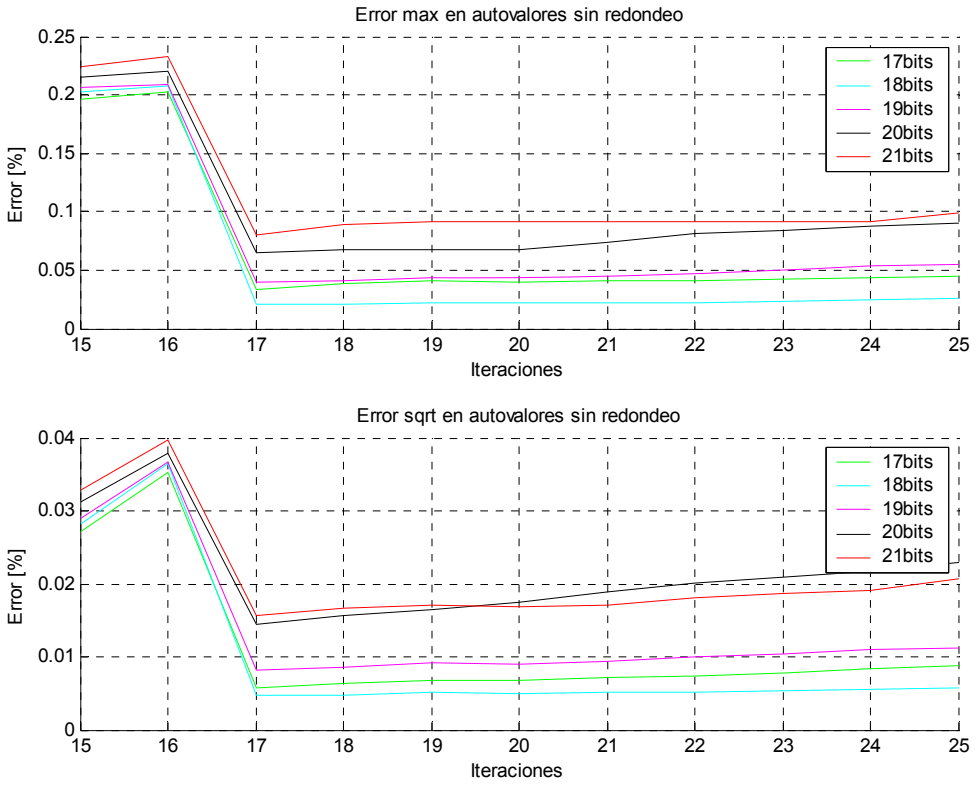


Figura 4.63. Error MAX/SQRT en autovalores para 17, 18, 19, 20, 21 bits con iteraciones desde 15 a 25 sin aplicar ningún redondeo.

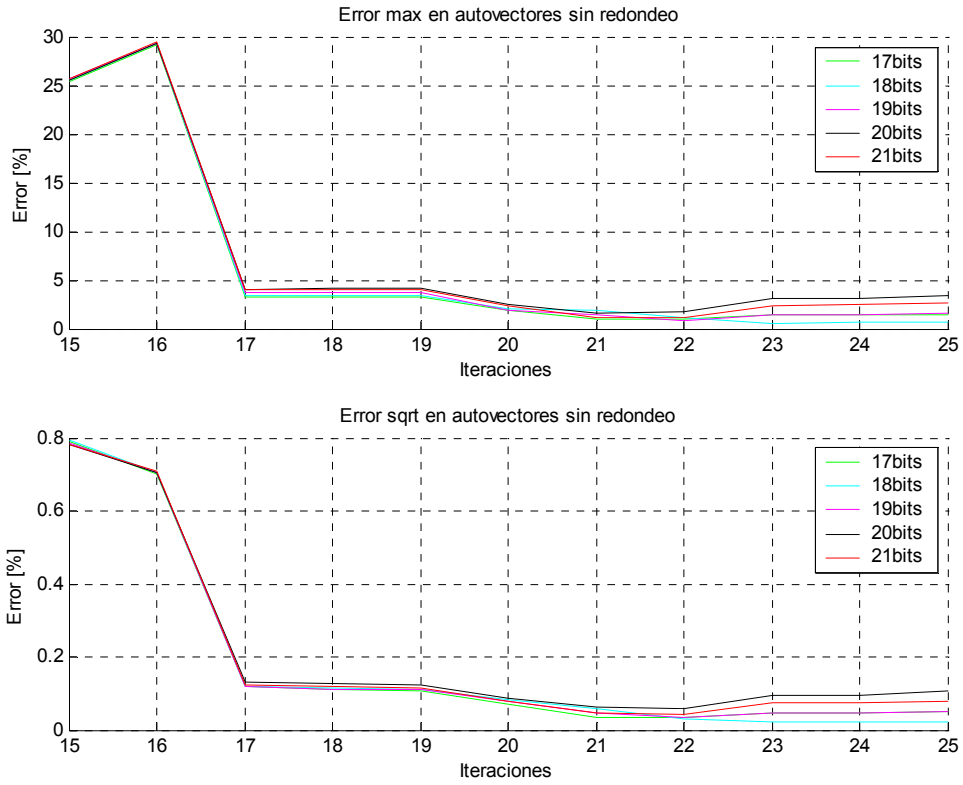


Figura 4.64. Error MAX/SQRT en autovectores para 17, 18, 19, 20, 21 bits con iteraciones desde 15 a 25 sin aplicar ningún redondeo.

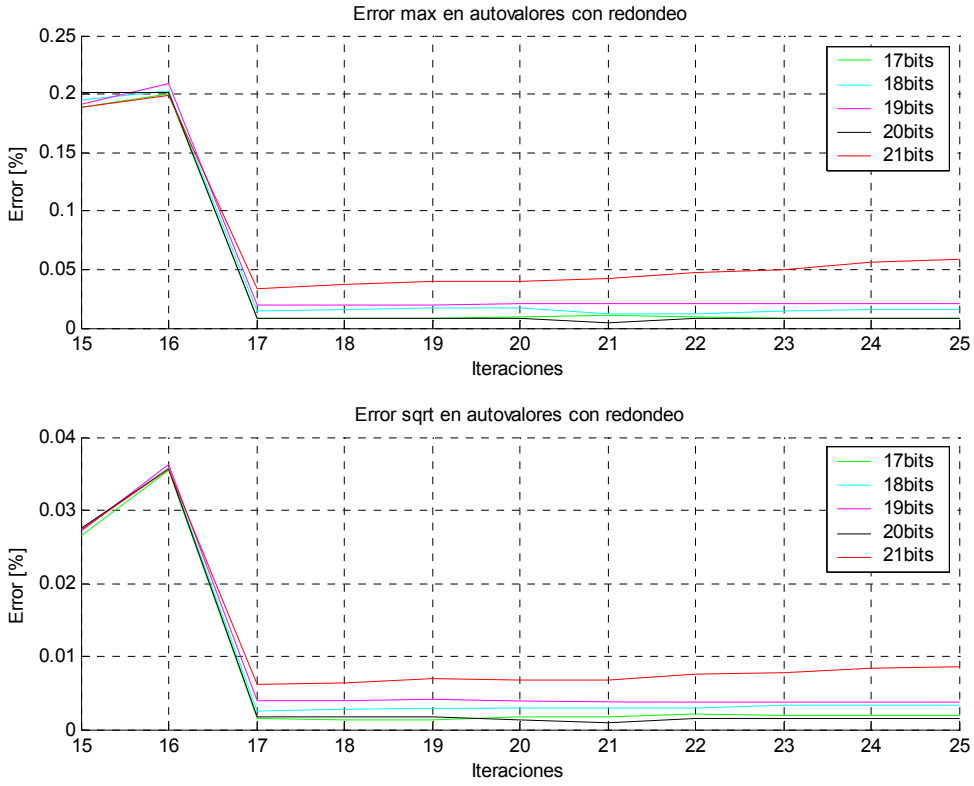


Figura 4.65. Error MAX/SQRT en autovalores para 17, 18, 19, 20, 21 bits con iteraciones desde 15 a 25 aplicando todos los redondeos.

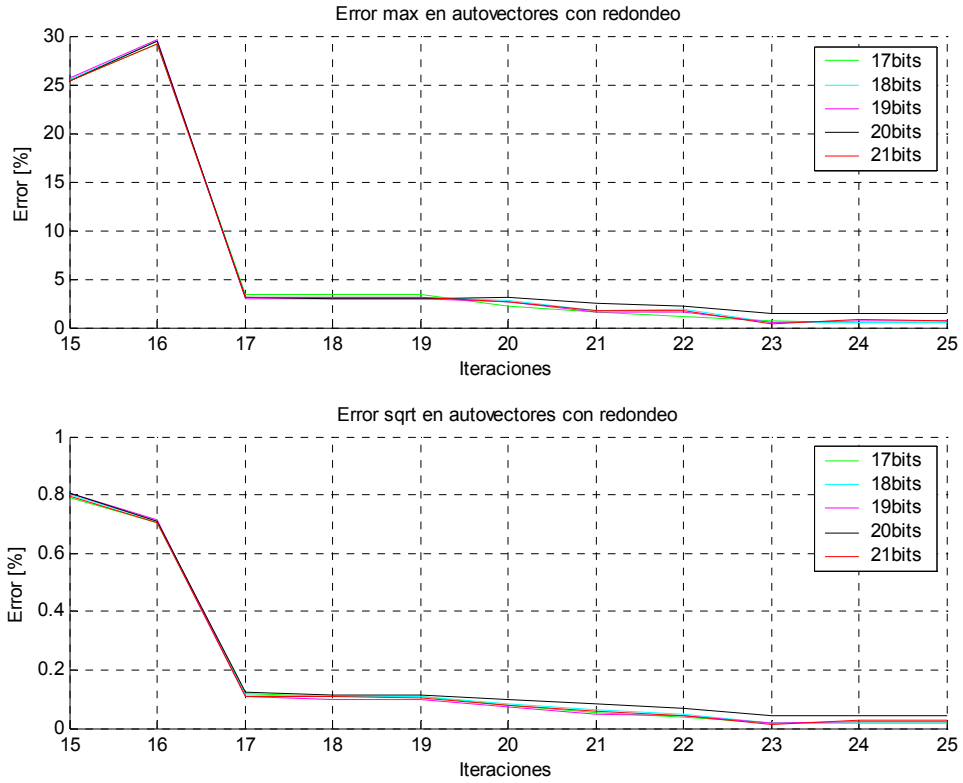


Figura 4.66. Error MAX/SQRT en autovectores para 17, 18, 19, 20, 21 bits con iteraciones desde 15 a 25 aplicando todos los redondeos.

Tabla 4.5. Comparativa del error en autovectores para número de iteraciones según [Brent, 1983] y error mínimo en función del número de bits y de redondeo.

Bits	Redondeo	Error SQRT para 17 iteraciones	Error mínimo/Iteración	Mejora de error (%) con respecto a la iteración 17
17	No	0.123783	0.0436872 / Iter=22	64.707
17	Sí	0.108751	0.0141377 / Iter=23	87
18	No	0.119684	0.036184 / Iter=22	69.767
18	Sí	0.116534	0.0156298 / Iter=24	86.588
19	No	0.119813	0.0205207 / Iter=23	82.873
19	Sí	0.11464	0.0172452 / Iter=23	84.957
20	No	0.11995	0.0323626/Iter=22	73.02
20	Sí	0.108751	0.0177133 / Iter=23	83.712
21	No	0.130127	0.0573677/Iter=22	55.914
21	Sí	0.121445	0.0428907 / Iter=24	64.683

Analizando los resultados ofrecidos por la Figura 4.65, la Figura 4.66 y la Tabla 4.5, se extraen las siguientes conclusiones:

- El error, tanto en autovectores como en autovalores, se hace mínimo para el máximo número de iteraciones independientemente de n. Sin embargo, en el caso de autovectores que es el resultado final que interesa, dicho mínimo se alcanza primeramente en la iteración número 23, oscilando posteriormente el error sobre dicho valor mínimo. Por esta razón, el número de iteraciones de Jacobi se fija a este valor para $M = 8$.
- Con respecto al tamaño de n óptimo del sistema completo sin aplicar ningún tipo de redondeo, el valor idóneo se alcanza con 18 bits tanto para autovalores como autovectores. En caso de aplicar redondeo, la exactitud también es máxima para este valor y para 20 bits. Debido a esto, se fija el tamaño a 18 bits ya que éste consume menos recursos en la FPGA que para 20 bits.

Analizando los resultados generados con y sin redondeo de la Tabla 4.5, la solución de redondeo mejora los errores. Por lo que se justifica el empleo de redondeos. Ahora bien, ¿cuál es el redondeo óptimo? Para poder evaluar la importancia de cada uno de los tipos de redondeo en la Figura 4.67 y Figura 4.68, se presenta un estudio detallado de cada tipo de redondeo para 18 bits.

Tras analizar la información aportada por la Figura 4.68 se observa que el error para autovectores es mínimo con 18 bits y 23 iteraciones cuando el redondeo aplicado es el 3. Por lo que ésta sería sin tener en cuenta el número de recursos consumidos, la mejor opción.

Una vez visto el estudio de errores, a continuación se va a realizar un análisis de los recursos consumidos por cada opción. Inicialmente, para tener una idea del número total de recursos consumidos por el sistema propuesto en este apartado sin aplicar ningún redondeo y aplicando los tres redondeos a la vez, se presenta la Figura 4.69. Se puede observar como los recursos consumidos por todo el sistema son un poco más del doble que los empleados para un módulo CORDIC (Figura 4.57). Esto es así ya que los módulos CORDIC consumen la mayoría de los recursos internos de la FPGA y como cada sistema de cálculo de autovalores y autovectores está compuesto por dos módulos CORDIC, se justifica el número total de *slices* consumidos. Por su parte, la Figura 4.70 presenta de forma detallada el número de *slices* consumidos por cada tipo de redondeo.

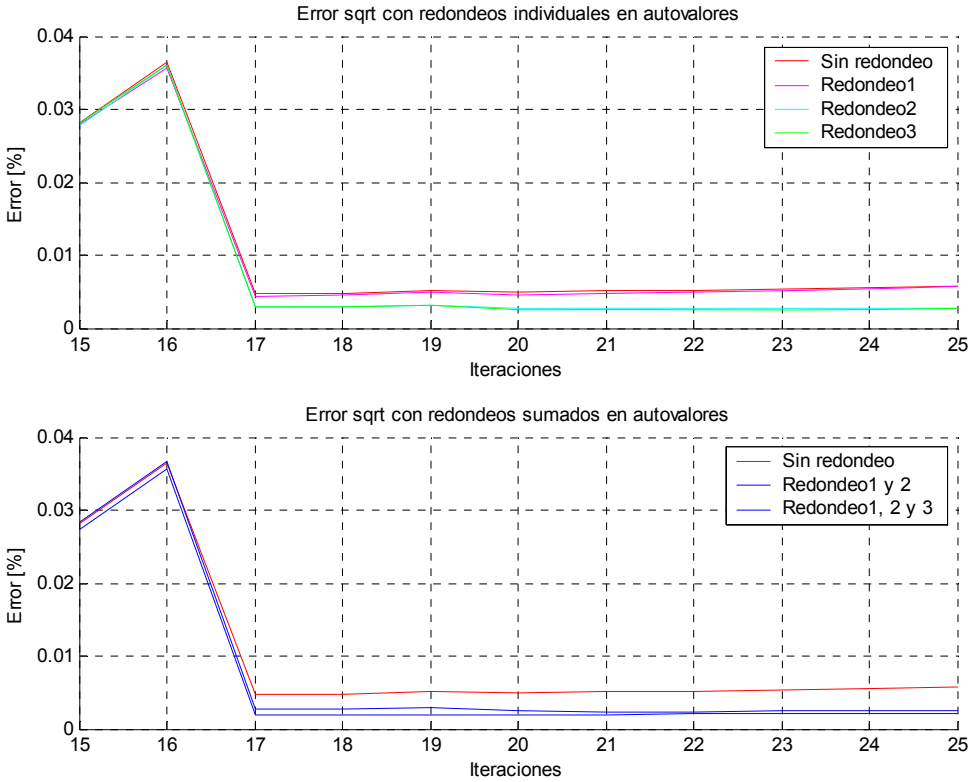


Figura 4.67. Error asociado a cada tipo de redondeo, en autovalores para datos de 18 bits, en función del número de iteraciones.

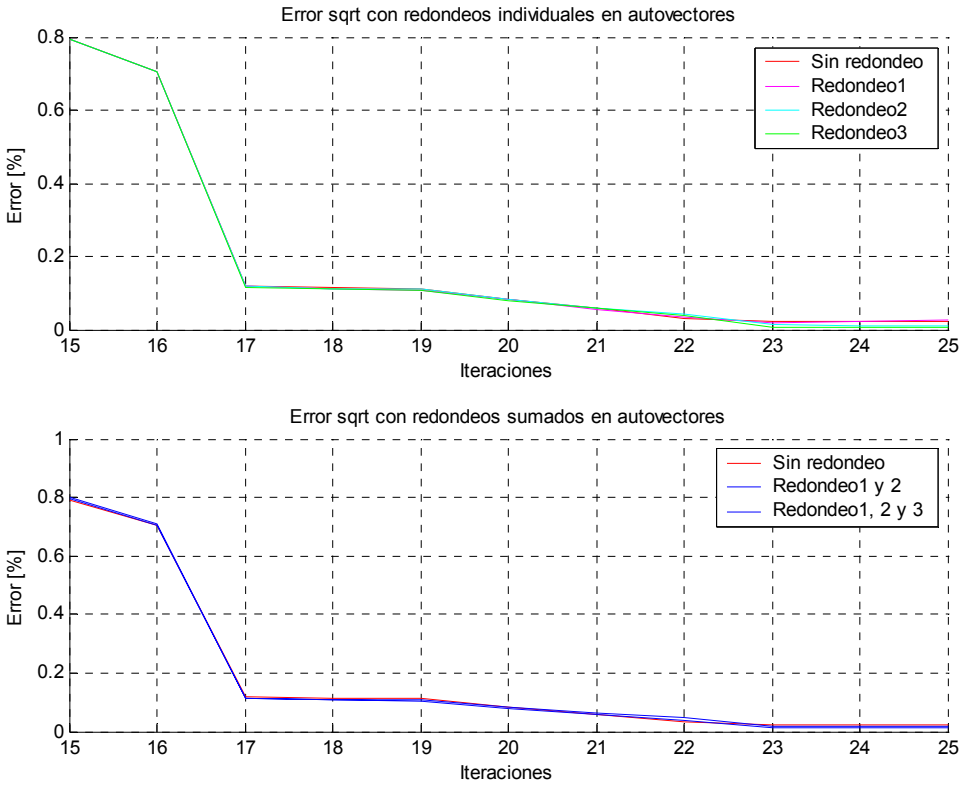


Figura 4.68. Error asociado a cada tipo de redondeo, en autovectores para datos de 18 bits, en función del número de iteraciones.

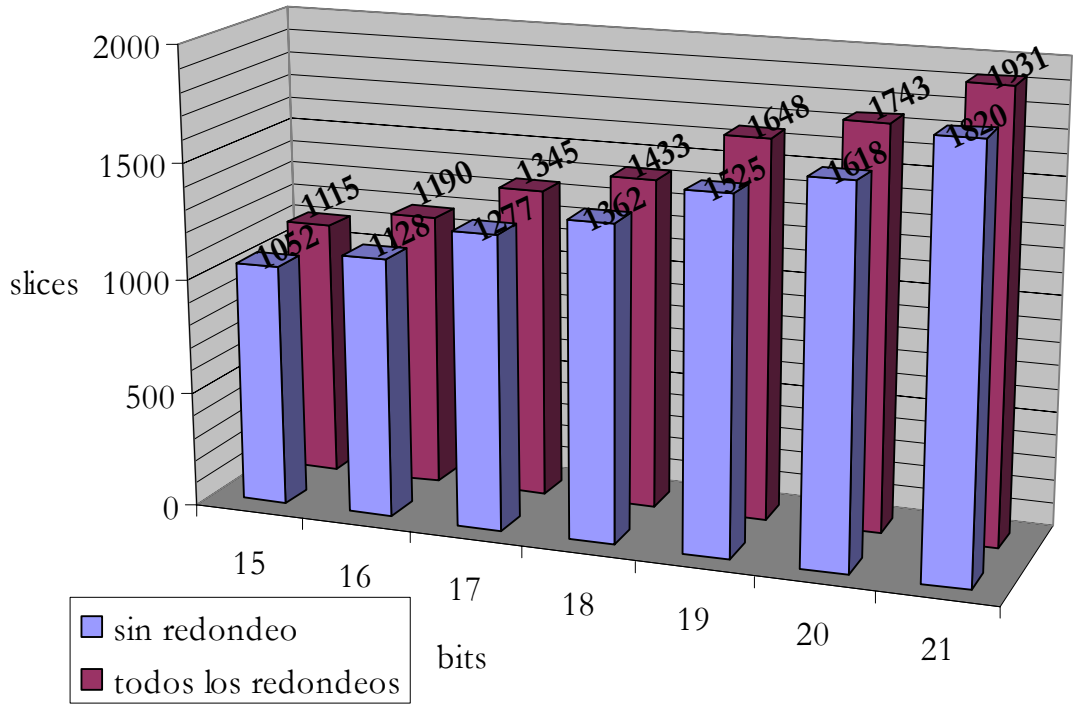


Figura 4.69. Comparativa *slices* consumidos de la arquitectura propuesta para el cálculo de autovalores y autovectores, sin aplicar redondeo y aplicando los tres tipos de redondeo, para los diferentes tamaños de datos de entrada.

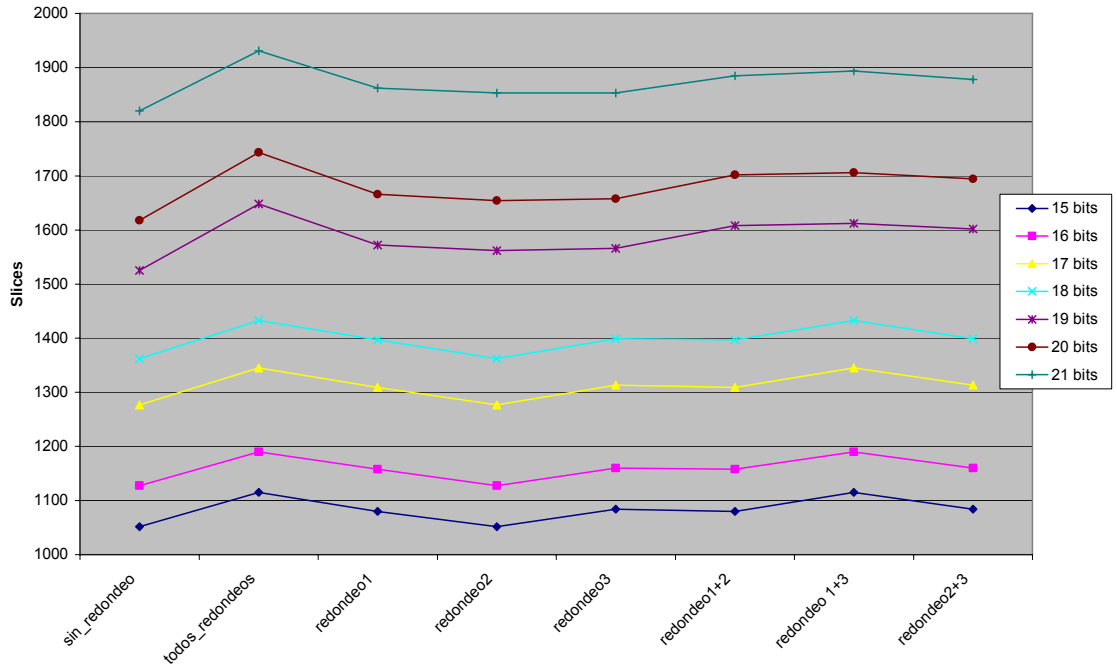


Figura 4.70. Comparativa *slices* consumidos por la arquitectura propuesta para el cálculo de autovalores y autovectores, para los diferentes tamaños de datos de entrada, con los diferentes tipos de redondeo.

Con respecto al resto de recursos de la FPGA consumidos, para todos los tamaños de bits y tipos de redondeos, se emplean los mismos elementos. Así, todas las variantes expuestas necesitan cuatro multiplicadores hardware (dos para cada CORDIC, encargados de realizar el factor de escalado). En cuanto al número de

elementos BRAM empleados, es de dos hasta 18 bits y tres para valores superiores a 18 bits.

Si el estudio se centra en los recursos consumidos para 18 bits que es el tamaño propuesto desde el punto de vista de exactitud, se puede comprobar en la Figura 4.71, que el empleo de todos los redondeos incrementa en un 5% el número de *slices* empleados. También se observa como el efecto del redondeo 2, tal y como se justificó anteriormente, no afecta al número de recursos ya que para este tamaño de datos no se realiza este redondeo. Comparando los recursos consumidos por el redondeo 1 y 3, al igual que ocurría en el estudio de los módulos CORDIC, el valor es análogo incrementando cada uno de ellos aproximadamente un 2.5% el número de *slices*. Empleando la propuesta óptima desde el punto de vista de exactitud (18 bits, 23 iteraciones y redondeo 3), el número de *slices* consumidos es 1399 (29% de la capacidad total de una XC2VP7).

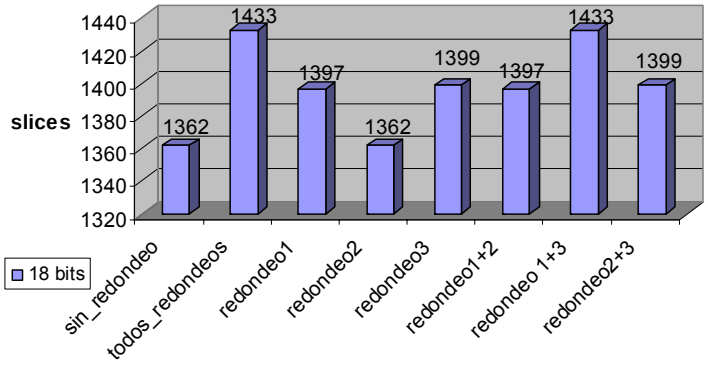


Figura 4.71. *Slices* consumidos para 18 bits, con los diferentes tipos de redondeo.

Una vez analizado el número de recursos consumidos, a continuación se va a realizar el estudio temporal de la propuesta de esta tesis para el cálculo de autovalores y autovectores. Antes de comenzar, es importante indicar que el sistema funciona con dos señales de reloj: CLK y CLK2X. Esta última tiene el doble de frecuencia que la primera y habitualmente se emplea en el direccionamiento de las memorias.

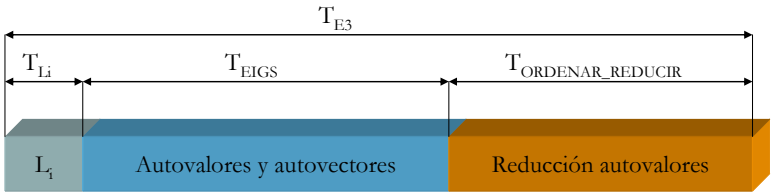


Figura 4.72. Análisis temporal del cálculo de autovectores y autovalores, para la segunda propuesta desarrollada.

En la Figura 4.72 se presenta la secuencia básica de funcionamiento de este sistema, donde se observa como el tiempo total del cálculo y reducción de autovalores y autovectores (T_{E3}) se divide en tres fases. A continuación se detalla cada una de ellas:

- El primer tiempo mostrado en la Figura 4.72 es el denominado como T_{L_i} siendo éste el asociado a la carga de la matriz de covarianza en la memoria DP de la Figura 4.38. Este tiempo se descompone en la suma de dos tiempos (4.60): tiempo de carga de autovalores ($T_{LOADAUTOVAI}$) y tiempo de carga de autovectores ($T_{LAODAUTOVEC}$). El primero es el empleado en la acumulación de los valores de la matriz de covarianza, siendo este tiempo igual a FT_{CLK2X} ya que al ser la matriz de covarianza una matriz simétrica únicamente se carga la matriz triangular superior. Por su parte, el tiempo de carga de autovectores es el tiempo que tarda en almacenarse la matriz identidad de tamaño $M \times M$ que se necesita para poder determinar los autovectores asociados a los autovalores de la matriz de covarianza.

$$T_{L_i} = T_{LOADAUTOVAL} + T_{LOADAUTOVEC} = FT_{CLK2X} + M^2 T_{CLK2X} \tag{4.60}$$

- El segundo tiempo a analizar es el denominado como T_{EIGS} (4.61). Este tiempo es el que se consume por todas las iteraciones del proceso de cálculo de autovalores y autovectores, descomponiéndose en el tiempo de una iteración (T_{ITER}) (tiempo de vectorización (T_{VEC}) más tiempo de cálculo de un autovalor (T_{EV})) multiplicado por el número de iteraciones (h) y una latencia final (L_f).

$$T_{EIGS} = hT_{ITER} + L_f = h(T_{VEC} + T_{EV}) + L_f \tag{4.61}$$

Siendo el tiempo de vectorización (T_{VEC}) el que tarda en generarse los ángulos de rotación. De nuevo, éste se divide en una secuencia de tiempos describiéndose en la Una vez finaliza la etapa de vectorización, comienza la fase de cálculo de autovalores y de autovectores dentro de una iteración (T_{EV}). Tal y como se describió en el apartado 4.2.3.9 y como muestra la Figura 4.42, de nuevo, se ha optado por la implementación de un *pipeline* que optimice la velocidad. Así, el tiempo total que consume esta fase (T_{EV}) se muestra en (4.63).

Tabla 4.6 cada uno de estos y en la Figura 4.73 su secuencia temporal. Así, el valor de T_{VEC} se expone en (4.62).

$$\begin{aligned} T_{VEC} &= T_{RD\_DP} + T_{B1} + T_{REG} + T_{CORDIC\_VEC} = \\ &2T_{CLK} + \left(n + \left(M/2 - 1\right)\right)T_{CLK} = \left(1 + n + M/2\right)T_{CLK} \end{aligned} \tag{4.62}$$

Una vez finaliza la etapa de vectorización, comienza la fase de cálculo de autovalores y de autovectores dentro de una iteración (T_{EV}). Tal y como se describió en el apartado 4.2.3.9 y como muestra la Figura 4.42, de nuevo, se ha optado por la implementación de un *pipeline* que optimice la velocidad. Así, el tiempo total que consume esta fase (T_{EV}) se muestra en (4.63).

Tabla 4.6. Descripción de los diferentes tiempos que forman el tiempo de vectorización.

Tiempos	Descripción
T_{RD_DP}	Tiempo que se necesita para leer de la DP los datos para el cálculo del ángulo. Como en ésta se usa f_{CLK2x}, en un ciclo de reloj se tienen los 4 datos necesarios para poder calcular el ángulo
T_{B1}	Tiempo que consume el Bloque 1 de la Figura 4.38 en realizar la resta de los datos del denominador de la expresión del cálculo del ángulo (4.38) Su tiempo es inferior a un ciclo de reloj.
T_{REG}	Posee una duración de un ciclo de reloj y es el tiempo que necesitan los registros previos a las tres entradas del CORDIC B
T_{CORDIC_VEC}	Tiempo que tarda el CORDIC B en calcular todos los ángulos necesarios para realizar las rotaciones.

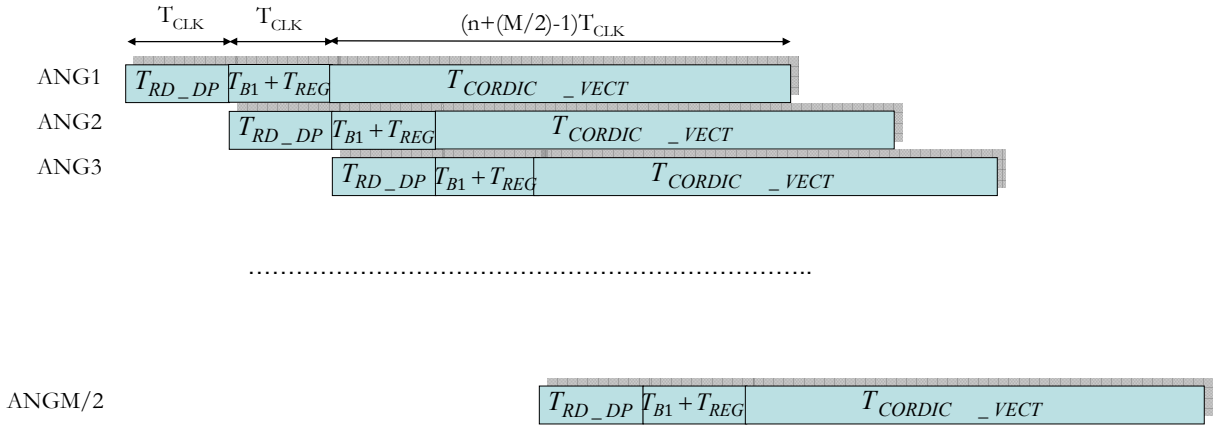


Figura 4.73. Distribución de tareas dentro del tiempo de vectorización.

$$T_{EV} = T_{ROT1} + T_{AUTOVEC} + T_{ROT2} + T_{AUTOVEC2} + T_{WRAUTOVAL} = \\ = \left(7 + 2n + \frac{M^2}{4} - \frac{F}{4}\right) T_{CLK} \tag{4.63}$$

$$T_{ROT1} = \left(n - \frac{F}{4} + 1\right) T_{CLK} \quad T_{AUTOVEC} = \left(\frac{F}{4} + 1\right) T_{CLK}$$

$$T_{ROT2} = \left(n - \frac{F}{4} + 1\right) T_{CLK} \quad T_{AUTOVEC2} = \left(\frac{M^2 - F}{4}\right) T_{CLK}$$

$$T_{WRAUTOVAL} = L + \left(\frac{F}{4} + 1\right) T_{CLK} = \left(4 + \frac{F}{4}\right) T_{CLK}$$

Por último el término L_f de (4.61) es una latencia final y está asociada al volcado de los autovectores de la última iteración a la memoria DP de la Figura 4.38.

$$L_f = \left(M^2 / 2\right) \cdot T_{CLK2X} = \left(M^2 / 4\right) \cdot T_{CLK} \tag{4.64}$$

Por tanto, el tiempo consumido en el cálculo de todos los autovalores y autovectores (T_{EIGS}) será:

$$T_{EIGS} = \left(h \cdot T_{ITER}\right) + L_f = \frac{M^2}{4} + h \cdot \left(8 + 3n + \frac{M^2}{4} + \frac{M}{2} - \frac{F}{4}\right) \tag{4.65}$$

donde h es el número de iteraciones del algoritmo de Jacobi (en nuestro caso para $M = 8$, será 23).

- El último tiempo a contemplar, es el asociado a la reducción de autovectores y autovalores ($T_{ORDENAR\_REDUCIR}$ de la Figura 4.72). Según lo descrito en el apartado 4.2.3.11, el circuito diseñado necesita, en el peor de los casos, es decir si no se eliminara ningún autovector ($t = M$), el número de ciclos de reloj mostrados en (4.59).

Por tanto, el tiempo total empleado por la propuesta desarrollada en esta tesis (Figura 4.38) para el cálculo de autovectores y autovalores (T_{E3}), consume el número de ciclos mostrado en (4.66). En la Tabla 4.7 se muestran los diferentes resultados obtenidos para el caso de $M = 8$ (máximo número de imágenes permitidas) y $f_{CLK} = 50MHz$ y $f_{CLK} = 100MHz$, variando el tamaño de los datos (n) y el número de iteraciones del método de Jacobi (h).

$$T_{E3} = L_i + T_{EIGS} + T_{ORDENAR\_REDUCIR} = \left(\frac{3M^2}{4} + \frac{F}{2} + 100 + h \cdot \left(8 + 3n + \frac{M^2}{4} + \frac{M}{2} - \frac{F}{4}\right)\right) T_{CLK} \tag{4.66}$$

Para poder evaluar los resultados alcanzados en esta sección [Bravo, 2006b], estos se han comparado con los expuestos en [Ahmedsaid, 2004b]. Éste es el único trabajo encontrado que implementa el algoritmo de Jacobi en FPGAs. Así, se ha construido la Figura 4.74 que presenta los resultados de ambos trabajos para un tamaño de matriz de 8×8 ($M=8$) y un ancho de datos de 16 bits ($n=16$). A la vista de los resultados, se observa como la propuesta desarrollada en esta tesis mejora la máxima frecuencia de trabajo (entorno a un 33%) y como es notablemente superior (casi un factor de 16 veces menos) el número de recursos consumidos.

Tabla 4.7. Tiempos máximos consumidos en el cálculo de autovalores y autovectores, para diferentes tamaños de datos (n) e iteraciones (h).

			L_i	T_{EIGS}	T_{RED}	T_{E3}
M=8 f=50MHz	**n=18**	**h=23**	50 T_{CLK} (1µs)	1591 T_{CLK} (31.82µs)	100 T_{CLK} (2µs)	1753 T_{CLK} (34.82µs)
		h=17;	50 T_{CLK} (1µs)	1291 T_{CLK} (25.82 µs)	100 T_{CLK} (2µs)	1453 T_{CLK} (25.82µs)
	n=19	**h=23**	50 T_{CLK} (1µs)	1810 T_{CLK} (36.20µs)	100 T_{CLK} (2µs)	1877 T_{CLK} (39.20µs)
		h=17	50 T_{CLK} (1µs)	1342 T_{CLK} (26.84µs)	100 T_{CLK} (2µs)	1421 T_{CLK} (29.84µs)

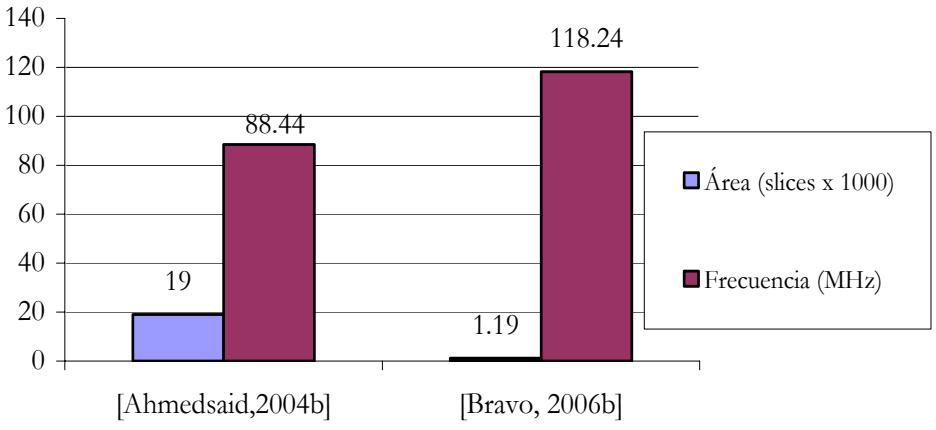


Figura 4.74. Comparativa resultados obtenidos mediante la nueva propuesta [Bravo, 2006b], con respecto a [Ahmedsaid, 2004b] para el cálculo de autovalores y autovectores, para $M=8$ y $n=16$.

Con objeto de evaluar las prestaciones que presenta la nueva propuesta desarrollada en esta tesis para el cálculo de autovalores y autovectores, en la Figura 4.75 se muestran los recursos ocupados por la nueva arquitectura propuesta y por la

arquitectura sistólica clásica basada en [Brent, 1985] desarrollada con el módulo CORDIC diseñado en esta tesis y expuesto en 4.2.3.10, para una matriz de entrada con $M = 8$ y con un $n = [15,21]$. Se observa como el número de *slices* consumidos por la nueva propuesta es claramente inferior a la de [Brent, 1985] no viéndose penalizado el tiempo de ejecución de la nueva propuesta con respecto a la clásica (Figura 4.76).

Por su parte, la Figura 4.77 presenta la evolución del ratio de tiempos de ejecución y *slices* de ambas arquitecturas, para $n = 18$ y $M \in [6,16]$. En este caso se observa como a medida que aumenta el tamaño de la matriz de entrada, los recursos consumidos en la SYST_ ARCH se incrementan exponencialmente, mientras que el ratio de tiempos de ejecución permanece prácticamente constante.

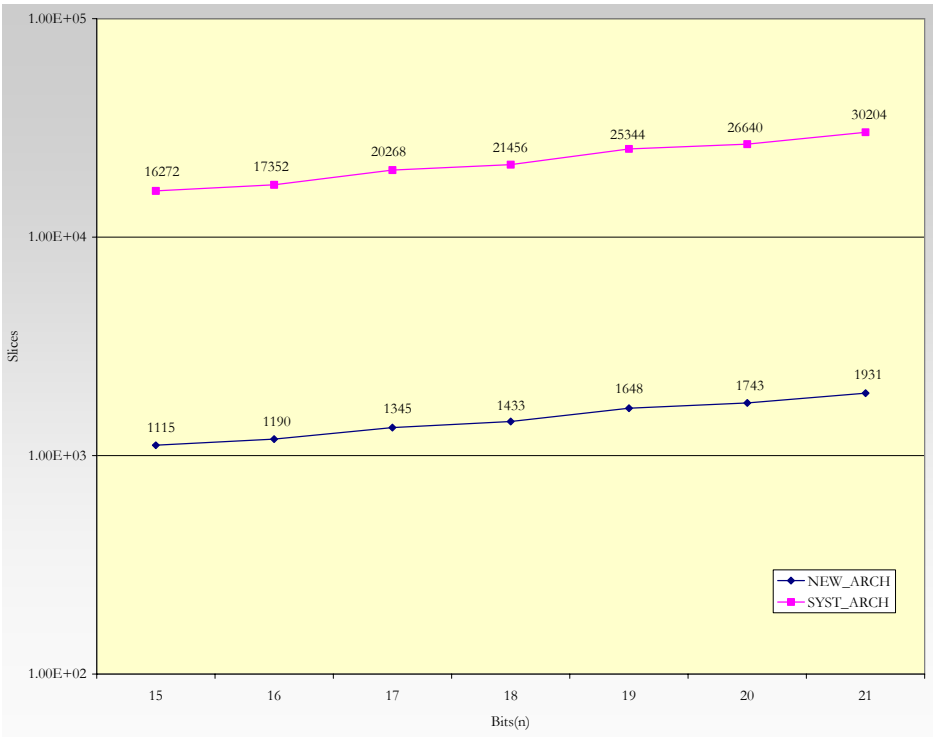


Figura 4.75. Recursos consumidos y tiempos de ejecución de la arquitectura desarrollada en esta tesis y la propuesta en [Brent, 1985] para un tamaño n variable.

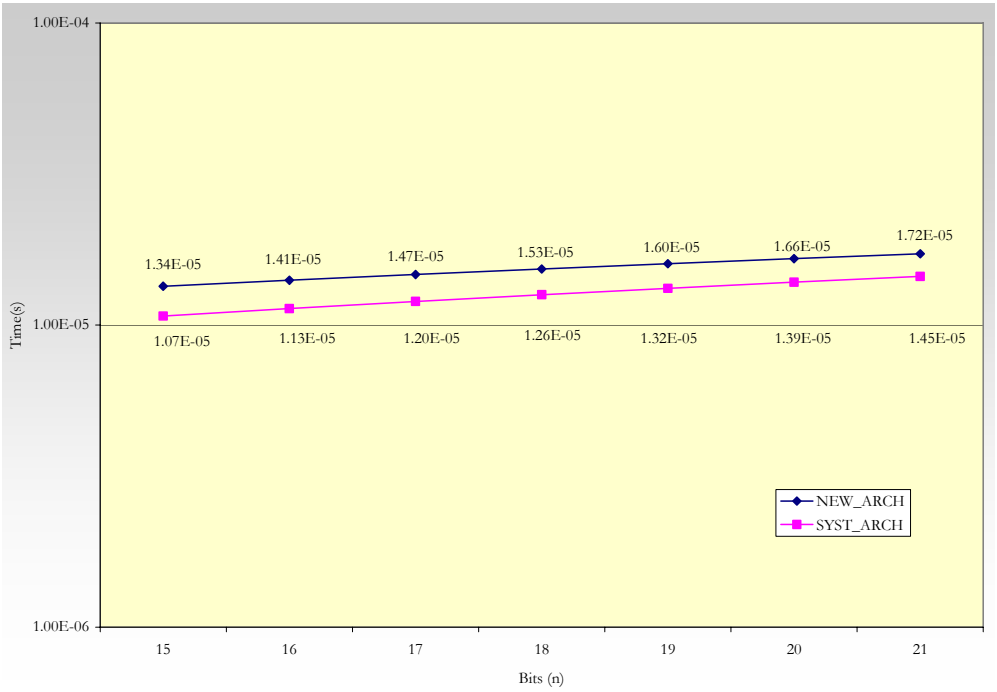


Figura 4.76. Tiempos de ejecución de la nueva arquitectura desarrollada (NEW_ARCH) y la sistólica de [Brent, 1985] (SYST_ARCH)) para n variable.

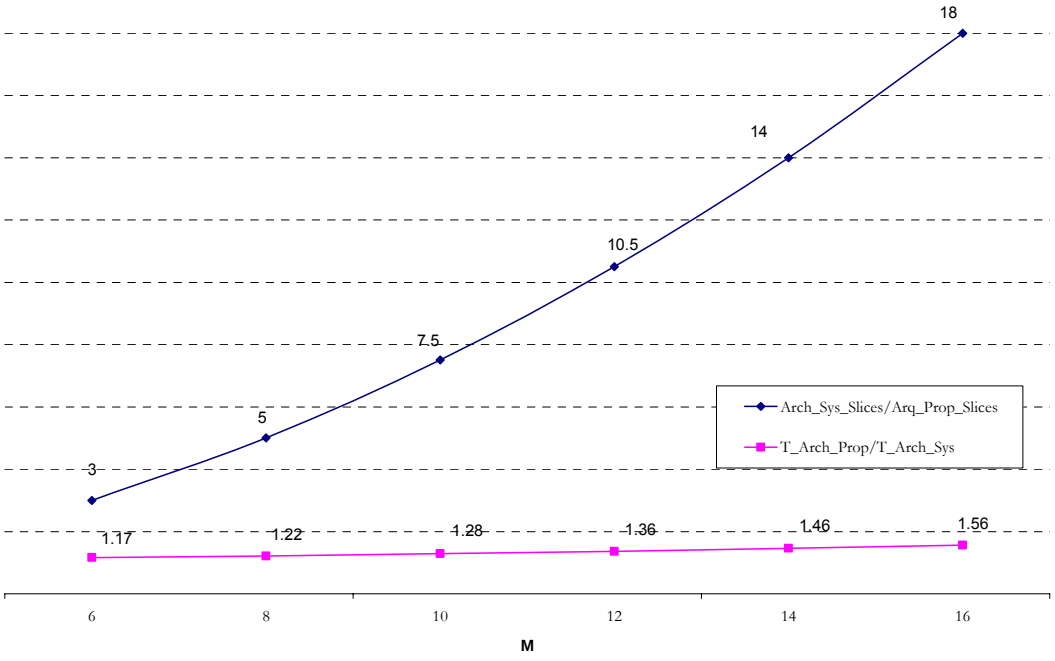


Figura 4.77. Ratio entre *slices* y tiempos de ejecución de la arquitectura propuesta en esta tesis (Arch_Prop) y la sistólica de Brent (Arch_Sys).

Con respecto a la exactitud alcanzada y tiempo de ejecución necesario, los resultados han sido comparados con los generados en un PC. La Tabla 4.8 muestra el tiempo consumido por un PC (PIV-2.66 GHz) para obtener una precisión similar a la obtenida por el diseño en FPGA. El programa del PC ha sido escrito en lenguaje C usando datos de tipo *double* (64 bits en coma flotante). Según muestra la Tabla 4.8, se

puede decir que la propuesta desarrollada en este trabajo disminuye el tiempo de cómputo con respecto a un PC como mínimo en un 40%, aumentándose este porcentaje a medida que aumenta el tamaño de las matrices.

Tabla 4.8. Comparación de tiempos de ejecución, para el cálculo de autovectores entre un PC y una FPGA.

Tamaño Matriz (M)	FPGA – 19 Iteraciones (f_{CLK}=100MHz)				PC – 32 bits (5 iteraciones)			*Mejora tiempo ejecución (%)*
	Bits	Error [%]	*Slices*	T_{Total} [us]	Bits	Error [%]	T_{Total} [us]	$\frac{T_{totalPC} - T_{totalFPGA}}{T_{totalPC}} \times 100$
8x8	18	0.304	1300	15.1	64	0.204	25.06	39.75%
10x10	18	0.324	1400	16.46	64	0.124	36.53	54.94%
12x12	18	0.373	1500	18.09	64	0.273	48.25	62.51 %
16x16	18	0.432	1800	22.17	64	0.332	64.15	65.44 %

4.2.4. Obtención de la matriz de autovectores $\mathbf{U}_t$ a partir de $\mathbf{V}_t$.

Una vez finalizado el proceso de generación de la matriz de autovectores $\mathbf{V} \in \Re^{M \times M}$ y la posterior reducción de ésta a $\mathbf{V}_t \in \Re^{M \times t}$ (apartado 4.2.3.11), el siguiente paso dentro del algoritmo PCA, es la obtención de la matriz $\mathbf{U}_t \in \Re^{N^2 \times t}$. Para conseguir esto, según (4.9), se debe realizar la multiplicación de la matriz $\mathbf{A} \in \Re^{N^2 \times M}$ por $\mathbf{V}_t$. Para ello, de nuevo se utiliza el *array* semi-sistólico expuesto en la Figura 4.11. Sin embargo para esta segunda multiplicación de matrices no se deben activar todas las celdas de multiplicación del *array*, sino que debido al tamaño de los datos y a la procedencia de éstos, únicamente se activan las 3 primeras filas de celdas de multiplicación del *array* semi-sistólico de la Figura 4.11. A cada celda se le ha dotado de una memoria de 3 posiciones de n bits que se denomina como MEM_V, funcionando como una memoria intermedia entre la DP que contiene los autovectores (Figura 4.38) y las celdas del *array*. En estas memorias se acumulan de forma dinámica hasta 3 autovectores diferentes de $\mathbf{V}_t$. El empleo de éstas va a permitir acelerar la ejecución de la multiplicación de $\mathbf{A}$ por $\mathbf{V}_t$.

Para obtener los datos de la matriz $\mathbf{A}$, ésta se debe volver a generar a partir de las M imágenes almacenadas en memoria externa. Esto es así porque esta matriz no se almacenó en memoria durante la generación de la matriz de covarianza ($\mathbf{S} \in \Re^{M \times M}$) debido a su elevado tamaño ($M^2 \times N^2 \times 9bits$). Otra ventaja de no almacenar $\mathbf{A}$ es la sincronización óptima dentro del *array* sistólico en la ejecución de

esta segunda multiplicación de matrices, ya que tal y como se verá a continuación, la cadencia de generación de los datos de $\mathbf{A}$ no es de un ciclo de reloj, sino de tres.

Para explicar el funcionamiento del *array* sistólico para esta segunda Mult. Mat., se ha construido la Figura 4.78 y Figura 4.79 donde $M=8$. En esta última se muestran las celdas activas del *array* de la Figura 4.11 para esta segunda multiplicación. También la Figura 4.78 presenta los datos que recibe cada celda distinguiendo cada operador con dos colores: en color azul para los procedentes de la matriz $\mathbf{A}$, donde $i=1,...,N^2$ y en rojo para los autovectores de la matriz $\mathbf{V}_t$, donde $l=1,...,t$. Por su parte, la Figura 4.79 muestra la secuencia ordenada de los datos de la matriz $\mathbf{A}$ y $\mathbf{V}_t$ en la generación de los datos de $\mathbf{U}_t$.

Los primeros datos de la matriz $\mathbf{A}$, al igual que ocurría en la generación de la matriz de covarianza, son para $i=1$. Por su parte, los operandos asociados a la matriz $\mathbf{V}_t$ dependen de la reducción de autovectores (t) expuesto en el apartado 4.2.3.11. Si $t\geq 3$, primero se almacenan en las MEM_V los datos procedentes de 3 autovectores, y luego, los restantes en múltiplos de 3. Los datos de cada autovector almacenado en las MEM_V se acumulan en las celdas de multiplicación tal y como se muestra en color rojo en la Figura 4.78.

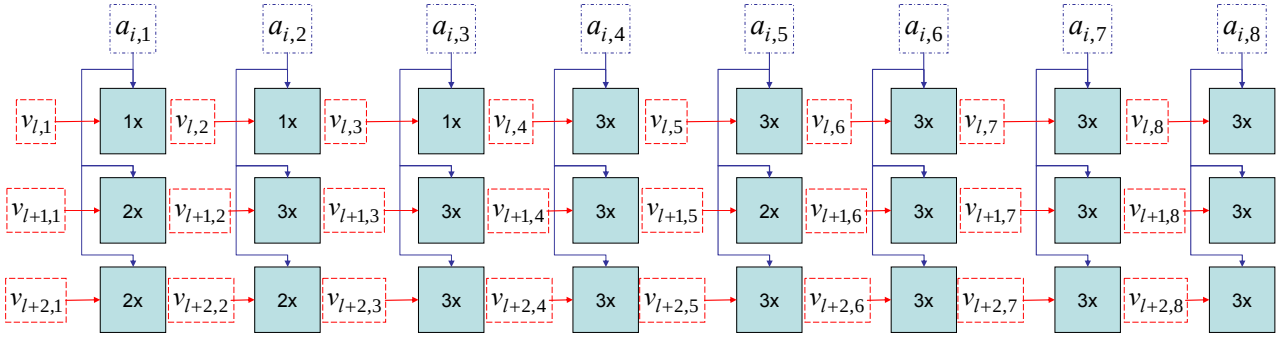


Figura 4.78. Secuencia de funcionamiento de las celdas de multiplicación del *array* sistólico, para la segunda multiplicación de matrices.

Cuando todas las celdas tienen los dos operandos para la multiplicación, se emite la orden de multiplicación y acumulación, y todas las celdas operan de forma concurrente. En este momento, cada unidad MAC de cada celda contiene un resultado parcial. Este resultado se debe sumar entre las diferentes celdas asociadas a la misma fila para poder generar un dato final de $\mathbf{U}_t$. Para realizar esto, se emplea el *array* de sumadores que tiene cada una de las tres primeras filas de celdas de la Figura 4.11. Una vez sumados, obtendremos los primeros datos de $\mathbf{U}_t$.

A continuación, se carga en las MEM_V los datos asociados a los autovectores restantes, para poder multiplicar esta primera fila de operandos de $\mathbf{A}$ por los autovectores restantes de $\mathbf{V}_t$ y generar así la primera fila de $\mathbf{U}_t$. Por tanto, se

debe realizar una nueva multiplicación y suma de datos parciales de cada celda para generar los datos restantes de la primera fila de $\mathbf{U}_t$. Una vez generada la primera fila de valores de $\mathbf{U}_t$, se solicita una nueva fila de datos de $\mathbf{A}$. El proceso se repite de forma análoga para las siguientes filas de $\mathbf{U}_t$.

Con respecto a la temporización asociada a esta segunda multiplicación de matrices, todas las celdas operan en paralelo mostrándose en la Figura 4.80 las diferentes tareas a realizar en cada celda, donde: LD_MEM_V es la lectura de datos desde las MEM_V, LD_A es la carga de los datos de la matriz $\mathbf{A}$, MULT es la fase de multiplicación de los dos operandos y ADD la fase de suma de los resultados parciales mediante los sumadores hardware mostrados en la Figura 4.11. Como se ve se realiza la carga de 3 autovectores por cada lectura de la matriz $\mathbf{A}$.

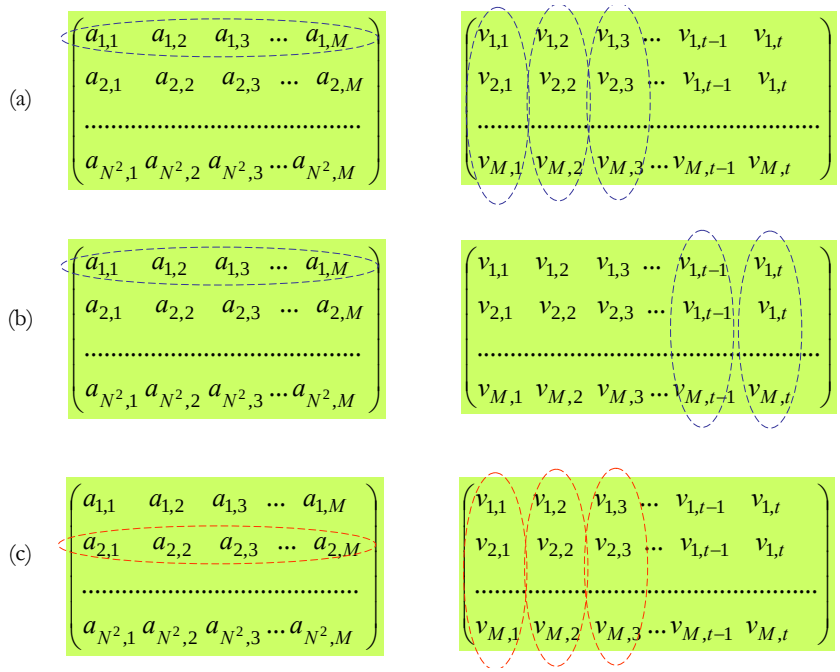


Figura 4.79. Secuencia de datos de las matrices $\mathbf{A}$ y $\mathbf{V}_t$: (a) Generación de los primeros datos de la primera fila de $\mathbf{U}_t$, (b) Generación de los últimos datos de la primera fila de $\mathbf{U}_t$, (c) Generación de los primeros datos de la segunda fila de $\mathbf{U}_t$.

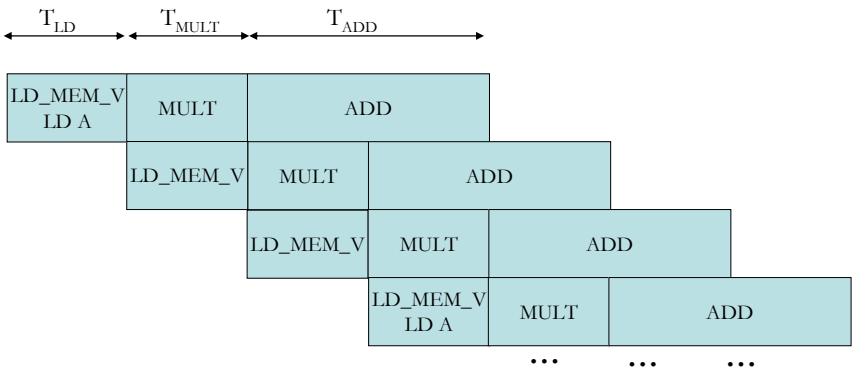


Figura 4.80. Temporización de las tareas en la segunda multiplicación de matrices.

Para determinar el número total de ciclos de reloj consumidos en esta segunda multiplicación de matrices, es necesario conocer el tiempo consumido por cada una de las tareas de la Figura 4.80. Así, en (4.67) se muestra el tiempo empleado por cada tarea para el caso práctico empleado en esta tesis: $M=8$ y $N=256$. A la hora de evaluar el tiempo consumido en la generación de la matriz $\mathbf{U}_t$ (T_{E4}), cabe destacar 3 casos posibles: $t\leq 3$, $3<t\leq 6$ y $t=7$. Para el primer caso ($d=1$ en (4.68)), el contenido de las celdas de las MEM_V no cambia ya que el tamaño de las MEM_V es igual al del número de autovectores. Por su parte, la generación de las filas de $\mathbf{A}$ se realiza en cada ciclo de reloj ya que se lee simultáneamente un píxel de cada imagen según se expuso en 4.2.1. En el segundo caso ($d=2$ en (4.68)), al ser mayor el número de autovectores que las memorias MEM_V, éstas se deben actualizar en cada ciclo de reloj. Por su parte, las filas de $\mathbf{A}$ se generan cada dos ciclos de reloj para así sincronizar su generación con la lectura de las MEM_V. Por último, en el tercer caso ($d=3$ en (4.68)) los datos de $\mathbf{A}$ se generan cada 3 ciclos de reloj y de nuevo las MEM_V se actualizan en cada ciclo de reloj. Tal y como se verá en el apartado 4.3.1, el caso $t=7$ en la práctica raramente ocurre, saturándose en este caso t a 6, con objeto de acelerar el tiempo de ejecución.

$$T_{LD}=T_{CLK},\ T_{MULT}=T_{CLK},\ T_{ADD}=(\log_2 M)T_{CLK}=3\cdot T_{CLK} \tag{4.67}$$

$$T_{E4}=\left(T_{LD}+T_{MULT}+T_{ADD}\right)+d\left(N^2-1\right)T_{CLK}\quad d=1,2,3 \tag{4.68}$$

Con respecto a la precisión del sistema, teóricamente la matriz $\mathbf{U}_t$ obtenida en el *array* semi-sistólico, podría alcanzar los 30 bits (4.69). Sin embargo, los valores obtenidos en esta operación son números reales donde la parte entera no supera los 8 bits, por lo que directamente se ha realizado un truncado a 18 bits (tamaño patrón) sin perder mucha precisión al eliminarse parte decimal del dato (ver Figura 4.93 y Figura 4.94).

$$n^o\_bits\_\mathbf{U}_t=\log_2\left(\sum_{i=1}^{M}\left(2_i^{p+1}\times 2_i^n\right)\right)=\log_2\left(\sum_{i=1}^{8}2_i^{27}\right)=30 \tag{4.69}$$

4.2.5. Normalización de autovectores. Cálculo de las normas.

Dado que los autovectores deben tener módulo unidad y los autovectores $\mathbf{u}_i$ obtenidos a partir de $\mathbf{v}_i$ no lo son todavía, se deben convertir los vectores de la matriz $\mathbf{U}_t$ obtenida mediante el *array* semi-sistólico de la Figura 4.11, a una matriz de vectores normalizada $\left(\mathbf{U}_{tn}\in\Re^{N^2\times t}\right)$. Para ello se calcula la norma de cada autovector de la matriz $\mathbf{U}_t$ (4.70) y se divide cada autovector por su norma (4.71) obteniendo así $\mathbf{U}_{tn}$.

$$n_j = \sqrt{\sum_{i=1}^{N^2} \left(\mathbf{u_{i,j}}\right)^2} \qquad j = 1,...,t \tag{4.70}$$

$$\mathbf{U}_{tn} = \frac{\mathbf{U}_t}{\mathbf{normas}} = \left(\frac{\mathbf{u}_1}{n_1} \frac{\mathbf{u}_2}{n_2} .... \frac{\mathbf{u}_t}{n_t} \right) \qquad \mathbf{normas} = \left[n_1, n_2, ..., n_t \right] \in \Re^{1 \times t} \tag{4.71}$$

Las operaciones aritméticas mostradas en las expresiones (4.70) y (4.71), a la hora de implementarse en hardware, poseen una alta complejidad así como un alto consumo de recursos. Con objeto de ahorrar operaciones matemáticas, se va a analizar en detalle cómo el algoritmo PCA emplea $\mathbf{U}_{tn}$:

La matriz de autovectores normalizada $\left(\mathbf{U}_{tn}\right)$, se emplea únicamente en el proceso *on-line*, concretamente durante la proyección de una imagen $\mathbf{\Phi}_j \in \Re^{N^2 \times 1}$ en el espacio transformado ($\mathbf{\Omega} \in \Re^{t \times 1}$) (4.72) y en la recuperación de la imagen proyectada ($\hat{\mathbf{\Phi}}_j \in \Re^{N^2 \times 1}$) (4.73). Si en (4.73) se sustituye el valor de $\mathbf{\Omega}$ por (4.72) y el de $\mathbf{U}_{tn}$ por el expuesto en (4.71), se obtiene (4.74). Por lo tanto, a la hora de obtener $\hat{\mathbf{\Phi}}_j$ se puede trabajar con la matriz de autovectores sin normalizar ($\mathbf{U}_t$) y con $\mathbf{\Omega}' \in \Re^{t \times 1}$ (4.75). De esta forma, la raíz cuadrada de (4.70) se puede eliminar, ahorrándose la ejecución de esta operación en la FPGA.

$$\mathbf{\Omega} = \mathbf{U}_{tn}^T \cdot \mathbf{\Phi}_j \tag{4.72}$$

$$\hat{\mathbf{\Phi}}_j = \mathbf{U}_{tn} \cdot \mathbf{\Omega} \tag{4.73}$$

$$\hat{\mathbf{\Phi}}_j = \mathbf{U}_{tn} \cdot \mathbf{\Omega} = \mathbf{U}_{tn} \cdot \mathbf{U}_{tn}^T \cdot \mathbf{\Phi}_j = \frac{\mathbf{U}_t}{\mathbf{normas}} \cdot \frac{\mathbf{U}_t^T}{\mathbf{normas}} \cdot \mathbf{\Phi}_j = \frac{\mathbf{U}_t \cdot \left(\mathbf{U}_t^T \cdot \mathbf{\Phi}_j \right)}{\mathbf{normas}^2} = \mathbf{U}_t \cdot \mathbf{\Omega}' \tag{4.74}$$

$$\mathbf{\Omega}' = \frac{\mathbf{\Omega}}{\mathbf{normas}^2} = \frac{\mathbf{U}_t^T \cdot \mathbf{\Phi}_j}{\mathbf{normas}^2} \tag{4.75}$$

Por tanto, para la obtención de $\hat{\mathbf{\Phi}}_j$ primeramente se obtiene $\mathbf{U}_t^T \cdot \mathbf{\Phi}_j$, posteriormente se realiza la división de este factor por las normas al cuadrado generándose así $\mathbf{\Omega}' \in \Re^{t \times 1}$ y finalmente, se realiza el producto $\mathbf{U}_t \cdot \mathbf{\Omega}'$. Posteriormente, en la descripción del proceso *on-line* se explica cómo se realiza esta operación de división. En este apartado, únicamente se va a describir el sistema que determina la matriz de normas al cuadrado.

Con respecto al diseño hardware desarrollado, en la Figura 4.81 se presenta la solución empleada. El funcionamiento de este módulo se inicia en el momento que se almacena la matriz $\mathbf{U}_t$ en memoria externa. Como el tamaño de los datos de $\mathbf{U}_t$

es de 18 bits, y el ancho del bus de memoria de escritura es de 64 bits, se puede escribir en un ciclo de reloj hasta 3 datos de $\mathbf{U}_t$, perteneciendo cada uno de estos tres datos a tres autovectores diferentes. Por tanto, en cada ciclo de reloj funcionan 3 bloques sumadores/acumuladores. Cuando finaliza de almacenarse $\mathbf{U}_t$ en memoria externa, en los registros Norma1, Norma2, …, Norma*t* de la Figura 4.81 queda almacenada la norma al cuadrado de cada uno de los t autovectores de $\mathbf{U}_t$.

El tiempo de ejecución de esta tarea se solapa con el almacenamiento de $\mathbf{U}_t$ en memoria externa. Únicamente tendrá una latencia final de un ciclo de reloj, correspondiente al volcado de cada una de las sumas/acumulaciones en los registros correspondientes. Por lo tanto, $T_{E5} = T_{CLK}$.

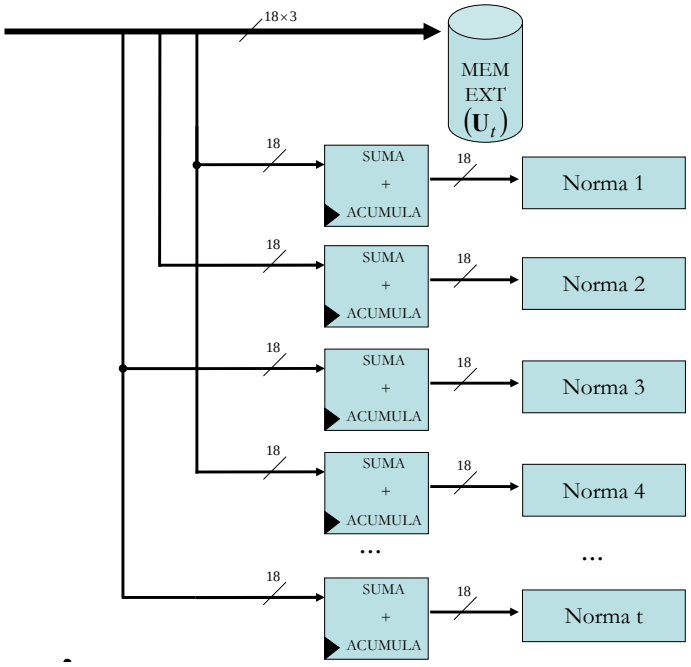


Figura 4.81. Diagrama de bloques del módulo de cálculo de las normas de la matriz $\mathbf{U}_t$.

4.3. PROYECCIÓN Y RECUPERACIÓN DE UNA IMAGEN SOBRE EL ESPACIO TRANSFORMADO (PROCESO *ONLINE*).

Una vez obtenida la matriz de autovectores $\mathbf{U}_t$ a continuación se describe la fase en la cual se determina si hay un nuevo objeto en la escena estática (fase *on-line*). En ésta, de forma continua, cada nueva imagen capturada se proyecta al espacio transformado generado con la matriz de autovectores, para posteriormente recuperarla y determinar así la existencia o no de un nuevo objeto en la escena. Para ello, acorde al algoritmo PCA, los pasos a realizar son los siguientes:

1. *Resta de la media con la imagen actual*: Al igual que ocurría en la fase de cálculo de matrices, ahora en la imagen actual sobre la que se desea determinar la existencia de un nuevo objeto en un nuevo vector imagen

($\mathbf{I}_j$), se le resta la media aritmética calculada en la fase anterior ($\mathbf{\Psi}$), creándose así $\mathbf{\Phi}_j$.

2. *Proyección de una imagen sobre el espacio transformado*: A continuación se multiplica el vector imagen $\mathbf{\Phi}_j$ por la matriz de autovectores reducida $\mathbf{U}_t^T$. Por tanto, de nuevo se está ante una nueva multiplicación de matrices a realizar en la FPGA. Para ello, se emplea de nuevo el *array* semi-sistólico expuesto en el apartado 4.2.2.2. El resultado de dicho producto deberá ser dividido entre las normas al cuadrado, ya que cabe recordar que la matriz de autovectores $\mathbf{U}_t$ no estaba normalizada, tal y como se justificó en 4.2.5, obteniéndose en esta fase el vector $\mathbf{\Omega}'$. Este proceso de normalización aprovecha ciertas condiciones de paralelismo dentro del sistema, que posteriormente son expuestas.
3. *Recuperación de una imagen del espacio transformado*: Una vez obtenido $\mathbf{\Omega}'$, el siguiente paso es la obtención del vector imagen estimado o recuperado ($\hat{\mathbf{\Phi}}_j$). Este vector sirve para determinar el grado de parecido entre las características fundamentales del vector imagen actual y las de la escena estática.
4. *Determinación del error de recuperación*: En esta última fase, se evalúa el grado de similitud entre $\mathbf{\Phi}_j$ y $\hat{\mathbf{\Phi}}_j$, para poder determinar la existencia o no, de un nuevo objeto. En el capítulo 5 de esta tesis doctoral se analizará a partir de la distancia euclídea entre $\mathbf{\Phi}_j$ y $\hat{\mathbf{\Phi}}_j$, cómo determinar si existe o no un nuevo objeto en base a un umbral Th_H el cuál se describe en el capítulo 5.

En la Figura 4.82 se observa un flujograma, de esta fase, junto con el resto de fases de PCA.

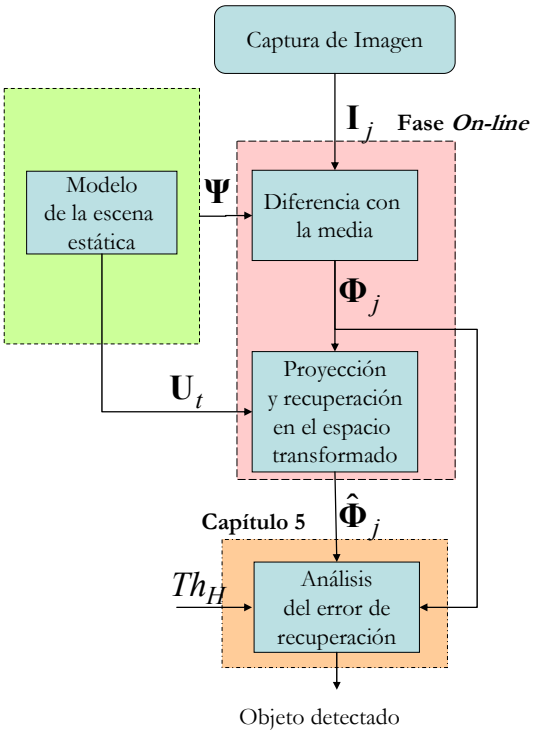


Figura 4.82. Flujograma de la fase *on-line* del algoritmo PCA.

El flujograma de la Figura 4.82 también ha sido codificado en VHDL, realizando un diseño modular, cuya estructura se puede observar en la Figura 4.83. En dicha figura aparece, de nuevo, el *array* semi-sistólico utilizado en fases anteriores el cuál se multiplexa en el tiempo.

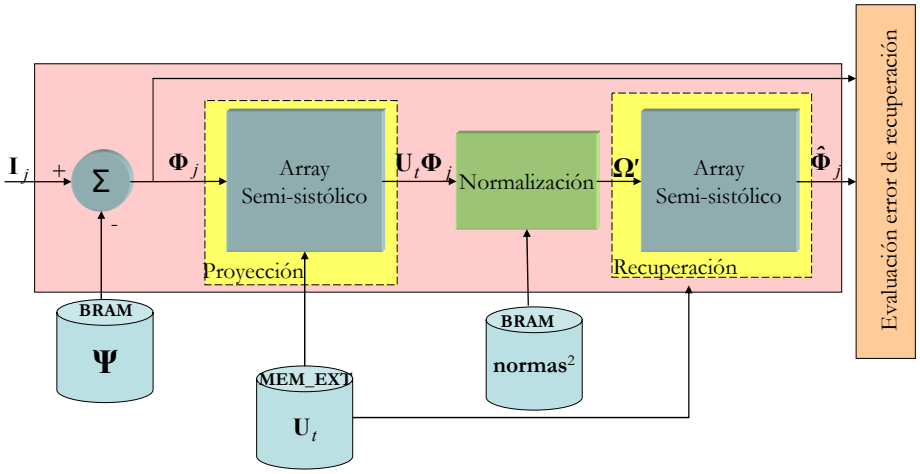


Figura 4.83. Diagrama de bloques de los módulos que forman el diseño realizado en VHDL para la fase *on-line* de PCA.

En este caso la ejecución de la parte *on-line*, se ha podido segmentar totalmente, con lo que si se ejecuta de forma continua esta fase se alcanza un alto rendimiento. En la Figura 4.84 se presentan las 4 etapas que forman esta fase *on-line*:

- *Resta* (Etapa 6).

- *Proyección* (Etapa 7).
- *Normalización* (Etapa 8).
- *Recuperación* (Etapa 9).

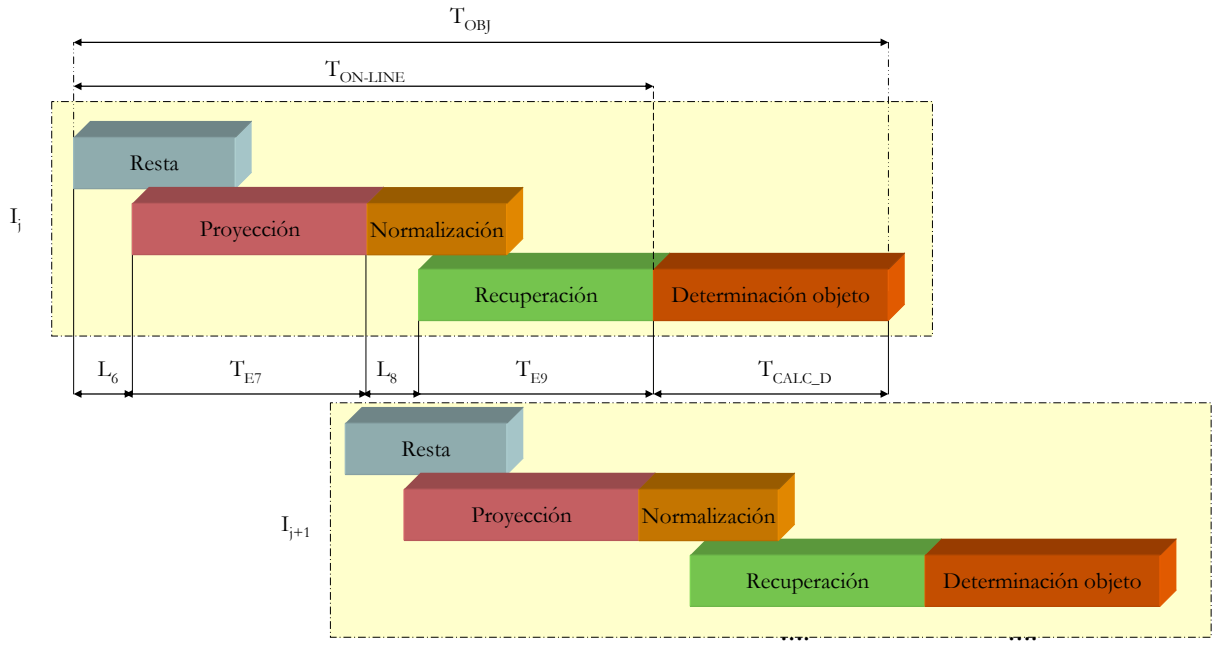


Figura 4.84. Segmentación del flujo de datos de la fase *on-line* de PCA.

Además, en la Figura 4.84 aparece la etapa de determinación de objeto nuevo, que se analiza en el capítulo 5. Se observa también como se puede iniciar el análisis de una nueva imagen, cuando finaliza la fase de normalización.

A continuación, se describe el proceso de proyección y recuperación de una imagen, sobre el espacio transformado.

4.3.1. Proyección de una imagen sobre el espacio transformado.

Una vez captado un nuevo vector imagen $\mathbf{I}_j \in \Re^{N^2 \times 1}$, éste es inicialmente almacenado en memoria externa. Posteriormente, es extraído de memoria y antes de realizar su proyección sobre el espacio transformado acorde a (3.25), el primer paso a realizar es la resta de $\mathbf{I}_j$ con el valor de la media ($\mathbf{\Psi}$) obtenida en la fase de cálculo de la matriz de autovectores, generándose así $\mathbf{\Phi}_j = \mathbf{I}_j - \mathbf{\Psi}$. Para ello, se emplea un restador de iguales características al empleado en el cálculo de la media del modelo de fondo. La operación resta consume un ciclo de reloj, por lo que vencida una latencia inicial (L_6) el primer dato ya está disponible para comenzar la etapa de proyección (E_7) (ver Figura 4.84).

En estos momentos, ya se puede realizar la proyección sobre el espacio transformado (3.25). Para ello, se utiliza el *array* semi-sistólico construido en esta tesis

para la multiplicación de matrices, siendo en este caso las matrices a multiplicar $\mathbf{U}_t^T$ y $\mathbf{\Phi}_j$. Es importante resaltar en este punto, que el tiempo de ejecución de esta Mult. Mat. es función del número de autovalores significativos (t). En todo momento, se va a evaluar la peor de las situaciones, estando en este caso acotada a $t=6$ para $M=8$. En función del porcentaje de autovalores significativos (ver (3.10)), puede darse la situación $t=8$ si el porcentaje es del 100% y $t=7$ para porcentajes inferiores al 100%. La primera situación nunca se dará ya que justamente lo que se busca es manejar un número de autovalores inferior al total de estos. Por otra parte, el caso $t=7$ tras analizar un banco de 1000 imágenes, no es una situación muy frecuente ya que normalmente como mínimo aparece un autovalor con un peso porcentual significativo, por lo que se ha decidido saturar el valor máximo de t a 6. Esta saturación provoca un aumento en el error de recuperación (ε) (4.76), tal y como se puede comprobar en la Figura 4.85. En esta figura se observa como a excepción de un caso (pico máximo de la Figura 4.85) el error inducido es aproximadamente del 1%. En ningún caso este incremento de ε afecta a la detección de objetos, tal y como se verá en el capítulo 5. Por su parte, esta limitación del número máximo de autovectores permite optimizar la ejecución de la Mult. Mat. al posibilitarse accesos múltiples simultáneos a memoria externa.

$$\varepsilon = \left\| \mathbf{\Phi}_j - \hat{\mathbf{\Phi}}_j \right\| = \sqrt{\sum_{i=1}^{N^2} \left(\Phi_{j_i} - \hat{\Phi}_{j_i} \right)^2} \tag{4.76}$$

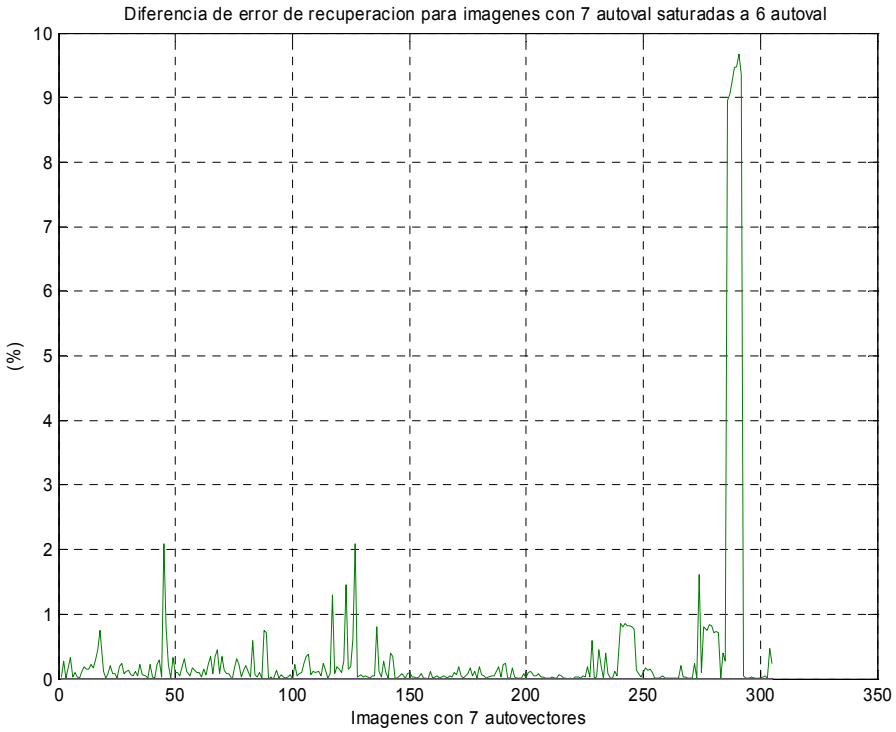


Figura 4.85. Diferencia del error de recuperación, para casos en los que el número de autovectores significativos es 7 ($t=7$), saturando $t_{MAX}=6$.

Para explicar el uso del *array* multiplicador semi-sistólico diseñado para esta tercera Mult. Mat. se ha construido la Figura 4.86. En ésta se observa en color verde el orden en el que son leídos los datos de $\mathbf{U}_t^T$ y $\mathbf{\Phi}_j$, mientras que en color azul se presenta el contenido de las unidades MAC2 de las celdas 1x del *array,* en cada paso. El funcionamiento del sistema es el siguiente:

1. En el primer ciclo de reloj, llegan al *array* simultáneamente los tres primeros datos de $\mathbf{U}_t^T$ y el primero de $\mathbf{\Phi}_j$. Estos, son multiplicados convenientemente (Figura 4.86.a), almacenando cada resultado parcial en la unidad MAC2 correspondiente.
2. En este momento (siempre y cuando $t \geq 4$), el primer dato de $\mathbf{\Phi}_j$ es retenido, y los siguientes datos de $\mathbf{U}_t^T$ son almacenados en el *array*. De esta forma, se realiza la multiplicación de cada nuevo elemento de $\mathbf{U}_t^T$ por el primer dato de $\mathbf{\Phi}_j$, acumulándose los resultados parciales en la unidad MAC2 correspondiente (Figura 4.86.b). En caso que t fuera menor de 4, este segundo paso no se realizaría.
3. Una vez multiplicado el primer dato de $\mathbf{\Phi}_j$ por todos los elementos de $\mathbf{U}_t^T$, se carga el segundo dato de $\mathbf{\Phi}_j$ y se repite lo mismo que en los pasos 1 (Figura 4.86.c) y 2 (Figura 4.86.d), hasta recorrer todos los elementos de $\mathbf{U}_t^T$.

Una vez multiplicados todos los elementos, cada unidad MAC2 dispone de los elementos que forman el vector resultado del producto $\mathbf{U}_t^T\mathbf{\Phi}_j$. Este nuevo vector, posee unas dimensiones bastantes reducidas ($t \times 1$), por lo que éste es almacenado en registros internos de la FPGA.

Figura 4.86. Secuencia de datos de $\mathbf{U}_t^T$ y $\mathbf{\Phi}_j$ y contenido de las celdas MAC2 del *array* semi-sistólico: (a) Generación de los primeros datos para los 3 primeros autovectores, (b) Generación de los primeros datos para los 3 segundos autovectores, (c) Generación de los segundos datos para los 3 primeros autovectores (d), Generación de los segundos datos para los 3 segundos autovectores.

Otro aspecto a analizar, es el tamaño de los datos de salida. Los elementos de la matriz $\mathbf{U}_t^T$, poseen 18 bits (n) mientras que los de $\mathbf{\Phi}_j$ poseen 8 bits (p). Esto hace que teóricamente el tamaño máximo de los datos de $\mathbf{U}_t^T\mathbf{\Phi}_j$, pueda alcanzar 42 bits (4.77). Sin embargo, debido al tamaño de datos manejado por el módulo que divide por las normas al cuadrado (módulo que recoge los datos generados en este apartado), el resultado debe truncarse al tamaño patrón empleado en esta tesis (18 bits). Para ver el error inducido por este truncamiento, sobre el funcionamiento del sistema, en el apartado 4.4 se justificará que dicho truncamiento no introduce mucho error en el resultado final.

$$n^{o}\_bits\_\left(\mathbf{U}_t\mathbf{\Phi}_j\right)=\log_2\left(\sum_{i=1}^{N^2}\left(2_i^n\times 2_i^p\right)\right)=\log_2\left(\sum_{i=1}^{256^2}2_i^{26}\right)=42 \qquad (4.77)$$

Con respecto al tiempo de ejecución de este producto de matrices (T_{E7}), su valor depende del valor de t. Si $t\geq 4$ se deben realizar dos accesos consecutivos a memoria externa, para obtener los tríos de datos de $\mathbf{U}_t^T$ (situaciones Figura 4.86.a y b). Este caso se presenta la Figura 4.87 donde RD_U es la etapa de lectura desde memoria externa de un trío de datos de $\mathbf{U}_t^T$, RD_Φ es la lectura de datos del vector $\mathbf{\Phi}_j$, MULT es el tiempo consumido en realizarse la multiplicación y ACC en la suma y acumulación de datos parciales. Cada una de estas etapas consume un ciclo de reloj. Por su parte si $t\leq 3$ no haría falta ese doble acceso a memoria externa para $\mathbf{U}_t^T$, eliminándose por tanto un tiempo de acceso a memoria. De esta forma, en (4.78) y (4.79), se muestran los tiempos totales (T_{E7}) para $t\geq 4$ (Figura 4.87) y $t\leq 3$, respectivamente.

$$T_{E7}=\left(T_{RD\_U}+T_{MULT}+2T_{ACC}\right)+2T_{ACC}\cdot\left(N^2-1\right)=\left(2+2N^2\right)T_{CLK} \qquad (4.78)$$

$$T_{E7}=\left(T_{RD\_U}+T_{MULT}+T_{ACC}\right)+T_{ACC}\cdot\left(N^2-1\right)=\left(1+N^2\right)T_{CLK} \qquad (4.79)$$

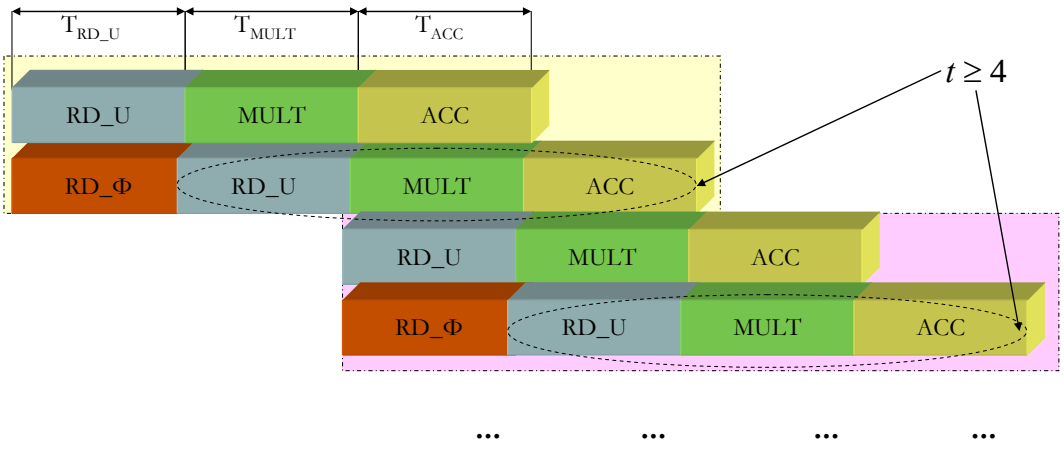


Figura 4.87. Temporización realizada en la generación del vector $\mathbf{\Omega}$.

Es esta multiplicación de matrices, sin duda una de las operaciones más lentas que se producen dentro de PCA, debido principalmente a la magnitud de las matrices a manejar. Para tener una idea aproximada del tiempo consumido por esta etapa, en la Tabla 4.9 se presenta un resumen de los tiempos generados para diferentes frecuencias de reloj.

Tabla 4.9. Tiempos consumidos en la generación del vector $\mathbf{U}_t^T\mathbf{\Phi}_j$ (T_{E7}), dentro de PCA, para diferentes frecuencias de reloj, particularizado para $N^2 = 256^2$ y $M = 8$.

	$t \leq 3$	$t \geq 4$
f_{CLK}=50MHz	1,3 ms	2,6 ms
f_{CLK}=66MHz	992,98 µs	2 ms
f_{CLK}=100MHz	655,37 µs	1,3 ms
f_{CLK}=120MHz	546,14 µs	1,1 ms

4.3.2. Normalización de la imagen proyectada.

Tal y como se expuso en el apartado 4.2.5, la matriz $\mathbf{U}_t$ generada en la fase de modelado del fondo, no está normalizada. Según (4.74) dicha normalización se puede realizar antes o después de la obtención de $\hat{\mathbf{\Phi}}_j$. Debido a las condiciones de segmentación empleadas en este diseño, con idea de disminuir el tiempo de cómputo, dicha normalización se va a hacer a la salida del producto $\mathbf{U}_t^T\mathbf{\Phi}_j$ (Figura 4.88). Como cada uno de los elementos de $\mathbf{U}_t^T\mathbf{\Phi}_j$ se genera en un ciclo de reloj, según van saliendo del *array*, estos son divididos por la norma correspondiente al cuadrado generándose así cada uno de los elementos de $\mathbf{\Omega}'$ (4.75). Es importante recordar que cada una de las normas fue calculada en la fase de modelado de fondo. Por tanto, se debe implementar en una FPGA una operación de división.

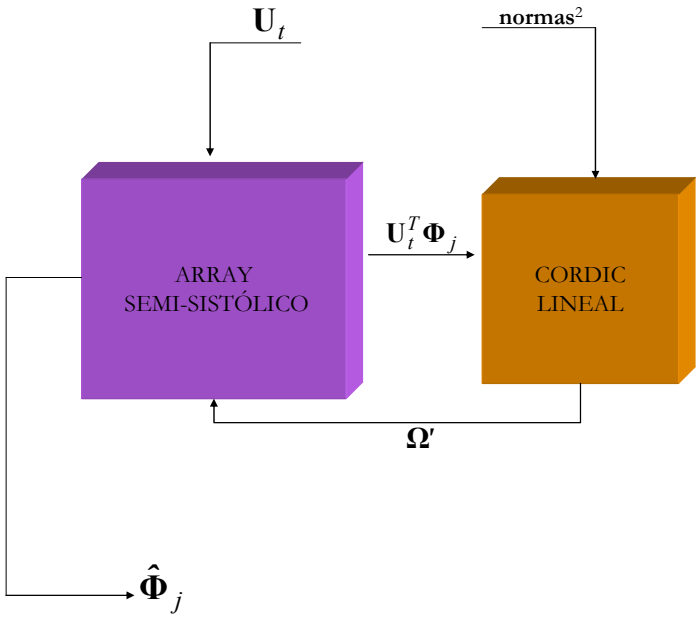


Figura 4.88. Diagrama de bloques del sistema de normalización.

Para realizar esta operación matemática en una FPGA, básicamente existen dos posibilidades: diseño de una unidad de división de propósito específico o el empleo del algoritmo CORDIC. La opción escogida en esta tesis (Figura 4.88) es justamente esta última, ya que el número de recursos consumidos es inferior con respecto a la otra opción.

La división de dos números es factible en CORDIC si éste se utiliza en modo vectorización con un sistema de coordenadas lineales. Por consiguiente, se debe diseñar un nuevo módulo CORDIC que funcione bajo estas condiciones, donde de nuevo, su estructura es bastante similar a la utilizada para el cálculo de autovectores mediante el método de Jacobi (apartado 4.2.3.9). También, se ha optado por utilizar una arquitectura CORDIC paralelo por lo que vencida una latencia inicial se obtienen los resultados correspondientes. El tiempo total del CORDIC lineal es (4.80), donde n es el tamaño de los datos de entrada y t el número de datos de entrada (en nuestro caso es el número de autovectores significativos). Sin embargo, como este proceso se realiza en paralelo con la generación de $\hat{\mathbf{\Phi}}_j$, en el cómputo total de $T_{ON-LINE}$ sólo se introduce una latencia (L_8 de la Figura 4.84) cuyo valor se puede ver en (4.81).

$$T_{CORDIC\_Lineal} = (n+t)T_{CLK} \tag{4.80}$$

$$L_8 = n \cdot T_{CLK} \tag{4.81}$$

Con respecto a los recursos consumidos por el CORDIC Lineal, en la Tabla 4.10 se presenta un resumen de los índices más significativos de su implementación en una XC2VP7.

Tabla 4.10. Recursos consumidos y máxima frecuencia de reloj, del CORDIC Lineal implementado.

Área (*Slices*)	Multiplicadores	BRAM	Máxima frecuencia
295 (8%)	0	0	135MHz

4.3.3. Recuperación de una imagen desde el espacio transformado.

Tras proyectar una imagen sobre el espacio transformado y haber dividido por las normas al cuadrado, dentro del proceso *on-line*, el siguiente paso a realizar es la recuperación de ésta desde el espacio transformado, o lo que es lo mismo, la generación del vector $\hat{\mathbf{\Phi}}_j$ (ver (4.74)). Para ello, de nuevo se tiene que realizar otra Mult. Mat., concretamente, la de la matriz $\mathbf{U}_t$ por el vector $\mathbf{\Omega}'$.

Al igual que ocurría en el apartado 4.3.1, cabe distinguir dos situaciones de funcionamiento diferentes: cuando $6 > t > 4$ y cuando $t \leq 3$. La primera situación consume mayor número de ciclos de reloj que la segunda, ya que se deben realizar más accesos a memoria externa.

De nuevo, para realizar esta cuarta multiplicación de matrices, se emplea el *array* semi-sistólico descrito en el apartado 4.2.2.2. Para explicar el funcionamiento de esta Mult. Mat. dentro del *array*, es importante previamente destacar los siguientes aspectos:

- En esta operación ($\hat{\mathbf{\Phi}} = \mathbf{U}_t \cdot \mathbf{\Omega}'$), el tamaño de las matrices de entrada son $N^2 \times t$ para $\mathbf{U}_t$ y $t \times 1$ para $\mathbf{\Omega}'$. El vector de salida (de dimensiones $N^2 \times 1$) es la imagen recuperada del espacio transformado, y se denomina como $\hat{\mathbf{\Phi}}_j$.

- Los datos de entrada al *array* semi-sistólico, son por una parte la matriz $\mathbf{U}_t$ y por otra $\mathbf{\Omega}'$ (4.75) (salida del CORDIC lineal) (Figura 4.88). De esta forma se obtiene $\hat{\mathbf{\Phi}}_j$ según se expuso en (4.74), eliminando así la operación raíz cuadrada de (4.70).

- El tamaño de los datos es tanto de 18 bits para $\mathbf{U}_t$ como para $\mathbf{\Omega}'$. Por tanto, cada uno de los elementos de $\hat{\mathbf{\Phi}}_j$ puede alcanzar como máximo hasta 39 bits cuando $t = 6$ (4.82). Sin embargo, como $\hat{\mathbf{\Phi}}_j$ se va a restar con $\mathbf{\Phi}_j$ en el cálculo del error de recuperación, el tamaño de los elementos de ambos vectores deben ser iguales. Por tanto los elementos de $\hat{\mathbf{\Phi}}_j$ deben truncarse a 8 bits que es el tamaño de los elementos de $\mathbf{\Phi}_j$. Este truncamiento de 39 a 8 bits, no tiene asociado un gran error de precisión, tal y como se observa en la Figura 4.93 y Figura 4.94. Esto es debido al orden de los valores prácticos obtenidos en $\hat{\mathbf{\Phi}}_j$. Estos poseen valores muy pequeños, no superando nunca el valor 2^7, por lo que directamente se truncan los bits de mayor peso, que son siempre cero.

$$n^o\_bits\_\hat{\mathbf{\Phi}} = \log_2\left(\sum_{i=1}^{t}\left(2_i^n \times 2_i^n\right)\right) = \log_2\left(\sum_{i=1}^{t=6} 2_i^{36}\right) = 39 \qquad (4.82)$$

Para explicar el funcionamiento de esta multiplicación dentro del *array* semi-sistólico, se ha realizado la Figura 4.89 la cual ayuda a comprender la secuencia de funcionamiento en el producto $\mathbf{U}_t \cdot \mathbf{\Omega}'$. En esta multiplicación, de nuevo se emplea únicamente las 3 celdas del *array* denominadas como 1x (ver Figura 4.11). Dentro de este tipo de celdas, cada celda posee dos unidades MAC de propósito específico denominadas como MAC2. En éstas se almacenan los resultados parciales de la Mult. Mat. A continuación se detalla la forma de operar del *array* para esta Mult. Mat.:

1.- Los primeros datos de $\mathbf{\Omega}'$ (4.75), son generados por el CORDIC lineal tras ser vencida su latencia inicial (nT_{CLK}), almacenándose posteriormente en

registros internos de la FPGA. Simultáneamente, se accede a memoria externa para la captura de los 3 primeros datos de $\mathbf{U}_t$. Estos deben ser multiplicados por los 3 primeros de $\mathbf{\Omega}'$, según se muestra en Figura 4.89.a. Los resultados de esta primera multiplicación son almacenados en las unidades MAC2X de las celdas 1x del *array* semi-sistólico (ver Figura 4.89.b). Si $4 \le t < 6$, todavía no ha finalizado la primera fila de datos, por lo que hay que realizar una segunda multiplicación en esta fase tal y como muestra en la Figura 4.89.c y d.

2.- Cuando se tienen completados todos los datos parciales de la primera fila, falta sumarlos entre sí. Para ello, dentro del *array* se ha diseñado un bloque sumador que opera según lo mostrado en la Figura 4.89.e. Si $4 \le t < 6$ funcionan SUM1, SUM2 y SUM3, pero si $t \le 3$ sólo lo hacen SUM1 y SUM2.

3.- Una vez obtenido el primer dato de $\hat{\mathbf{\Phi}}_j$ ($\hat{\phi}_{1,1}$) se deben repetir los pasos 1 y 2, para el resto de filas ($N^2 - 1$).

En cuanto a la temporización de esta multiplicación, el flujo de datos ha sido segmentado en 5 etapas: RD_U (lectura de $\mathbf{U}_t$), RD_Ω' (lectura de $\mathbf{\Omega}'$), MULT (multiplicación de un elemento de $\mathbf{U}_t$ por el correspondiente de $\mathbf{\Omega}'$), SUM1 (suma de los dos primeros resultados parciales), SUM2 (suma de los siguientes resultados parciales) y SUM3 (suma de los últimos resultados cuando $4 \le t < 6$). En función de t se tiene dos segmentaciones diferentes: en (4.83) y la Figura 4.90.a se muestra para el caso de $4 \le t < 6$, mientras que (4.84) y la Figura 4.90.b es el asociado a la situación $t \le 3$. La duración de cada una de las etapas segmentadas es de un ciclo de reloj.

$$\begin{aligned} T_{E9} &= T_{\hat{\Phi}} = L_i + (N^2 - 1)(T_{SUM2} + T_{SUM3}) \\ L_i &= T_{RD\_U} + T_{MULT} + T_{SUM1} + 2T_{SUM2} + T_{SUM3} \end{aligned} \tag{4.83}$$

$$\begin{aligned} T_{E9} &= T_{\hat{\Phi}} = L_i + (N^2 - 1)(T_{SUM2}) \\ L_i &= T_{RD\_U} + T_{MULT} + T_{SUM1} + 2T_{SUM2} \end{aligned} \tag{4.84}$$

Esta Mult. Mat. también consume bastante tiempo para poder generarse dentro de la FPGA. Así, de forma análoga a lo realizado en la generación de $\mathbf{U}_t^T \mathbf{\Phi}_j$, en la Tabla 4.11 se muestra a modo orientativo los tiempos consumidos, con diferentes frecuencias de reloj.

Tabla 4.11. Tiempos consumidos en la generación del vector $\hat{\mathbf{\Phi}}_j$ (T_{E9}), dentro de PCA, para diferentes frecuencias de reloj, particularizado para $N^2 = 256^2$ y $M = 8$.

	$t \leq 3$	$t \geq 4$
f_{CLK}=50MHz	1,31 ms	2,62 ms
f_{CLK}=66MHz	993,01 µs	1.98 ms
f_{CLK}=100MHz	655,39 µs	1,31 ms
f_{CLK}=120MHz	546,16 µs	1,1 ms

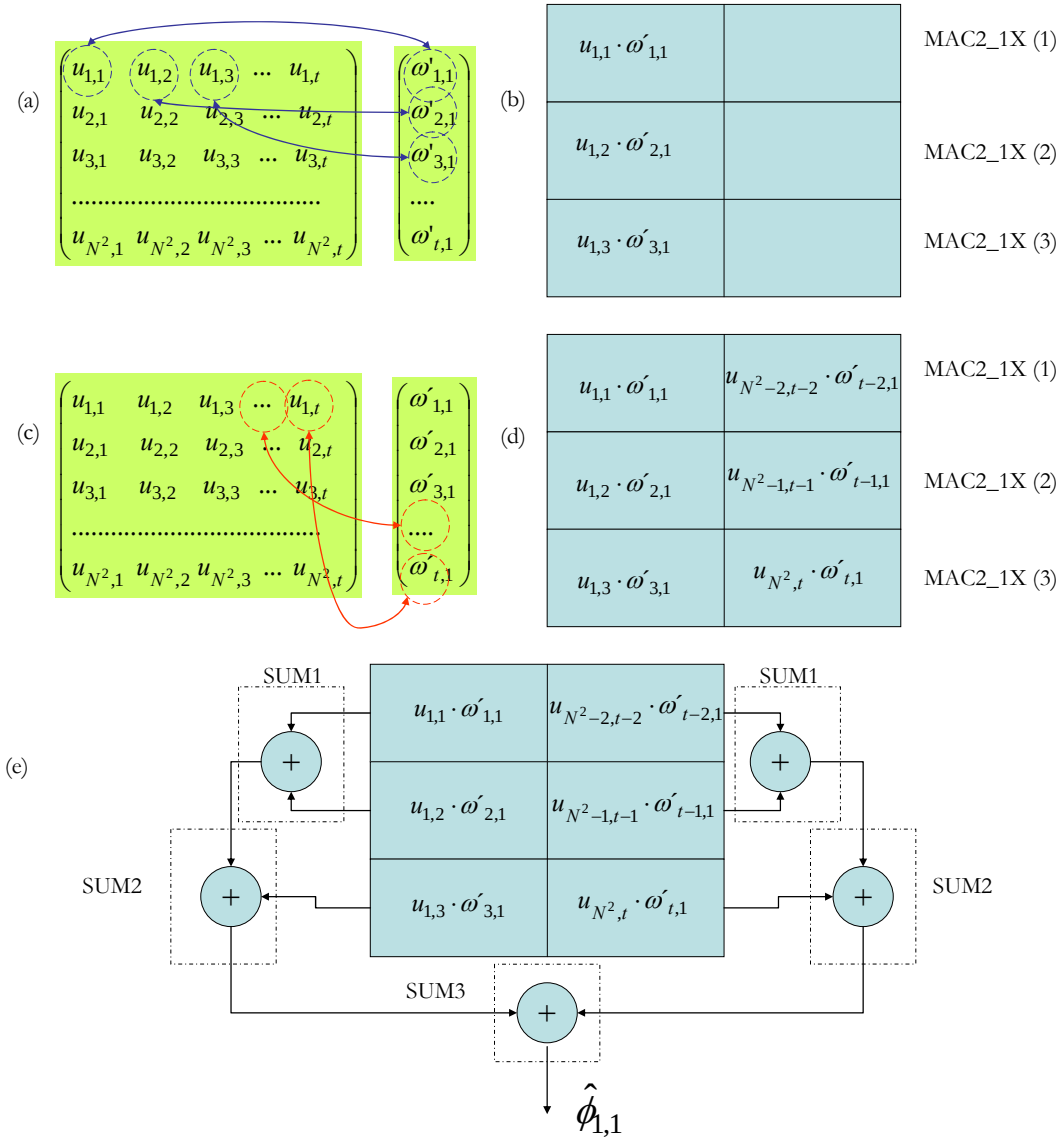


Figura 4.89. Evolución de la multiplicación de $\mathbf{U}_t$ por Ω': a) Primeros datos a multiplicar, b) almacenamiento de a) en las unidades MAC2, c) Segunda fase de multiplicación de los primeros datos para $4 \leq t < 6$ d) almacenamiento de las fases a) y c) en unidades MAC2X, e) Estructura y contenido del sistema sumador del *array* semi-sistólico para esta Mult. Mat.

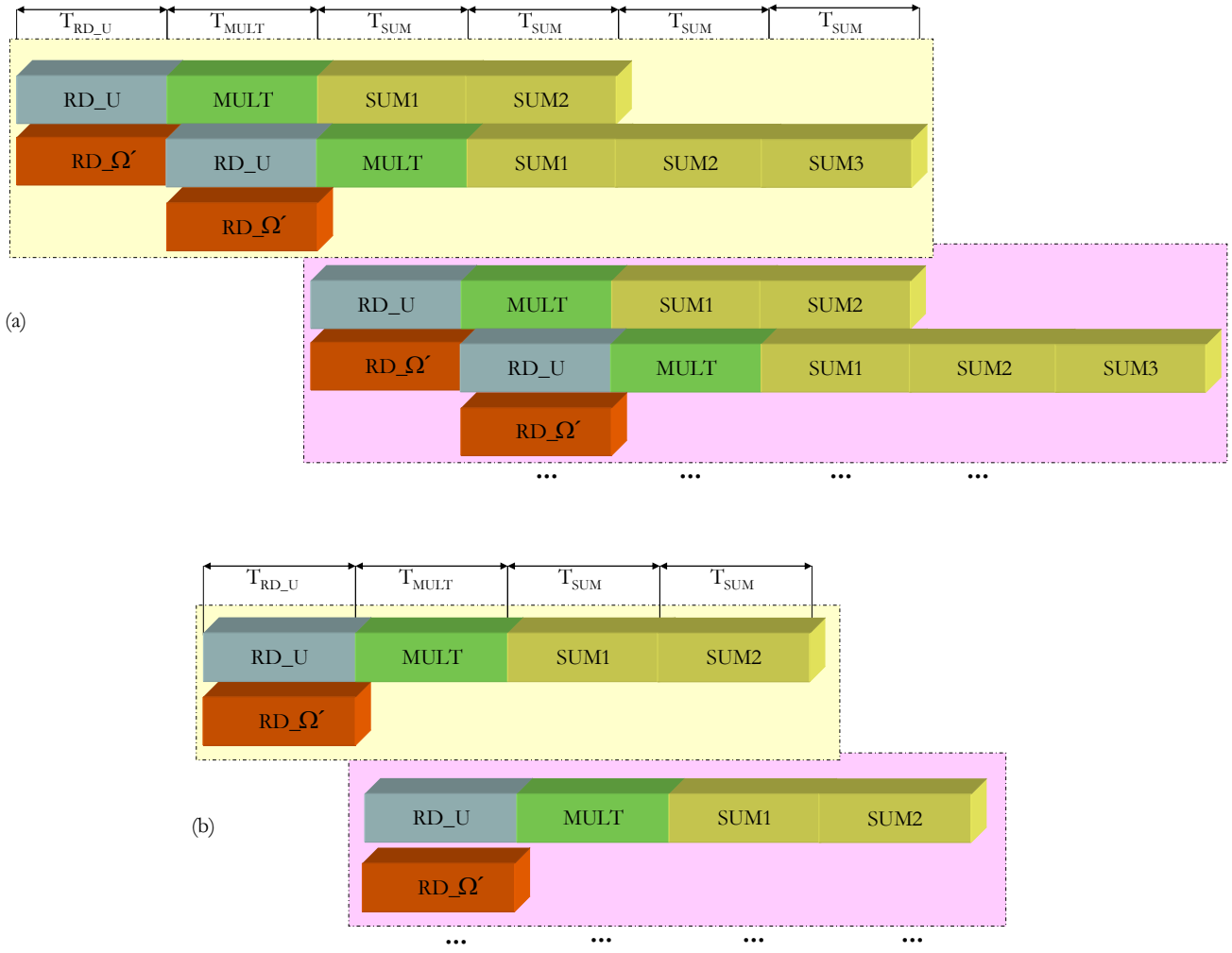


Figura 4.90. Temporización realizada en la generación de la matriz $\hat{\mathbf{\Phi}}$. a) Si $4 \leq t < 6$ b) Si $t \leq 3$.

4.4. RESULTADOS DE LA IMPLEMENTACIÓN DE PCA SOBRE FPGAs.

En este apartado se van a analizar fundamentalmente, los tiempos consumidos por la unión de las dos fases principales de PCA (generación de matriz de autovectores, y *on-line*), así como los recursos consumidos por cada una de ellas. De forma individualizada, se han estudiado previamente los recursos y tiempos consumidos por cada una de las partes de la generación de la matriz de autovectores, así como de la fase *on-line*.

Otro aspecto importante a evaluar es la exactitud del sistema. Para ello, se ha estudiado el efecto que tienen los truncamientos realizados en distintas partes de PCA sobre el error de recuperación (4.76). Concretamente, se va a analizar el truncamiento en la generación de la media, de la matriz de covarianza, de la matriz $\mathbf{U}_t$, de la matriz $\mathbf{\Omega}'$ y del vector $\hat{\mathbf{\Phi}}$.

A la hora de determinar el tiempo consumido por la fase *on-line* y la de generación de autovectores, es muy complicado establecer un tiempo exacto. Esto es debido a la existencia de varios factores variables, los cuales condicionan el tiempo consumido. Estos son:

a) Porcentaje de autovalores deseado (P de la Figura 4.61). En función de este porcentaje se fija el valor que caracteriza el número de autovectores principales (factor t). Así, dependiendo del valor de t obtenido se tiene que analizar dos situaciones: cuando $t \leq 3$ y cuando $4 \leq t \leq 6$, ya que cada una de ellas implica un tiempo diferente con varios accesos a memoria en las operaciones de proyección y recuperación de una imagen (apartado 4.3).

b) Actualización de la matriz de autovectores. Una de las cuestiones que todavía no se ha resuelto es la siguiente, ¿cuándo se debe actualizar la matriz de autovectores con una nueva imagen? Aunque esta pregunta será resuelta en el capítulo 5, es importante tener presente cuándo puede ser necesario calcular la nueva matriz de autovectores (matriz de transformación) incorporando la última imagen capturada.

c) Otro aspecto importante, es el tamaño de los datos de entrada (p), así como el tamaño patrón de los datos intermedios del sistema (n). En nuestro caso, tal y como se ha ido justificando, se ha fijado debido fundamentalmente a optimizaciones de velocidad y área, $p = 8$ y $n = 18$.

d) Por último, otros aspectos que condicionan la velocidad de ejecución son el número de imágenes captadas para formar la matriz de autovectores (M), así como el tamaño de los vectores imagen (N^2). Debido principalmente a limitaciones en el ancho del bus de memoria externa, se ha fijado $M = 8$ y $N = 256$.

En base a lo anteriormente expuesto, el funcionamiento del algoritmo PCA hasta el momento, se realizaría de la siguiente forma:

1.- *Generación de la matriz de autovectores*. Al tiempo consumido por esta fase se le denomina como T_{GEN_U}. Este tiempo internamente está segmentado tal y como muestra la Figura 4.3 y como se ha descrito en el apartado 4.2. Esta fase se ejecuta nada más arrancar y cuando se desea actualizar la matriz de autovectores.

2.- *Fase on-line*. Esta fase se está ejecutando cíclicamente hasta que se decida actualizar la matriz de autovectores. El tiempo empleado por esa fase se denomina como $T_{ON\text{-}LINE}$ estando también segmentado internamente (ver Figura 4.84).

En la Figura 4.91 se muestra el flujograma de funcionamiento de las fases *on-line* y generación de autovectores dentro del sistema final.

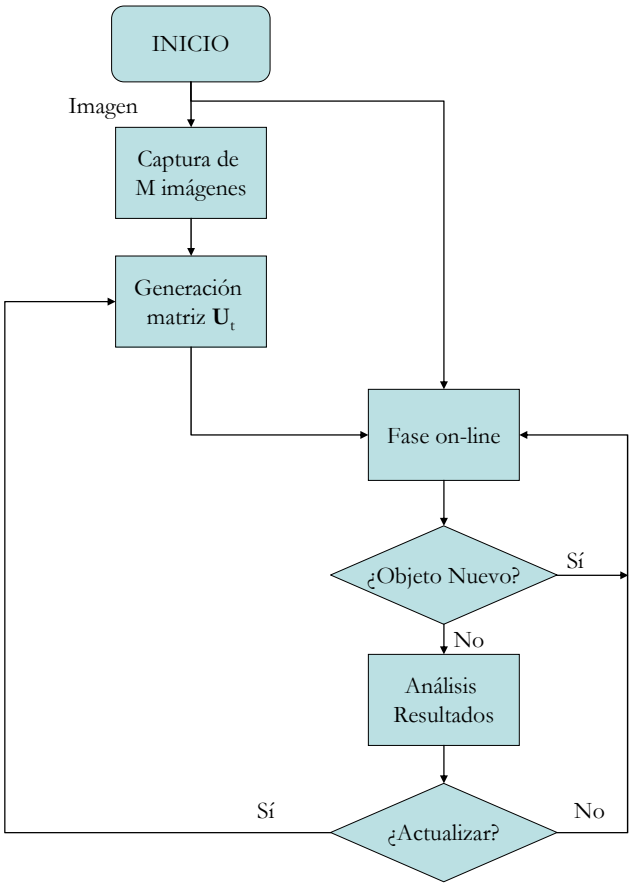


Figura 4.91. Flujograma de funcionamiento continuo de la fase on-line, generación de la matriz de autovectores, detección de objetos y actualización del modelo de fondo.

Una vez expuesto cómo funciona el sistema, a la hora de modelar los tiempos consumidos se van a considerar dos situaciones: tiempos máximos y tiempos mínimos que se podrían consumir. Para obtener las expresiones de ambos, según lo expuesto en el apartado 4.2 y 4.3, en (4.85), (4.86), (4.87) y (4.88) se muestran sus valores particularizando para las situaciones expuestas anteriormente ($M = 8$, $n = 18$, $N = 256$), para las dos fases de PCA.

$$T_{GEN\_U\_MIN} = T_{E1} + T_{E2} + T_{E3} + T_{E4MIN} + T_{E5} = 132789 \cdot T_{CLK} \tag{4.85}$$

$$T_{GEN\_U\_MX} = T_{E1} + T_{E2} + T_{E3} + T_{E4MX} + T_{E5} = 198325 \cdot T_{CLK} \tag{4.86}$$

$$T_{ON-LINE\_MIN} = L_6 + T_{E7MIN} + L_8 + T_{E9MIN} = 131096 \cdot T_{CLK} \tag{4.87}$$

$$T_{ON-LINE\_MX} = L_6 + T_{E7MX} + L_8 + T_{E9MX} = 262169 \cdot T_{CLK} \tag{4.88}$$

Para poder calcular los tiempos totales, también se consideran dos situaciones diferentes:

a) *Actualización continua*: Esta situación sucede cuando cada una de las imágenes evaluadas en la fase *on-line* no tiene objetos nuevos y se

introduce en la fase de generación de autovectores. En este caso el tiempo consumido para b imágenes, se presenta para el mejor de los casos en (4.89), y en (4.90) para el peor.

$$T_{ACT\_CONTINUA\_MX} = \left(T_{GEN\_U\_MX} + T_{ON-LINE\_MX}\right) \cdot b \tag{4.89}$$

$$T_{ACT\_CONTINUA\_MIN} = \left(T_{GEN\_U\_MIN} + T_{ON-LINE\_MIN}\right) \cdot b \tag{4.90}$$

b) *Actualización no continua*: En este caso se supone que de cada b nuevas imágenes captadas se actualiza la matriz de autovectores. En base a la segmentación realizada en la fase *on-line* (ver Figura 4.84), antes de que finalice el procesado de una imagen actual se puede iniciar el de la siguiente. De esta forma, se reduce el tiempo generándose (4.91) para el mejor de los casos y (4.92) para el peor.

$$T_{ACT\_NOCONT\_MX} = T_{GEN\_U\_MX} + b \cdot T_{ON-LINE\_MX} \tag{4.91}$$

$$T_{ACT\_NOCONT\_MIN} = T_{GEN\_U\_MIN} + b \cdot T_{ON-LINE\_MIN} \tag{4.92}$$

Para poder evaluar cuantitativamente estos tiempos, se ha construido la Figura 4.92 en la que se presentan los diferentes tiempos consumidos por los distintos casos expuestos anteriormente manejando diferentes frecuencias de reloj. Se puede observar en dicha figura como a medida que aumenta b, el modo de actualización no continuo ofrece los mejores resultados. En el mejor de los casos evaluados ($b = 10$ y f_{CLK}=120MHz), el sistema desarrollado en esta tesis emplea un tiempo total de unos 12ms en volver a actualizar la matriz de autovectores. Esto implica que en ese intervalo se ha logrado procesar b imágenes sin tener en cuenta en éste el tiempo necesario para realizar la captura desde el sensor, así como el de detección de objeto (capítulo 5). En el peor de los casos, es decir trabajando con f_{CLK}=50MHz y con actualización continua, el tiempo empleado en actualizar la matriz de autovectores y procesar una sola imagen es de unos 9 ms. Esto conlleva una tasa de procesado de imágenes de unos 110 frames/sg. De nuevo, en estos tiempos no se han considerado los de adquisición de imágenes así como el de detección de objetos.

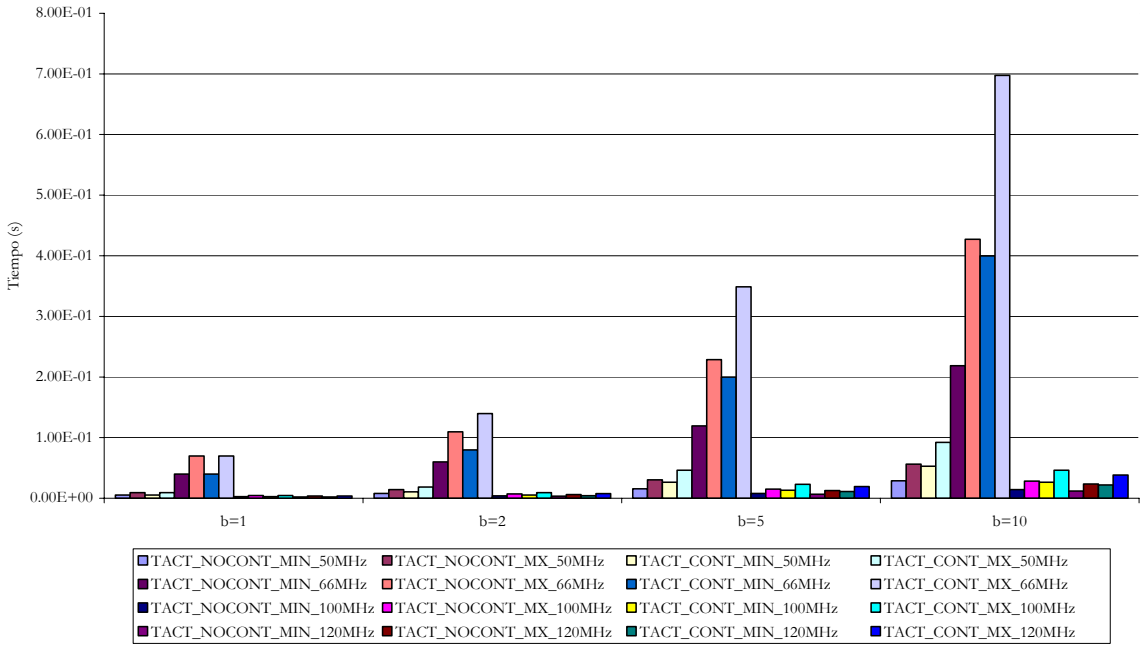


Figura 4.92. Tiempos máximos y mínimos consumidos, para modo de actualización continuo y no continuo, con frecuencias de reloj de 50MHz, 66MHz , 100MHz y 120MHz, en el caso de b=1, 2, 5, 10.

Una vez evaluados los tiempos consumidos, a continuación se analiza la exactitud alcanzada con el sistema desarrollado con respecto a los resultados que se podrían alcanzar en coma flotante. El sistema tal y como se ha ido exponiendo a lo largo de este capítulo ha sido codificado íntegramente en coma fija. Los datos de partida, originalmente vienen codificados con 8 bits, sin embargo debido a las numerosas operaciones aritméticas que se realizan, el tamaño de los datos intermedios va aumentando. Por tanto, estos deben ser truncados a la salida de las diferentes etapas. Para evaluar la importancia de cada truncamiento tanto en escenas con o sin objetos nuevos, en la Figura 4.93 y Figura 4.94 se presentan los errores de recuperación si: se trabajara en coma flotante (*float* en la Figura 4.93 y Figura 4.94), si se truncara únicamente los datos de la media de 9 a 8 bits (*media* en la Figura 4.93 y Figura 4.94), si se truncan los datos de la matriz de covarianza (*C* en la Figura 4.93 y Figura 4.94) a 16 bits para respetar los bits de salvaguarda, si se trunca la salida de la matriz $\mathbf{U}_t$ a 18 bits (*U* en la Figura 4.93 y Figura 4.94), si se trunca la salida del vector $\mathbf{\Phi}$ a 8 bits (FI_2 en la Figura 4.93 y Figura 4.94), si se trunca el vector $\mathbf{\Omega}'$ a 18 bits (*OMEGA* en la Figura 4.93 y Figura 4.94), si se trunca $\hat{\mathbf{\Phi}}$ a 8 bits (FI_{est} en la Figura 4.93 y Figura 4.94), si se aplican todos los truncamientos descritos previamente (*TODO* en la Figura 4.93 y Figura 4.94) y por último si se aplica una máscara de 3x3 elementos para reducir el error de recuperación (capítulo 5) tanto para coma flotante ($float_{V3}$ en la Figura 4.93 y Figura 4.94), como para fija con todos los truncamientos ($TODO_{V3}$ en la Figura 4.93 y Figura 4.94).

Si se analiza la aportación que hace cada uno de los truncamientos sobre el error de recuperación total, tomando el error de recuperación en coma flotante como índice de comparación, se obtiene que:

- El truncamiento de la media genera un incremento en el error de recuperación de un 0.5%.
- El truncamiento de la covarianza genera un incremento en el error de recuperación de un 4%.
- El truncamiento de la matriz $\mathbf{U}_t$ genera un incremento en el error de recuperación de un 0.1%.
- El truncamiento de $\mathbf{\Phi}$ genera un incremento en el error de recuperación de un 5.5%. Este error, aunque en porcentaje es el mayor de todos, no es muy significativo ya que los datos a manejar en esta etapa son muy pequeños, por lo que pequeñas variaciones implican un alto error.
- Tanto el truncamiento de $\mathbf{\Omega}'$ como el de $\hat{\mathbf{\Phi}}$ genera cada uno de ellos dos un incremento de un 0.2%.
- La suma de todos los truncamientos introduce un error con respecto al valor obtenido en coma flotante de un 12%.

Por último, es importante evaluar los recursos consumidos por el sistema descrito en este capítulo. En la descripción detallada de cada uno de los bloques que lo forman se ha ido evaluando el número de recursos consumidos por cada uno de ellos. Especial importancia tienen los módulos de cálculo de autovalores y autovectores, así como el *array* semi-sistólico. Estos dos elementos, debido a su alta carga computacional y complejidad, ocupan casi el 62% de todos los recursos disponibles. En la Tabla 4.12 se presenta un resumen de todos los recursos consumidos por los diferentes módulos que forman este apartado.

Tabla 4.12. Resumen de recursos consumidos y frecuencia de reloj máxima, alcanzado por el módulo de generación de autovectores junto con el de la fase *on-line*.

Área (*Slices*)	BRAM	Multiplicadores	Máxima frecuencia
3761 (75,6%)	35 (80%)	43 (98%)	114MHz

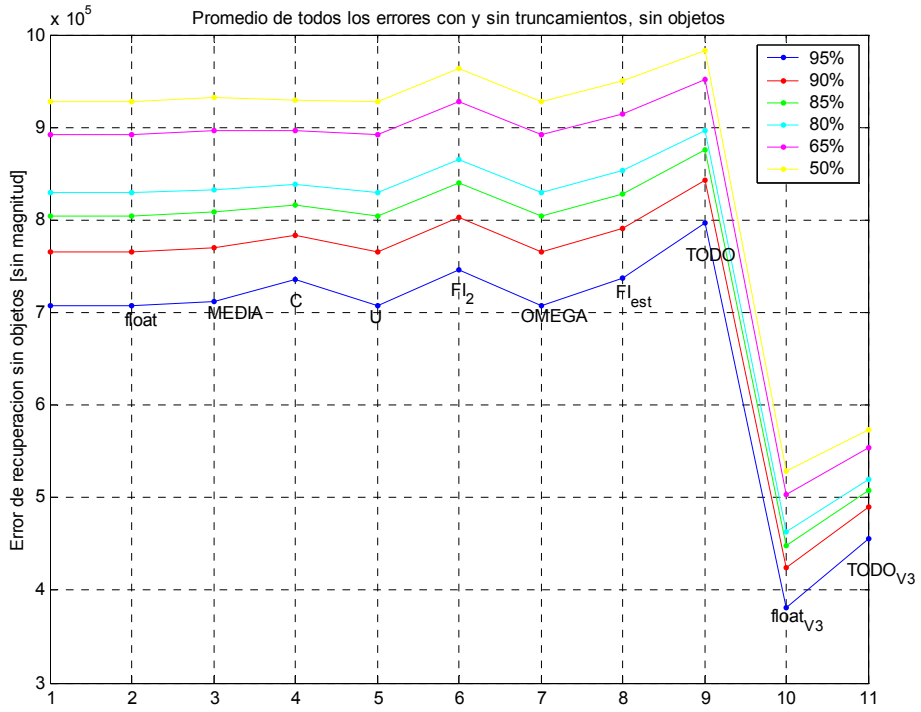


Figura 4.93. Error de recuperación debido a cada uno de los truncamientos, para diferentes porcentajes de autovalores, sin objetos nuevos en la escena a evaluar.

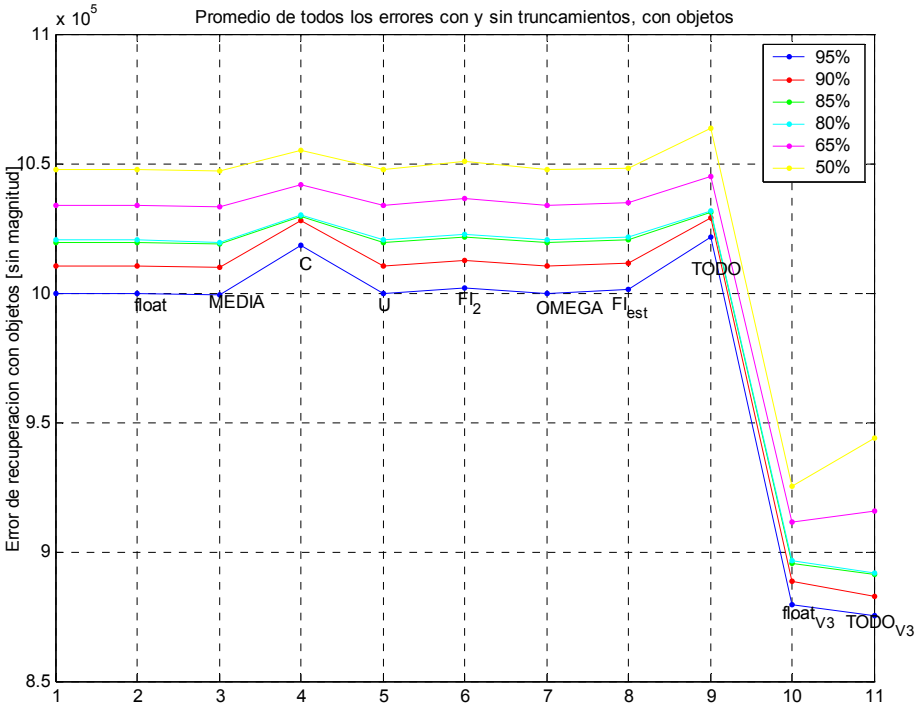


Figura 4.94. Error de recuperación debido a cada uno de los truncamientos, para diferentes porcentajes de autovalores, con objetos nuevos en la escena a evaluar.

4.5. CONCLUSIONES.

Como conclusión general de este capítulo se puede decir que se ha propuesto una alternativa concurrente para la implementación del algoritmo PCA en dispositivos FPGAs. Esta alternativa, orientada a la detección de nuevos objetos en escenas estáticas, describe de una forma robusta el modelo de dicha escena (mediante sus autovectores fundamentales). La detección de nuevos objetos en una escena se realiza mediante el análisis del error de recuperación. En este capítulo se ha calculado dicho valor.

Mediante la arquitectura propuesta en este capítulo se permite ejecutar, con una alta velocidad, el procesado del algoritmo PCA en un dispositivo hardware. Esto conlleva fundamentalmente dos ventajas:

- Mayor aceleración de la ejecución con respecto a un sistema basado en un microprocesador.
- No es necesario emplear un PC como elemento de procesado o control.

Tras los múltiples ensayos realizados la propuesta desarrollada en este capítulo presenta resultados óptimos tanto en entornos exteriores como interiores. Además, el sistema está preparado para poder incorporar a la matriz de autovectores, nuevas imágenes de la escena. De esta forma se consigue añadir al conjunto de características fundamentales nueva información de la escena (por ejemplo cambios de iluminación).

Desde un punto de vista diseño, en este capítulo se ha dado solución a dos importantes aspectos, no encontrándose hasta el momento otros trabajos que mejoren los resultados alcanzados:

1.- *Resolución del cálculo de autovalores y autovectores*. La nueva propuesta desarrollada en esta tesis [Bravo, 2006b], permite acelerar la ejecución de dicho algoritmo hasta un 60% con respecto a un PC, mejorando además índices como frecuencia o recursos consumidos con respecto a otros trabajos implementados en FPGAs. Además, una de las principales aportaciones de la alternativa propuesta es el empleo de una novedosa arquitectura, la cual ocupa un número muy bajo de recursos internos de la FPGA, comparada con la arquitectura sistólica clásica.

2.- *Multiplicación de matrices en FPGAs*. Otro aspecto al que se ha dado solución en este capítulo es el desarrollo de un sistema de multiplicación de matrices de gran tamaño en una FPGA. La alternativa desarrollada permite ejecutar, dentro de una FPGA, cuatro tipos diferentes de multiplicación de matrices, siendo todas ellas de gran tamaño. Tras evaluar diferentes algoritmos se optó por implementar un *array* semi-sistólico, el cuál permite alcanzar unos bajos tiempos de cómputo.

El número de recursos consumidos por estas dos soluciones es bastante elevado. Sin embargo, ambas forman el núcleo de PCA por lo que el resto de partes que faltan, no ocuparan muchos elementos internos. Además, todo el sistema ha sido desarrollado para una FPGA de gama media-baja. Esto permite abaratar costes en una posible transferencia de este sistema, al mundo comercial.

Desde el punto de vista de exactitud, se ha hecho hincapié en alcanzar la mayor precisión posible. Aunque el sistema ha sido codificado íntegramente en coma fija, se han analizado todas las posibles causas que puedan mejorar dicha exactitud, intentando en todo momento aportar soluciones para asemejarla a los resultados generados en coma flotante.

5. IMPLEMENTACIÓN DE UN UMBRAL ADAPTATIVO PARA LA DETECCIÓN DE OBJETOS EN MOVIMIENTO. ACTUALIZACIÓN DEL MODELO DE FONDO

5.1. INTRODUCCIÓN.

En este capítulo se presenta la solución desarrollada para la implementación en hardware reconfigurable de un umbral dinámico (Th_{MD}) (5.1) que permite detectar la aparición de nuevos objetos a partir de la imagen actual menos la media ($\mathbf{\Phi}_j \in \Re^{N^2 \times 1}$) y la imagen proyectada y recuperada del espacio transformado ($\hat{\mathbf{\Phi}}_j \in \Re^{N^2 \times 1}$). Además también se describe la alternativa empleada para la actualización selectiva del modelo de fondo.

El cálculo de Th_{MD} presenta dificultades ya que debe ser adaptativo y su valor es función tanto de las características de la escena a analizar como de las condiciones de iluminación. Por ello, en este capítulo se propone un nuevo método de cálculo del umbral de forma dinámica, que minimice las falsas detecciones de nuevos objetos dentro de la escena de interés, tendiendo en cuenta que cuando el error de recuperación sea superior a ese umbral, se considerará que existe como mínimo un objeto candidato a ser clasificado como objeto nuevo en la escena (5.1).

$$\begin{aligned} \left\| \mathbf{\Phi}_j - \hat{\mathbf{\Phi}}_j \right\| &\leq Th_{MD} \quad \textit{No hay objetos nuevos en la escena} \\ \left\| \mathbf{\Phi}_j - \hat{\mathbf{\Phi}}_j \right\| &> Th_{MD} \quad \textit{Hay objetos nuevos en la escena} \end{aligned} \tag{5.1}$$

Además de la detección de objetos nuevos en una escena, para lo cual sería suficiente con el cálculo del error de recuperación (4.76), también resulta interesante conocer su ubicación espacial aproximada. Por ello, se propone en esta tesis la construcción de un mapa de errores de recuperación o mapa de distancias que permite localizar espacialmente los objetos nuevos. La primera aproximación que se puede plantear es la construcción de un mapa de distancias cuyo tamaño sea el mismo que $\hat{\mathbf{\Phi}}_j$ y $\mathbf{\Phi}_j$, donde cada una de sus posiciones sea la distancia euclídea píxel a píxel entre $\hat{\mathbf{\Phi}}_j$ y $\mathbf{\Phi}_j$. Esta opción, cuya implementación hardware es inmediata, presenta el inconveniente de ser poco robusta frente al ruido, ya que en el caso de que un píxel bien de $\hat{\mathbf{\Phi}}_j$ o de $\mathbf{\Phi}_j$ se viera perturbado por ruido, el error de recuperación asociado sería elevado. Para ello, se propone un sistema basado en máscaras de tal forma que se promedie el valor de la distancia euclídea entre los píxeles adyacentes. De esta forma, posibles perturbaciones de ruido se verán atenuadas.

Una vez obtenido el mapa de distancias, para calcular el grado de similitud entre una nueva imagen y la escena de referencia, se realiza una umbralización del mapa de distancias con un umbral dinámico Th_{MD}, detectando así la presencia de nuevos objetos en la escena bajo estudio (no es objeto de esta tesis efectuar una clasificación de los objetos que aparecen, sino simplemente detectar su aparición).

La primera propuesta de aproximación de todo este proceso, para su implementación sobre una FPGA, se muestra en la Figura 5.1. Los bloques mostrados en la Figura 5.1 son:

1. El primer bloque que se presenta es el denominado como *Generación de imagen promediada*. Como su nombre indica este bloque se encarga de realizar el promediado de cada píxel con sus $q \times q$ píxeles vecinos con objeto de minimizar el ruido asociado a cada píxel. Para ello se aplica una máscara de $q \times q$ elementos sobre $\hat{\mathbf{\Phi}}_j$ y $\mathbf{\Phi}_j$, obteniéndose $\hat{\mathbf{\Phi}}_{vj} \in \Re^{N^2 \times 1}$ (vector $\hat{\mathbf{\Phi}}_j$ promediado) y $\mathbf{\Phi}_{vj} \in \Re^{N^2 \times 1}$ (vector $\mathbf{\Phi}_j$ promediado).
2. El siguiente bloque que se presenta en la Figura 5.1 es el de *Construcción del mapa de distancias*. Una vez obtenidos $\hat{\mathbf{\Phi}}_{vj}$ y $\mathbf{\Phi}_{vj}$, en este bloque se genera el mapa de distancias. Cada uno de los elementos de este mapa ($\varepsilon_{w,i}$) corresponde al error de recuperación o distancia euclídea del elemento $\Phi_{vj_{w,i}} \in \mathbf{\Phi}_{vj}$ menos $\hat{\Phi}_{vj_{w,i}} \in \hat{\mathbf{\Phi}}_{vj}$ donde

$w,i=1,..,N$. Al mapa creado por esta primera propuesta se le denomina como MD_{V1} (Mapa de Distancias promediado de la primera propuesta). La Figura 5.2 muestra la generación de cada elemento perteneciente a MD_{V1} ($\varepsilon_{w,i}$) particularizado para una máscara de 3x3 elementos.

3. El bloque de *Análisis del Mapa de distancias y Detección de Objetos*, evalúa el MD_{V1} para determinar la existencia de un nuevo objeto en la escena.

4. Por último los bloques *Decisión Actualización Modelo de Fondo* y *Actualización del Modelo de Fondo*, determinan si la imagen actual bajo estudio en base a la información característica que aporta, debe o no ser añadida al modelo de fondo.

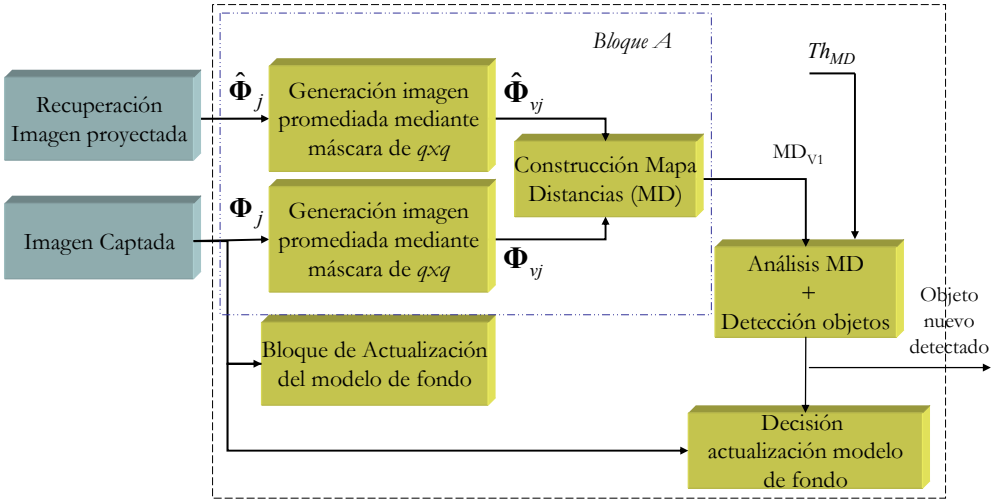


Figura 5.1. Primera propuesta para la actualización del modelo de fondo y para la detección de nuevos objetos.

La arquitectura propuesta en la Figura 5.1 puede optimizarse desde un punto de vista de recursos consumidos sin penalizar su exactitud ni su velocidad de ejecución. Concretamente el Bloque A de la Figura 5.1 puede ser sustituido por el Bloque A de la Figura 5.3. De esta forma se elimina un elemento de *Generación de la imagen promediada*, quedando el flujo de diseño de la siguiente forma (Figura 5.3):

1. *Construcción del Mapa de Distancias*: El mapa de distancias generado se construye a partir de $\hat{\mathbf{\Phi}}_j$ y $\mathbf{\Phi}_j$. A este nuevo mapa se le denominará como $\mathbf{MD}_2 \in \Re^{N\times N}$ (Mapa Distancias de la propuesta 2).

2. *Promediado del Mapa de Distancias*: Este bloque se encarga de aplicar una máscara de $q\times q$ elementos a cada uno de los elementos del mapa de distancias $\mathbf{MD}_2$, obteniéndose un nuevo Mapa de Distancias promediado $\mathbf{MD}_{V2} \in \Re^{N\times N}$.

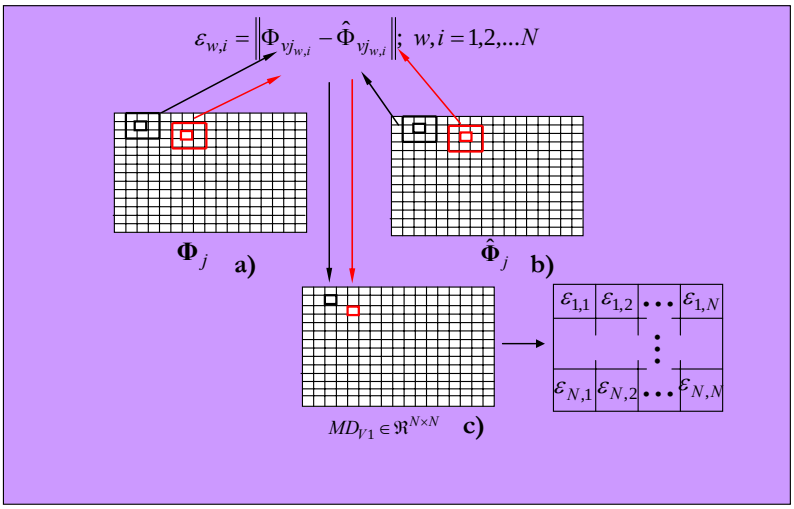


Figura 5.2. Representación gráfica de: a) Máscara 3x3 elementos aplicada sobre un elemento de la imagen captada menos la media ($\mathbf{\Phi}_{j,i}$), b) Ventana 3x3 elementos aplicada sobre un elemento de la imagen recuperada ($\hat{\mathbf{\Phi}}_{j,i}$), c) Mapa de distancias euclídeas promediado (MD_{V1}).

3. Una vez generados todos los elementos de $\mathbf{MD}_{V2}$, el bloque *Análisis de Mapa de Distancias* determina la existencia de nuevos objetos mediante la umbralización del $\mathbf{MD}_{V2}$ con un umbral Th_h. De nuevo el valor del umbral debe obtenerse de forma adaptativa. El tiempo que tarda esta etapa en ejecutarse se denomina como T_{UMBRAL}.
4. Por último el bloque *Actualización del Modelo de Fondo*, el cual se ejecuta en paralelo con el bloque de *Construcción del Mapa de Distancias*, determina en base a la luminosidad de todos los píxeles de $\mathbf{\Phi}_j$ si esa nueva imagen debe ser añadida al modelo de la escena.

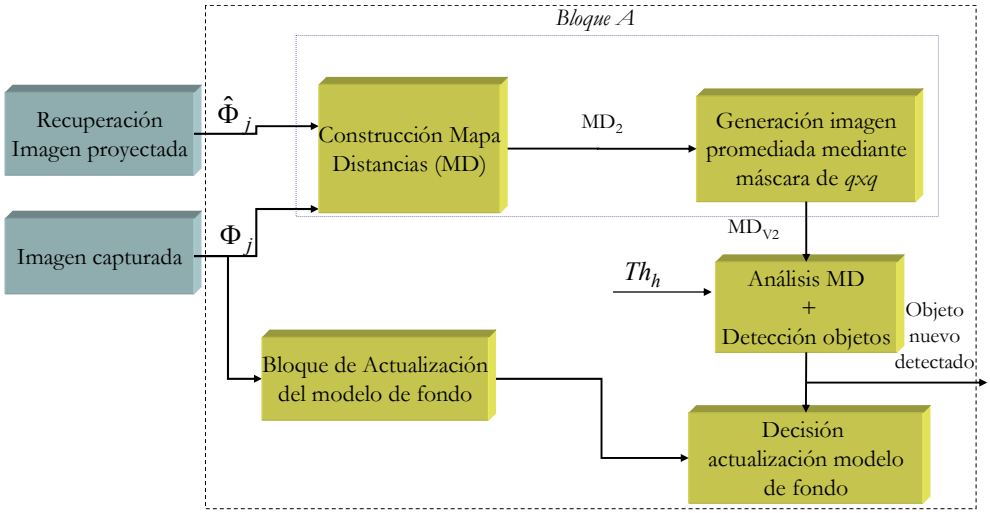


Figura 5.3. Segunda propuesta para el sistema formado por la construcción del MD, detección de nuevos objetos y actualización del modelo de fondo.

Debido al ahorro de recursos internos de esta segunda alternativa, ésta es la que se ha desarrollado finalmente en esta tesis

Para poder modelar completamente, desde el punto de vista temporal, la secuencia del sistema mostrado en la Figura 5.3 es necesario previamente cuantificar los tiempos consumidos por las fases de *Construcción del Mapa de Distancias* y *Etapa de Promediado del Mapa de Distancias*. Con respecto a la fase de *Construcción del Mapa de Distancias*, ésta se inicia en el momento que se recibe el primer elemento de $\mathbf{\Phi}_j$ y de $\hat{\mathbf{\Phi}}_j$. Tras vencer una latencia inicial (L_{MD}) se obtiene el primer elemento del $\mathbf{MD}_2$ generándose a continuación en cada ciclo de reloj un nuevo elemento. Al tiempo total de esta fase se le denomina como T_{MD}.

Por otra parte, la *Etapa de Promediado del Mapa de Distancias* se inicia tras haberse recibido los primeros $2N + p$ elementos, siendo $N \times N$ la dimensión de las imágenes $\mathbf{\Phi}_j$ y $\hat{\mathbf{\Phi}}_j$, y q el tamaño de la máscara de promediado. Por consiguiente, para generarse en este bloque el primer elemento también hace falta una latencia denominada como L_V. El tiempo total empleado por este segundo bloque viene dado por T_V.

Una vez considerados todos los tiempos consumidos por cada bloque de esta segunda alternativa, en la Figura 5.4 se muestra la secuencia temporal de los pasos que conlleva su procesamiento, particularizado para el caso de no actualización del modelo de fondo. En caso de haber actualización del fondo, el tiempo total que consumiría esta fase viene dado por (5.2) donde b es el número de imágenes analizadas desde la última actualización de fondo.

$$T_{CALC\_D} = b \cdot \left(L_{MD} + L_V + T_{UMBRAL} \right) \tag{5.2}$$

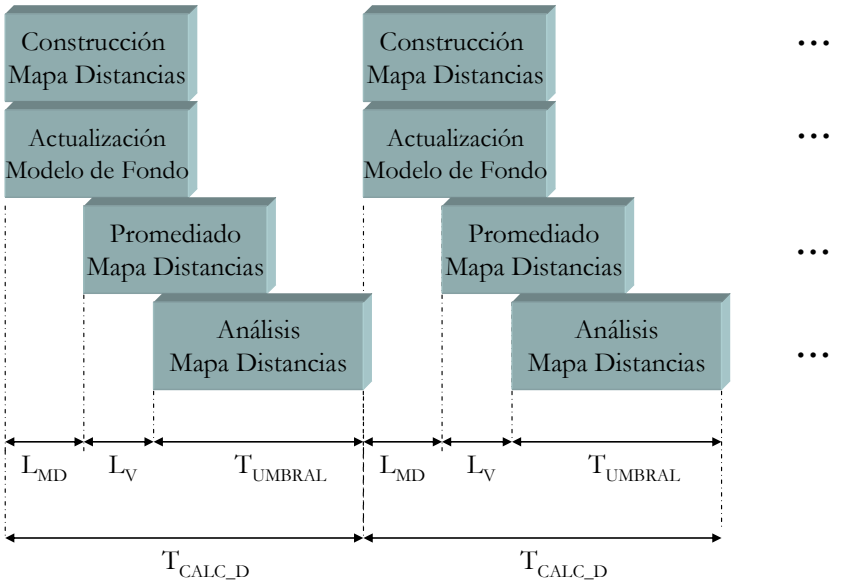


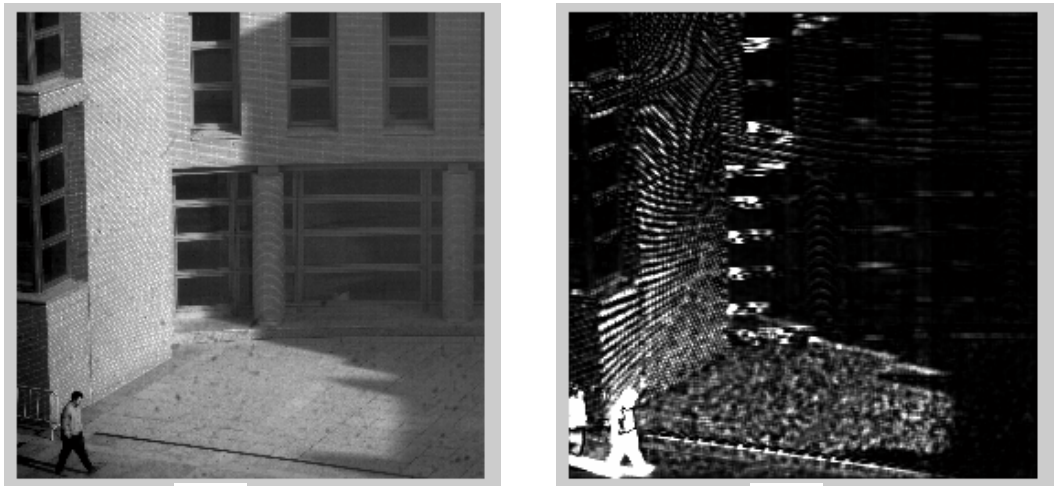
Figura 5.4. Secuencia temporal de funcionamiento para la detección de nuevos objetos mediante el sistema propuesto en la Figura 5.3.

5.2. CONSTRUCCIÓN DEL MAPA DE DISTANCIAS.

Siguiendo la propuesta mostrada en la Figura 5.3, la primera fase para la detección de nuevos objetos es obtener el mapa de distancias $\mathbf{MD}_2 \in \Re^{N \times N}$ a partir de $\mathbf{\Phi}_j$ y de $\hat{\mathbf{\Phi}}_j$. Este mapa de distancias ($\mathbf{MD}_2$) se obtiene a partir de (5.3), siendo ε'_i el cuadrado de la distancia euclídea entre cada elemento $\Phi_{ji} \in \mathbf{\Phi}_j$ y $\hat{\Phi}_{ji} \in \hat{\mathbf{\Phi}}_j$ $i = 1,...,N^2$ (para imágenes de tamaño $N \times N$). En la Figura 5.5.b se muestra el contenido del $\mathbf{MD}_2$ para el ejemplo de imagen mostrado en la Figura 5.5.a.

$$\varepsilon'_i = \left\| \Phi_{j_i} - \hat{\Phi}_{j_i} \right\|^2 \quad i = 1,..., N^2 \tag{5.3}$$

El hecho de trabajar con el cuadrado de la distancia euclídea en vez de la distancia euclídea (4.76), facilita el diseño del hardware asociado a esta función, ya que con ello se evita tener que realizar la operación de la raíz cuadrada. Por otra parte se comprueba que el uso del cuadrado de la distancia euclídea provoca que los valores de distancia asociados a los píxeles pertenecientes a objetos nuevos en el mapa de distancias $\mathbf{MD}_2$, estén más alejados que los que pertenecen al fondo, con lo cual se facilita su posterior umbralización; y por otra parte, además se reduce el número de recursos consumidos en la FPGA y se acelera el tiempo de ejecución del algoritmo de obtención del mapa de distancias. Por tanto la obtención de $\mathbf{MD}_2$ conlleva solamente una operación de resta y otra de multiplicación. En la Figura 5.6 se muestra el diagrama de bloques del hardware asociado al cálculo de MD_2 utilizando (5.3).



(a) (b)

Figura 5.5. Ejemplo de mapa de distancias $\mathbf{MD}_2$. (a) Imagen actual Φ_j. (b) $\mathbf{MD}_2$ de la imagen actual (a).

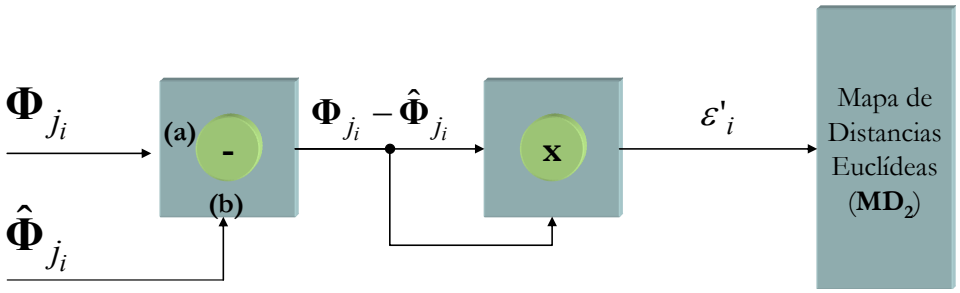


Figura 5.6. Diagrama de bloques del sistema hardware que calcula el cuadrado de la distancia euclídea.

Para analizar el comportamiento temporal de este circuito es importante resaltar la recepción secuencial de cada $\hat{\mathbf{\Phi}}_{j_i}$ y $\mathbf{\Phi}_{j_i}$. De esta forma tras dos ciclos de reloj (uno para la resta (T_{RESTA}) y otro para la multiplicación (T_{MULT})) se genera cada ε'_i correspondiente. En la Figura 5.7 se presenta esta secuencia de funcionamiento.

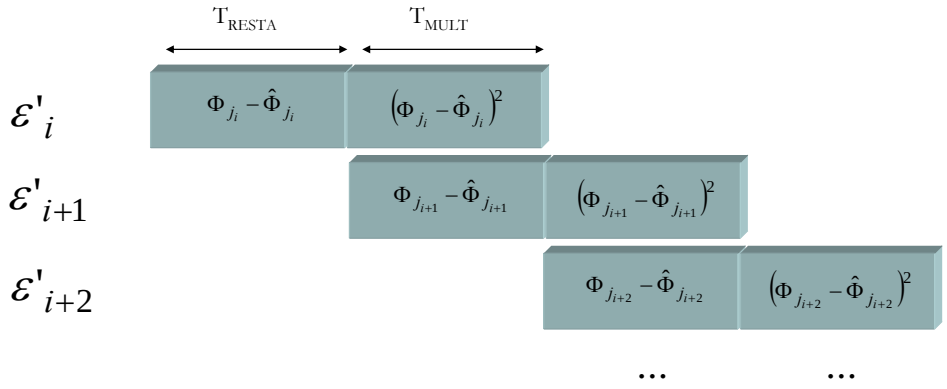


Figura 5.7. Secuencia temporal de funcionamiento para la generación del $\mathbf{MD}_2$.

Por tanto, en base al funcionamiento descrito en la Figura 5.7, el tiempo total de ejecución de esta fase (T_{MD}) es:

$$T_{MD} = (T_{MULT} + T_{RESTA}) + (N^2 - 1)T_{MULT} = 2T_{CLK} + (N^2 - 1)T_{CLK} \qquad (5.4)$$

Una vez que se generen los primeros elementos del mapa de distancias ($\mathbf{MD}_2$), se puede iniciar la generación del mapa de distancias promediado ($\mathbf{MD}_{V2} \in \Re^{N \times N}$). Esto implica que la ejecución del $\mathbf{MD}_2$ se solapa con la generación del ($\mathbf{MD}_{V2}$) (ver Figura 5.4) por lo que el único tiempo a tener en cuenta en la fase de generación de los elementos ε'_i del $\mathbf{MD}_2$ es el denominado como L_{MD} (latencia de generación de los primeros datos del $\mathbf{MD}_2$). Suponiendo un ciclo de reloj para realizar la fase de multiplicación y otro para la fase de resta del circuito de la Figura 5.6, el valor de L_{MD} es de 2 ciclos de reloj ($L_{MD} = 2T_{CLK}$).

En cuanto al número de recursos internos consumidos, hacer notar que la arquitectura desarrollada apenas consume recursos de la FPGA ya que únicamente emplea para su funcionamiento un multiplicador hardware y un restador.

5.2.1. Generación de imagen promediada mediante una máscara de *qxq* elementos en FPGAs.

Desde un punto de sensibilidad al ruido, el $\mathbf{MD}_2$ obtenido previamente es poco inmune, ya que cualquier píxel bien de $\hat{\mathbf{\Phi}}_j$ o de $\mathbf{\Phi}_j$ que esté contaminado por ruido, será detectado como píxel de nuevo objeto al realizar la distancia euclídea asociada al mismo tomando valores distintos de cero. Por ello, con objeto de atenuar ruidos en píxeles aislados dados en $\mathbf{MD}_2$ se propone el uso de una máscara de $q \times q$ elementos que promedie los píxeles adyacentes del $\mathbf{MD}_2$, esto es, se debe hacer un filtrado paso bajo de $\mathbf{MD}_2$. La matriz resultante de aplicar la máscara de $q \times q$ elementos sobre cada elemento de $\mathbf{MD}_2$ se le denominará como $\mathbf{MD}_{V2}$ y a cada uno de sus elementos como $\varepsilon'_{v_i,w}; i,w = 1,..,N$.

Por tanto, el primer paso que se debe realizar es la determinación del tamaño óptimo de q tanto desde un punto de vista de resultados como de recursos consumidos. Una vez obtenido éste se elegirá la arquitectura hardware óptima que implemente este algoritmo en la FPGA.

5.2.1.1. Determinación del tamaño de la máscara óptima.

A la hora de encontrar la dimensión óptima de la máscara, se debe hacer notar que cuanto mayor sea la dimensión de la máscara mayor es el ruido filtrado, pero ello puede suponer la eliminación de datos pertenecientes a $\mathbf{MD}_2$ cuyo valor distinto de cero no sea debido al ruido sino debido a la presencia de un nuevo objeto y la pérdida de localización espacial. Para justificar esta afirmación, se han simulado diferentes tamaños de máscaras con diferentes $\mathbf{MD}_2$ generados en la FPGA todos ellos codificados en coma fija. Concretamente, se han evaluado máscaras de tamaños: 3x3, 5x5 y 7x7 elementos sobre un banco de pruebas de $\mathbf{MD}_2$ construido en base a 1000 imágenes captadas, tanto en entornos interiores como exteriores. En la Figura 5.8 se presenta el porcentaje de reducción del error de recuperación $e(\%)$ (5.5), donde $e(\%)$ representa la diferencia entre el error de recuperación del $\mathbf{MD}_2$ (E) y el error de recuperación del mapa de distancias promediado ($\mathbf{MD}_{V2}$) por una máscara de tamaño $q \times q$ (E_{vq}), estando todo ello normalizado con respecto a E. Si en la Figura 5.8 el 100% simboliza el valor óptimo a alcanzar, se justifica en todos los casos la mejora del $e(\%)$ acentuándose dicha mejora para la máscara de mayor tamaño evaluado ($q = 7$). Sin embargo el empleo de tamaños grandes de máscara implica un aumento considerable en el tiempo de cómputo (Figura 5.9) y en el número de recursos de la FPGA empleados.

$$e(\%) = \left| \frac{E - E_v}{E} \right| \cdot 100;\ E = \sum_{i=1}^{N^2} \varepsilon'_i;\ E_{vq} = \sum_{i=1}^{N^2} \varepsilon'_{v_i};\ q = 3,5,7 \qquad (5.5)$$

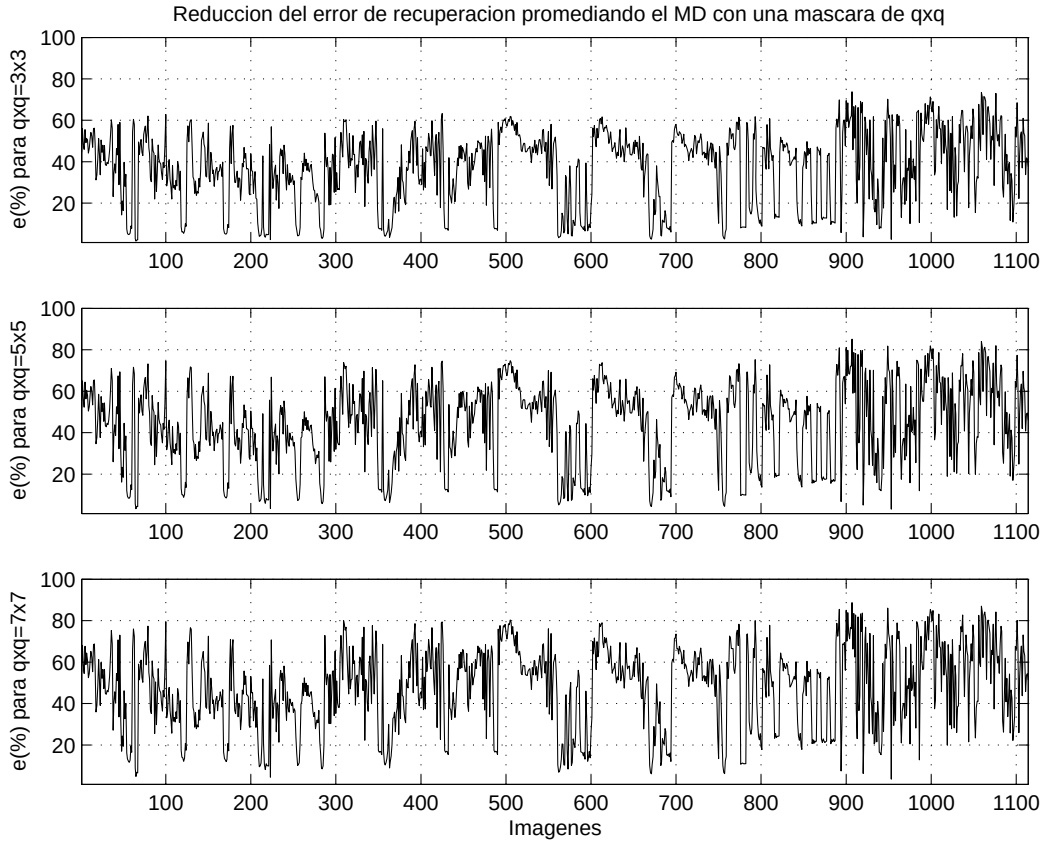


Figura 5.8. Reducción del error de recuperación en el mapa de distancias $\mathbf{MD}_2$ con respecto al mapa de distancias promediado para diferentes máscaras de tamaño $q \times q$.

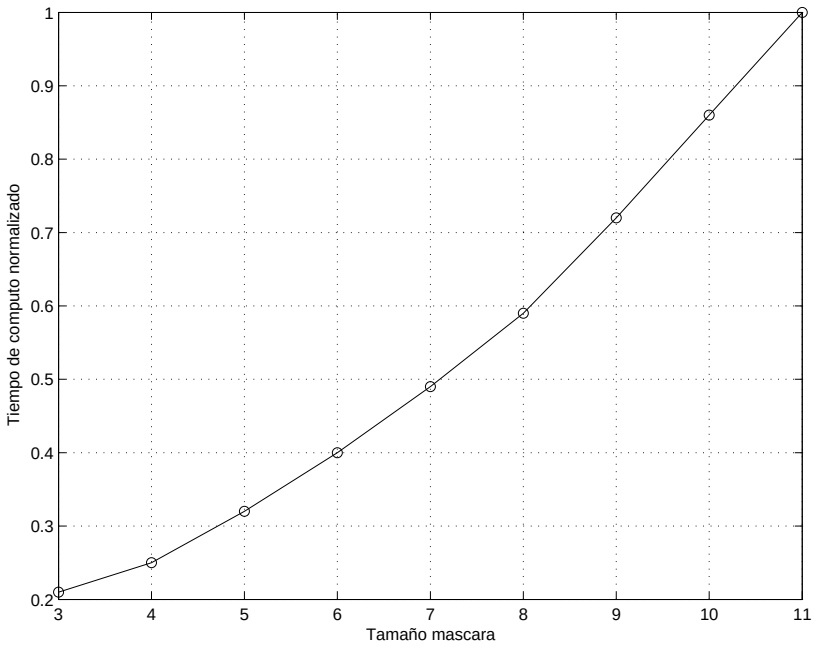


Figura 5.9. Tiempo de cómputo normalizado para determinar el cuadrado de la distancia Euclídea, con imágenes de 240 x 240 píxeles y dimensión de máscaras desde *q=3* hasta *q=11* [Vázquez, 2006].

En nuestro caso se ha establecido como principal objetivo mejorar las prestaciones en cuanto a tiempo de ejecución y recursos consumidos, por lo que se

ha optado por emplear una máscara de 3x3 elementos, la cual aporta un cierto grado de robustez y fiabilidad del sistema frente al ruido.

5.2.2. Implementación hardware del sistema de promediado mediante máscaras de 3x3 elementos.

Una vez justificado el empleo de una máscara de 3x3 elementos a continuación se va a describir la arquitectura hardware diseñada para ello. El sistema hardware que realiza dicho promediado debe ejecutar la convolución del mapa de distancias $\mathbf{MD}_2$ con una máscara de 3x3 elementos donde cada uno de ellos vale 1/9 (ver Figura 5.10) y donde el conjunto de cada uno de los elementos resultantes ($\varepsilon'_{v_{i,w}}$; $i,w=1,..,N$) forman el mapa de distancias promediado ($\mathbf{MD}_{V2}$).

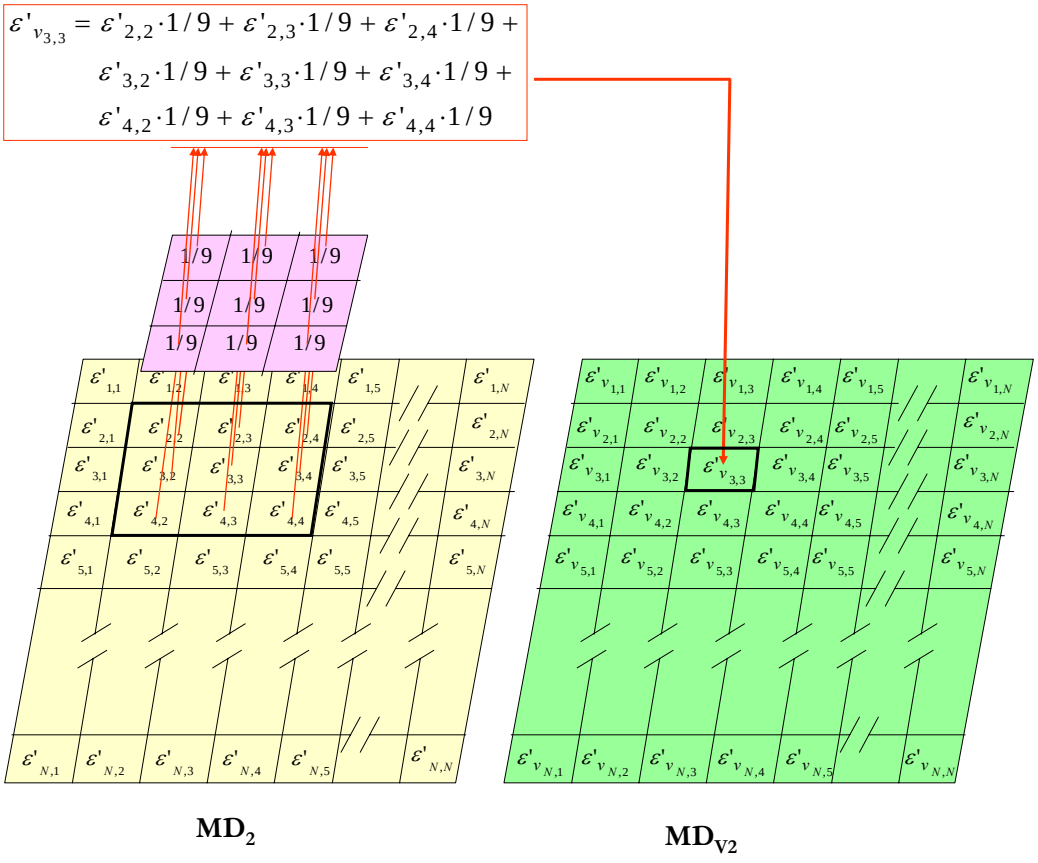


Figura 5.10. Convolución del $\mathbf{MD}_2$ por una máscara de 3x3 elementos de valor 1/9.

Para seleccionar la alternativa de implementación hardware óptima se han diseñado varias propuestas de convolucionadores, evaluando en todo momento el tiempo de ejecución así como el número de recursos internos consumidos de la FPGA [Bravo, 2003] [Bravo, 2003b] [Bravo, 2005].

La propuesta finalmente desarrollada ha sido la que mejores resultados ha ofrecido tanto desde el punto de vista de tiempos de ejecución como en el número de recursos internos consumidos. Su diseño surge ante la necesidad de mejorar el

ratio entre la cadencia de entrada de datos $\varepsilon'_{i,w}$ y la de salida de datos ($\varepsilon'_{v_{i,w}}$), consiguiendo en el diseño final tras vencer una latencia inicial (L_i) un ratio igual a uno, es decir, que la velocidad de salida de datos sea igual a la de entrada. Otras alternativas como las desarrolladas en [Bravo, 2003] [Bravo, 2003b] obligan a que la cadencia de entrada de datos sea como mínimo 5 veces la de salida.

Para realizar la convolución entre una matriz y una máscara genérica se deben realizar 9 operaciones de multiplicación y 8 de acumulación (ver Figura 5.10). Sin embargo, cuando todos los coeficientes de la máscara son idénticos, tal y como ocurre en nuestro caso, otra forma de realizar la convolución puede ser según (5.6) lo que permite reducir el número de multiplicaciones a 1. De esta forma, para obtener cada elemento $\varepsilon'_{v_{i,w}}$ del $\mathbf{MD}_{V2}$ es necesario realizar una suma-acumulación de 9 elementos y una multiplicación por el factor equivalente en coma fija.

$$\varepsilon'_{v_{i,w}} = 1/9 \cdot \left(\varepsilon'_{i-1,w-1}+\varepsilon'_{i-1,w}+\varepsilon'_{i-1,w+1}+\varepsilon'_{i,w-1}+\varepsilon'_{i,w}+\varepsilon'_{i,w+1}+\varepsilon'_{i+1,w-1}+\varepsilon'_{i+1,w}+\varepsilon'_{i+1,w+1}\right) \quad (5.6)$$

Debido a las limitaciones en cuanto al número de multiplicadores en la FPGA empleada en esta tesis, en nuestro caso la multiplicación por 1/9 se sustituye por una división entre 8 lo que equivale a un desplazamiento. Con ello se introduce un factor de escala al promediado por lo que no se preservan los valores máximos del mapa de distancias. Este incremento no supone un problema de saturación de los valores del mapa de distancias promediado dado que se trabaja con un número de bits mayor y por tanto realizar la división entre 8 en vez de entre 9 no modifica los resultados del algoritmo PCA obtenidos finalmente.

Con respecto a la etapa de suma-acumulación, el primer dato de salida no se puede obtener hasta que no se hayan recibido los primeros $2N+3$ elementos de $\mathbf{MD}_2$. Por tanto se deben sumar-acumular los diferentes $\varepsilon'_{i,w}$ que van llegando al sistema convolucionador. El hecho de tener que acumular previamente los datos a manejar, obliga a buscar un método de almacenamiento temporal de los elementos de forma que no sea necesario el acceso constante a memoria para no retardar la operación. Para ello los $\varepsilon'_{i,w}$ se van almacenando temporalmente en memorias FIFO (*First Input First Output*) implementadas dentro de la FPGA a modo de memoria CACHE.

Una vez recibidos los primeros $2N+3$ datos, se obtiene el primer dato de salida. Mientras se realiza la operación suma-acumulación se siguen recibiendo nuevos valores $\varepsilon'_{i,w}$ a la entrada, por lo que el sistema debe funcionar de manera segmentada para no ralentizar el tiempo de ejecución. Para conseguir esto, se ha implementado una arquitectura basada en *pipeline*, donde se van registrando los datos entre cada etapa para evitar problemas de retardos dentro de la FPGA que limiten la frecuencia máxima de trabajo.

El diagrama de bloques del sistema hardware implementado se puede ver en la Figura 5.11. En ella se pueden diferenciar claramente dos tipos de circuitos:

- Memorias FIFO. Son las encargadas de acumular los $\varepsilon'_{i,w}$ que se van recibiendo, así como de almacenar las sumas-acumulaciones parciales.
- FILTRO_LINE. Estos bloques se encargan de realizar las diferentes sumas a realizar en la convolución.

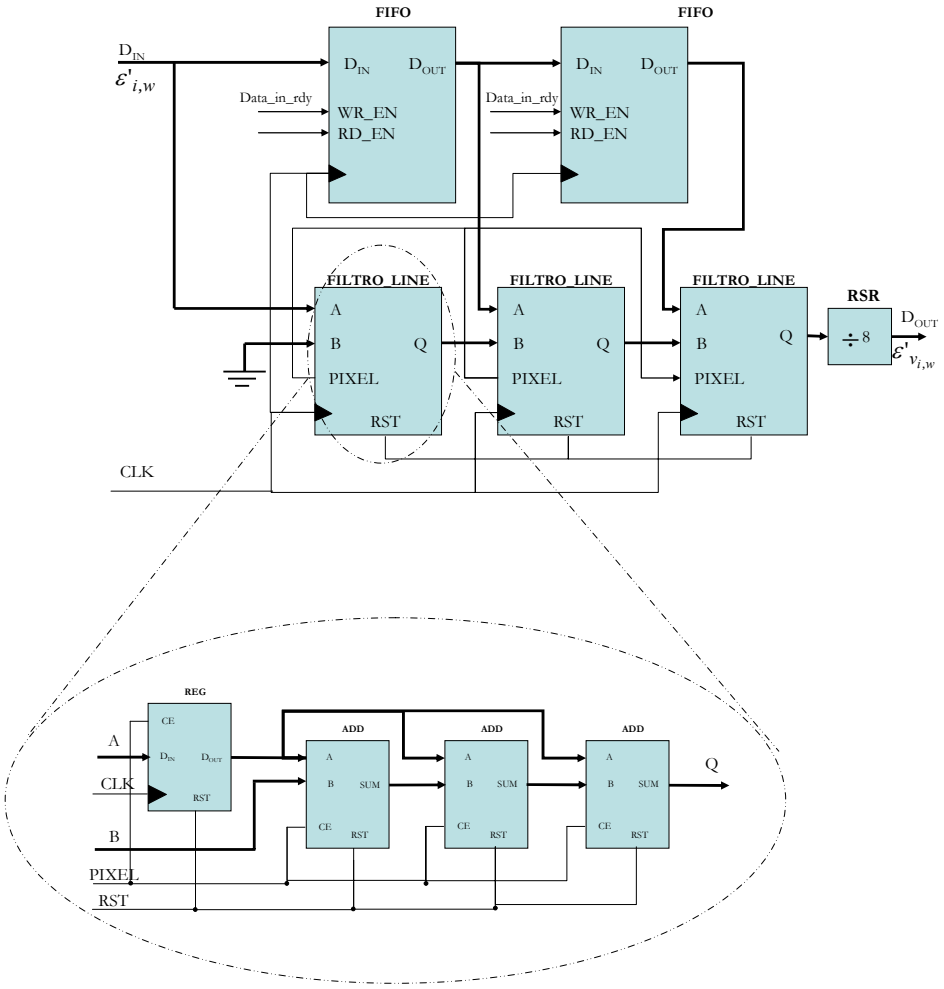


Figura 5.11. Diagrama de bloques del sistema hardware implementado para realizar la convolución con una máscara de 3x3 elementos.

El tiempo total de ejecución de esta convolución (T_{CONV}) se muestra en (5.7) donde el L_i es el valor de la latencia inicial en generarse los primeros datos, siendo su valor el expuesto en (5.8). Este valor viene justificado por el tiempo que tardan en recibirse los elementos $\varepsilon'_{i,w}$ necesarios para la generación del primer dato de salida, más un ciclo de reloj consumido internamente para registrar las entradas.

$$T_{CONV} = L_i + (N^2 - 1) \cdot T_{CLK} \tag{5.7}$$

$$L_i = (2N + 4) \cdot T_{CLK} \tag{5.8}$$

Por otra parte, el número de recursos consumidos por este bloque es muy bajo, tal y como se muestra en la Tabla 5.1. Cabe destacar el uso de 4 BRAMs para implementar en ellas los bloques FIFO de la Figura 5.11.

Tabla 5.1. Recursos consumidos en una FPGA XC2VP7 para un bloque convolucionador utilizando una máscara de 3x3.

Área (Slices)	BRAM	Multiplicadores	f_{CLKMAX}
124 de 4928 (2%)	4 de 44 (9%)	0	199,5 MHz

5.3. UMBRAL DINÁMICO.

Una vez obtenido el mapa de distancias promediado ($\mathbf{MD}_{V2}$) el siguiente paso que se realiza es el análisis de dicho mapa para evaluar la existencia o no de nuevos objetos en la escena de interés. Para ello se debe obtener un umbral Th_h que aplicado al $\mathbf{MD}_{V2}$ permita realizar la segmentación y en consecuencia detectar la presencia de nuevos objetos. El valor de Th_h debe ser dinámico ya que su valor se debe adaptar, entre otros, a cambios de iluminación de la escena. Para la obtención de este Th_h dinámico se han propuesto diferentes alternativas, [Vázquez, 2006] [Nakagawa, 1979], [Otsu, 1979], [Kittler, 1986], [Manay, 2003], [Shao, 2004]. La mayoría de ellas se basan en la distinción de dos clases (una clase que representa al fondo y otra clase que representa a los objetos que han entrado en la escena) por lo que siempre se elegirá un umbral que segmente los píxeles de la imagen resultante. En nuestro caso, cuando no existan objetos nuevos en la escena estática, todos los píxeles pertenecerán a la clase correspondiente al modelo del fondo.

El trabajo desarrollado en [Vázquez, 2006] muestra un método basado en la determinación empírica de una ecuación que define dicho umbral en función de la información del histograma del mapa de distancias (Figura 5.12) y de un parámetro modificador β obtenido experimentalmente (Figura 5.13). Así, a la hora de determinar Th_h, éste será la suma de β y de ε_{mf} (5.9), donde β es un parámetro en función de la varianza de $\mathbf{\Psi}$ y donde ε_{mf} es la distancia de mayor repetición del histograma del mapa de distancias. Tal y como se puede observar en el histograma del mapa de distancias de la Figura 5.12, las distancias que poseen una mayor repetición se concentran en la parte izquierda del eje de abcisas (valores de error en el mapa de distancia pequeño, esto es, coinciden en gran medida con el valor de fondo).

$$Th_h = \beta + \varepsilon_{mf} \tag{5.9}$$

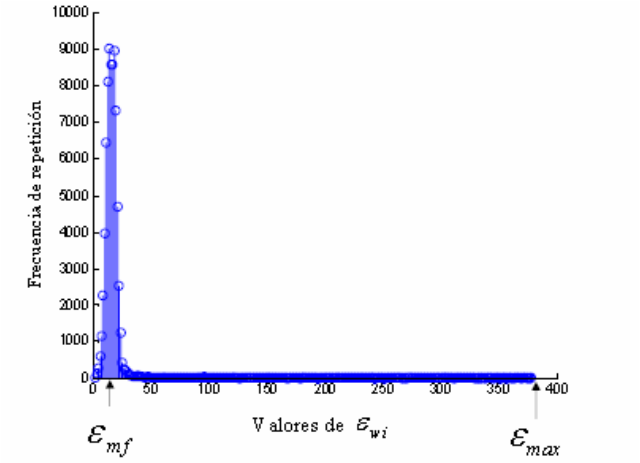


Figura 5.12. Histograma del mapa de distancias propuesto en [Vázquez, 2006].

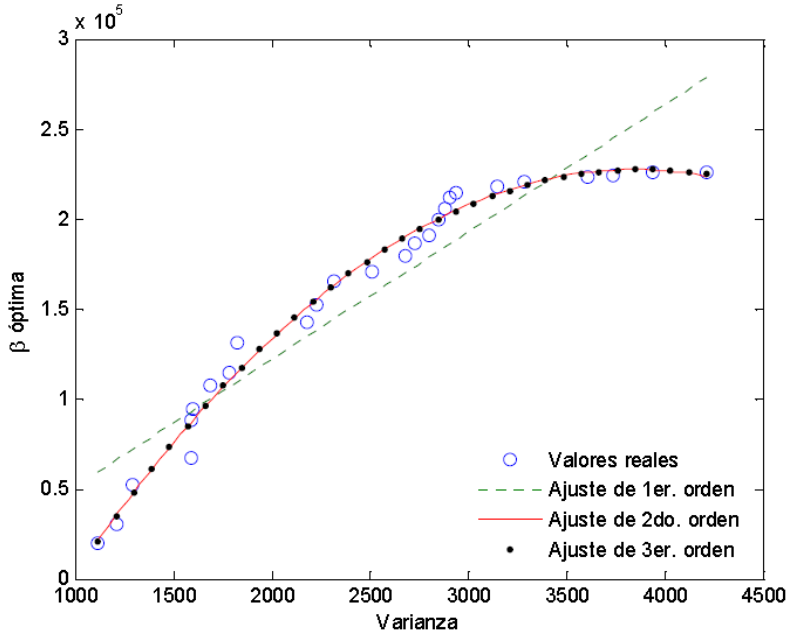


Figura 5.13. Obtención del parámetro β a partir de la varianza de Ψ [Vázquez, 2006].

Los resultados ofrecidos por esta propuesta se pueden calificar como muy buenos para el algoritmo PCA cuando se emplea un PC como elemento de procesado y una cámara con ajuste de ganancia automática tal y como se muestra en la Figura 5.14. Sin embargo, al trasladar esta propuesta sobre nuestra implementación en una FPGA no arroja tan buenos resultados como sobre un PC. La causa viene determinada por la codificación y el tamaño de los datos manejados en un PC y en una FPGA. Sobre la FPGA se trabaja con datos de coma fija y baja precisión en comparación con los 64 bits en coma flotante con los que generalmente trabaja la unidad aritmética de un PC convencional. Esto hace que el valor ε_{mf} sea inferior a la resolución con la que trabaja la FPGA, haciendo que todos los valores del mapa de distancias próximos o inferiores a ε_{mf}, se aglutinen en el primer intervalo del

histograma con valor igual a cero. Es decir, no hay definición suficiente para discernir el pico que define el máximo de ε_{mf}, siendo éste en la mayor parte de los casos cero tal y como se muestra en la Figura 5.15.

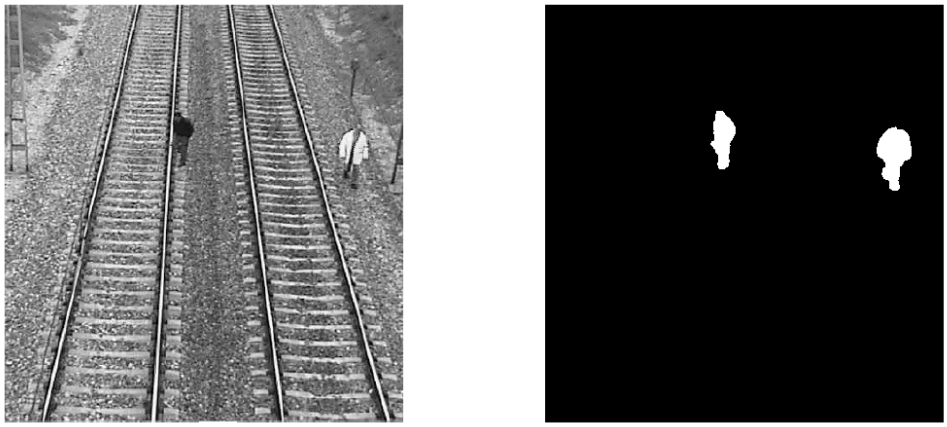


Figura 5.14. Resultados obtenidos al aplicar la propuesta de [Vázquez, 2006].

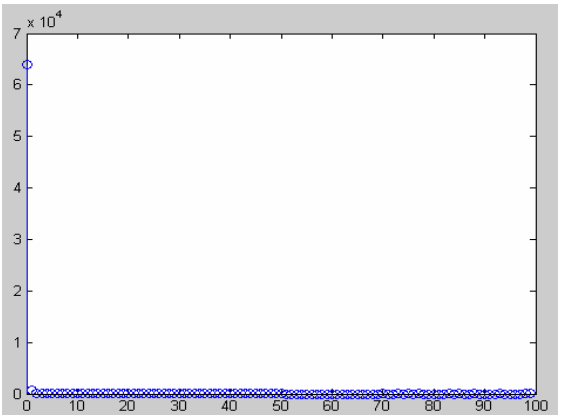


Figura 5.15. Histograma del mapa de distancias obtenido para una FPGA con la plataforma hardware empleada en esta tesis doctoral.

Puesto que ε_{mf} es cero en la mayor parte de las veces y unido a la naturaleza de las imágenes capturadas, no es posible calcular una relación directa entre la varianza de $\boldsymbol{\Psi}$ y el valor de β óptimo, es decir, no se puede trazar una curva que relacione β con la varianza, según se muestra en la Figura 5.16, para encontrar el valor del umbral óptimo acorde a (5.9). En esta figura, los datos empleados proceden de imágenes captadas y procesadas sobre una FPGA.

De esta forma, se hace necesario disponer de otro método para calcular el umbral dinámico, teniendo presente que dicho algoritmo se va a implementar sobre hardware programable, y que por consiguiente, no puede ser muy repetitivo o costoso en cuanto a número de operaciones. Por tanto, debe ser sencillo de ejecutar y planificar para que no requiera un secuenciador demasiado grande y complejo.

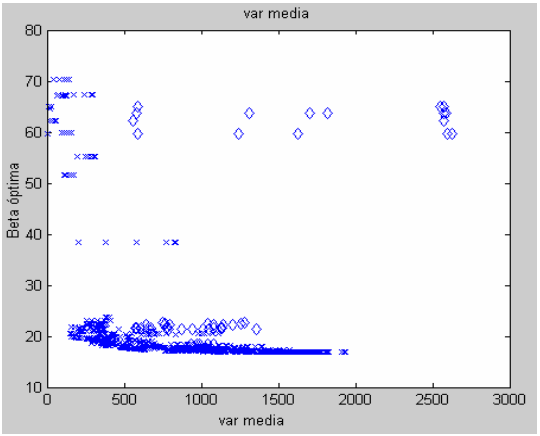


Figura 5.16. Diferentes valores de β para diferentes bancos de imágenes, en función de la varianza de $\boldsymbol{\Psi}$.

Centrando la atención en la Figura 5.15, se aprecia que el histograma del mapa de distancias aporta muy poca información, dado que siempre aparece un pico en las proximidades del cero y valores casi nulos en el resto de intervalos. Debido a la falta de precisión, ocasionada fundamentalmente por el tamaño de datos usados en la FPGA (18 bits), el histograma de mapa de distancias se ve muy afectado por el ruido producido por el error de cuantificación. Para mejorar este aspecto, se propone el cálculo del histograma de las distancias euclídeas máximas de cada columna de $\mathbf{MD}_{V2}$ (mapa de distancias promediado) (Figura 5.17) y la posterior obtención a partir de este histograma de un umbral dinámico nuevo. Esta propuesta implementada sobre una FPGA, genera resultados excelentes, tal y como se verá posteriormente.

La elección del histograma con los máximos de cada columna en vez de cada fila viene dada por la mejor definición que tiene un nuevo objeto, sobre todo si éste es más alto que ancho, sobre el histograma de los máximos de columnas. Esto se ha podido validar con las cerca de 1000 simulaciones realizadas con la base de imágenes captadas con la plataforma desarrollada en esta tesis. Sirva como ejemplo los histogramas mostrados en la Figura 5.18. Se puede observar en la Figura 5.18.a el histograma por columnas y en la Figura 5.18.d por filas., distinguiéndose la persona a detectar más claramente en el histograma por columnas (a) que en el de por filas (d).

Una vez construido el histograma de los máximos de cada columna del $\mathbf{MD}_{V2}$ se aprecia claramente (resaltado con un círculo en la imagen Figura 5.18 (b)) que se trata de un histograma bimodal donde el lóbulo más próximo al origen representa al fondo y el siguiente al nuevo objeto. Por tanto, se puede establecer un umbral Th_h para separar puntos del $\mathbf{MD}_{V2}$ pertenecientes al fondo y a los objetos nuevos.

Analizando la información proporcionada por el histograma de los máximos de las columnas del $\mathbf{MD}_{V2}$, se observa como al igual que ocurría en la propuesta de [Vázquez, 2006] la mayor parte de las distancias euclídeas máximas representadas, se

concentran en los intervalos inferiores (parte izquierda del histograma). Sin embargo, cuando aparece un nuevo objeto en la escena bajo estudio, las distancias euclídeas máximas de las columnas del $\mathbf{MD}_{V2}$ en las que está dicho objeto, se traducen en un valle en dicho histograma (ver Figura 5.23). En caso de no existir un nuevo objeto no aparece dicho valle tal y como se puede comprobar en el ejemplo de la Figura 5.21.

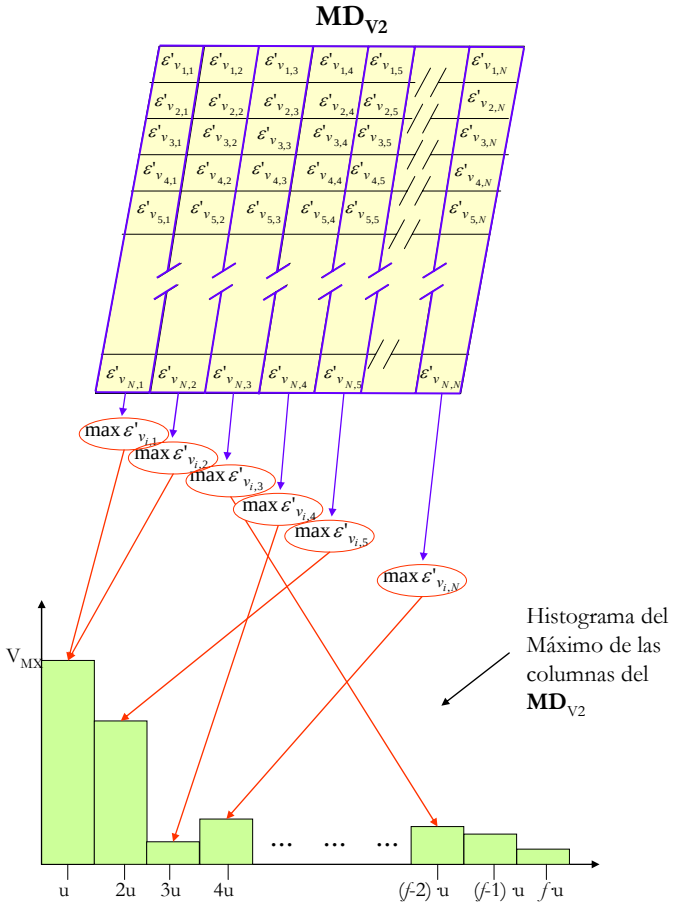


Figura 5.17. Ejemplo de construcción del histograma de los máximos de las columnas para un mapa de distancias promediado ($\mathbf{MD}_{V2}$).

La construcción del nuevo histograma propuesto obliga a definir el número de intervalos que forman dicho histograma (factor f en la Figura 5.17) así como el valor de cada uno (factor u en la Figura 5.17). La obtención de f y de u se ha realizado de forma empírica. Concretamente se utilizó un conjunto de 1000 imágenes como banco de test observándose primeramente que $f \cdot u$ nunca superaba el valor 12000, es decir, el valor más grande de los máximos de las columnas del mapa de las distancias promediadas nunca supera 12000. Por tanto, este valor fue fijado como $f \cdot u$. La siguiente cuestión a resolver es la búsqueda del valor u óptimo. También de forma experimental, se comprobó que para encontrar el valle de umbralización que separa de forma adecuada los píxeles de fondo de los píxeles pertenecientes a nuevos objetos, es necesario establecer un valor mínimo de $u = 1200$.

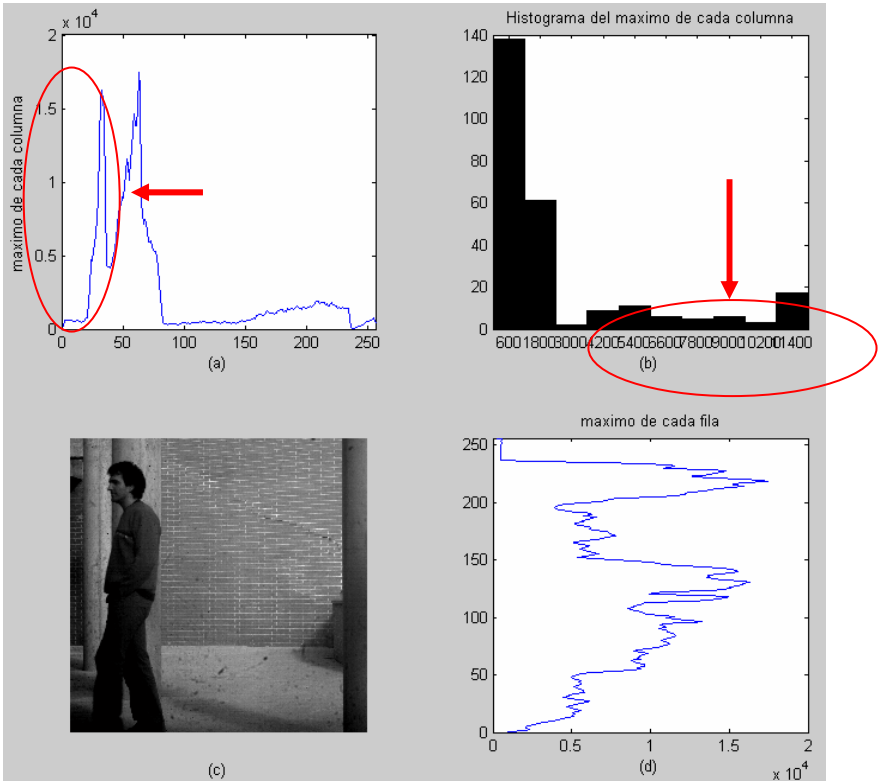


Figura 5.18. Representación gráfica de: (a) Máximos de cada columna del MD_{V2} (b) Histograma de los máximos de cada columna del MD_{V2}. (c) Imagen con objeto a detectar. (d) Máximos de cada fila de la MD_{V2}.

Para umbralizar el $\mathbf{MD}_{V2}$ se necesita encontrar el valor de Th_h que permita discriminar entre un nuevo objeto y el fondo. El valor mínimo de Th_h necesario para la correcta detección de los nuevos objetos, debe ser igual al valor del intervalo del histograma que contiene el valle asociado al nuevo objeto tal y como se puede apreciar en la Figura 5.20 y Figura 5.23 con una línea roja. Por tanto para la detección de nuevos objetos bastará con la determinación de la existencia de un valle en el histograma de los máximos de las columnas del $\mathbf{MD}_{V2}$. De esta forma el proceso de detección de nuevos objetos se muestra en la Figura 5.19.

Para validar la propuesta de detección de nuevos objetos mediante la plataforma hardware desarrollada en esta tesis, se han realizado diversas simulaciones de imágenes de diferentes escenas y con distintas condiciones de iluminación. Para mostrar la efectividad de la propuesta desde la Figura 5.20 a la Figura 5.23 se muestran las diferentes situaciones analizadas. En cada una de estas figuras, se muestra la imagen captada (parte superior derecha), el máximo de cada columna del MD_{V2} (parte superior izquierda), el histograma de los máximos del $\mathbf{MD}_{V2}$ (parte inferior derecha) y el resultado de la umbralización realizada sobre $\mathbf{MD}_{V2}$ a partir de un umbral Th_h cuyo valor coincide con el del intervalo del histograma que contiene al valle, delimitando el fondo de los nuevos objetos. En la Figura 5.20 y Figura 5.23 se aprecia la entrada en la escena de un nuevo objeto no contemplado en el modelo de fondo, y como se puede observar el algoritmo propuesto lo identifica

correctamente. Para el caso de la Figura 5.21 y Figura 5.22 no existen nuevos objetos, y como se puede observar se marca el umbral como 0 para indicar que no se detectó ningún objeto nuevo en la escena. Sobre la gráfica del histograma hay impresos varios datos referentes al número de imagen procesada, imágenes con las que se ha hecho el modelo de fondo, y datos sobre la media y la varianza tanto del $\mathbf{MD}_{V2}$ como de $\mathbf{\Psi}$. Por último, el nivel del umbral calculado aparece enmarcado y señalado en la gráfica mediante una línea roja (tanto en la determinación del valle como en la umbralización de los máximos de las columnas).

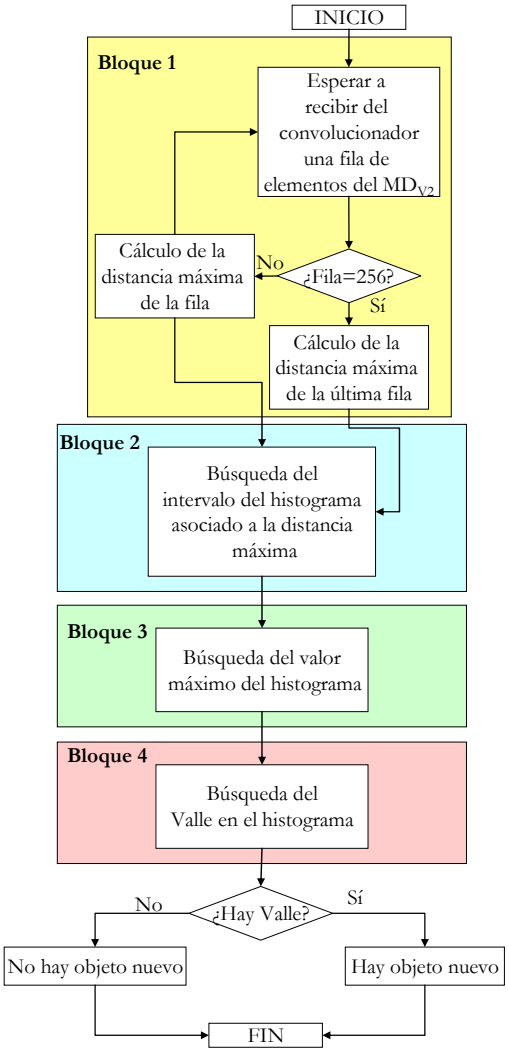


Figura 5.19. Diagrama de flujo para la obtención del umbral óptimo Th_h

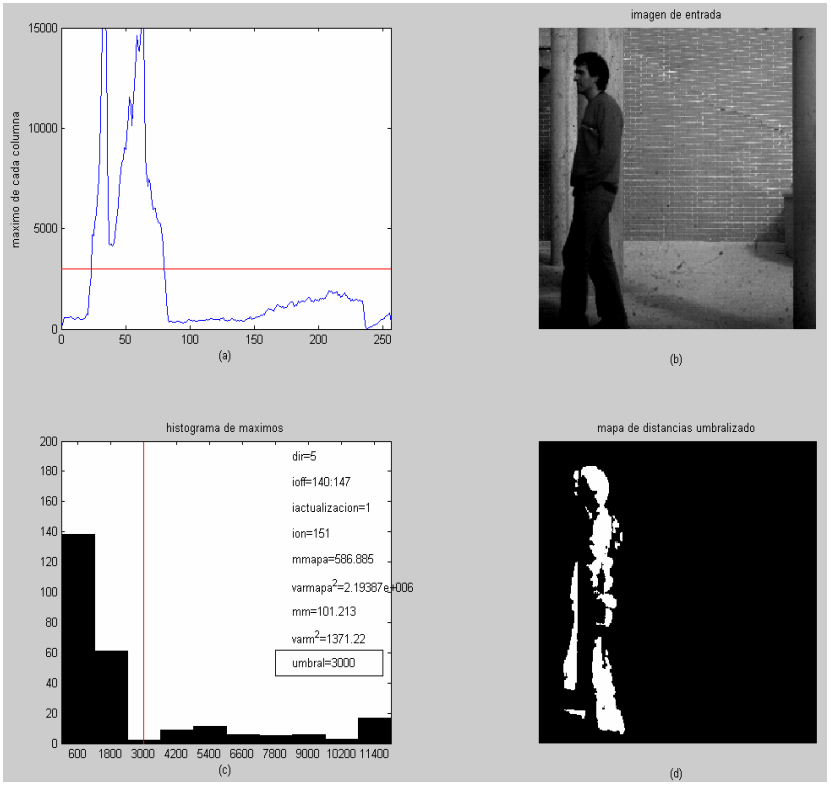


Figura 5.20. Representación gráfica de: (a) Máximos de cada columna del $\mathbf{MD}_{V2}$. (b) Imagen sobre la que detectar nuevos objetos. (c) Histograma de máximos de las columnas del $\mathbf{MD}_{V2}$. (d)Umbralización del $\mathbf{MD}_{V2}$ donde aparece un objeto introducido en la imagen.

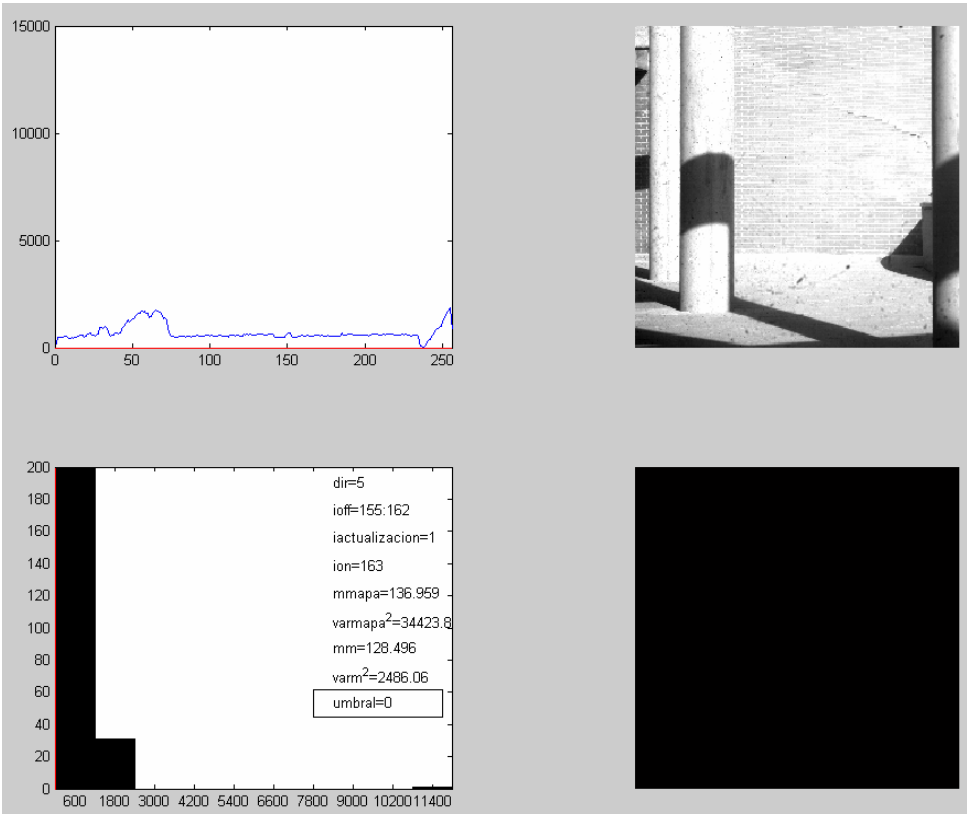


Figura 5.21. Imagen sin nuevo objeto, junto a su histograma de umbralización del $\mathbf{MD}_{V2}$.

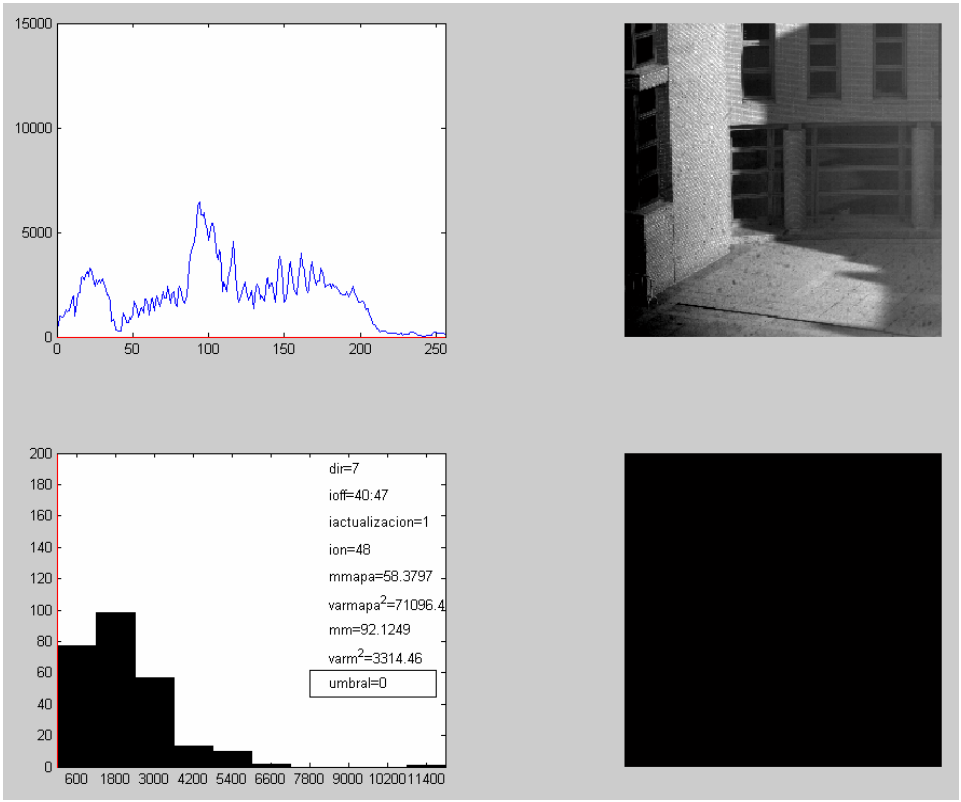


Figura 5.22. Imagen sin nuevo objeto, junto a su histograma de umbralización del $\mathbf{MD}_{V2}$.

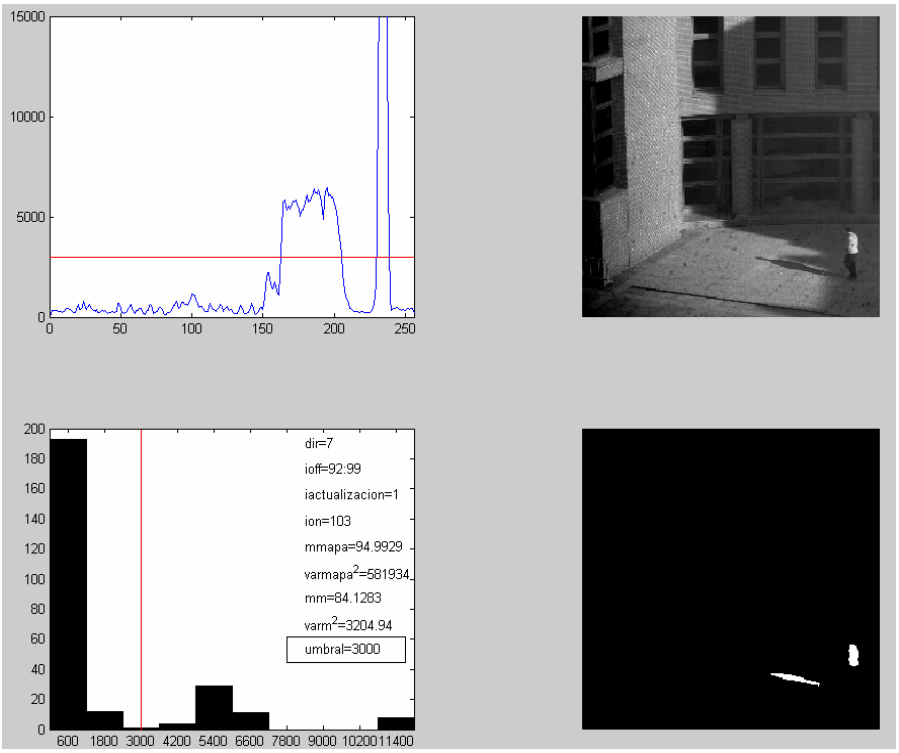


Figura 5.23. Imagen con nuevo objeto, junto a su histograma de umbralización del $\mathbf{MD}_{V2}$.

Para validar la nueva propuesta de umbral se realizaron simulaciones para un conjunto 1000 imágenes captadas, detectándose sólo dos casos en los que el sistema proporcionó falsos positivos, esto es, sin haber en la escena nuevos objetos el algoritmo indicó que sí los había. En uno de los casos el error fue debido a un cambio muy fuerte de iluminación en la escena que provocó una saturación en la cámara, y en el segundo debido a un balanceo del soporte de la cámara. Por tanto se puede concluir que la propuesta desarrollada en este apartado es válida para ser implementada en una FPGA.

5.3.1. Implementación Hardware del algoritmo de cálculo del umbral dinámico.

Una vez validada la propuesta de cálculo del umbral dinámico anteriormente expuesta, a continuación se aborda el diseño hardware para realizar dicha tarea. El comportamiento de este diseño se muestra en la Figura 5.19 indicándose sobre ésta la funcionalidad de cada uno de los bloques que forman el diseño. Partiendo del mapa de distancias promediado ($\mathbf{MD}_{V2}$), el diseño consta de 4 bloques, mostrándose en la Figura 5.24 su interconexión.

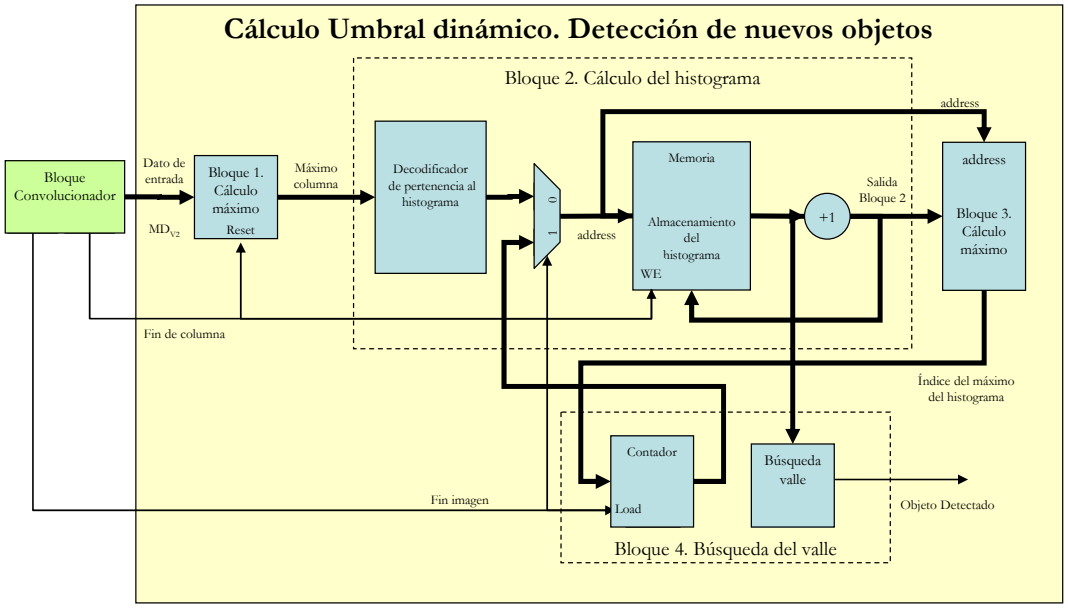


Figura 5.24. Diagrama de bloques del sistema de cálculo de umbral dinámico para la detección de nuevos objetos sobre una FPGA.

A continuación se describe cada uno de los bloques de la Figura 5.24:

- *Bloque 1*: Este bloque es el encargado de calcular el máximo de cada columna del mapa de distancias $\mathbf{MD}_{V2}$. Internamente está compuesto por un único registro que va almacenando el valor máximo y un comparador que evalúa si el nuevo dato es mayor o menor que el máximo temporal almacenado. Una señal generada por la lógica de control marca el final de la columna para resetear el registro y

empezar de nuevo con la siguiente columna (Figura 5.25). Como se puede desprender de lo anterior, el mapa de distancias se recorre columna a columna, para ir obteniendo de cada una de ellas su máximo, pudiendo hacer uso de un único registro. El tiempo consumido por este bloque ($T_{BLOQUE1}$) es de $N^2 T_{CLK}$ siendo N^2 el tamaño de $\mathbf{MD}_{V2}$. Su actividad se inicia en el momento en que se recibe el primer elemento de $\mathbf{MD}_{V2}$ desde el convolucionador descrito en el apartado 5.2.2.

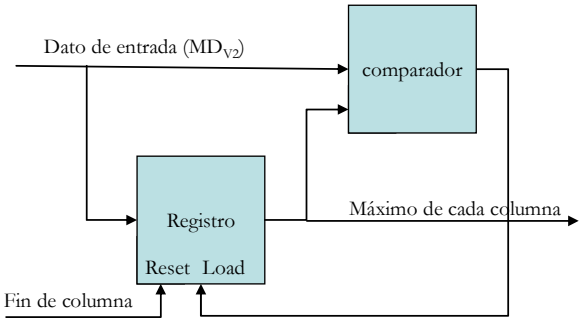


Figura 5.25. Estructura interna del Bloque 1 (cálculo del máximo de cada columna) de la Figura 5.24.

- *Bloque 2*: Tras haber calculado los máximos de las columnas de $\mathbf{MD}_{V2}$, el *Bloque 2* se encarga de construir el histograma de los máximos de las columnas. Su ejecución se realiza en paralelo con el *Bloque 1* una vez que se ha obtenido el máximo de la primera columna. Cada vez que se obtiene un máximo se debe buscar el intervalo del histograma al que pertenece dicho máximo. Para llevar a cabo esta tarea el máximo obtenido en el *Bloque 1* se compara con f comparadores en paralelo (donde f es el número de intervalos que se ha fijado para la compresión del histograma de los máximos de las columnas, que tal y como se expuso anteriormente en nuestro caso se fijará a 10). El resultado de los comparadores se decodifica para obtener una dirección de memoria correspondiente a cada intervalo. Una vez calculada la dirección de memoria, se suma una unidad al contenido de memoria de dicha dirección. En la Figura 5.26 se muestra la estructura interna del *Bloque 2*. El tiempo consumido por este bloque ($T_{BLOQUE2}$) es de $3T_{CLK}$ (un ciclo para comparar, otro para decodificar y otro para escribir en memoria).

- *Bloque 3*: Este módulo, cuya ejecución se inicia cuando finaliza el *Bloque 2*, se encarga de calcular el valor máximo del histograma (V_{MX} de la Figura 5.17). Es necesario buscar el valor máximo con el fin de buscar la existencia de un valle y por lo tanto determinar si la imagen captada posee o no un nuevo objeto. El funcionamiento de este bloque es el siguiente: cada vez que se obtiene un máximo de una

columna en el *Bloque 2*, se acumula un nuevo valor al intervalo del histograma que corresponda y se actualiza el número de intervalo del histograma que tiene el máximo valor de acumulación. En este momento, en el Bloque 3 se evalúa si el intervalo incrementado es el mayor de todos Si es así, se almacena el valor para comparar con las sucesivas salidas del Bloque 2 y se almacena la dirección de memoria generada por el *Bloque 2*, la cual indica la posición de dicho máximo (*Índice del máximo del histograma* de la Figura 5.27). Su tiempo de ejecución ($T_{BLOQUE3}$) es de sólo un ciclo de reloj.

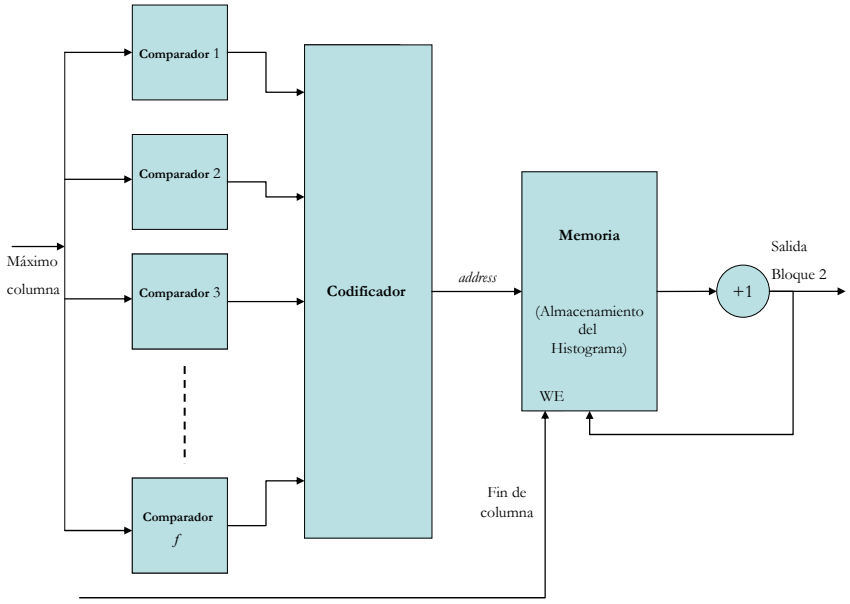


Figura 5.26. Estructura interna del Bloque 2 (cálculo del histograma de los máximos de las columnas del $\mathbf{MD}_{V2}$) de la Figura 5.24.

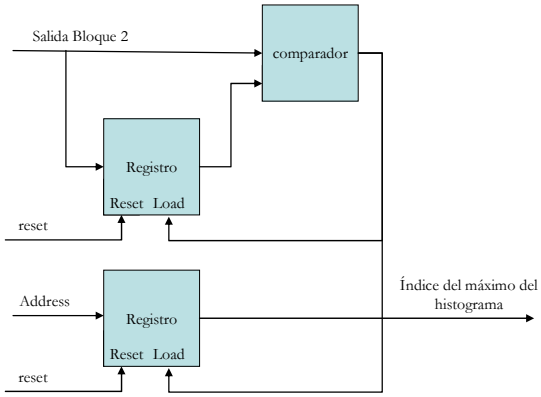


Figura 5.27. Estructura interna del Bloque 3 (cálculo de la dirección del valor máximo del histograma) de la Figura 5.24.

- *Bloque 4*: Por último, este elemento se encarga de buscar la existencia de un valle dentro del histograma una vez que han finalizado el *Bloque 2* y el *Bloque 3*. Tal y como se ha expuesto en el apartado anterior, la existencia de un valle conlleva la presencia de un nuevo objeto en la imagen bajo estudio. Debido a que en esta tesis, sólo se desea detectar

la existencia o no de nuevos objetos sin realizar ninguna clasificación de éstos, en la práctica, cuando se detecte la existencia de un valle se activará la señal objeto detectado (ver Figura 5.24 y Figura 5.28) finalizando así la fase de detección.

Para encontrar dicho valle se ha diseñado el sistema mostrado en la Figura 5.28. Internamente está compuesto por un contador que recorre la memoria del Bloque 2, la cual contiene el histograma de los máximos de las columnas de $\mathbf{MD}_{V2}$. El contador arranca a partir de la dirección almacenada en el *Bloque 3*, es decir, la dirección del máximo del histograma. Para hallar la existencia de un valle, simplemente se debe encontrar un valor en la memoria que sea mayor que el almacenado en la posición anterior. Si no existe un mínimo, el sistema irá aumentando el umbral hasta considerar que el umbral se sitúa en el intervalo extremo y clasificar así a todos los píxeles de la imagen como pertenecientes al fondo (Figura 5.28). Su tiempo de ejecución ($T_{BLOQUE4}$) es: $T_{BLOQUE4} = f \cdot 2T_{CLK}$.

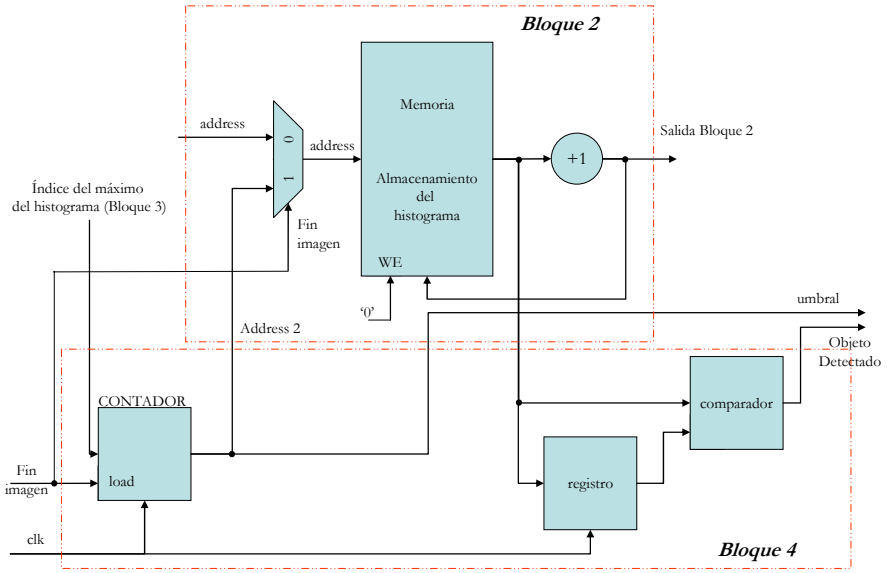


Figura 5.28. Estructura interna del Bloque 4 (búsqueda de un valle en el histograma de los máximos de columnas de $\mathbf{MD}_{V2}$) de la Figura 5.24.

Con respecto al tiempo total consumido por el sistema de la Figura 5.24 sobre una FPGA, tal y como se ha comentado anteriormente, los diferentes bloques que forman la Figura 5.24 se ejecutan en paralelo. De esta forma se ha conseguido una ejecución concurrente, mostrándose en la Figura 5.29 su secuencia de funcionamiento. En base a esta secuencia, el tiempo total de ejecución del sistema de cálculo de umbral dinámico aplicado para $N = 256$, que es el tamaño empleado en esta tesis, es el mostrado en (5.10). El número de intervalos del histograma (f) ha sido fijado de forma empírica a 10, ya que con este valor la propuesta desarrollada funciona correctamente.

$$T_{UMBRAL} = N \cdot T_{BLOQUE1} + L_f = \left(N^2 + 4 + 2f\right)T_{CLK} = 65560T_{CLK} \tag{5.10}$$

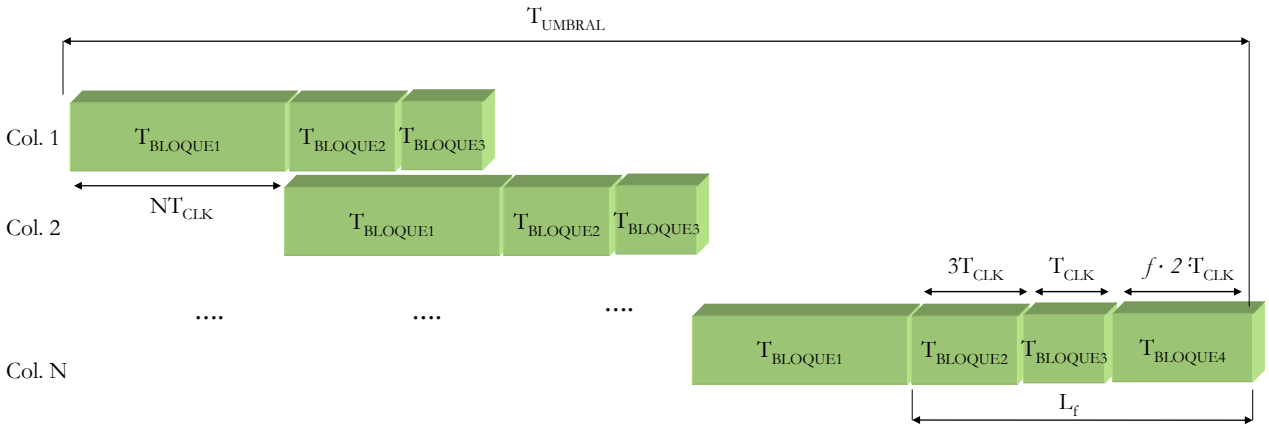


Figura 5.29 Secuencia de funcionamiento del sistema propuesto para el cálculo de umbral dinámico para una FPGA.

Por otra parte, el número de recursos consumidos por el sistema de cálculo de umbral, es relativamente pequeño tal y como se expone en la Tabla 5.2, ya que la lógica asociada es bastante elemental. Cabe destacar que debido al consumo elevado de BRAM por otros módulos de PCA, el hardware para obtener el umbral dinámico ha sido implementado en lógica distribuida, utilizando CLBs.

Tabla 5.2. Recursos consumidos en una FPGA XC2VP7 por el bloque de cálculo del umbral dinámico.

Área (Slices)	BRAM	Multiplicadores	f_{CLKMAX}
111 de 4928 (2.3 %)	0	0	144.12 MHz

5.4. ACTUALIZACIÓN DE LA MATRIZ DE AUTOVECTORES.

La actualización de la matriz de autovectores es un aspecto fundamental para el correcto funcionamiento del algoritmo PCA cuando la escena está sometida a cambios de iluminación. La pregunta que se debe responder es ¿cuándo se debe actualizar la información característica de una escena (matriz de autovectores)? En primer término se podría pensar en realizar la actualización continua del modelado de la escena, es decir a medida que una nueva imagen es evaluada, incorporar su información característica al modelo de la escena siempre y cuando no se detecten nuevos objetos. Esta opción equivale a ralentizar el algoritmo global ya que la actualización de la matriz de autovectores tiene asociado un elevado tiempo de cómputo. Además del incremento del tiempo de ejecución, puede suceder tal y como se ha comprobado experimentalmente, que el grado de similitud de la nueva imagen captada con respecto al modelo de fondo sea muy grande por lo que no compensa incluir esa imagen al fondo ya que apenas va a añadir nueva información.

Por tanto, se debe fijar alguna condición que justifique la actualización de forma selectiva. En esta tesis, según el estudio desarrollado en [Vázquez, 2006], se establece como condición de actualización un cambio de luminosidad significativo de la imagen sobre la que se detecta la aparición de nuevos objetos, con respecto a la luminosidad del modelo de fondo. En este caso, la pregunta que ahora surgiría sería: ¿qué criterio se debe cumplir para realizar la actualización? En [Vázquez, 2006] se exponen diferentes alternativas, justificándose que la denominada *Actualización Selectiva del Modelo de Fondo* ofrece unos óptimos resultados. Esta alternativa se basa en la obtención de un vector característico que aporte información sobre los cambios de iluminación de la escena. A continuación se describe el funcionamiento de este algoritmo.

Mientras se crea el modelo de fondo con la captura de M imágenes, de ese conjunto de imágenes se obtendrán dos nuevos vectores denominados como $\mathbf{I}_{\max} \in \Re^{N^2 \times 1}$ y como $\mathbf{I}_{\min} \in \Re^{N^2 \times 1}$, donde sus elementos contienen los valores máximo y mínimo respectivamente, de cada uno de los píxeles de las M imágenes, es decir:

$$\mathbf{I}_{\max} = \begin{bmatrix} \max(I_{i1}) \\ \max(I_{i2}) \\ .... \\ \max(I_{iN^2}) \end{bmatrix} \tag{5.11}$$

$$\mathbf{I}_{\min} = \begin{bmatrix} \min(I_{i1}) \\ \min(I_{i2}) \\ .... \\ \min(I_{iN^2}) \end{bmatrix} \tag{5.12}$$

donde $I_{i1}, I_{i2}, ... I_{iN^2}$ son los niveles de luminosidad de cada uno de los píxeles de las imágenes y el subíndice i representa a cada una de la M imágenes.

Para realizar una actualización más selectiva, se divide tanto $\mathbf{I}_{max}$ como $\mathbf{I}_{min}$ en c ventanas o regiones (υ_i) todas ellas de igual tamaño, tal y como se observa en Figura 5.30, calculando a continuación el valor medio de la luminosidad de cada una de estas ventanas ($\Lambda\upsilon_i$). Dichos valores medios obtenidos de las regiones de las imágenes de máximos y mínimos se almacenan en registros para su posterior uso en la fase *on-line*.

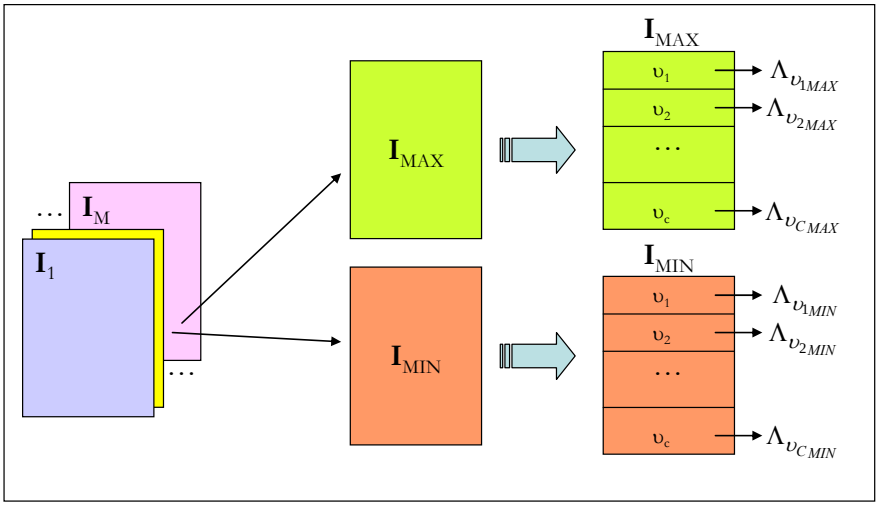


Figura 5.30. Obtención de las imágenes del modelo de fondo con mayor y menor luminosidad y posterior división en regiones.

Una vez determinados los vectores $\mathbf{I}_{max}$ y $\mathbf{I}_{min}$, para determinar en la fase *on-line* si una nueva imagen $\mathbf{I}_j \in \Re^{N^2 \times 1}$ debe ser añadida al modelo de fondo, ésta se divide en las mismas c ventanas en las que se dividía $\mathbf{I}_{max}$ y $\mathbf{I}_{min}$, calculándose en cada ventana su valor medio ($\Lambda\upsilon_i$). Posteriormente, cada valor medio es comparado con los valores correspondientes al valor medio máximo ($\Lambda\upsilon_{iMAX}$) y al mínimo ($\Lambda\upsilon_{iMIN}$). Si tras comparar todas las ventanas de la imagen actual con las del modelo de fondo, existen más de r ventanas cuyo valor medio $\Lambda\upsilon_i$ no cumple la relación $\Lambda\upsilon_{iMAX} > \Lambda\upsilon_i > \Lambda\upsilon_{iMIN}$ y además en $\mathbf{I}_j$ no hay ningún nuevo objeto, entonces se considerará que $\mathbf{I}_j$ debe añadirse al modelo de fondo. Todo este proceso se puede observar en la Figura 5.31.

El problema que ahora se plantea es la obtención del valor óptimo de c y r. Para resolver esta cuestión, se han simulado para un conjunto de 1400 imágenes diferentes valores de c y r, fijando los siguientes criterios para c y r:

- Se han evaluado los siguientes valores de c, tanto para la imagen actual como para $\mathbf{I}_{max}$ y $\mathbf{I}_{min}$: $c=2$ (tamaño de ventana de 128x128), $c=4$ (tamaño de ventana de 64x64), $c=8$ (tamaño de ventana de 32x32), $c=16$ (tamaño de ventana de 16x16) y $c=32$ (tamaño de ventana de 8x8).

- Para obtener el valor óptimo de r, en este caso se ha fijado un valor mínimo de porcentaje de ventanas que cumplen la relación $\Lambda\upsilon_{iMAX} > \Lambda\upsilon_i > \Lambda\upsilon_{iMIN}$. En caso de ser superior a este porcentaje mínimo, entonces la imagen actual es candidata a ser incorporada al modelo de fondo. Los diferentes porcentajes mínimos empleados han sido: 12.5%, 25%, 37.5%, 50%, 62.5%, 75%, 87.5% y 100% (en este caso nunca se actualiza el modelo del fondo).

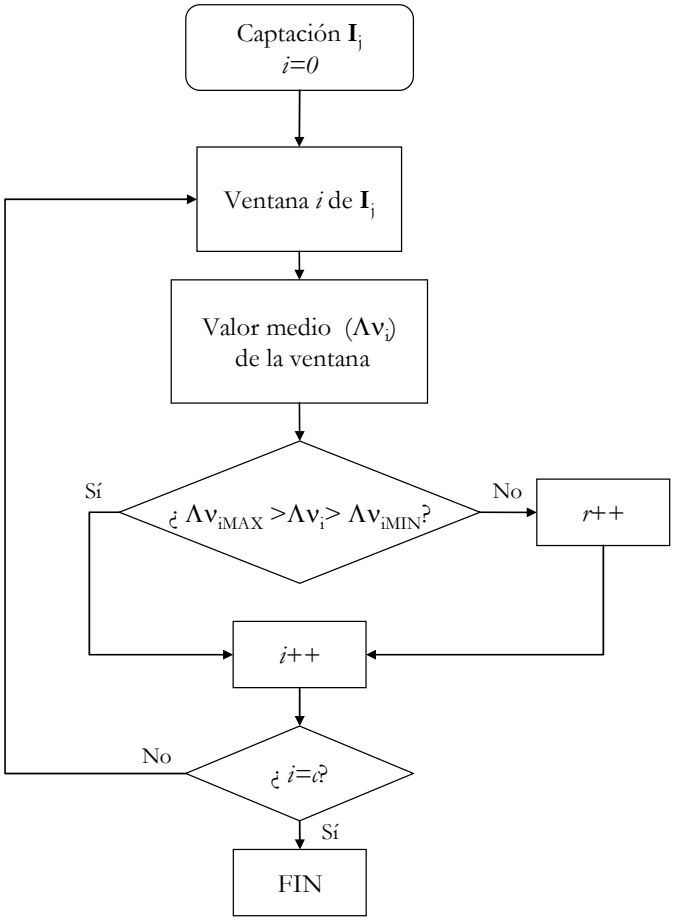


Figura 5.31. División y comparación de una nueva imagen para determinar si debe ser añadida al modelo de fondo.

Los índices empleados para obtener los valores idóneos de c y r han sido el número de falsos positivos (detección de nuevo objeto cuando no lo hay) y falsos negativos (omisión de detección de un nuevo objeto) obtenidos para los diferentes tamaños de c y r. Bajo estas condiciones se ha construido la Figura 5.32 y Figura 5.33. La primera de ellas muestra el número de falsos positivos para los diferentes tamaños de c y r mientras que la segunda presenta el número de falsos negativos.

Analizando la Figura 5.32 y la Figura 5.33 se extraen las siguientes conclusiones:

a) El porcentaje de falsos positivos siempre es superior al de falsos negativos. La generación de un falso positivo se produce por la detección errónea de un valle dentro del histograma de las columnas del $\mathbf{MD}_{V2}$. Esta falsa detección de valle es debida a la diferencia entre las condiciones de luminosidad de la imagen actual y las del fondo. Así, esta diferencia de luminosidad es confundida con un nuevo objeto. Además si hay un falso positivo no se actualiza el modelo de fondo, por lo que si las imágenes siguientes al falso positivo detectado poseen condiciones de luminosidad parecidas al falso positivo, de nuevo se volverá a generar un falso

positivo. Esta es la principal causa por la que el número de falsos positivos es superior.

b) En el caso de los falsos negativos, estos son generados principalmente cuando los objetos a detectar son pequeños. Si el tamaño de la ventana c es grande o el porcentaje de ventanas mínimo dentro de los valores medios es pequeño, entonces el valor de luminosidad de los nuevos objetos tiene una pequeña influencia en el histograma no generándose en éste un valle.

c) En el caso de no actualizar nunca el modelo de fondo el porcentaje de error de falsos positivos alcanza casi un 17% mientras que el de falsos negativos es de 9.5%. Ambos casos (errores máximos) se dan cuando el tamaño de la ventana es máximo (128x128).

d) En el caso de actualización continua sólo se ha detectado para el conjunto de imágenes empleadas un falso positivo y un falso negativo, siendo la causa en el primer caso una saturación excesiva de luminosidad y en el segundo un movimiento/balanceo de la cámara cuando se captó la secuencia de imágenes.

e) A la hora de obtener los valores idóneos de c y r, analizando la Figura 5.32 y la Figura 5.33, estos se obtienen para tamaños de ventanas pequeños y para porcentajes de r grandes.

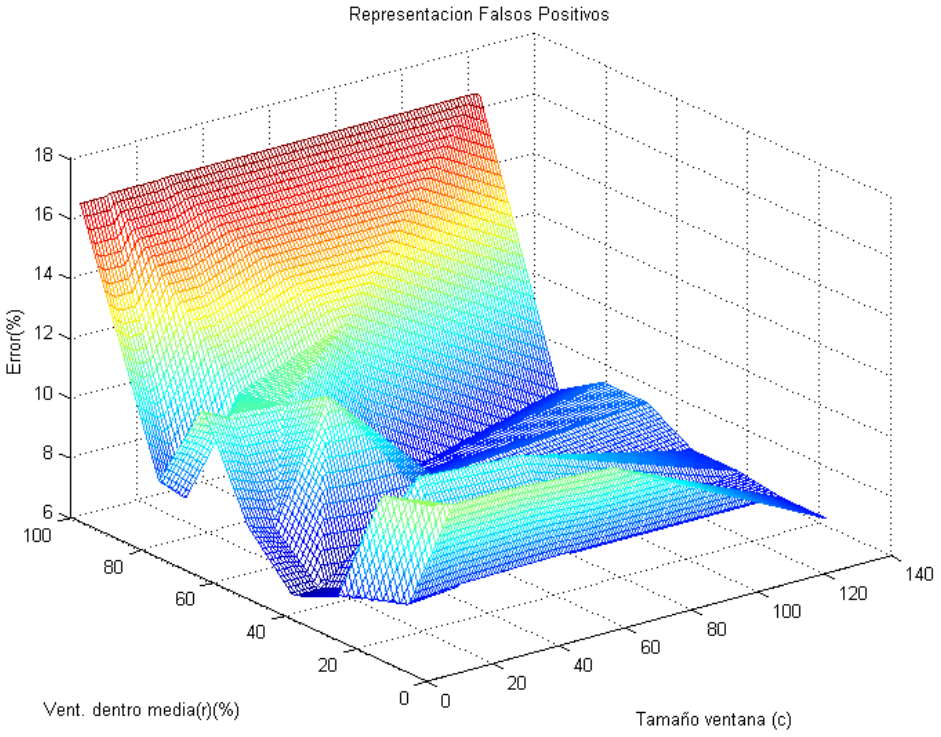


Figura 5.32. Representación del porcentaje de falsos positivos en función del tamaño de la ventana (c) y porcentaje de ventanas fuera del valor medio (r).

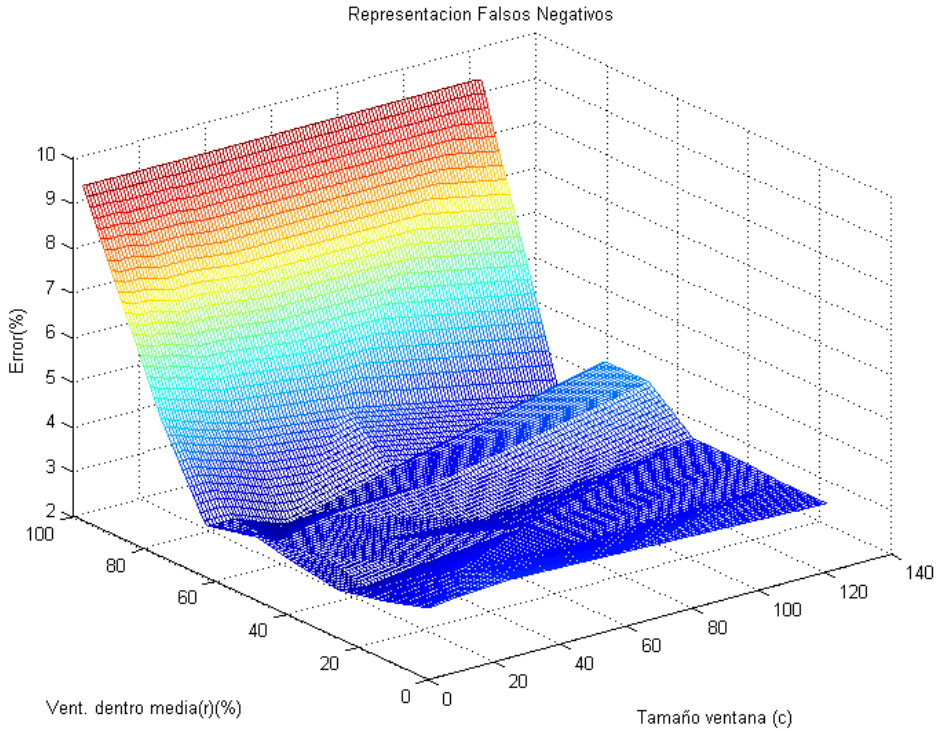


Figura 5.33. Representación del porcentaje de falsos negativos en función del tamaño de la ventana (c) y porcentaje de ventanas fuera del valor medio (r).

En nuestro caso se ha elegido un tamaño de ventana de 16x16 y un valor de r igual al 75%. Con estos dos factores el porcentaje de falsos negativos es del 3% y el de falsos positivos del 11% para un banco de test de 1000 imágenes.

Una vez justificado el valor óptimo de c y r a continuación se presenta la solución hardware propuesta en esta tesis para implementar el sistema de actualización selectiva del modelo de fondo. Esta solución está compuesta por dos bloques: uno para el cálculo de los valores medios de niveles de gris máximo y mínimo de las imágenes que forman el modelo de fondo (Figura 5.34) y otro bloque para la comparación de las c ventanas de la imagen actual con los valores medios del modelo de fondo (Figura 5.35).

Con respecto al bloque de cálculo de los valores medios del modelo de fondo (Figura 5.34), su estructura interna está formada por:

- Bloque de comparadores (BLOQUE_COMP): Conectados en una estructura en cascada (8 imágenes, 3 niveles de comparación) obtienen el valor máximo y mínimo de los niveles de gris de los píxeles que forman el modelo de fondo (I_{iMIN} e I_{iMAX}).

- Bloques acumuladores (ACC): Se encargan de calcular el valor medio máximo y mínimo de cada ventana.

La ejecución de este bloque dentro de PCA se realiza en el cálculo del modelo de fondo. Concretamente su ejecución se realiza en paralelo con el cálculo del vector

media ($\boldsymbol{\Psi}$). Debido a esta concurrencia, su tiempo de cómputo (5.13) no afecta al tiempo total de PCA.

$$T_{CALC\_V\_MEDIOS} = \left(3 \cdot T_{BLOQUE\_COMP} + T_{ACC}\right) \cdot N^2 = \left(3 \cdot 2 + 1\right) \cdot N^2 T_{CLK} = 458752 T_{CLK} \quad (5.13)$$

El número de recursos consumidos por el sistema mostrado en la Figura 5.34 se presenta en la Tabla 5.3.

Tabla 5.3. Recursos consumidos en una FPGA XC2VP7 por el bloque de cálculo de los valores medios de las ventanas del modelo de fondo.

Área (Slices)	BRAM	Multiplicadores	f_{CLKMAX}
168 de 4928 (3.4 %)	0	0	138.92 MHz

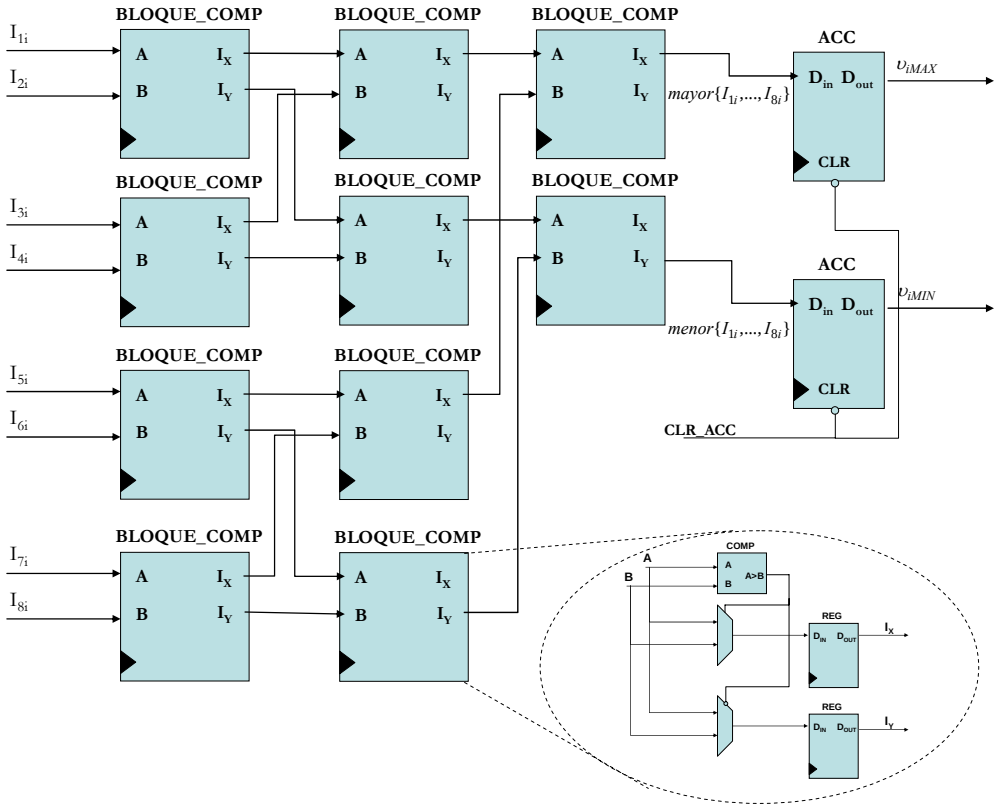


Figura 5.34. Estructura interna del bloque de obtención de los valores medios de las c ventanas del modelo de fondo.

Con respecto al bloque encargado de obtener el valor medio de nivel de gris de la imagen actual (Figura 5.35) internamente está formado por comparadores, acumuladores y un sumador. Este bloque se ejecuta en la fase *on-line* de PCA, concretamente a la vez que se realiza la resta de $\mathbf{I}_j$ con el valor de la media $\boldsymbol{\Psi}$

($\mathbf{\Phi}_j = \mathbf{I}_j - \mathbf{\Psi}$). De esta forma su tiempo de ejecución tampoco afecta al cómputo total del tiempo del algoritmo PCA.

$$T_{CALC\_VMEDIO\_I_j} = N^2 T_{ACC} + \left(T_{COMP} + T_{ADD} + T_{COMP}\right)c = \left(N^2 + 3c\right)T_{CLK} \tag{5.14}$$

Por su parte, el número de recursos internos de la FPGA consumidos para este último bloque es muy bajo, tal y como se expone en la Tabla 5.4, al tratarse de una estructura muy sencilla.

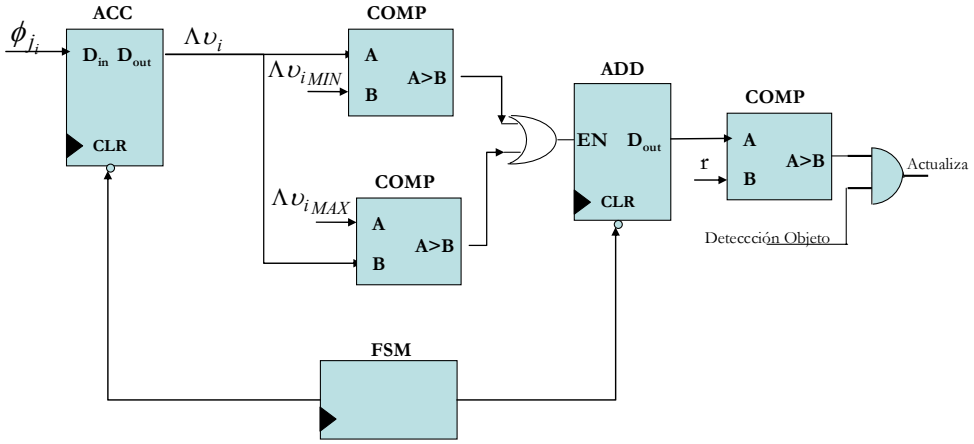


Figura 5.35. Estructura interna del bloque de comparación de los valores medios de las c ventanas de la imagen actual menos la media ($\mathbf{\Phi}_j$) con respecto a las valores de las c ventanas de $\mathbf{I}_{mx}$ e $\mathbf{I}_{min}$.

Tabla 5.4. Recursos consumidos en una FPGA XC2VP7 por el bloque de cálculo de los valores medios de las ventanas para decidir la actualización del modelo de fondo.

Área (Slices)	BRAM	Multiplicadores	f_{CLKMAX}
32 de 4928 (0.65 %)	0	0	148.72 MHz

5.5. CONCLUSIONES.

La propuesta realizada en esta tesis para la detección de nuevos objetos en una escena, mediante el análisis de la distancia euclídea del error de recuperación desarrollado en este capítulo, es una importante aportación de esta tesis doctoral debido a su alta tasa de efectividad.

Para lograr esta nueva propuesta, se ha diseñado una arquitectura segmentada (creación del mapa de distancias, convolucionador y umbral dinámico), la cual alcanza un alto rendimiento. Cada una de las etapas que forman esta arquitectura está optimizada desde el punto de vista de ejecución y de recursos internos consumidos. Este último aspecto ha sido muy importante a la hora de intentar implementar todo el sistema desarrollado en esta tesis sobre una FPGA. Para ello ha sido fundamental realizar una óptima codificación del diseño teniendo siempre muy presente el dispositivo hardware sobre el que se implementa.

La construcción del histograma de los máximos de las columnas del mapa de distancias promediado ($\mathbf{MD}_{V2}$) así como su posterior umbralizado es otra importante contribución de este capítulo. Esta nueva alternativa alcanza una alta efectividad en la detección de objetos debido fundamentalmente a su adaptación a las características del diseño realizado. El hecho de trabajar con una FPGA con datos en coma fija y baja resolución, si se compara con un PC, obliga en numerosas ocasiones a la búsqueda de nuevas propuestas específicas.

Otro aspecto resuelto en este capítulo ha sido la actualización selectiva del modelo de fondo para una FPGA. Este nuevo desarrollo ha permitido obtener un sistema que ocupa muy pocos recursos en una FPGA, donde su tiempo de ejecución es solapado con otras tareas del algoritmo PCA, por lo que su ejecución no introduce ninguna latencia adicional en todo el sistema desarrollado en esta tesis para la detección de nuevos objetos. Mediante un amplio estudio realizado se han obtenido los valores óptimos del número de ventanas de evaluación de la luminosidad (c) así como el porcentaje de ventanas cuyos valores medios difieren de los del modelo de fondo (r), para una correcta actualización del modelo de fondo, llegando a la conclusión que a mayor número de ventanas y a mayor porcentaje r mejor rendimiento se ofrece ya que más se parece al modelo de actualización continuo que sería el ideal.

6. RESULTADOS PRÁCTICOS

6.1. INTRODUCCIÓN.

En este capítulo se presentan los resultados obtenidos en la detección de objetos en movimiento utilizando visión computacional y técnicas PCA así como la arquitectura hardware de procesamiento propuesta y desarrollada al efecto en esta tesis doctoral.

La construcción de una plataforma específica para la captura y el procesamiento de imágenes sobre la que implementar las técnicas PCA presentada en los capítulos anteriores es otra importante aportación de esta tesis [Bravo, 2006]. Dicha plataforma consta básicamente de dos elementos: un sensor CMOS (encargado de captar y enviar imágenes a la FPGA) y una FPGA (encargada de controlar el sensor CMOS, implementar los algoritmos PCA desarrollados en los capítulos anteriores y gestionar la memoria externa de la plataforma). La elección de ambos es fundamental, ya que en función de sus características internas, se condicionan las prestaciones finales del sistema.

Para la implementación del sistema se ha optado por un sensor CMOS con un interfaz de control conocido y con una alta velocidad de captura de imágenes; concretamente, el sensor elegido es el MT9M413C36STM de Micron, el cual capta imágenes en blanco y negro con una velocidad máxima de 500 imágenes por segundo con una resolución espacial de 1280x1024 píxeles.

Con respecto a las características que debe poseer la FPGA del sistema, tal y como ha quedado de manifiesto en los capítulos anteriores, uno de los objetivos de esta tesis ha sido la implementación del sistema de detección de objetos en una escena, utilizando para ello herramientas de diseño hardware VHDL y sin el empleo de ningún *core*, permitiendo así su portabilidad a cualquier dispositivo FPGA. Debido a esto, los requisitos que debe poseer la FPGA únicamente son desde el punto de vista de recursos internos de propósito general (BRAM, *slices*, *buffers tri-state* y multiplicadores hardware), cumpliendo esta demanda múltiples familias de FPGAs. De las posibles alternativas, se ha optado por el empleo de una FPGA Virtex II-Pro de Xilinx, concretamente la XC2PV7.

Además del sensor CMOS y de la FPGA, la plataforma desarrollada también está dotada de un banco de memoria externa donde se almacenan las imágenes y la matriz de autovectores $\mathbf{U}_t$, así como de un circuito de interfaz para realizar la conexión vía puerto paralelo con un PC (ver Figura 6.1).

Este capítulo se ha estructurado en dos grandes apartados: en el primero de ellos se describen las características más importantes de la plataforma diseñada, mientras que el segundo presenta los resultados finales logrados con la implementación de la detección de objetos en movimiento mediante PCA sobre dicha plataforma.

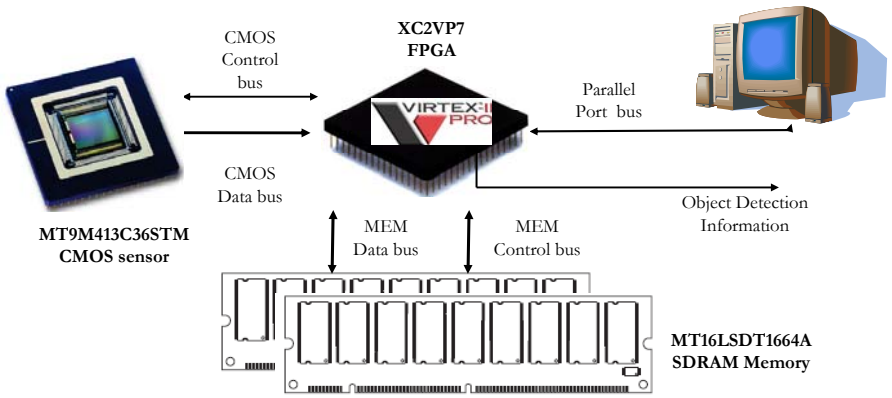


Figura 6.1. Diagrama de los elementos físicos que componen la plataforma desarrollada en esta tesis doctoral.

6.2. PLATAFORMA DESARROLLADA PARA LA APLICACIÓN DE PCA A LA DETECCIÓN DE OBJETOS EN MOVIMIENTO.

En este apartado se presentan las diferentes partes que componen la plataforma de captura, almacenamiento y procesado de imágenes desarrollada en esta tesis, mostrándose en la Figura 6.2 una fotografía de la misma.

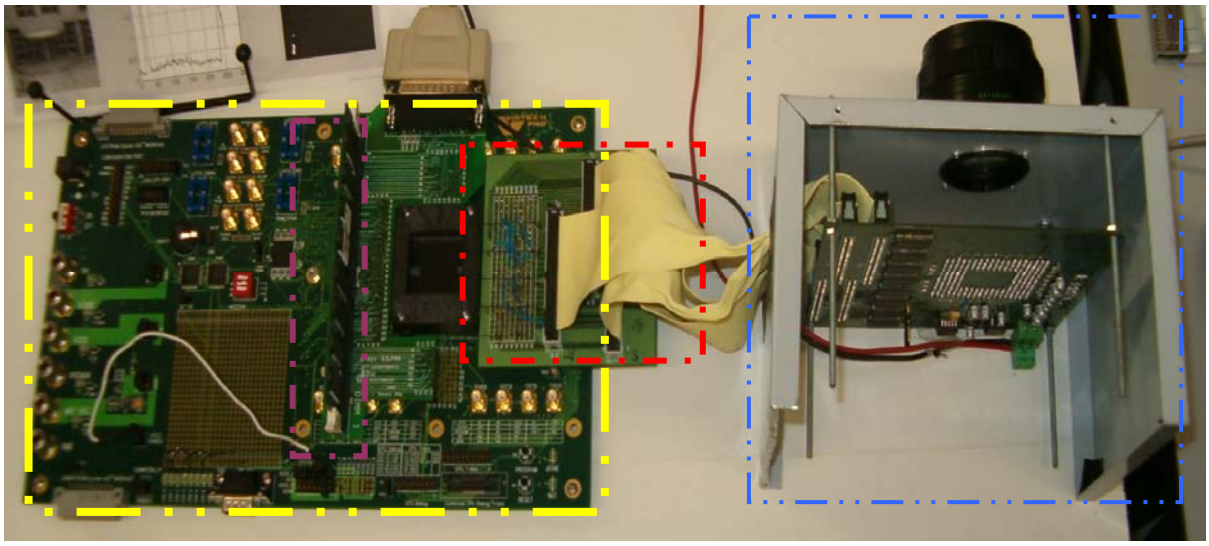


Figura 6.2. Vista de la plataforma basada en FPGAs desarrollada en esta tesis.

El sistema implementado ha sido construido en base a la tarjeta de desarrollo de Xilinx HW-AFX-FF672-300 (Figura 6.3). La elección de esta tarjeta frente a otras alternativas comerciales viene dada por la posibilidad de conectar a la FPGA diferentes periféricos, al disponer de una zona de prototipado contigua a la FPGA tal y como se puede ver en la Figura 6.3.

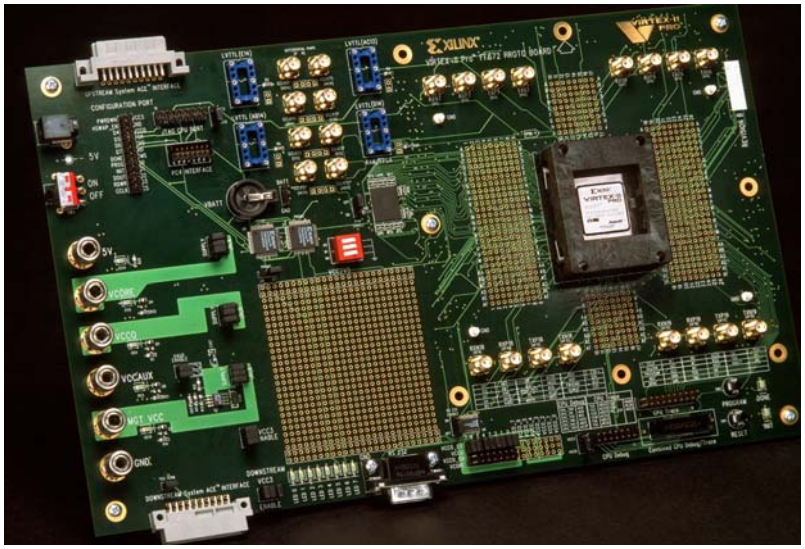


Figura 6.3. Vista general de la tarjeta de evaluación HW-AFX-FF672-300 empleada en esta tesis.

La tarjeta de desarrollo HW-AFX-FF672-300 está especialmente indicada para FPGAs de la familia Virtex II-Pro con encapsulado FF672 [Xilinx, 2007]. De esta forma es capaz de soportar los dispositivos de Xilinx: XC2VP2, XC2VP4, XC2VP7. Las características de los tres modelos, atendiendo a sus recursos internos, se pueden observar en la Tabla 6.1. Debido a las necesidades demandadas por los algoritmos desarrollados en los capítulos anteriores, de las posibles candidatas el modelo XC2VP7 es el que mayor número de recursos internos posee, por lo que se decidió utilizar esta FPGA para el diseño y pruebas a lo largo de toda esta tesis.

Tabla 6.1. Principales características de las FPGAs XC2VP2, XC2VP4, XC2VP7 con encapsulado FF672.

FPGA	XC2VP2	XC2VP4	XC2VP7
Logic Cells	3168	6768	11.088
Slices	1408	3008	4928
BRAM (Kbits)	216	504	792
Multiplicadores de 18x18 bits	12	28	44
DCMs	4	4	4
PowerPCs	0	1	1
Rocket I/Os	4	4	8
Número máximo de I/Os	204	348	396

Además la tarjeta HW-AFX-FF672-300 posee las siguientes características:

- Memoria externa SDRAM de 8 MBytes con bus de datos de 32 bits.
- Conectores para Rocket I/O, depuración PowerPC, JTAG y Puerto Serie.

Para dar cumplimiento a las necesidades de la presente tesis se han diseñado los siguientes sistemas que se han ido conectando a la tarjeta de evaluación de Xilinx (Figura 6.4):

- *Memorias externas SDRAM*. Se han construido dos tarjetas denominadas DIMM1 y DIMM2 que permiten insertar en cada una de ellas un módulo DIMM SDRAM de 64MB. Estos dos bloques se observan en color violeta y rosa en la Figura 6.4.
- *Interfaz de gestión del puerto paralelo*. Este módulo se encuentra insertado en la tarjeta DIMM1 tal y como se observa en color azul sobre la Figura 6.4, y permite la comunicación entre la FPGA y un PC para definir el formato de la imagen captada por el sensor.
- *Interfaz de control del sensor CMOS*. Se ha diseñado una tarjeta de adaptación (área roja de la Figura 6.2) para realizar la configuración y control del sensor CMOS. Esta tarjeta se inserta en la zona de prototipado enmarcada en rojo en la Figura 6.4.

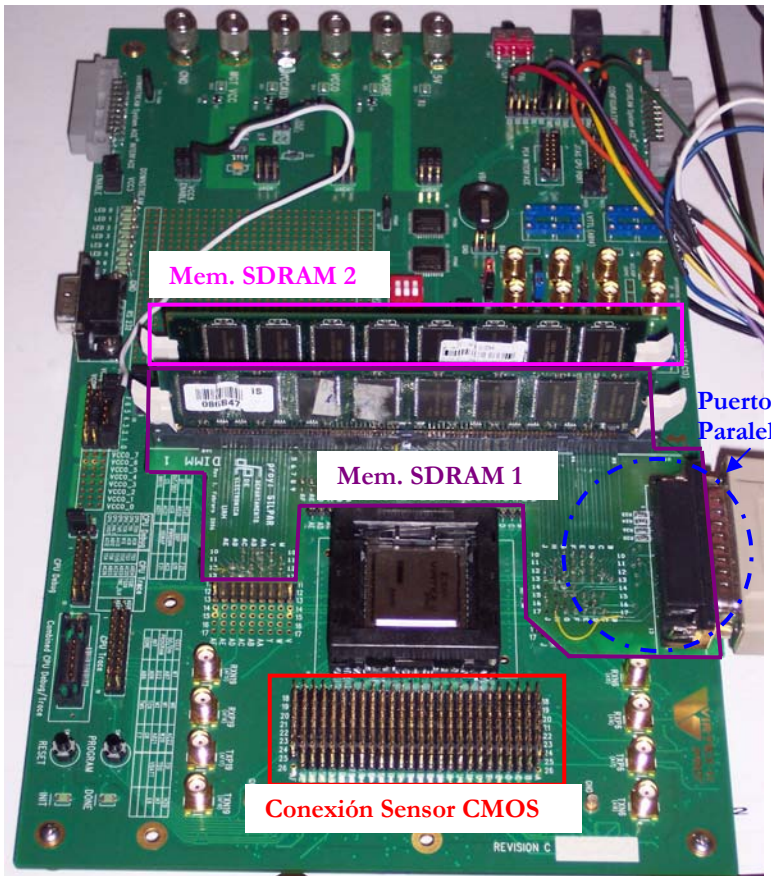


Figura 6.4. Vista de la tarjeta de evaluación de la FPGA, con las tarjetas de memoria externa SDRAM, gestión puerto paralelo y sensor CMOS.

En esta tesis se ha empleado una FPGA para realizar todo el proceso de detección de objetos en movimiento y el control de un sensor CMOS. La estructura lógica interna a la FPGA se divide en los siguientes bloques:

- *Comunicación con el puerto paralelo del PC*. Se establece este canal de comunicación como vía de recepción desde el PC de diversos parámetros como son el tamaño o ventana de imagen deseada. La elección de este canal viene justificada por la sencillez de implementación en la FPGA así como su velocidad de transmisión.

- *Captura de imágenes y controlador del sensor CMOS*. El sensor empleado posee varias señales de control que son activadas desde la FPGA. Además, ésta en función del tamaño de imagen deseado realiza un diezmado o un promediado de la imagen captada (*binning*).

- *Controlador de Memoria*. El sistema está dotado de dos memorias externas dónde se guardan las imágenes captadas y posteriormente las procesadas. El tipo de memoria empleado es SDRAM.

- *Procesamiento de imágenes*. En este caso, el algoritmo implementado ha sido PCA tal y como se ha expuesto en los capítulos anteriores. La

modularidad del diseño permite reutilizar la plataforma para otras aplicaciones de procesamiento de imágenes.

En la Figura 6.5 se muestra el diagrama de bloques correspondiente a la estructura lógica interna de la FPGA.

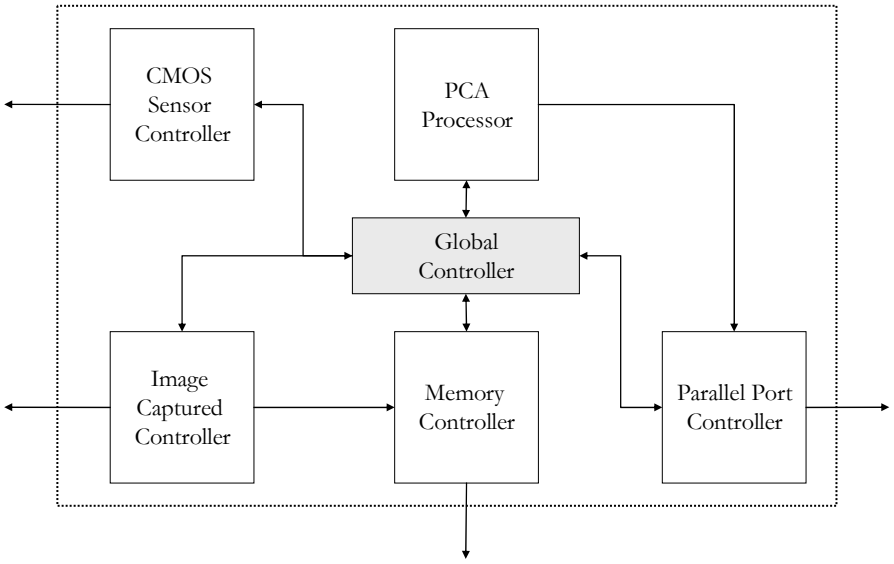


Figura 6.5. Diagrama de bloques principal implementado en la FPGA.

A continuación se describen las características más importantes de los módulos encargados de realizar la comunicación con el puerto paralelo, la captura de imágenes y el almacenamiento en memoria externa.

6.2.1. Comunicación por puerto paralelo.

En esta sección se describe tanto la interfaz desarrollada para dotar a la tarjeta de evaluación de una conexión IEEE-1284 así como su controlador VHDL.

La comunicación de datos a través del puerto paralelo está regida según la normativa IEEE-1284 [IEEE1284, 2000]. Dentro de los diferentes modos de funcionamiento que el estándar IEEE-1284 propone, el modo EPP (*Enhanced Parallel Port*) supone la alternativa más rápida de este tipo de puertos permitiendo un régimen de transmisión máximo de datos de hasta 2MBytes/s.

Debido a que el puerto paralelo sólo se utiliza para configurar ciertos parámetros del sensor CMOS y a tareas de mantenimiento/depuración, el volumen de datos a transferir entre el PC y la FPGA es bajo. Por esta razón se justifica el uso del canal del puerto paralelo. Así, se ha implementado en la FPGA un controlador o interfaz que maneje adecuadamente las líneas de datos (DATA[7:0]) y las señales de control (ASTRB, DSTRB, WRITE, BUSY, WAIT) del puerto paralelo. La estructura interna de este módulo diseñado en la FPGA se muestra en la Figura 6.6. En ella se observa cómo la comunicación entre el puerto paralelo y la FPGA se realiza mediante un conjunto de *P* registros, donde cada registro de la Figura 6.6 está

asociado a un parámetro que el usuario puede configurar en función de la aplicación deseada. Concretamente se han implementado un total de 14 registros mostrándose en la Tabla 6.2 las direcciones de cada uno y la operación que se realiza al escribir/leer en ellos.

El Puerto Paralelo del PC (que ejerce de maestro) solicita leer o escribir mediante el protocolo EPP en una dirección de la FPGA. Cada una de esas direcciones está vinculada a un registro de la FPGA, por lo que al decodificar la dirección, que el PC escribe en el bus de datos/direcciones, se habilita únicamente un registro sobre el que se realiza la operación de lectura/escritura. El controlador global de la FPGA (ver Figura 6.5) se encarga de manejar la información que hay en cada registro.

Con respecto al interfaz hardware necesario para conectar la tarjeta de evaluación de Xilinx con el puerto paralelo del PC, únicamente se debe implementar un buffer bidireccional que adapte los diferentes de niveles de tensión entre la FPGA (3.3V) y el puerto paralelo (5V). En el rectángulo azul de la Figura 6.4 se muestra el área de la tarjeta DIMM1 en la que se ha implementado la zona de comunicación con el puerto paralelo de un PC.

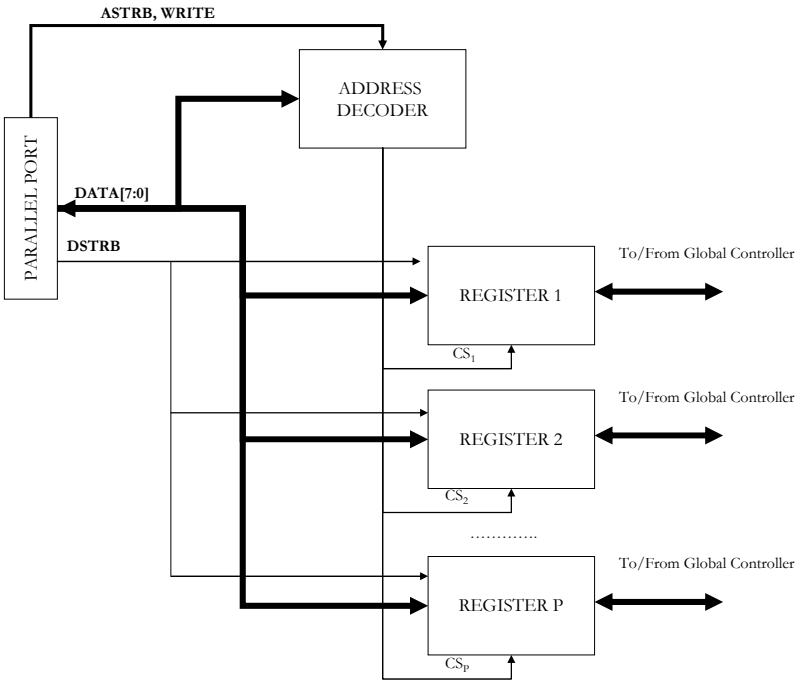


Figura 6.6. Diagrama de bloques del interfaz puerto paralelo de la FPGA.

Tabla 6.2. Registros del controlador del puerto paralelo de la FPGA para realizar la comunicación PC-FPGA.

Dirección	Nombre del registro	Modo	Operación
0	EXPOSURE_TIME1	WR	Tiempo de exposición [7:0]
1	EXPOSURE_TIME2	WR	Tiempo de exposición [15:8]
2	EXPOSURE_TIME3	WR	Tiempo de exposición [23:16]
3	START_ROW	WR	Fila inicial de la ventana seleccionada
4	END_ROW	WR	Fila final de la ventana seleccionada
5	ROW_INC	WR	Selección de modo *binning* o diezmado
6	CONTROL	WR	Registro de control del sistema
7	CONTROL_CAMERA	WR	Registro de control de la cámara
8	REG_1	WR	Reservado para futuras expansiones
9	START_COL	WR	Columna inicial de la ventana seleccionada
10	END_COL	WR	Columna final de la ventana seleccionada
0	STATUS	RD	Registro de estado del sistema
1	EPP_COUNTER_1	RD	Contador de transferencia de datos [7:0]
2	EPP_COUNTER_2	RD	Contador de transferencia de datos [15:8]

6.2.2. Sensor CMOS.

El sensor escogido es el modelo MT9M413C36STM perteneciente a la empresa Micron [Micron, 2004b]. La elección de éste viene justificada por las siguientes necesidades:

- El interfaz de control del sensor está preparado para ser gestionado desde un dispositivo externo a él, pudiendo ser un DSP, microprocesador, o una FPGA (ver Figura 6.7).
- Alta velocidad de generación de imágenes. Esta característica permite que la FPGA proporcione una alta tasa de imágenes procesadas.

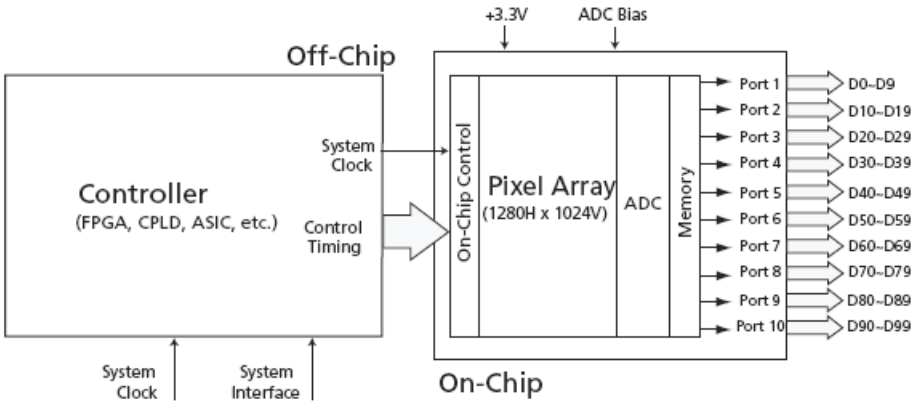


Figura 6.7. Conexión del sensor CMOS MT9M413C36STM con un controlador externo.

Las principales características de este sensor son:

- Resolución espacial de hasta 1280x1024 píxeles y 10 bits de resolución de nivel de gris. Sin embargo el sensor permite variar el tamaño de la imagen captada (número de filas) desde 1x1280 hasta 1024x1280. En la Tabla 6.3 se muestran las diferentes velocidades alcanzadas por el sensor para distintos tamaños de imagen.
- Bus de datos de salida de 100 bits (10 píxeles).
- Exposición simultánea de todos los píxeles (modo *TrueSNAP*).
- Posibilidad de extracción manual de una fila, gracias a un decodificador interno que permite seleccionar mediante la escritura en un bus de direcciones la fila de salida del sensor CMOS.
- 1280 ADCs de 1 bit con muestreo simultáneo.
- Configuración externa de las señales de control y calibración de los ADCs.
- Idéntico encapsulado para las versiones en blanco/negro y color.
- Configuración externa del tiempo de exposición.

Con respecto al funcionamiento del sensor, éste posee tres modos de captación diferentes: simultáneo, secuencial e imagen única. La diferencia entre los tres modos viene determinada por el inicio de la exposición y el inicio de lectura de los píxeles.

- *Modo Simultáneo*: En este modo se van generando continuamente imágenes de salida. Así, una vez vencida una latencia inicial, el inicio de la exposición de una nueva imagen se solapa con la lectura de los últimos píxeles de la imagen anterior.

- *Modo Secuencial*: En este modo también se generan las imágenes continuamente. A diferencia con el simultáneo, el inicio de la exposición de una nueva imagen no se inicia hasta que se ha finalizado la lectura del último píxel de la imagen anterior.
- *Modo Imagen única*: En este modo la generación de cada imagen no es continua, sino que una señal externa determina cuándo debe empezar la captura de una nueva imagen. Por esta razón, no se puede solapar el inicio de la exposición con la lectura de los píxeles de la imagen anterior.

Tabla 6.3. Velocidad de transferencia de imágenes para diferentes resoluciones con un reloj de cámara de 66MHZ (máximo valor posible) [Micron, 2004b].

Tamaño de imagen (filas x columnas)	Velocidad: Imágenes/segundo
1 x 1280	500.000
2 x 1280	250.000
10 x 1280	5000
256 x 1280	2000
512 x 1280	1000
1024 x 1280	500

Una vez expuestas las características del sensor empleado en esta tesis, a continuación se describe el controlador VHDL diseñado para la gestión y manejo del sensor así como el módulo VHDL encargado de modificar el tamaño de la imagen captada (*controlador imágenes captadas*). Por último se presentan las diferentes tarjetas de circuito impreso diseñadas para conectar el sensor CMOS con la FPGA de la tarjeta de evaluación.

6.2.2.1. Controlador VHDL para el sensor CMOS.

El primer aspecto a resolver a la hora de diseñar el controlador para el sensor CMOS es la selección del modo de funcionamiento que se ajuste mejor. En nuestro caso, la FPGA no es capaz de soportar un flujo continuo de imágenes desde el sensor CMOS a la velocidad que emite tanto el modo simultáneo como el secuencial. La razón principal viene impuesta por el tratamiento y procesado que se realiza sobre las imágenes en la FPGA. Tal y como se expondrá posteriormente, cuando la imagen deseada no coincide con el tamaño proporcionado por el sensor, la reducción de tamaño es realizada por la FPGA, siendo enviada a la memoria externa la nueva imagen, donde permanece almacenada para su posterior procesado. Estos pasos se ejecutan mediante un modelo segmentado. Sin embargo, el tiempo que consume cada etapa segmentada junto con la máxima frecuencia de reloj permisible para el diseño implementado en la FPGA, hace inviable el uso del modo secuencial y simultáneo.

Por tanto, la FPGA será más lenta que el sensor y así ella será la que imponga el ritmo de generación de imágenes. Por estas razones el modo de funcionamiento que mejor se adapta es el de imagen única.

Con respecto al controlador VHDL diseñado para la gestión del sensor CMOS elegido, éste se encarga de:

- *Seleccionar una ventana del sensor*: El sensor genera los datos fila a fila. Así, si se desea un ancho de filas diferente al máximo, desde la FPGA se envían las direcciones de las filas inicial y final deseadas. Con respecto al tamaño de columnas, el sensor entrega todos los píxeles correspondientes a una fila; no existiendo posibilidad de seleccionar directamente solamente los píxeles que interesen entre dos columnas. Por ello, es el algoritmo de control del sensor implementado en la FPGA el que se encarga de eliminar aquellos elementos de columnas no deseadas.

- *Ajuste del tiempo de exposición*: El tiempo que los fotodiodos internos del sensor están activados (tiempo de exposición) es seleccionado por el usuario mediante registros accesibles a través del interfaz del puerto paralelo (ver Tabla 6.2).

- *Control de ADCs internos del sensor CMOS*: El sensor empleado obliga a controlar el tiempo de exposición de los elementos fotosensibles, así como sus respectivas señales de control para obtener su dato digital equivalente.

- *Calibración del sensor*: El sensor está dotado de un conjunto de señales de calibración que permite reducir el denominado "*column-wise fixed-pattern noise*" [Micron, 2004b] lo que es gestionado desde la FPGA.

En la Figura 6.8 se muestra el diagrama de bloques de todo el sistema asociado al manejo y control del sensor CMOS, así como al enventanado de imágenes. En él se observa cómo el bloque denominado *CMOS Sensor Controller,* en función de las consignas enviadas desde la máquina de estados de este bloque (*FSM Control*), activa dos bloques de señales de control del sensor CMOS: las asociadas a la imagen (*Pixel Signals*) y las asociadas a la calibración de los ADCs internos (*Calibration Signals*). Los píxeles captados por el sensor y enviados a la FPGA (bus DATA_CAPTURE de la Figura 6.8) son registrados en la FPGA mediante los registros internos de los IOBs (DATA_REG de la Figura 6.8). Mediante esta opción se obtiene un ahorro notable de registros al no consumir recursos de los *slices*, permitiendo además la captura simultánea de todos los píxeles consiguiendo el menor *skew* posible. Con idea de minimizar la información recibida, los píxeles recibidos en la FPGA desde el sensor son truncados de 10 a 8 bits en el proceso de registrado interno (DATA_REG de la Figura 6.8).

Internamente el bloque *CMOS Sensor Controller* está formado por dos máquinas de estados.

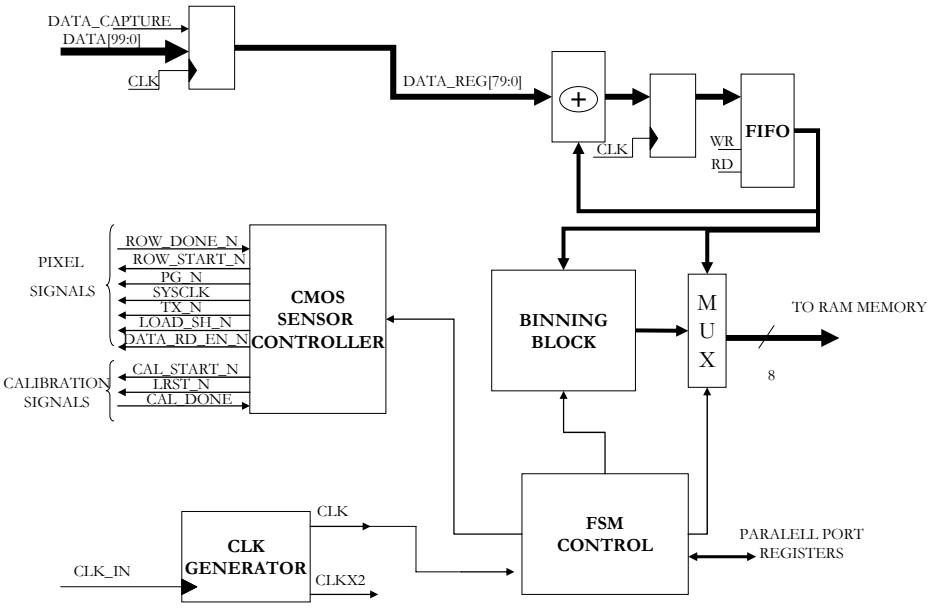


Figura 6.8. Estructura interna del bloque controlador del sensor CMOS y captura de imágenes.

El tiempo necesario para captar una imagen depende del número de filas que tenga la imagen deseada así como de la frecuencia de reloj de la cámara (T_{CLK_CAMERA}). Es importante recordar que el sensor genera 10 píxeles (100 bits) en un ciclo de reloj por lo que el tiempo consumido en la captura de una imagen de *r* columnas por *s* filas es el mostrado en (6.1), donde T_{ROW} es el tiempo que se tarda en generar una fila. Este tiempo será igual a $128T_{CLK\_CAMERA}$ (128 ciclos para generar 10 píxeles) más una latencia inicial en la generación de los primeros 10 píxeles cuyo valor es 4 T_{CLK_CAMERA}.

$$T_{CAPTURE} = T_{ROW} \times s = (128 + L_{ROW})sT_{CLK\_CAMERA} = 132rT_{CLK\_CAMERA} \quad (6.1)$$

La frecuencia máxima de reloj que admite la cámara (T_{CLK_CAMERA}) es de 66MHz y es justamente con este valor con el que se trabaja en nuestro diseño, siendo la FPGA la encargada de generar dicho reloj.

6.2.2.2. Controlador de imágenes captadas.

Este módulo es el encargado de la selección del tamaño y de la posición deseada de la ventana de interés (Bloque *Binning* de la Figura 6.8). Sus parámetros son configurados desde el puerto paralelo mediante la escritura en los registros expuestos en la Tabla 6.2. Una vez que dichos registros han sido configurados, el controlador global del sistema se encarga de enviar al sensor las direcciones de las filas que están dentro de la ventana de interés. Si el tamaño de la imagen deseada es distinto de los expuestos en la Tabla 6.3 se han desarrollado dos posibilidades para generar la imagen final:

a) La primera opción desarrollada ha sido la implementación de un *diezmado* (D) de valor 8, 4 ó 2. Esta alternativa es la más sencilla de realizar ya que de cada D filas el sistema almacena sólo una y de cada D píxeles de cada fila se queda únicamente con una columna. De esta forma el tamaño final de la imagen es reducido en un factor DxD. Sin embargo esta alternativa posee el principal problema de la aparición de un excesivo *aliasing* [Bouman, 2007].

b) La segunda opción, pasa por la implementación de un módulo que promedie las muestras (1 de cada bloque BxB) que se desean diezmar (*binning block*). Así, si se elige esta opción se realiza un promediado (vertical y horizontal) de todas las muestras diezmadas para así obtener un resultado con menor *aliasing* que en el caso anterior, siendo el tamaño final de la imagen BxB veces menor que el tamaño original.

Para ver el efecto del *aliasing* sobre una imagen original diezmada y sobre una promediada, se presenta la Figura 6.9. En esta figura se puede comprobar como el hecho de aplicar *binning* en vez de diezmado mejora notablemente la calidad de la imagen resultante. Ambas opciones pueden ser seleccionadas por el usuario mediante una aplicación de control software desarrollada en esta tesis cuya pantalla principal se puede ver en la Figura 6.10.

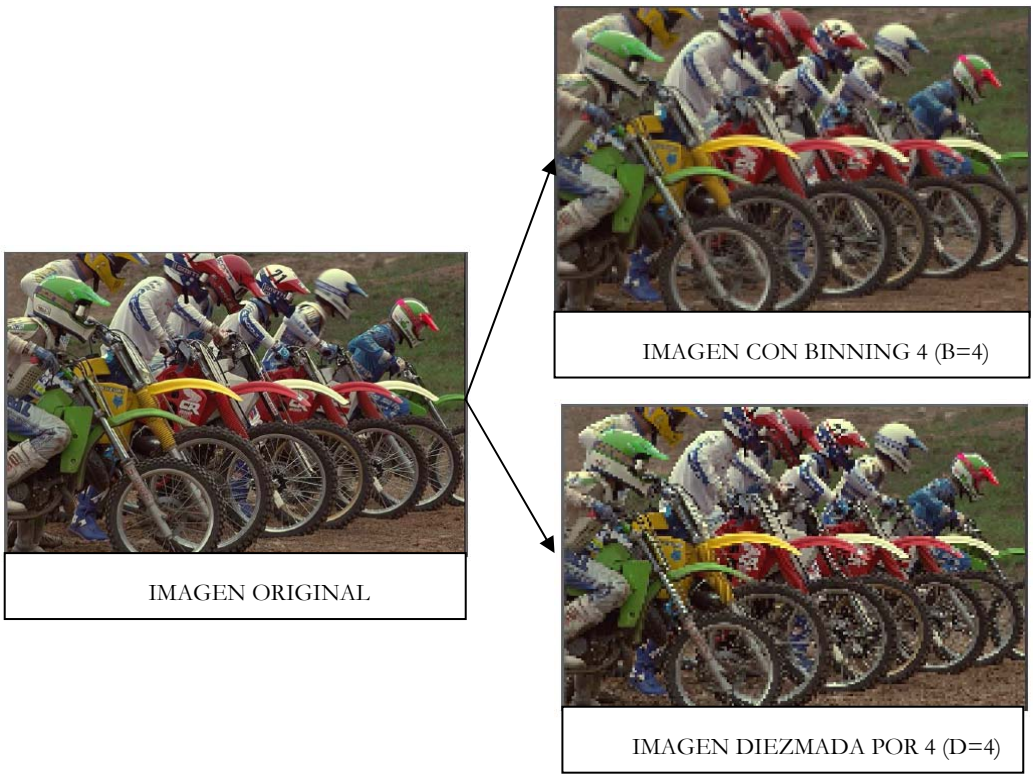


Figura 6.9. Ejemplo de aplicación de *binning* y diezmado a una imagen [Bouman, 2007].

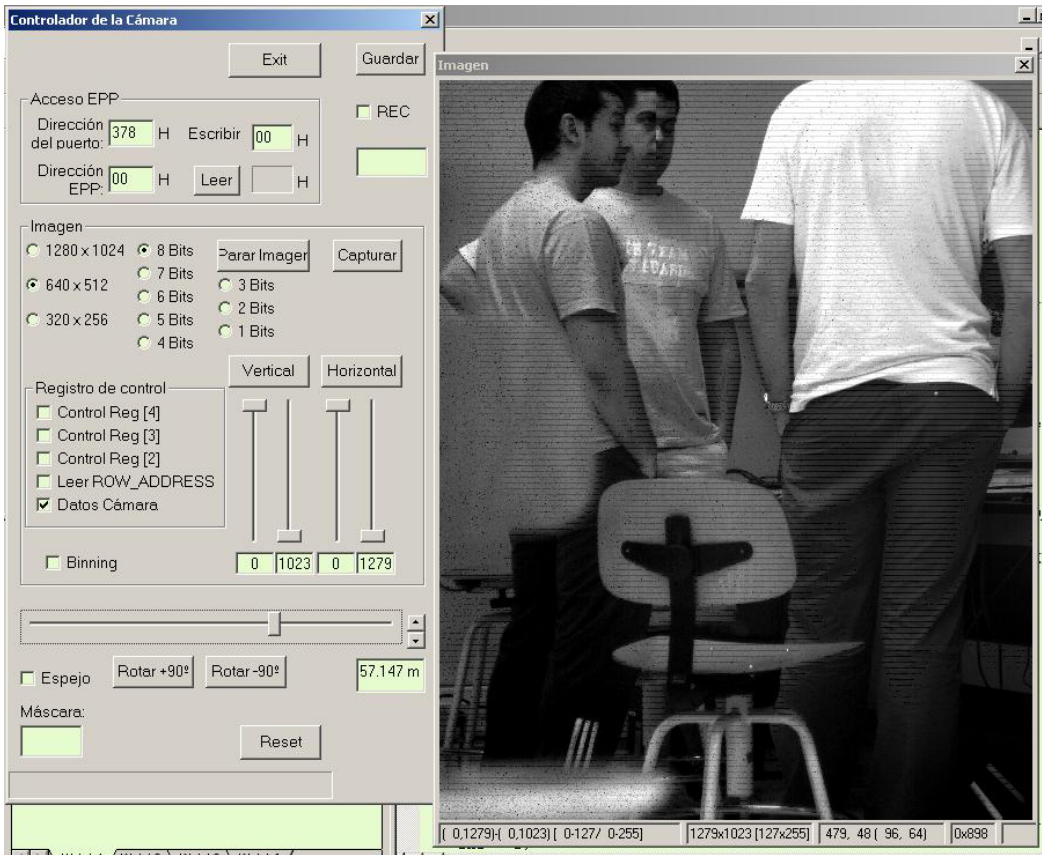


Figura 6.10. Pantalla principal de la aplicación desarrollada para la configuración de la ventana de interés.

En los dos modos de trabajo del sistema mostrado en la Figura 6.8 (diezmado o *binning*) el funcionamiento inicial es idéntico: una vez que el sensor CMOS está habilitado y éste empieza a enviar los píxeles de la primera fila de la imagen, éstos son registrados mediante los registros internos de los IOBs. Durante la recepción de los píxeles de la primera fila el sumador de la Figura 6.8 estará deshabilitado, entrando los datos directamente a una FIFO (ver Figura 6.11), donde $p_{i,j}$ es el píxel *j-ésimo* de la fila i. El tamaño de la FIFO es de 128 palabras de 110 bits. Debido a que la operación suma se realiza únicamente en el modo *binning*, se ha ampliado el ancho de palabra de 80 a 110 bits para evitar desbordamiento. Por su parte, el ancho de la memoria es de 128 palabras ya que si en cada ciclo se reciben 10 píxeles y el tamaño máximo es de 1280 píxeles por línea, son necesarias 128 posiciones para poder guardar todos los píxeles de una línea.

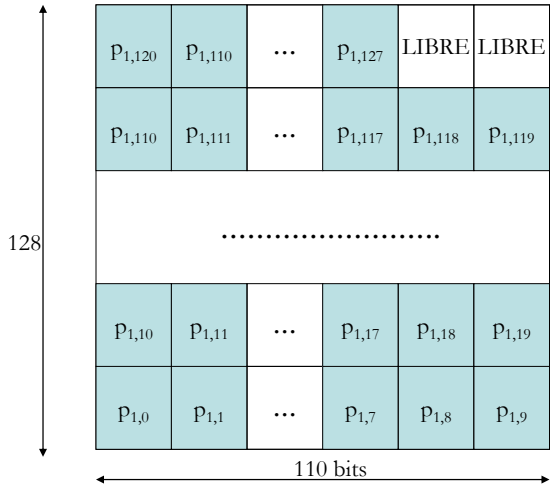


Figura 6.11. Mapa de la memoria FIFO tras la recepción de los píxeles de la primera línea.

Una vez almacenados los píxeles de la primera fila en la FIFO (Figura 6.11), el flujo de datos difiere en función del modo escogido. Si se escoge el modo diezmado, los datos salen de la FIFO hacia un multiplexor (ver Figura 6.8) a medida que se van introduciendo los siguientes píxeles de la segunda línea de la imagen. El multiplexor está habilitado para enviar directamente los píxeles diezmados hacia la memoria externa.

Si el modo escogido es el de *binning*, se deben promediar los píxeles a diezmar. Así, si denominamos como B al factor de *binning* escogido, es decir, el diezmado promediado deseado, una vez recibida la primera línea se habilita el sumador y a medida que llegan los píxeles de la segunda línea se van sumando a los anteriores (ver Figura 6.12.a). Este proceso se repetirá hasta llegar a la *B-ésima* fila (ver Figura 6.12.b).

Una vez registradas las primeras B filas, la memoria FIFO contiene en cada posición el promediado vertical de B filas. Para finalizar la etapa de *binning*, se debe realizar el promediado vertical. Para ello la máquina de estados de la Figura 6.8, habilita convenientemente el bloque de *binning* (ver Figura 6.13) y el multiplexor previo a la memoria externa. En función del factor de *binning* escogido la máquina de estados de la Figura 6.8 activa las señales de control de los sumadores (C_1, C_2, C_3, C_4) así como el valor del registro de desplazamiento RSR para realizar la división asociada al promediado (señal *Load_Shift* de la Figura 6.13).

La ejecución del bloque de *Binning*/Diezmado se realiza de forma simultánea a la captación de píxeles. Una vez que se ha registrado la primera fila este bloque comienza su actividad, poseyendo una latencia de ejecución ($L_{BINNING\_DEC}$) de $4T_{CLK}$. Estos ciclos de reloj están asociados al almacenamiento de la fila de píxeles en la FIFO de la Figura 6.8 y al bloque sumador de la Figura 6.13.

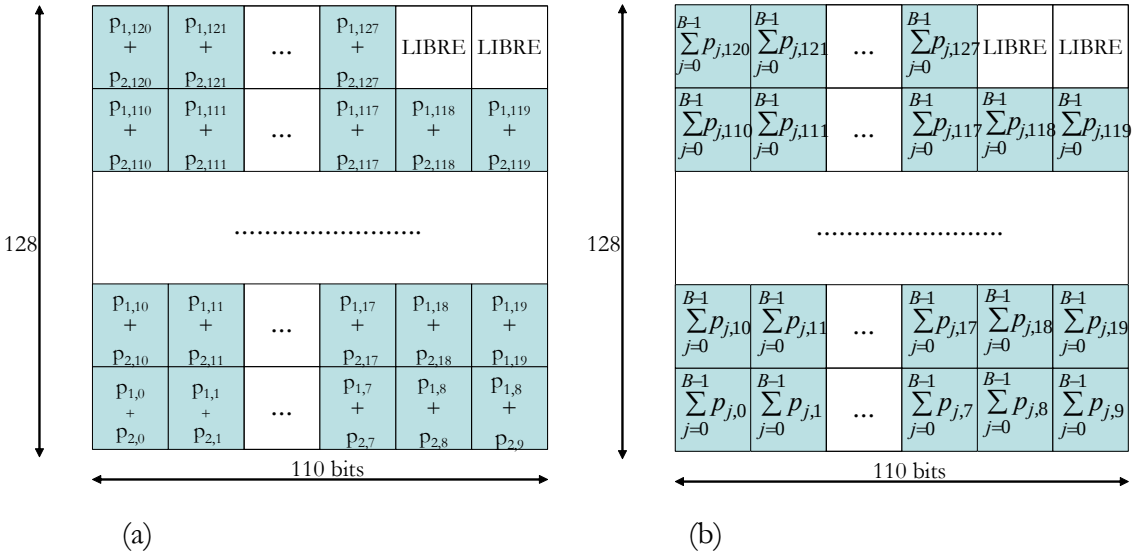


Figura 6.12. Mapa de la memoria FIFO tras la recepción de los píxeles en modo binning de: a) la segunda línea, b) la B-ésima línea.

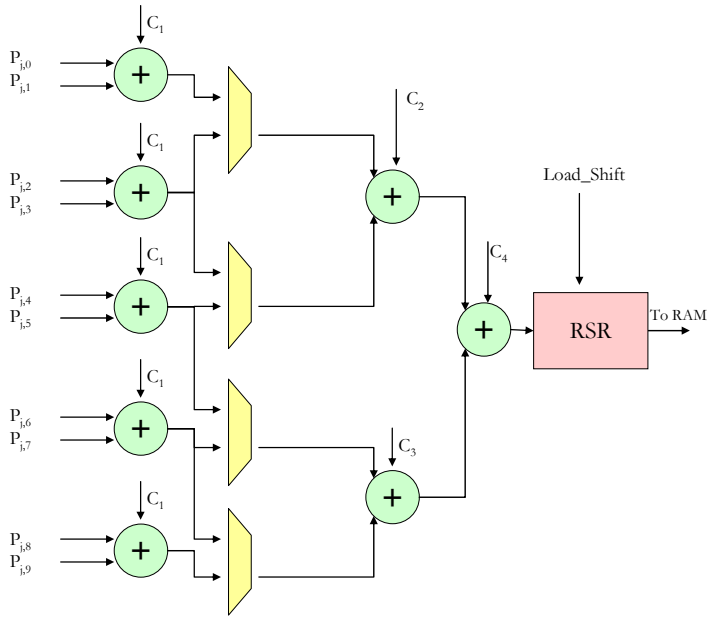


Figura 6.13. Estructura universal del bloque binning, para un factor B= 1, 2, 4, 8.

6.2.2.3. Interfaz de conexión CMOS-FPGA.

Para conectar el sensor y la electrónica adicional que éste requiere para su correcto funcionamiento se ha diseñado una tarjeta de circuito impreso (CMOS_PCB) con el aspecto que se puede ver en la Figura 6.14. En esta tarjeta, además del sensor CMOS, hay un conjunto de potenciómetros para ajustar los valores analógicos del sensor, el módulo de alimentación y los *buffers* de interconexión de las señales digitales del sensor. Esta tarjeta se encuentra insertada en una estructura construida al efecto y que contiene las lentes necesarias para captar de forma adecuada la escena en el sensor minimizando posibles aberraciones ópticas (ver Figura 6.15). Para poder conectar esta estructura del sensor CMOS con la tarjeta

de evaluación conteniendo a la FPGA se ha dotado al sistema de cierta libertad de movimiento al utilizar conexiones mediante cables planos y una tarjeta adicional para su interconexionado (CMOS-FPGA_PCB). De esta manera, esta última tarjeta se conecta por un lado a una de las zonas de pines de expansión de la tarjeta de evaluación de la FPGA y por el otro a 4 cables planos que enlazan con la tarjeta que contiene el sensor CMOS.

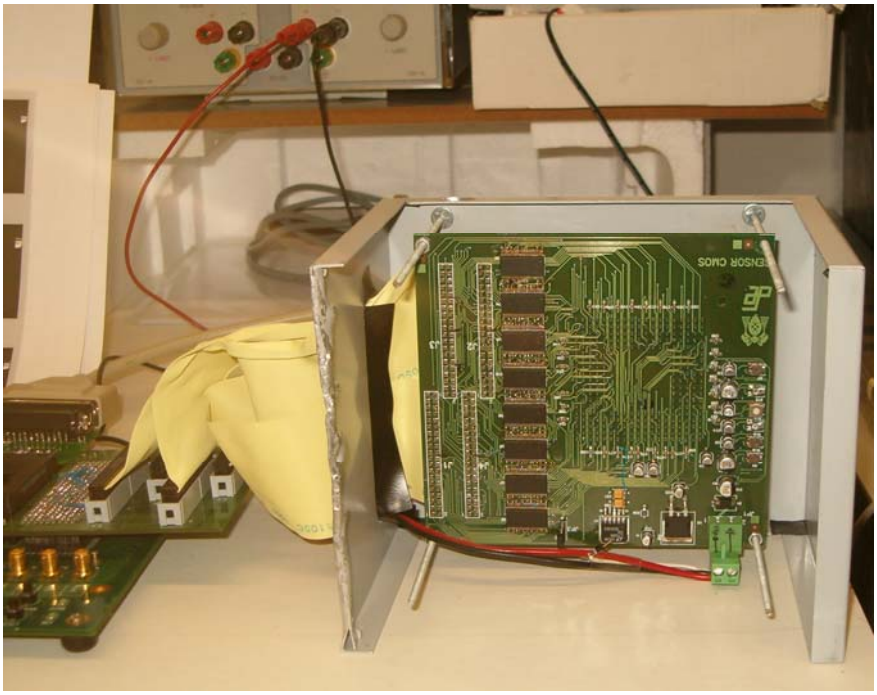


Figura 6.14. Vista de la tarjeta de circuito impreso para alojar el sensor (CMOS_PCB).

6.2.3. Memorias externas.

En la mayoría de los algoritmos de procesamiento de imágenes es necesario almacenar imágenes así como datos temporales para su correcto tratamiento. El caso de PCA no es una excepción siendo necesario en este caso almacenar las M imágenes empleadas para generar la matriz de autovectores, la imagen a procesar ($\mathbf{I}_j \in \Re^{N^2 \times 1}$) y la matriz de autovectores resultante ($\mathbf{U}_l$). Debido al tamaño de las imágenes manejadas en esta tesis (256x256), su almacenamiento dentro de la FPGA se hace inviable ya que los BRAM internos alcanzan un tamaño máximo próximo a 100KB. Por ello se hace necesario el uso de memoria externa.

Aunque la tarjeta de evaluación de Xilinx empleada en esta tesis (HW-AFX-FF672-300) dispone de una memoria externa SDRAM de 8MB con un bus de datos de 32 bits, tras analizar el flujo de lectura/escritura de datos que PCA demanda sobre esta memoria, se produce un gran cuello de botella en la fase *on-line* en los accesos a memoria debido al ancho del bus de datos. Esto provoca la inserción de ciclos de espera en el algoritmo PCA ralentizando así el tiempo total de PCA. Por tanto, con idea de acelerar la ejecución de PCA y, en la medida de lo posible, eliminar el cuello de botella en el acceso a memoria, se ha optado por ampliar el bus de datos de la

memoria externa a 128 bits. Para ello, a la tarjeta de evaluación de Xilinx se le ha dotado de dos tarjetas de circuito impreso auxiliares (DIMM1 y DIMM2) en las que se inserta memoria SDRAM PC133 de 64MB (8Mx64). La diferencia entre ambas es que la DIMM1 además incluye la parte de conexión con el puerto paralelo (Figura 6.16).

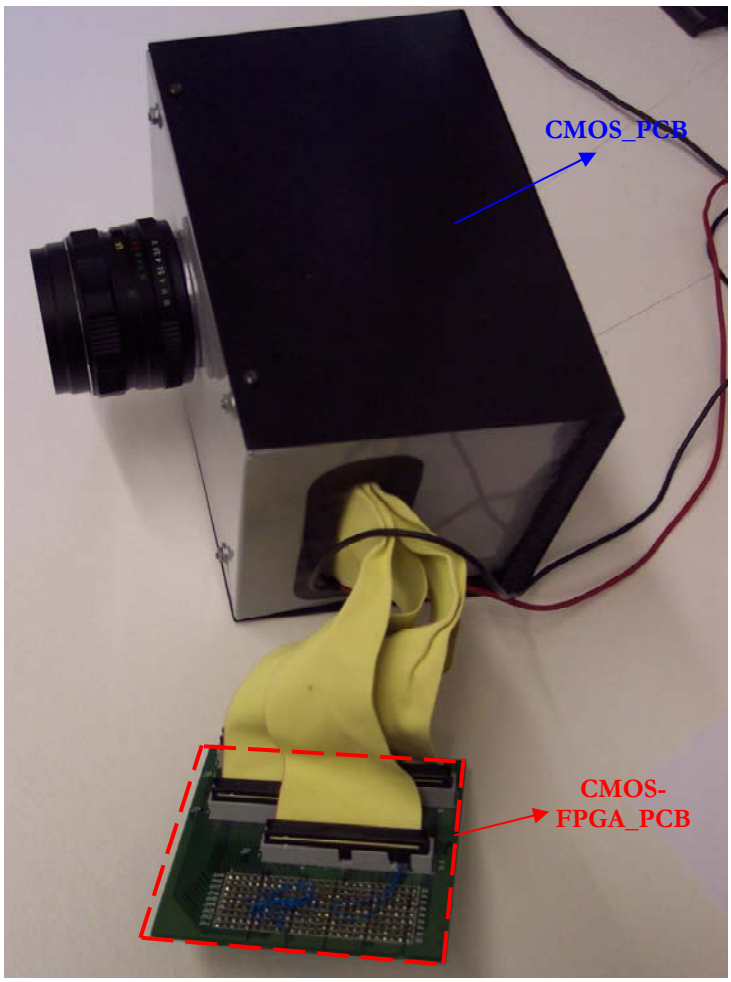


Figura 6.15. Vista de la estructura que contiene la lente y la tarjeta CMOS_PCB, así como la tarjeta de interconexión sensor CMOS-FPGA (CMOS-FPGA_PCB).

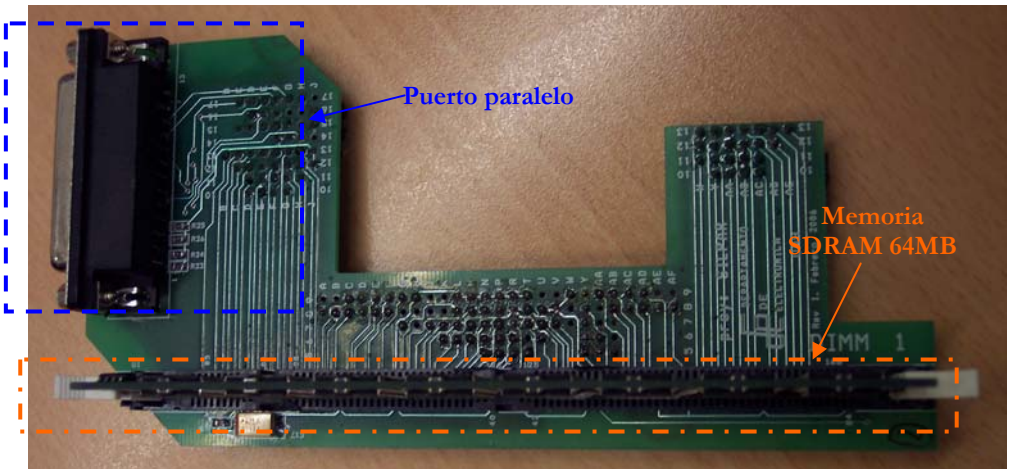


Figura 6.16. Vista de la tarjeta de expansión de memoria SDRAM DIM 1.

Las memorias SDRAM poseen como característica principal la necesitad de estar refrescándose continuamente. Además las memorias disponen de múltiples señales de control para dirigir los procesos de lectura, escritura y refresco. Según el fabricante, cada una de las celdas de memoria es capaz de contener un dato durante 64ms, tiempo tras el cual la celda debe ser refrescada para no perder la información. El refresco puede realizarse de forma distribuida, refrescando posiciones de memoria en cualquier instante de tiempo siempre que cada celda no supere su tiempo máximo de retención. Esto permite, por ejemplo, realizar ciclos de refresco durante los tiempos muertos entre acceso y acceso. También es posible realizar el refresco completo de la memoria de forma atómica una vez transcurridos 64ms desde el anterior refresco.

Las características fundamentales de la memoria empleada son:

- Dual In-line Memory Module (DIMM) de 168 pines.
- Máxima frecuencia de reloj 133 MHz.
- 64MB (8Mx64).
- Funcionamiento interno en *pipeline* (el acceso a columnas puede ser cambiado en cada ciclo de reloj).
- Estructura interna distribuida en 4 bancos.
- 4096 filas y 512 columnas.
- Longitud de ráfaga variable.

Para el acceso a una posición de memoria es necesario activar el banco y la fila en la que se encuentra mediante un proceso que conlleva el uso de varios comandos tal y como se define en [Micron, 2004]. Una vez activada una fila, ésta puede ser leída y/o escrita, pero antes de leer una posición de memoria que no se encuentra en esa fila es necesario realizar un proceso de precarga que desactiva la fila. Antes de iniciar un proceso de refresco es necesario que no haya ninguna fila que esté activada.

Cada memoria contiene internamente 2 registros de configuración denominados *Mode Register* y *Extended Mode Register*, que contienen todos los parámetros referentes a tipo y longitud de ráfaga, latencia de los datos de salida, temperatura de funcionamiento máxima permitida y configuración del estado de auto-refresco. Las lecturas y escrituras pueden ser realizadas de diferentes formas en función de la longitud de la ráfaga escogida, la latencia de los datos y si se utiliza o no precarga automática. A continuación se resumen los aspectos más significativos que se parametrizan en los registros de configuración:

- *Latencia de los datos de salida* (CAS *Latency*). El retardo en ciclos de reloj hasta que los datos son transmitidos a la salida depende de este

parámetro. Con retardos mayores se permite la utilización de la memoria a una frecuencia mayor.

- *Utilización del modo ráfaga*. La longitud de la ráfaga puede ser de 1, 2, 4 u 8 posiciones, o de página completa. La ráfaga de página completa no permite el uso de autoprecarga.

- *Autoprecarga*. En los modos de escritura en ráfaga con tamaño de ráfaga conocido es posible configurar la memoria para que se precargue justo después de terminar el acceso.

- *Características de bajo consumo*. Es posible configurar otros parámetros como por ejemplo, indicar a la memoria la temperatura máxima a la que funcionaría de tal forma que pueda ajustar el sistema de refresco automático, extendiendo el tiempo entre refrescos en función de esa temperatura. También es posible detener el oscilador dejando la memoria en un estado de bajo consumo.

En base a estas especificaciones anteriores se ha diseñado un controlador de memoria codificado en VHDL que permite la lectura y escritura en memorias externas SDRAM. El controlador diseñado permite su empleo en cualquier memoria de tipo SDRAM sin más que variar la longitud del bus de datos y del de direcciones.

Como nuestro diseño posee dos memorias externas, hacen falta dos módulos controladores los cuales funcionan de manera idéntica. En la Figura 6.17 se presenta un diagrama de bloques del sistema de gestión y manejo de las dos memorias externas. Además de los citados controladores de memorias (RAM_CTRL_MEM) aparecen dos memorias DP que funcionan a modo de memoria *cache* del sistema y una máquina de estados (FSM) encargada de secuenciar todo el proceso de lectura/escritura en memoria.

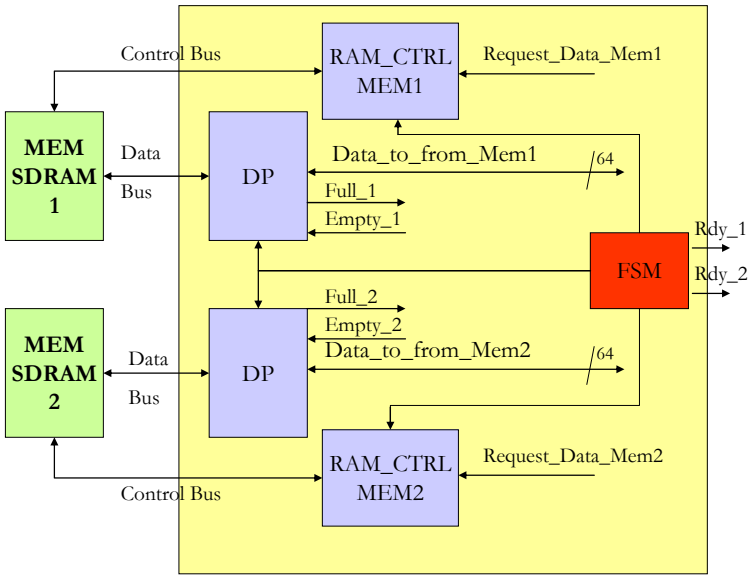


Figura 6.17. Diagrama de bloques del controlador de memorias externas.

Con respecto al mapa de memoria externa, en la Figura 6.18 se muestra el contenido de cada memoria. Se observa como la Memoria 1 sólo se emplea para almacenar las imágenes necesarias para generar la matriz de autovectores. Aprovechando que estas imágenes sólo se emplean para la obtención de la matriz de autovectores, la imagen actual (fase *on-line*) sobre la que se determina la existencia de nuevos objetos, es almacenada en la posición de la imagen captada en primer lugar Por tanto, en caso de actualización de $\mathbf{U}_t$ se van suprimiendo las primeras imágenes captadas. De esta forma se optimiza la actualización de las imágenes necesarias para generar la matriz de autovectores, renovándose siempre las imágenes captadas en primer lugar.

La Memoria 2 sólo almacena la matriz de autovectores la cual será empleada por la fase *on-line* de PCA. Debido al ancho del bus de datos de esta memoria (64 bits) en cada fila se pueden insertar 3 elementos de $\mathbf{U}_t$ (recuérdese que cada elemento estaba codificado con 18 bits) quedando libre en cada fila 10 bits.

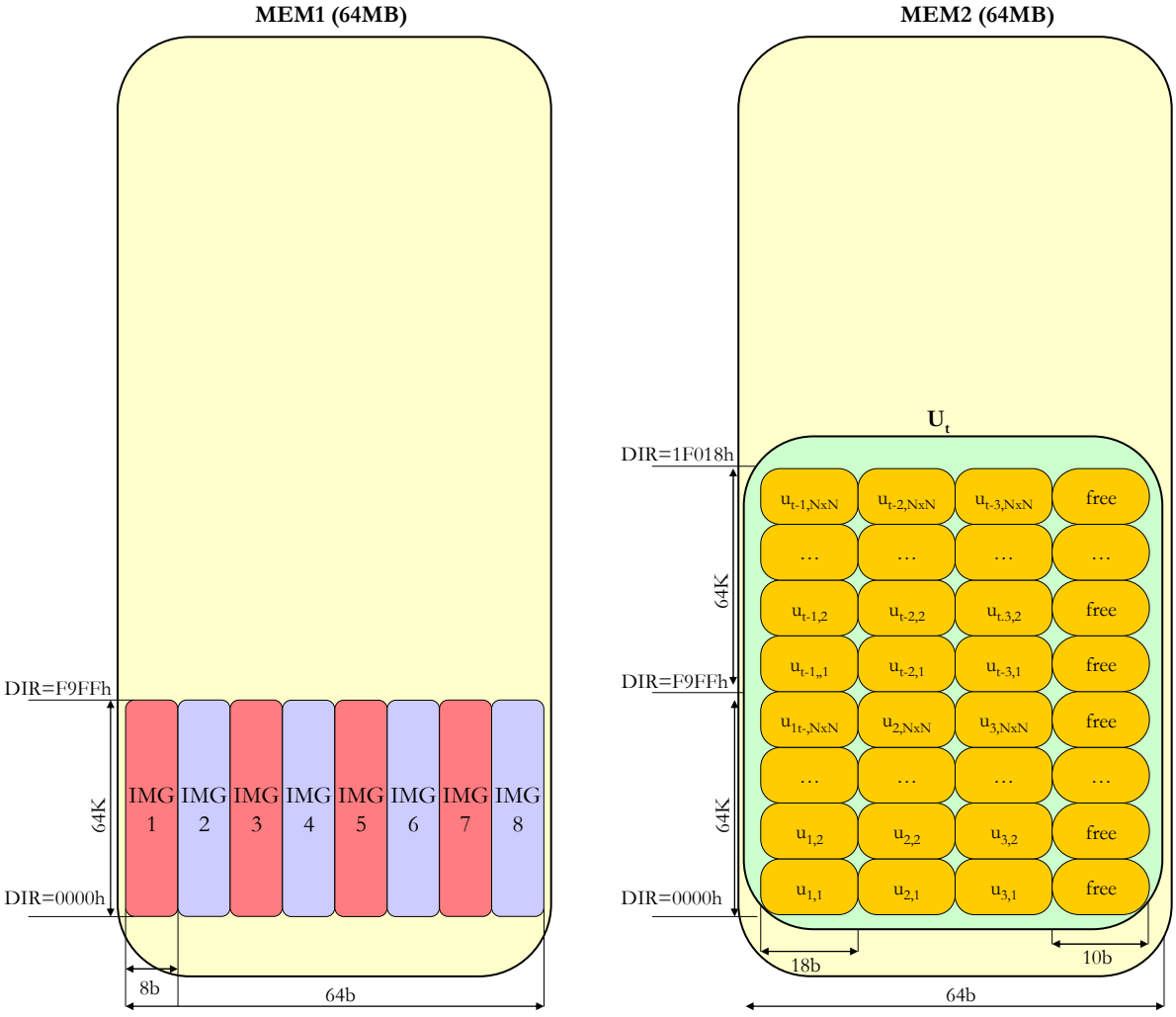


Figura 6.18. Mapa de Memoria de las memorias externas.

6.2.4. Recursos necesarios en la FPGA por los controladores de puerto paralelo, sensor CMOS y memoria externa.

En este apartado se presentan los resultados de recursos consumidos por los controladores del puerto paralelo, del sensor CMOS y de la memoria externa, así como su tiempo de ejecución.

Comenzando con el *puerto paralelo* este módulo, cuyo resumen de recursos consumidos dentro de la FPGA se muestra en la Tabla 6.4, tiene un comportamiento isócrono. El tiempo consumido viene impuesto por la normativa [IEEE1284, 2000] la cual establece que la máxima velocidad alcanzada asciende a 2MB/s.

Tabla 6.4. Recursos consumidos por el bloque controlador de puerto paralelo.

Área (Slices)	BRAM	Multiplicadores	f_{CLKMAX}
13 (0.25%)	0	0	251.12MHz

El tiempo consumido por la FPGA en cada acceso del puerto paralelo es de 2 ciclos de reloj (uno para decodificar la dirección del registro sobre el que escribir/leer y otro parar realizar la escritura/lectura en cada registro asociado al puerto paralelo (Tabla 6.2)).

Con respecto al módulo *controlador del sensor CMOS* la actividad de este elemento es también de tipo isócrona. Su ejecución se inicia cuando se deseen captar las M imágenes necesarias para la construcción del modelo de la escena o cuando se desee captar una imagen sobre la que determinar la existencia de un nuevo objeto ($\mathbf{I}_j$). En ambos casos el tiempo de captación de una imagen ($T_{CAPTURE}$) depende del tamaño de imagen deseado así como del *Binning*/Diezmado a realizar. En nuestro caso para atenuar el efecto de *aliasing* siempre se va a realizar un *Binning* de 4 ($B = 4$) sobre la imagen regular de máxima resolución posible (1024x1024). El tiempo de ejecución necesario desde que se da la orden de captación hasta que finaliza su escritura en la memoria externa SDRAM 1 (T_{IMAGE}) se muestra gráficamente en la Figura 6.19 siendo su valor dado por (6.2), mostrándose en la

Tabla 6.5 el valor y significado de cada uno de sus tiempos parciales.

$$T_{IMAGE} = T_{CAPTURE} + L_{ROW} + L_{BINNING} + L_F + T_{WR\_SDRAM} \qquad (6.2)$$

Tabla 6.5. Descripción de los tiempos parciales de T_{IMAGE}.

$T_{CAPTURE}$	Tiempo que tarda en entregar una imagen completa a máxima resolución el sensor (6.3)
L_{ROW}	Latencia en salir los primeros píxeles del sensor ($4T_{CLK\_CAMERA}$).
$L_{BINNING}$	Latencia del bloque de *Binning* para la última fila de una imagen, siendo su valor de $4T_{CLK}$ (tiempo consumido por los sumadores de la Figura 6.13).
L_F	Latencia que hay hasta que se llena la DP de la Figura 6.17 (T_{CLK})
T_{WR_SDRAM}	Tiempo que tarda en escribirse el bloque completo de memoria DP de la Figura 6.17 sobre la memoria SDRAM (6.4)

$$T_{CAPTURE} = \frac{\left(1280\,pixeles/col \times 1024 cols\right)}{10\,pixeles/T_{CAMERA}} = 131072 T_{CLK\_CAMERA} \qquad (6.3)$$

$$T_{WR\_SDRAM} = \frac{Width\_DP}{BW\_BusData\_DP} T_{CLK} = \frac{1024}{64} T_{CLK} = 16 T_{CLK} \qquad (6.4)$$

Por tanto, el tiempo consumido en la captación y escritura en SDRAM 1 de las M imágenes será:

$$T_{CAPT\_M\_IMAGS} = M T_{IMAGE} = M\left(131076 \cdot T_{CLK\_CAMERA} + 21 T_{CLK}\right) \qquad (6.5)$$

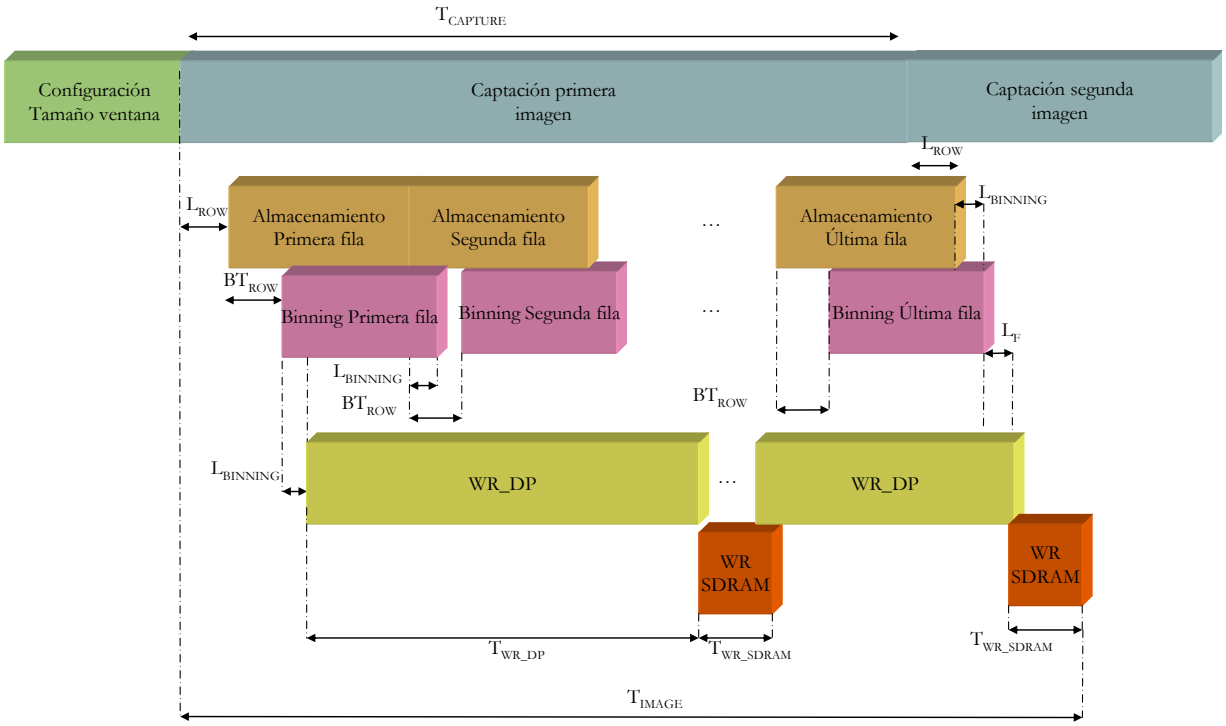


Figura 6.19. Secuencia temporal de funcionamiento para la captación de la primera imagen.

Por otra parte, los recursos internos de la FPGA asociados al controlador CMOS y al bloque de *Binning* se muestran en la Tabla 6.6. Es importante hacer notar que aunque la frecuencia máxima de reloj permisible por este bloque es de 174.63MHz, la cámara externa se manejará con una frecuencia de 66MHz (T_{CLK_CAMERA}) que es la máxima que permite el sensor.

Tabla 6.6. Recursos consumidos por el módulo de diezmado/*binning* constante e igual a 4 y por el controlador del sensor CMOS.

Área (Slices)	BRAM	Multiplicadores	f_{CLKMAX}
344 (7%)	0	0	174.63MHz

Por último, con respecto al análisis del módulo implementado en la FPGA para la *gestión de las memorias SDRAM*, tal y como se ha expuesto en el apartado 6.2.3, el controlador diseñado en VHDL está compuesto por un bloque de memoria DP, una máquina de estados y un módulo de gestión específico de cada memoria (RAM_CTRL_MEM de la Figura 6.17). En la Tabla 6.7 se presenta un resumen de los recursos internos consumidos por todos los elementos encargados de controlar las dos memorias SDRAM. Cabe destacar en este aspecto que aunque la frecuencia de funcionamiento de las memorias SDRAM puede llegar hasta los 133MHz, debido a la existencia de una única señal de reloj dentro de la FPGA, la cual no alcanza los valores máximos posibles de la SDRAM, la frecuencia real de este módulo será de 100MHz, que es la frecuencia patrón utilizada en la FPGA.

Tabla 6.7. Recursos consumidos por el bloque controlador de memoria

Área (Slices)	BRAM	Multiplicadores	f_{CLKMAX}
147 (3%)	4	0	165.53MHz

El sistema mostrado en la Figura 6.17 funciona como el interfaz entre la memoria SDRAM y los diferentes módulos internos de la FPGA. Así, el tiempo de ejecución de este interfaz (T_{CTRL_MEM}) será por una parte el de escritura/lectura de la DP ($T_{WR\_RD\_DP}$) y por otra el de escritura/lectura en la SDRAM (T_{WR_SDRAM} (6.4)), tal y como se expone en (6.6).

$$T_{CTRL\_MEM} = T_{WR\_RD\_DP} + T_{WR\_SDRAM} = Width\_DP \cdot T_{CLK} + 16T_{CLK} = \\ = (1024+16)T_{CLK} = 1040T_{CLK} \quad (6.6)$$

6.3. RESULTADOS FINALES DEL SISTEMA COMPLETO.

En este apartado se muestran los resultados obtenidos de la unión de todos los módulos implementados para la detección de nuevos objetos que han sido expuestos en los diferentes capítulos que forman esta memoria. Al igual que en capítulos anteriores, se presentan los tiempos de ejecución de cada bloque así como los recursos internos consumidos en la FPGA. Además, se exponen diferentes ejemplos de imágenes captadas con la plataforma desarrollada y diversas imágenes preprocesadas en las diferentes etapas que componen la propuesta desarrollada en esta tesis.

6.3.1.1. Tiempos de ejecución.

En la descripción detallada de cada bloque del sistema implementado en la FPGA para la detección de objetos en base a PCA presentada en los capítulos 4 y 5, se han ido justificando los diferentes tiempos de ejecución de los diferentes bloques. Tal y como se ha puesto de manifiesto en numerosas ocasiones en esta tesis, en todo momento se ha buscado segmentar las diferentes tareas para lograr así la máxima velocidad de ejecución posible. Sin embargo, debido principalmente a la dependencia de datos y en otras ocasiones al cuello de botella existente en los accesos a memoria externa, ciertas tareas no se han podido segmentar. En la Figura 6.20 se presenta la secuencia temporal de todo el sistema en función de los diferentes tiempos de ejecución que se han calculado en secciones previas. Así, se pueden distinguir claramente 4 fases en el flujo de funcionamiento cuyo orden temporal es el siguiente:

- *Fase 1:* Captación y escritura de las M imágenes en memoria ($T_{CAPT\_M\_IMGS}$) (6.5).
- *Fase 2:* Generación de la matriz $\mathbf{U}_t$ ($T_{GEN\_WR\_U}$).
- *Fase 3:* Captación y escritura de la imagen $\mathbf{I}_j$ sobre la que detectar la existencia de nuevos objetos (T_{IMAGE}).

- *Fase 4:* Detección de nuevos objetos (T_{OBJ}).

Las fases 1 y 2 están asociados a la generación del modelo de fondo mientras que la 3 y la 4 lo están a la detección de nuevos objetos en la escena.

Una vez finalizada la fase 4, se pasa a la 2 cuando se actualice el modelo de fondo siempre y cuando en $\mathbf{I}_j$ no exista un nuevo objeto (recuérdese que $\mathbf{I}_j$ se escribía directamente en una de las posiciones de memoria de las M imágenes). Por tanto la fase 1 sólo se ejecuta al arrancar el sistema.

Desde un punto de vista cuantitativo, en el cálculo del tiempo de ejecución de toda la propuesta presentada en esta tesis (T_{PCA_TOTAL}), si se supone una actualización continua del modelo de fondo que es el peor de los casos, una vez ya captadas las primeras M imágenes (6.5), el tiempo total consumido se expone en (6.7), mostrándose en la Tabla 6.8 la descripción de cada uno de los tiempos de (6.7).

$$T_{PCA\_TOTAL} = T_{GEN\_WR\_U} + T_{IMAGE} + L_{MEM} + T_{OBJ} \tag{6.7}$$

Tabla 6.8. Descripción de los tiempos parciales de T_{PCA_TOTAL}.

$T_{GEN\_WR\_U}$	Tiempo que tarda la FPGA en generar y escribir en SDRAM la matriz de autovectores $\mathbf{U}_t$.
T_{IMAGE}	Tiempo empleado en la captación de una nueva imagen y su posterior escritura en SDRAM.
L_{MEM}	Latencia de la memoria SDRAM1 desde que se da la orden de lectura de una imagen hasta que se obtienen los primeros datos.
T_{OBJ}	Tiempo consumido en la detección de nuevos objetos tras obtener la imagen recuperada ($\hat{\mathbf{\Phi}}_j$) del espacio transformado

Debido a las diferentes segmentaciones realizadas en el diseño de esta tesis, los tiempos mostrados en la Tabla 6.8 a su vez se dividen en diferentes expresiones. Comenzando con $T_{GEN\_WR\_U}$ el valor de este tiempo viene definido según (6.8) donde el valor de cada uno de los tiempos de esta expresión se expone en la Tabla 6.9.

$$T_{GEN\_WR\_U} = T_{GEN\_U} + \left(L_{MEM2} - T_{E5}\right) \tag{6.8}$$

Tabla 6.9. Descripción de los tiempos parciales de $T_{GEN\_WR\_U}$.

T_{GEN_U}	Tiempo que tarda la FPGA en generar la matriz de autovectores $\mathbf{U}_t$.
L_{MEM2}	Latencia de la memoria SDRAM2 hasta que finaliza la escritura de $\mathbf{U}_t$. Su valor coincide con el denominado anteriormente como T_{WR_SDRAM} (6.4)

T_{E5}	latencia generada en la obtención de las normas de los autovectores (apartado 4.2.5) siendo su valor igual a T_{CLK}

A la hora de calcular el número de ciclos de reloj totales empleados por $T_{GEN\_WR\_U}$ (6.8), el valor obtenido no es constate ya que depende del número de autovectores significativos, del tamaño de la matriz y del número de iteraciones del algoritmo Jacobi tal y como se expone en el apartado 4.2.3.12 y en el 4.2.4. Así, el valor de $T_{GEN\_WR\_U}$ para 6 autovectores (peor caso), con un tamaño de matriz de 8x8 ($M=8$), un ancho de datos internos de 18 bits ($n=18$) y 23 iteraciones para el algoritmo de Jacobi ($h=23$), en base a las expresiones obtenidas en (4.67) y (4.69), sería:

$$T_{GEN\_WR\_U} = 16 \cdot T_{CLK} + 198370 \cdot T_{CLK} + (16-1) \cdot T_{CLK} = 198401 \cdot T_{CLK} \qquad (6.9)$$

El siguiente tiempo a determinar en la expresión (6.7) es T_{IMAGE}. Su expresión fue justificada anteriormente en el apartado 6.2.4 siendo su valor el expuesto en (6.2).

El último a tiempo a considerar en T_{PCA_TOTAL} (6.7) es T_{OBJ}. Tal y como se puede comprobar en la Figura 6.20 este tiempo viene dado por (6.10), mostrándose en la Tabla 6.10 los diferentes tiempos que componen esta última expresión.

$$T_{OBJ} = T_{ON-LINE} + T_{CALC\_D} = (262169 + 66332) T_{CLK} = 328501 \cdot T_{CLK} \qquad (6.10)$$

Tabla 6.10. Descripción de los tiempos parciales de T_{OBJ}.

$T_{ON\text{-}LINE}$	Tiempo consumido desde que se empieza a leer la imagen captada de la memoria SDRAM 1 hasta que se obtiene la imagen proyectada y recuperada ($\hat{\mathbf{\Phi}}_j$). Su expresión también es función del número de autovectores significativos (4.88) (4.89) siendo su valor para el peor de los casos igual a 262169 ·T_{CLK}.
T_{CALC_D}	Tiempo consumido en la construcción del mapa de distancias promediado y en la búsqueda de nuevos objetos sobre él. En (5.2) se muestra su expresión siendo el peor de los casos igual a 66332 ·T_{CLK}

Con todos los tiempos ya definidos, se presenta el valor total de T_{PCA_TOTAL} en la expresión (6.11)

$$\begin{aligned} T_{PCA\_TOTAL} &= 198401 T_{CLK} + (131076 T_{CLK\_CAMERA} + 21 T_{CLK}) + 16 T_{CLK} + 328501 T_{CLK} = \\ &= 131076 T_{CLK\_CAMERA} + 526939 T_{CLK} \end{aligned} \qquad (6.11)$$

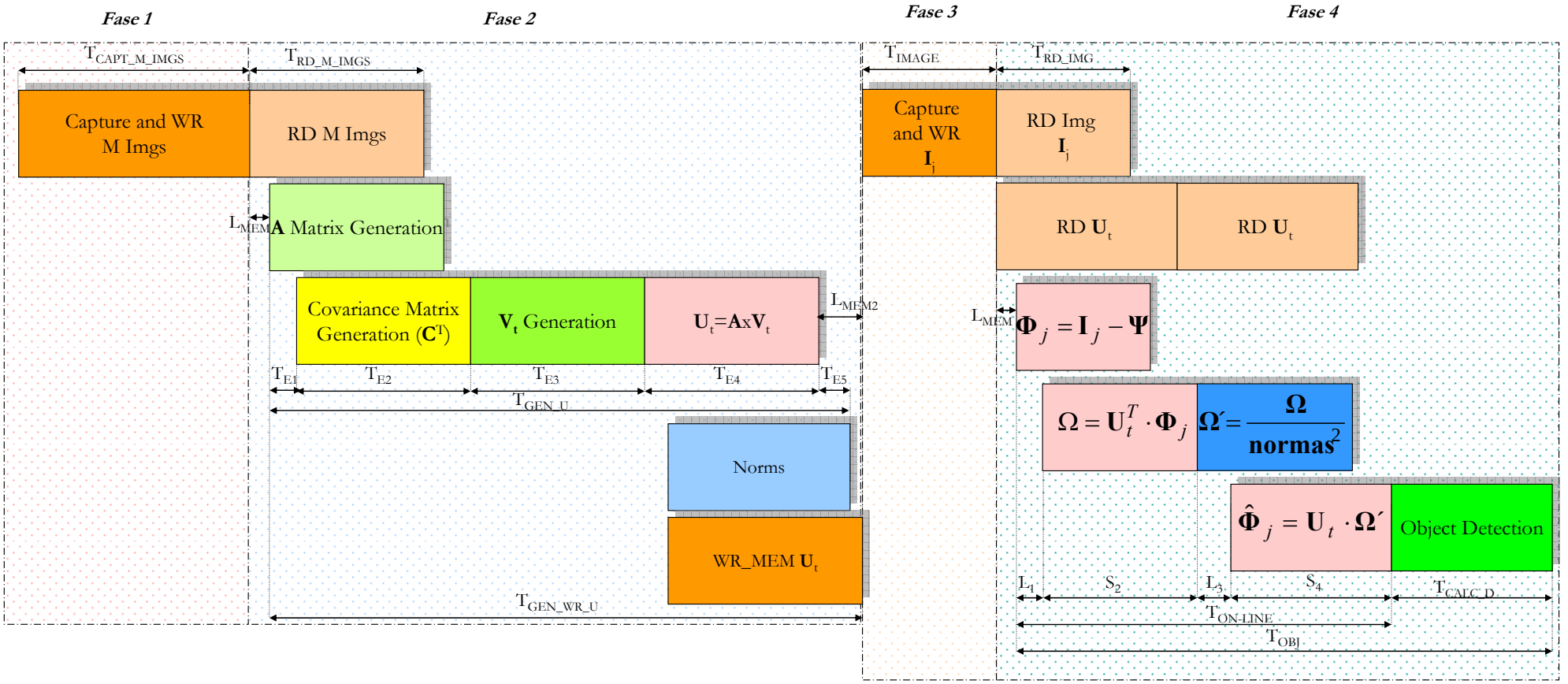


Figura 6.20. Secuencia temporal del sistema de detección de objetos empleando PCA.

Para obtener una relación del número de imágenes que procesa el sistema, suponiendo una actualización continua del modelo de fondo (peor de los casos), si el reloj del sensor CMOS (T_{CLK_CAMERA}) es de 66MHz y el patrón de la FPGA es de 100MHz (frecuencia que se ha logrado alcanzar tras la implementación total del sistema) se alcanza un mínimo de 121 imágenes de 256x256 por segundo.

Este ratio aumenta notablemente si se da alguna de las siguientes situaciones:

a) *Número de autovectores significativos (t) menor de cuatro*. En este caso las operaciones de multiplicación de matrices $\mathbf{U}_t = \mathbf{A} \cdot \mathbf{V}_t$ (4.9), $\mathbf{U}_t^T \cdot \mathbf{\Phi}_j$ (4.73) y $\hat{\mathbf{\Phi}} = \mathbf{U}_t \cdot \mathbf{\Omega}'$ (4.74) se reducen notablemente, tal y como se justificó en el capítulo 4. De esta forma el nuevo valor de T_{TOTAL} con actualización continua del modelo de fondo sería el mostrado en (6.12), alcanzando en este caso un ratio de imágenes por segundo igual a 189.

$$\begin{aligned} T_{PCA\_TOTAL} &= 132867T_{CLK} + \left(131076T_{CLK\_CAMERA} + 21T_{CLK}\right) + 16T_{CLK} + 197428T_{CLK} = \\ &= 131076T_{CLK\_CAMERA} + 330332T_{CLK} \end{aligned} \tag{6.12}$$

b) *Actualización selectiva*. En el caso de que no se actualice continuamente la matriz de autovectores, sino que entre actualización y actualización transcurran b imágenes, el nuevo ratio de imágenes por segundo que se obtiene se muestra en la Figura 6.21. Se observa en dicha figura como a partir de $b = 100$ independientemente del número de autovectores significativos, el sistema alcanza su máximo valor entorno a 250 imágenes por segundo para $t \leq 3$ y entorno a 190 para $t \geq 4$. Esto se debe a que en este número de imágenes la segmentación del sistema mostrada en la Figura 6.20 alcanza el máximo rendimiento.

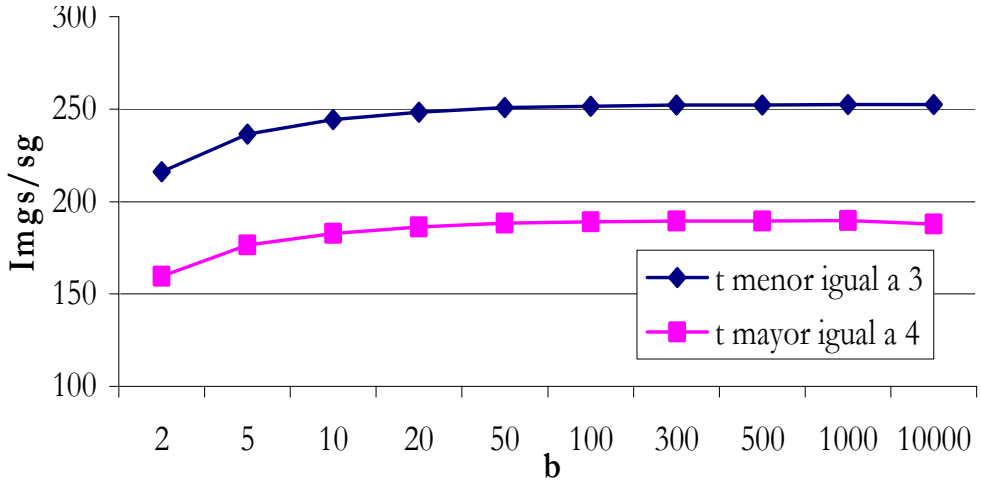


Figura 6.21. Ratios de imágenes por segundo alcanzadas para $b \neq 1$.

6.3.1.2. Recursos internos consumidos.

En la Tabla 6.11 se presenta el resumen de recursos finales consumidos por los diferentes bloques implementados en la FPGA. Es importante destacar que debido al número limitado de recursos de la FPGA se ha intentando optimizar el diseño en todo momento, con idea de minimizar el uso de recursos internos. Gracias a esto se ha conseguido implementar todo el sistema en una FPGA de gama media-baja como es la XC2VP7, desde un punto de vista de número de elementos BRAM y de *slices*.

Tabla 6.11. Resumen de todos los recursos consumidos por todo el sistema desarrollado en esta tesis para una XC2VP7.

Área (Slices)	BRAM	Multiplicadores	f_{CLKMAX}
4225 (86%)	40 (91%)	43 (98%)	112,4MHz

Con respecto a la máxima frecuencia de reloj de la FPGA, según los informes generados por la herramienta de implementación, se asegura para toda la FPGA un valor máximo de 112,4MHz. Sin embargo la frecuencia maestra escogida para nuestro diseño ha sido la de 100 MHz ya que a partir de este valor se pueden generar el resto de frecuencias necesarias (frecuencia de cámara y de memorias externas).

6.3.1.3. Ejemplos de diferentes detecciones de nuevos objetos aplicando la propuesta implementada.

En este apartado se presentan distintos ensayos reales en la detección de nuevos objetos en 3 escenas diferentes. Todas las imágenes presentadas han sido captadas con la plataforma presentada en el apartado 6.2, adquiriendo en todos los casos las imágenes a máxima resolución (1280x1024) reduciendo su tamaño a imágenes cuadradas de 256x256 mediante *binning* (B=4) en la FPGA. Todos los ejemplos corresponden a escenas tomadas en los alrededores del Edificio Politécnico de la Universidad de Alcalá bajo diferentes condiciones de iluminación.

El primer ejemplo que se presenta detecta la presencia de una persona en un área sometida a grandes zonas de sobras. El conjunto de las 8 imágenes que se tomaron al principio para construir la primera matriz de autovectores se presenta en la Figura 6.22. Con estas imágenes se generó el vector media cuya vista se muestra en la Figura 6.23.

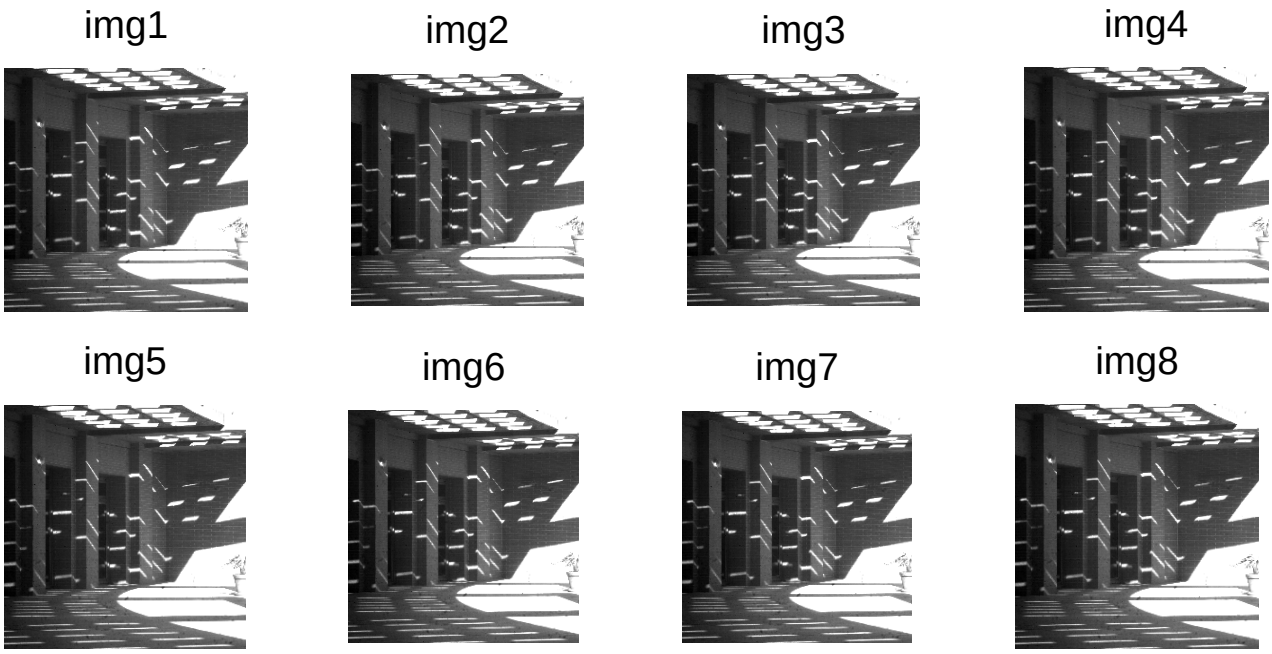


Figura 6.22. Conjunto de imágenes empleadas para calcular la matriz de autovectores ($M = 8$).

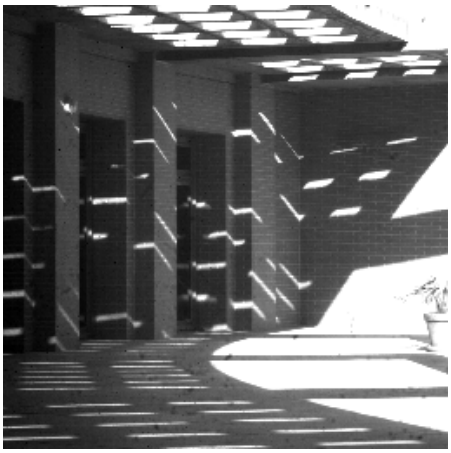


Figura 6.23. Vista de la imagen del vector media construido en base a las imágenes de la Figura 6.22.

Una vez obtenida la matriz de autovectores $\mathbf{U}_t$, la fase *on-line* se activa y se capta una nueva imagen (Figura 6.24). Comparando la imagen actual (Figura 6.24) con la escena sin objetos (Figura 6.22) se manifiesta la presencia de un nuevo objeto, en este caso una persona, la cual debe ser detectada por el algoritmo implementado en la FPGA. Para ello el primer paso es la construcción del vector de la imagen actual menos la media ($\mathbf{\Phi}_j$). En la Figura 6.25 se muestra dicho vector imagen. Una vez obtenido $\mathbf{\Phi}_j$ éste es proyectado y recuperado en el espacio transformado obteniéndose el denominado vector estimada ($\hat{\mathbf{\Phi}}_j$) cuya vista se puede ver en la Figura 6.26.

Figura 6.24. Imagen sobre la que detectar nuevos objetos.

Figura 6.25. Imagen actual menos la media con objeto.

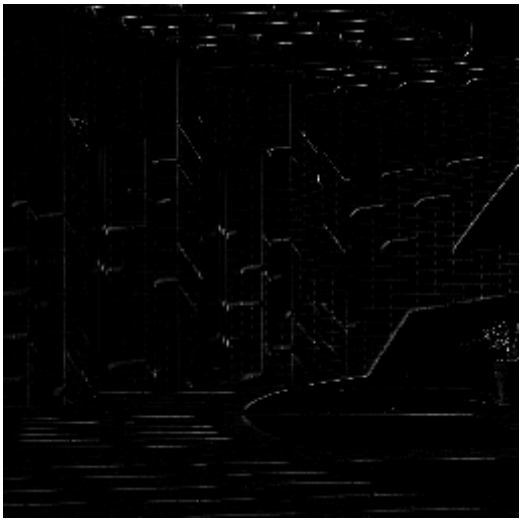


Figura 6.26. Imagen proyectada y recuperada**.**

El siguiente paso es el cálculo del mapa de distancias en base a $\mathbf{\Phi}_j$ y $\hat{\mathbf{\Phi}}_j$. Éste se presenta en la Figura 6.27 sin realizar ningún promediado y en la Figura 6.28 mediante un promediado con una ventana de 3x3.

Figura 6.27. Mapa de distancias.

Figura 6.28. Mapa de distancias promediado con ventana de 3x3 elementos.

Por último, con la información proporcionada por el mapa de distancias promediado se debe determinar la presencia de un nuevo objeto sobre éste. Para ello según la propuesta descrita en esta tesis, se calcula el histograma de los máximos de las columnas para buscar sobre él la existencia de un valle que servirá como valor del umbral (capítulo 5). En la Figura 6.29 se presenta en la parte superior izquierda el máximo de cada columna, en la parte superior derecha la imagen sobre la que determinar la existencia de nuevos objetos, en la parte inferior izquierda el histograma de los máximos y por último en la parte inferior derecha la umbralización hecha del mapa de distancias a partir del umbral Th_h calculado según el algoritmo descrito en el capítulo 5.

En el resto de ejemplos que se presentan se muestran los resultados finales generados por el algoritmo PCA implementado para una secuencia continua de imágenes. De esta forma en la Figura 6.30 se muestran las 8 imágenes tomadas para construir $\mathbf{U}_t$ en un área del Edificio Politécnico dónde la distancia entre los objetos a detectar y la cámara ascendía a unos 25 metros. De nuevo los objetos a detectar serán personas, tal y como se muestra en la Figura 6.31. El resultado de la detección se muestra en la Figura 6.32.

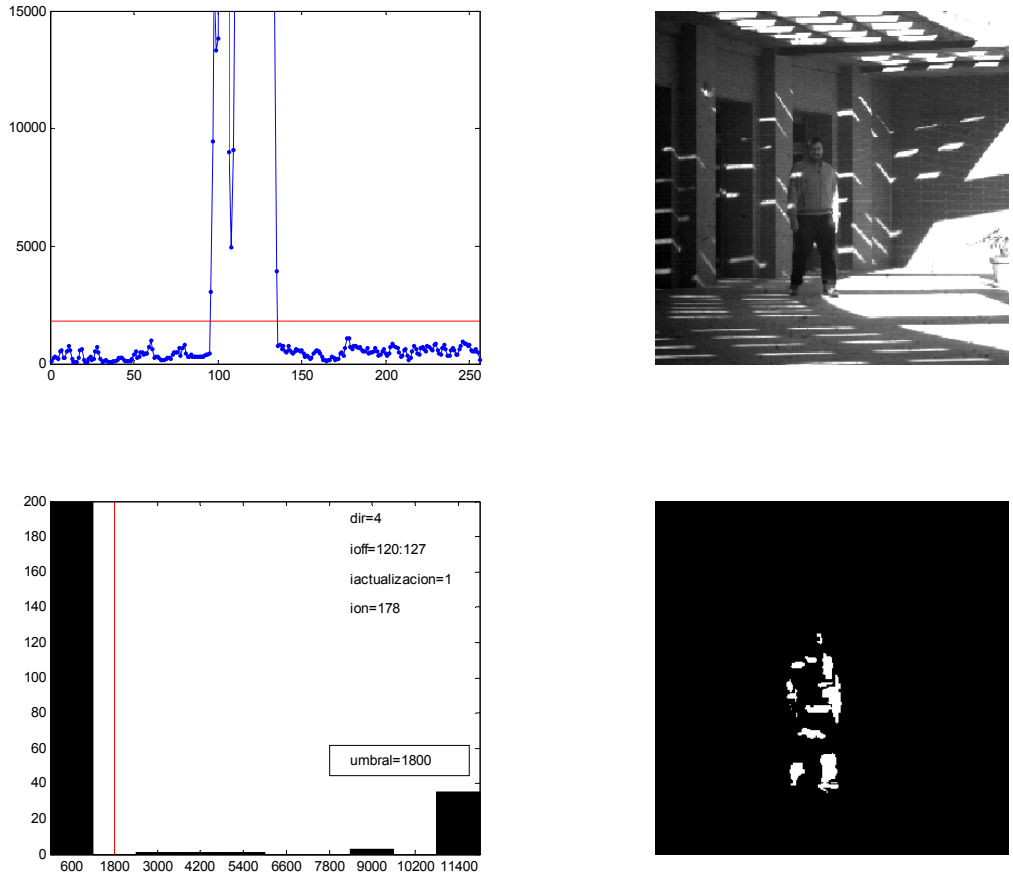


Figura 6.29. Máximos de cada columna de la imagen (arriba izquierda), histograma de los máximos de cada columna (abajo izquierda), imagen captada (arriba derecha) e imagen umbralizada (abajo derecha).

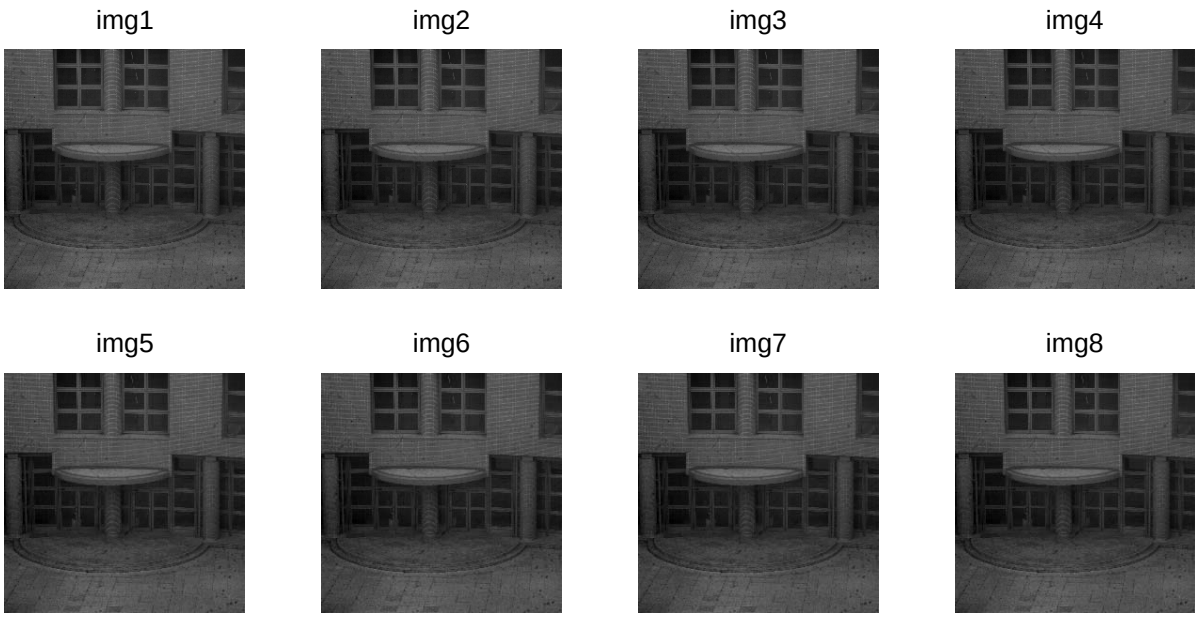


Figura 6.30. Conjunto de imágenes captadas para generar $\mathbf{U}_t$ en el segundo ejemplo.

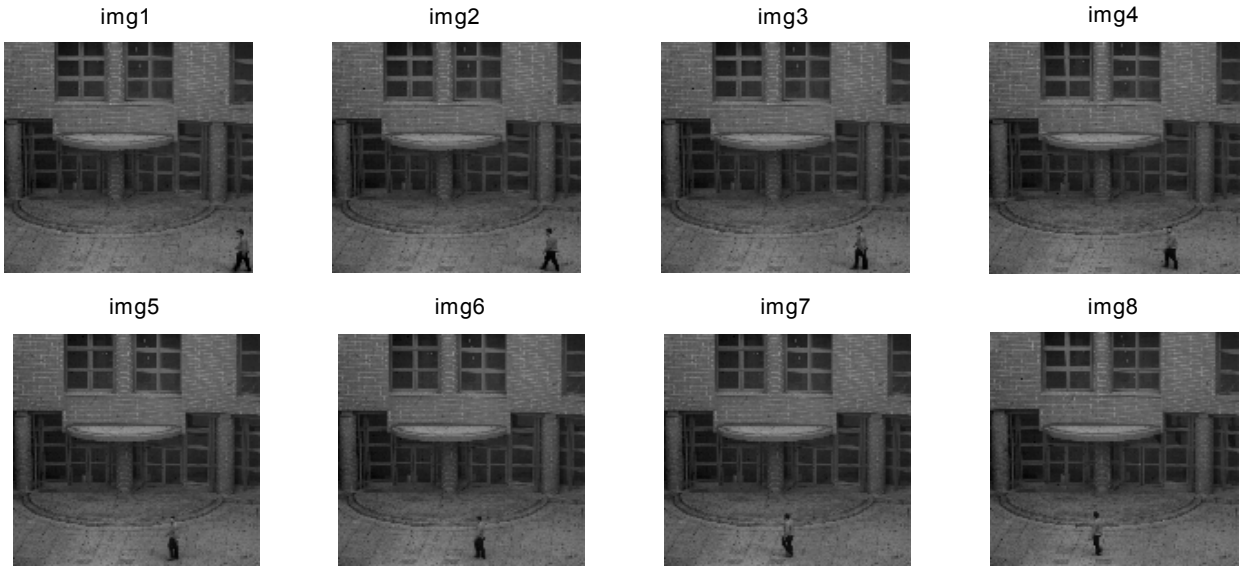


Figura 6.31. Secuencia de imágenes tomadas para determinar nuevos objetos con respecto a la escena de la Figura 6.30.

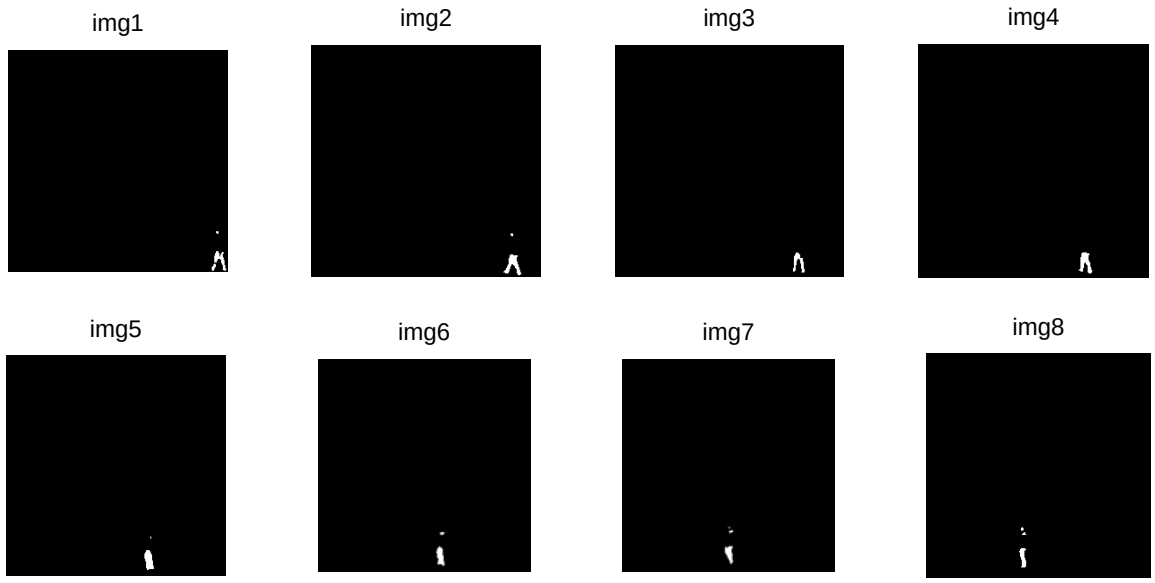


Figura 6.32. Secuencia de imágenes detectadas para determinar un nuevo objeto a partir de las tomadas en la Figura 6.31.

Por último, el tercer ejemplo detecta una secuencia de nuevos objetos sobre una escena inicial captada (Figura 6.33). La Figura 6.34 presenta los objetos a detectar mientras que la Figura 6.35 presenta la detección de una persona en la escena de la Figura 6.33.

Figura 6.33. Conjunto de imágenes captadas para generar $\mathbf{U}_t$ en el tercer ejemplo.

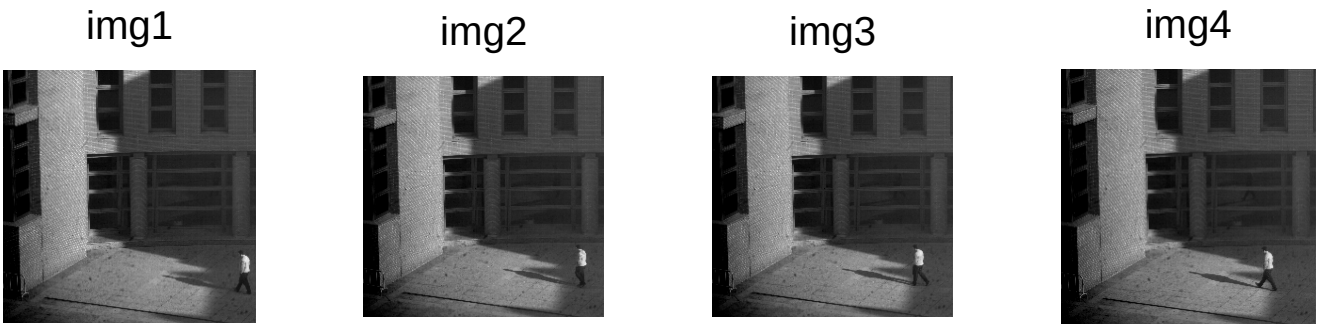


Figura 6.34. Secuencia de imágenes tomadas para determinar nuevos objetos con respecto a la escena de la Figura 6.33.

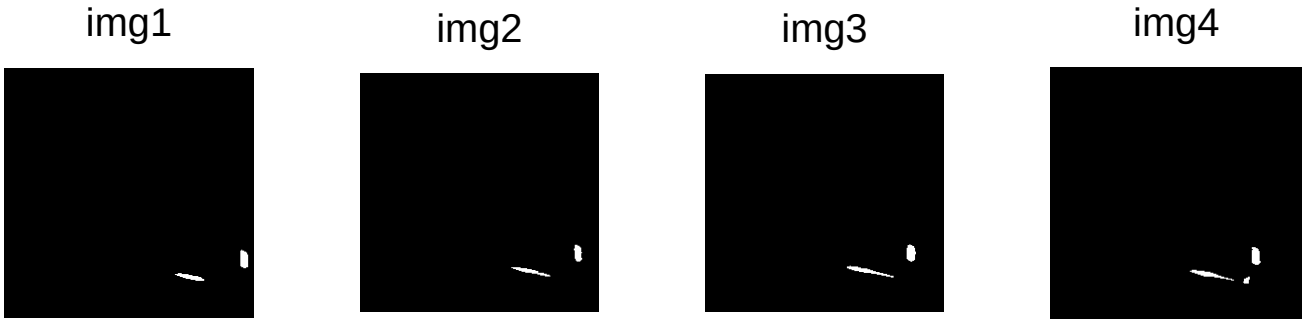


Figura 6.35. Secuencia de imágenes detectadas para determinar un nuevo objeto a partir de las tomadas en la Figura 6.34.

6.4. CONCLUSIONES.

En este capítulo se ha puesto de manifiesto que la arquitectura propuesta en esta tesis responde, desde el punto de vista práctico, a las expectativas que de ella se esperaban a partir de la propuesta teórica y de las diferentes simulaciones realizadas. Se ha podido comprobar en las pruebas prácticas realizadas su excelente comportamiento tanto desde el punto de vista de tiempo de proceso como de precisión en las operaciones que incluye la técnica PCA aplicada al procesamiento de imágenes; más concretamente a la detección de movimientos dentro de una escena captada por una cámara. Indicar asimismo que en nuestro caso la implementación en FPGA incluye además de los cálculos de autovalores y autovectores de matrices (base de la técnica PCA) todos los algoritmos de obtención de proyección y recuperación de imágenes desde el espacio transformado, así como todo los procesos de obtención de umbrales, etc., necesarios para la detección de nuevos objetos dentro de una escena que se toma como referencia.

Las diversas pruebas realizadas en entornos reales y para diferentes condiciones de iluminación, tamaño de objetos avalan la propuesta realizada tanto desde el punto de vista del algoritmo como de la solución hardware que da respuesta muy satisfactoria a las demandas de capacidad de proceso que se necesitan para un correcto funcionamiento en tiempo real.

7. CONCLUSIONES Y TRABAJOS FUTUROS

7.1. CONCLUSIONES.

En esta tesis se ha propuesto un nuevo sistema de adquisición y procesado de imágenes basado en FPGAs para la detección de nuevos objetos en una escena, partiendo de un modelo de referencia de la misma. Para ello se ha utilizado la técnica de Análisis de Componentes Principales (PCA), siendo el objetivo principal de esta tesis el de su paralelización, para lograr así una ejecución concurrente que permite alcanzar ratios de velocidad de proceso entorno a 120 imágenes por segundo. Esta velocidad de proceso, incluyendo todas las fases que incluye la técnica PCA (cálculo de autovalores y autovectores, proyección y recuperación de imágenes al/desde el espacio transformado, obtención del mapa de distancias, etc.) da respuesta a muchas de las aplicaciones donde el objetivo es la detección de nuevos objetos en la escena, incluso en aquellos casos donde se requiera, por razones diversas (cambios de iluminación, por ejemplo) una actualización continua del modelo de fondo. La solución propuesta mejora de forma muy significativa las soluciones basadas en el uso de un PC [Vázquez, 2006]. El desarrollo íntegro del algoritmo PCA en una FPGA era una tarea que hasta el momento no se había conseguido/realizado, o no consta en la revisión, amplia, de trabajos relacionados con este tema llevada a cabo

Otra contribución destacable de esta tesis ha sido el diseño de una plataforma específica, basada en una FPGA de gama media/baja desde el punto de vista de

recursos internos, para la captura y procesado de imágenes procedentes de un sensor CMOS. Esta plataforma ha sido dotada de todos los periféricos necesarios para asegurar un correcto funcionamiento de la aplicación final.

A continuación se resumen, de forma ordenada, las principales aportaciones realizadas en el desarrollo de esta tesis:

- *Implementación íntegra de PCA en una FPGA.*

  Esta tesis constituye el primer trabajo que implementa completamente en una FPGA el algoritmo PCA. Los trabajos previos, como el mostrado [Fleury, 2004], derivaban parte de la ejecución de PCA a un PC o DSP. Además, se ha conseguido implementar todo el sistema desarrollado en esta tesis sobre una FPGA de bajo número de recursos internos.

- *Revisión de técnicas de Multiplicación de Matrices en hardware. Diseño de un array semi-sistólico para la Multiplicación de Matrices.*

  Dado que una de las operaciones más importantes en la técnica PCA es la multiplicación de matrices (son necesarias cuatro operaciones de multiplicación de matrices cada vez que se ejecuta PCA), se ha realizado una exhaustiva revisión de las diferentes alternativas para llevar a cabo la multiplicación de matrices y, así, poder realizar una propuesta susceptible de de ser implementada sobre FPGAs.

  Como las cuatro operaciones de multiplicación de matrices necesarias para la técnica PCA manejan diferentes tipos de matrices, el primer aspecto a resolver dentro de esta tesis fue la de determinar si era posible realizar estos cuatro tipos de multiplicaciones utilizando un único algoritmo, con el objetivo de simplificar el diseño y optimizar los recursos hardware, y sin que ello supusiera pérdida de velocidad de proceso. Por ello se evaluaron diferentes algoritmos de multiplicación de matrices y se implementaron tres de ellos, con el fin de verificar sus prestaciones tanto desde el punto de vista de velocidad de proceso como de recursos consumidos Tras analizar los tiempos de ejecución y los recursos consumidos se propuso una solución basada en un *array* semi-sistólico que permite realizar las cuatro operaciones de multiplicación con un mismo sistema.

  Además esta solución aprovecha al máximo el número de recursos disponibles en la FPGA empleada y consigue tiempos de cómputo que mejoran en un factor superior a 10^3 con respecto a otras soluciones como por ejemplo la propuesta por Strassen [Bravo, 2007].

  Tal y como se expuso en el capítulo 2, la multiplicación de matrices en FPGAs no es una operación que se implemente habitualmente sobre estos dispositivos debido al número de recursos consumidos, a la dependencia del tipo de matiz de entrada y a la dificultad del diseño e implementación. Por

tanto una notable contribución de esta tesis ha sido la construcción de un sistema de multiplicación de matrices universal.

La estructura modular del *array* semi-sistólico diseñado permite su portabilidad a diferentes operaciones de multiplicación de matrices, sin aumentar el número de recursos consumidos. Además, la regularidad y simetría de los elementos que forman el *array* permiten la expansión del *array* para así acelerar el tiempo de ejecución.

- *Cálculo de autovalores y autovectores.*

Una de las aportaciones más significativa de esta tesis la constituye la propuesta de una nueva alternativa para el cálculo, utilizando FPGAs, de autovalores y autovectores de matrices cuadradas y simétricas. Desde el punto de vista de recursos internos de la FPGA, su porcentaje de ocupación es notablemente inferior a las propuestas sistólicas implementadas previamente [Ahmedsaid, 2004b], y ello sin apenas penalizar el tiempo de ejecución. Esta disminución en el número de recursos es debida, en primer lugar, al manejo de la matriz triangular en vez de la matriz completa aprovechando la simetría de la matriz de entrada, y en segundo lugar, a la disminución del número de módulos CORDIC con respecto a otras alternativas previas.

A modo de ejemplo indicar que para un tamaño de la matriz de entrada de 8x8 ($M = 8$) donde cada uno de sus datos está codificado con 18 bits ($n = 18$), siendo M y n los tamaños patrones empleados en esta tesis, el número de recursos consumidos con respecto a la arquitectura sistólica optimizada propuesta en [Brent, 1985] es 5 veces inferior, siendo el incremento del tiempo de ejecución de sólo un 22%. La nueva propuesta desarrollada en esta tesis mejora en aproximadamente un 95% el número de recursos internos consumidos en la FPGA con respecto a la arquitectura sistólica de [Brent, 1985] implementada en FPGAs en [Ahmedsaid, 2004].

Con respecto al tiempo de ejecución de la propuesta realizada en esta tesis, su valor es como mínimo un 40% inferior al requerido por un PC de altas prestaciones en el momento de la realización de la presente tesis (PIV-2.66 GHz, 1GB RAM).

Otra importante contribución de la arquitectura propuesta es su flexibilidad a la hora de adaptarse a diferentes tamaños de matrices de entrada. A diferencia de las arquitecturas sistólicas, el número de recursos internos consumidos en la FPGA apenas varía constituyendo esto otra de las ventajas de esta nueva propuesta. Debido a su eficiente estructura interna, el uso de este módulo para el cálculo de autovalores y autovectores en cualquier otra aplicación queda justificado.

- *Diseño de un módulo CORDIC óptimo para el cálculo de autovalores y autovectores.*

Dado que el sistema de cálculo de autovalores y autovectores emplea el algoritmo CORDIC con coordenadas circulares para resolver operaciones trigonométricas, se ha realizado un exhaustivo estudio de las diferentes alternativas de implementación sobre una FPGA.

Con idea de disminuir al máximo el número de recursos internos necesarios, se ha diseñado un módulo CORDIC de propósito específico, el cual, a diferencia de los *cores* comerciales, no posee tanta flexibilidad y versatilidad pero ocupa menos recursos internos de la FPGA sin incrementar su tiempo de ejecución con respecto a los *cores* (objetivo primordial perseguido en esta tesis).

Para conseguir la máxima eficiencia posible del módulo diseñado, se han evaluado aspectos del modulo CORDIC relacionados con la exactitud en el resultado final y coste adicional de recursos que está asociado a cada situación desde diferentes puntos de vista: tamaño de datos ideal, número de iteraciones del CORDIC, y errores de truncamiento. Gracias a este análisis se ha permitido caracterizar el incremento en número de recursos que llevan asociados las diferentes opciones para mejorar la precisión.

- *Estudio de los truncamientos de datos internos de PCA.*

Debido a las numerosas operaciones aritméticas que se realizan en PCA (sumas, multiplicaciones de matrices, etc.) se ha realizado un profundo análisis de la influencia del tamaño óptimo de los diferentes datos intermedios que se manejan en PCA, evaluando la exactitud, recursos consumidos y tiempo de ejecución. Así, se ha buscado para cada fase intermedia de PCA (cálculo de la media, generación matriz covarianza, cálculo matriz autovectores ($\mathbf{U}_t$), cálculo imagen proyectada y recuperada) la importancia que cada truncamiento de datos aporta sobre el error de recuperación, llegando a la conclusión que la fase más crítica desde este punto de vista, es la asociada a la obtención de la matriz de covarianza. Esto implica que un tamaño óptimo en esta fase mejora el error de recuperación.

- *Construcción del mapa de distancias promediado.*

Con idea de obtener una umbralización óptima e identificar mejor el nuevo objeto, se ha propuesto la implementación de un sistema de promediado del mapa de distancias $\mathbf{MD}_{V2}$. Aunque el promediado es una de las operaciones que se pueden realizar de forma muy eficiente en una FPGA, el número de recursos consumidos así como el tiempo de ejecución aumentan exponencialmente a medida que aumenta el tamaño de la ventana. Por esta causa y debido a la limitación del número de recursos internos disponibles en la FPGA, se ha optado por una solución de compromiso empleando una máscara de 3x3 elementos.

El sistema de promediado para máscaras de 3x3 elementos empleado en esta tesis está basado en un convolucionador genérico de 3x3 elementos. Este convolucionador ha sido el que mejores resultados ha ofrecido tanto en el consumo de recursos internos como en el tiempo de ejecución. Su estructura interna ha sido escogida tras analizar e implementar 3 propuestas diferentes [Bravo, 2003].

- *Cálculo de un umbral dinámico en FPGA.*

El valor del umbral óptimo que debe aplicarse sobre el mapa de distancias promediado para determinar la aparición de nuevos objetos, debe ser dinámico para así adaptarse a diferentes escenas así como a distintas condiciones de iluminación.

Para obtener el valor del umbral óptimo en función de la información proporcionada por el mapa de distancias promediado ($\mathbf{MD}_{V2}$) se ha desarrollado una nueva alternativa para su implementación en una FPGA. Otras propuestas como [Vázquez, 2006] no funcionan correctamente debido principalmente al tamaño de los datos manejados en esta tesis (codificación en coma fija con 18 bits). La alternativa desarrollada en esta tesis se basa en la construcción de un histograma con los máximos de las columnas del $\mathbf{MD}_{V2}$. En este tipo de histogramas la presencia de un nuevo objeto, queda definida mediante la aparición de un valle en el histograma, es decir, una zona alejada del origen en la cual la repetición de valores es escasa, equivaliendo esto a la aparición de un nuevo objeto que proporciona un segundo pico con valores mayores a los proporcionados por el fondo. Este nuevo sistema de cálculo de umbral, proporciona una alta eficiencia en la búsqueda de nuevos objetos sobre una FPGA.

- *Actualización selectiva del modelo de fondo.*

En esta tesis se propone una nueva alternativa para la actualización selectiva del modelo de fondo basada en el análisis del valor medio de luminosidad de un número variable de ventanas en las que se divide la imagen actual, con respecto a una imagen con valor de luminosidad media máxima y otra con valor mínimo, obtenidas a partir de las M imágenes que forman el modelo de fondo.

Para determinar el tamaño óptimo de las ventanas así como el porcentaje de ventanas cuyos valores medios difieren de los del modelo de fondo, se ha realizado un profundo análisis de los mismos, llegando a la conclusión que a mayor número de ventanas analizadas dentro de la imagen actual, mejor eficiencia presenta el sistema desde el análisis de falsas o erróneas detecciones de objetos. Una gran ventaja de este sistema de actualización del modelo de fondo implementado en una FPGA es el reducido número de recursos internos consumidos. Además, su tiempo de ejecución no tiene una

repercusión significativa en el tiempo total de todo el sistema de detección de objetos ya que su actividad se solapa con otras tareas.

- *Diseño de una arquitectura de propósito específico para captar y almacenar imágenes.*

Otra de las contribuciones de la presente tesis ha sido el diseño e implementación de una plataforma para procesamiento de imágenes. La plataforma desarrollada constituye un sistema autónomo que permite:

  - Captura de imágenes de un sensor CMOS. Aunque la plataforma ha sido diseñada para el sensor MT9M413C36STM de Micron, el controlador VHDL diseñado puede ser reutilizado para otros sensores de la misma compañía sin más que variar algunas señales de control. De esta forma, el sistema desarrollado permite mucha versatilidad y flexibilidad.
  - Comunicación con Puerto Paralelo del PC. El sistema desarrollado está dotado de un canal de comunicación vía puerto paralelo con un PC. Mediante este canal se configuran parámetros de la captura de imagen además de ser de utilidad en fases de depuración del sistema desarrollado.
  - Almacenamiento en memoria externa. En la mayoría de las aplicaciones de visión artificial es necesario almacenar imágenes y datos temporales en memoria externa. Debido al tamaño de las imágenes es necesario hacer uso de la memoria externa en vez de la BRAM de una FPGA, ya que el tamaño de esta última suele ser inferior al tamaño de una imagen. Esta es la causa principal por la que se ha añadido a la plataforma un banco de memorias SDRAM de capacidad 128MB con un bus de datos de 64 bits. Debido a la similitud de las diferentes memorias SDRAM, el controlador diseñado en VHDL para la FPGA, permite su portabilidad a otros fabricantes y tipos de memorias SDRAM.

7.2. FUTURAS LÍNEAS DE TRABAJO.

A lo largo de esta tesis se ha descrito la implementación en una FPGA de la técnica PCA para la detección de nuevos objetos dentro de una escena captada por una cámara. A partir de los resultados alcanzados en esta tesis se plantean los siguientes trabajos futuros:

- *Implementación de nuevos algoritmos de visión sobre la plataforma desarrollada.* Debido a la gran modularidad alcanzada en esta tesis, la plataforma desarrollada posee una alta versatilidad lo que permite su uso en otros algoritmos de visión cuya portabilidad a una FPGA sea factible. Para ello habrá que paralelizar la ejecución del algoritmo de visión y conectarlo al resto de módulos que contiene la placa (gestión sensor CMOS, controlador de memoria externa y controlador global). De esta forma se logrará una alta velocidad de cómputo en comparación a la alcanzada en plataformas secuenciales.

- *Implementación de GPCA (Generalized Principal Component Analysis).* La variante de PCA expuesta en [Ye, 2004] trabaja con matrices en vez de vectores en el cálculo de autovectores. Además, esta alternativa maneja una segunda matriz de transformación. Estos condicionantes según [Ye, 2004] mejoran notablemente la precisión de los resultados finales así como el tiempo de ejecución en un PC. La ejecución del cálculo de autovalores en esta tesis mejora como mínimo en un 40% el tiempo requerido en un PC, por lo que la implementación de GPCA sobre una FPGA puede mejorar notablemente los resultados alcanzados con PCA.

- *Cambio de la memoria externa de la plataforma.* Una de las posibles soluciones que permiten el aumento del número de imágenes procesadas por segundo es el cambio de la memoria externa. En nuestro caso se han empleado memorias SDRAM las cuales están sobredimensionadas en cuanto a su capacidad, poseyendo un bus de datos total de 128 bits. Este ancho permite alcanzar unos ratios de procesamiento de imágenes muy buenos, los cuales se pueden mejorar.

  Dentro del sistema desarrollado en la FPGA, las tareas que consumen mayor tiempo son las multiplicaciones de matrices, debido al continuo acceso que se realiza a las memorias externas. El ancho de datos del bus actual (128 bits) obliga a insertar ciclos de espera en las multiplicaciones de matrices ya que el número de operandos está limitado al ancho del bus. Por tanto, para eliminar estos ciclos de espera y así poder acceder a más operandos en el mismo ciclo de reloj, el aumento del ancho del bus de datos permitiría acelerar las operaciones de multiplicaciones de matrices y por tanto disminuir el tiempo total del sistema.

- *Empleo de la plataforma desarrollada como co-procesador en algoritmos de visión.* En aquellas aplicaciones en las que sea inviable una implementación total de un algoritmo de visión sobre una FPGA, se puede emplear la plataforma desarrollada en esta tesis como elemento de procesamiento de ciertas partes del algoritmo cuya implementación en hardware consiga una velocidad superior a la conseguida en un microprocesador.

- *Identificación de los objetos detectados.* Otra línea de trabajo que se abre a partir de los resultados logrados, es el análisis de objetos detectados. Esto puede permitir realizar una clasificación de los diferentes nuevos objetos que aparecen en la escena.

- *Aplicación de la tesis realizada a una aplicación específica.* Los resultados logrados avalan su empleo en diferentes áreas en las que puede ser interesante la detección de nuevos objetos como pueden ser la seguridad de recintos o la seguridad ferroviaria.

8. REFERENCIAS

[Ahmedsaid, 2004] Ahmedsaid, A.; Amira, A.; Bouridane, A.; "Improved SVD systolic array and implementation on FPGA". White paper from www.celoxica.com . 2004

[Ahmedsaid, 2004b] Ahmedsaid, A.; Amira, A.; Bouridane, A.; "Accelerating MUSIC method on reconfigurable hardware for source localisation". Proceedings of the 2004 International Symposium on Circuits and Systems (ISCAS '04). vol. 3, pp:III - 369-372, 2004.

[Altera, 2006] Implementing Multipliers in FPGA Devices http://www.altera.com/products/devices/cyclone2/features/multipliers/cy2-multipliers.html

[Amira, 2000] Amira, A.; Bouridane, A.; Milligan, P.; Sage, P.; "A high Throughput FPGA Implementation of a Bit Level Matrix Product". Proceedings of IEEE Workshop on Signal Processing Systems Design and Implementation (SIPS 2000), vol. 1, pp: 396-399. 2000.

[Amira, 2001] A. Amira, A. Bouridane, P. Milligan. "Accelerating Matrix Product on Reconfigurable Hardware for Signal Processing". Proceedings of Field Programmable Logic and Applicacions (FPL), pp: 101-111, 2001

[Amira, 2002] Amira, A.; Bensaali, F.; "An FPGA based parameterisable system for matrix product implementation". Proceedings of IEEE Workshop on Signal Processing Systems, 2002 (SIPS '02), pp: 75 – 79. 2002.

[Andraka, 1998] Andraka R. "A survey of CORDIC algorithms for FPGA based computers". Proceedings of the sixth International symposium on Field Programmable Gate Array (FPGA'98), pp: 191-200, 1998.

[Antelo, 1995] Antelo, E; "Algoritmos y Arquitectura CORDIC en Aritmética Redundante para Procesamiento de Alta Velocidad". Tesis Doctoral. Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela. 1995

[Antelo, 1997] Antelo, E.; Brugera, J.D.; Lang, T.; Zapata, E.L.; "Error Analysis and Reduction for Angle calculation usign the CORDIC algorithm". IEEE Transactions on Computers, vol. 46, no. 1, pp: 1264-1271. 1997.

[Arias, 2001] Arias-Estrada, M.; Torres-Huitzil, C.; "Real time field programmable gate array architecture for computer vision". Journal of Electronic Imaging, vol. 10, pp: 289-296, 2001.

[Arias, 2001b] Arias-Estrada, M.; Rodríguez-Palacios, E.; "An FPGA co-processor for Real-time visual tracking". Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL 01), pp: 710 – 719, 2001.

[Arnold, 1992] Arnold, J. M.; Buell, D. A.; Davis, E. G.; "SPLASH 2".Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures, pp: 316-322, 1992.

[Battle, 2002] Battle, J.; Martí, J.; Ridao, P.; Amat, J.; "A new FPGA/DSP- Based Parallel architecture for real-time image processing". Journal Real-Time imaging, vol.8, pp: 345-356, 2002.

[Bensaali, 2003] Bensaali, F.; Amira, A.; Bouridane, A.; "An FPGA Based Coprocessor for Large Matrix Product Implementation". Proceedings of IEEE International Conference on Field-Programmable Technology (FPT'03), pp. 292-295. 2003

[Bobda, 2001] Bodda, C.; Steenbock, N.; "Singular value decomposition on distributed reconfigurable systems". Proceedings of the 12th International Workshop on Rapid System Prototyping, pp: 38 – 43, 2001.

[Boluda, 2000] Boluda, J.A.; "Arquitectura de procesamiento de imágenes basada en lógica reconfigurable para navegación de vehículos autónomos con visión foveal" Tesis Doctoral. Departamento de Informática de la Universidad de Valencia. 2000

[Boonkumklao, 2001] Boonkumklao, W.; Miyanaga, Y.; Dejhan, K.; "Flexible PCA architectura realized on FPGA". Proceedings of International Symposium on Communications and Information Technologies (ISCIT 2001), pp: 590-593, 2001.

[Boonkumklao, 2003] Boonkumklao, W.; Miyanaga, Y.; Dejhan, K.; "A flexible architecture for Digital Signal Processing". IEICE Transaction Informatics & Systems, vol. 10, pp: 2179-86, 2003.

[Bouman, 2007] Bouman, C.A. "Notes of Course of Digital Image Processing I (Spring 2007)" Purdue University. School of Electrical and Computer Engineering. 2007

[Bouridane, 1999] Bouridane, A.; Crookes, D.; Donachy, P.; Alotaibi, K.; Benkrid, K.; "A high level FPGA-based abstract machine for image processing". Journal of Systems Architecture, vol. 45, pp: 809-824, 1999.

[Bravo, 2003] Bravo, I.; Hernández, A.; Gardel, A.; Mateos, R..; Lázaro, J.L.; Díaz, V.; "Different proposals to the multiplication of 3x3 vision mask in VHDL for FPGAs". Proceedings IEEE Conference on Emerging Technologies and Factory Automation (ETFA), vol. 2, pp: 208-211. 2003.

[Bravo, 2004] Bravo, I.; Padilla, D.; Mazo, M.; Hernández, A.; Gardel, A.; Mateos, R.; Lázaro, J.L.; "Algoritmo de cálculo de autovalores y autovectores en Xilinx System Generator.". Actas de las IV Jornadas de Computación Reconfigurable y Aplicaciones (JCRA'04), pp: 415-422, 2004

[Bravo, 2005] Bravo, I.; Jiménez, P.; Hernández, A.; Gardel, A,; Mateos, R.; "Convolución de imágenes con máscaras de FPGAs". Actas del I Congreso Español de Informática (CEDI), V Jornadas de Computación Reconfigurable y Aplicaciones (JCRA05), 2005.

[Bravo, 2006] Bravo, I.; Jiménez, P.; Mazo, M.; Lázaro, J.L., Martín, E. "Architecture Based on FPGA's for Real-Time Image Processing". Lecture Notes in Computer Science, Reconfigurable Computing: Architectures and Applications, vol. 3985, pp: 153-157. ISSN: 0302-9743 Ed. Springer-Verlag, 2006.

[Bravo, 2006b] Bravo, I.; Jiménez, P.; Mazo, M.; Lázaro, J.L.; Gardel, A. "Implementation in FPGAs of Jacobi Method to solve the eigenvalue and eigenvector problem". Proceedings of the IEEE Conference on Field Programmable Logic and Applications 2006 (FPL 2006), pp: 729-732, 2006.

[Bravo, 2006c] Bravo, I.; Jiménez, P.; Mazo, M.; Lázaro, J.L.;de las Heras, J.J. "Algoritmos no sistólicos para la multiplicación de matrices en FPGAs". Proceedings de las VI Jornadas sobre Computación Reconfigurable y Aplicaciones (JCRA 2006), pp: 86-91. 2006.

[Bravo, 2006d] Bravo, I., Jiménez, P. "Análisis de errores en una arquitectura CORDIC orientada al cálculo de Autovalores y Autovectores". Proceedings de las VI Jornadas sobre Computación Reconfigurable y Aplicaciones (JCRA 2006), pp: 92-97, 2006.

[Bravo, 2007] Bravo, I.; Jiménez, P; Mazo, M.; Lázaro, J.L.; de las Heras, J.J.; Gardel, A. "Different proposals to matrix multiplication based on FPGAs" Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE), 2007.

[Brent, 1983] Brent, R. P.; Luk, F. T.; Van Loan, C. F.; "Computation of the generalized singular value decomposition using mesh-connected processors". Proceedings Real Time Signal Processing VI SPIE, vol. 431, pp: 66-71, 1983.

[Brent, 1985] Brent, R. P.; Luk, F. T.; "The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays". SIAM J. Scientific and Statistical Computing. vol. 6, pp: 69-84, 1985.

[Brent, 1985b] Brent, R. P.; Luk, F. T.; Van Loan, C. F.; "Computation of the singular value decomposition using mesh-connected processors". Journal of VLSI and Computer Systems no.1, vol. 3, pp:242-270, 1985.

[Buell, 1996] Buell, D. A.; Arnoll, J.M.; Kleinfelder, W.J.; "Splash 2, FPGAs in a Custom Computing Machine". Willey-IEEE Computer Society Press. ISBN: 0-8186-7413-X. 1996.

[Casselman, 1993] Casselman, S. "Virtual Computing and The Virtual Computer". Proceedings IEEE Workshop on FPGAs for Custom Computing Machines, pp: 43-48, 1993.

[Cattell, 1966] Cattell, R.B.; "The Scree Test for the Number of Factors"; Multivariate Behavioral Research, Nº 1, pp. 245 -276. 1966.

[Cavallaro, 1987] Cavallaro, J.R.; Luk, F.T.; "Architectures for a CORDIC SVD Processor". Proceedings Real Time Signal Processing IX SPIE, vol. 698, pp: 45-53, 1986.

[Cavallaro, 1988] Cavallaro, J.R.; Luk, F.T.; "CORDIC Arithmetic for an SVD Processor". Journal of Parallel and Distributed Computing, vol. 5, pp 271-290, 1998

[Cavallaro, 1989] Cavallaro, J.R.; Keleher, M.P.; Price, R.H.; Thomas, G.S. "VLSI implementation of a CORDIC SVD processor" Proceedings Eighth 12-14 University/Government/Industry Microelectronics Symposium, pp: 256 – 260, 1989.

[Chen, 1999] Chen, S.G.; Chang, C.; "A new efficient algorithm for singular value decomposition". Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '99), vol. 5, pp: 523 – 526, 1999.

[Cheney, 2001] Cheney, M.; "The Linear Sampling Method and the MUSIC Algorithm". Inverse Problems, vol. 17, pp: 591-595, 2001.

[Cheng, 2002] Cheng A., "Real-Time Systems: Scheduling, Analysis, and Verification" Publisher: Wiley-Interscience; (2002) ISBN: 0471184063

[Choi, 1996] Choi, J.; Dongarra, J.J.; Ostrouchov, L.S.; Petitet, A.P.; Walker, D.W.; Whaley, R.C.; "The design and implementation of the Scalapack LU, QR and Cholesky". Factorization Routines. Scientific Programming, vol. 5 pp. 173-184. 1996.

[Choi, 1997] Choi, S.; Yun, D.; Lee, H.; "Blind adaptive beamforming algorithms based on the extreme eigenvalue problem". Proceedings of the IEEE International Symposium on Antennas and Propagation Society, vol. 4, pp: 2418 – 2421. 1997.

[Choi, 2003] Choi, S.; Prassana, V.K.; "Time and Energy Efficient Matrix Factorization using FPGAs". Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays (FPGA 03), pp: 225 – 234, 2003.

[Cuenca, 2002] Cuenca, S.; "Propuesta de algoritmos y arquitecturas reconfigurables para sistemas de inspección visual automáticos de alto rendimiento." Tesis Doctoral. Universidad Miguel Hernández. 2002

[Dawood, 2002] Dawood, A.S.; Visser, S.J.; Willianms, J.A.; "Reconfigurable FPGAs for real time processing in Space". Proceedings of the 14th International Conference on Digital Signal Processing (DSP 2002), vol. 2, pp: 845 - 848, 2002.

[Díaz, 2006] Díaz, J. "Multimodal bio-inspired vision system. High performance motion and stereo processing architecture" Tesis doctoral. Departamento de Arquitectura y Tecnología de Computadores, Universidad de Granada. 2006.

[Draper, 2003] Draper, B.A.; Beveridge, J.R.; Willem Böhm, A.P.; Ross, C.; Chawathe, M.; "Accelerated Image Processing on FPGAs". IEEE Transactions on image processing, vol 12, Issue: 12, pp: 1543 – 1551, 2003.

[ElGindy, 2002] ElGindy, H.; Shue, E.; "On Sparse Matriz-vector Multiplication with FPGA-based system". Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp: 273-274, 2002.

[Ercegovac, 1990] Ercegovac M.D., Lang T. "Redundant and on-line CORDIC: Application to matrix triangularization and SVD". IEEE Transactions on computers, vol.39, no.6, pp: 725-740, 1990

[Fleury, 2004] Fleury, M.; Self, R.P.; Downton, A.C.; "Development of a fine-grained parallel Karhunen-Loeve transform". Journal of Parallel and Distributed Computing, vol. 64, pp: 520-535, 2004.

[Forsythe, 1960] Forsythe G.E., Henrici P. "The cyclic Jacobi method for computing the principal values of a complex matrix." Trans. Amer. Math. Soc. 94, pp: 1-23, 1960

[Francis, 1961] Francis J.G. "The QR Transformation, Parts I and II". The Computer Journal, No. 4, pp: 265–271 and 332–345, 1961–1962.

[García, 2005] García de Jalón, J. "Valores y vectores propios (métodos numéricos)". Apuntes de Métodos Matemáticos. http://mat21.etsii.upm.es/matesp/index.htm 2005.

[Giménez, 1996] Giménez, D.; Van de Geijn, R.; Hernández, V.; Vidal, A.M.; "Exploiting the Symmetry on the Jacobi Method on a Mesh of Processors". Proceedings 4th Euromicro Workshop on Parallel and Distributed Processing (PDP '96), pp: 377-384, 1996.

[Givens, 1953] Givens, J.W.; "A method of computing eigenvalues and eigenvectors suggested by classical results on symmetric matrices". National Bureau of Standards Applied Mathematics Series, 29, 117-122, 1953.

[Golub, 1989] Golub, G.H.; Van Loan, C.F.; "Matrix Computations", 2nd edn., The John Hopkins University Press, 1989.

[Gottumukkal, 2003] Gottumukkal, R.; Asari, V.K.; "System Level Design of Real Time Face Recognition Architecture Based on Composite PCA". Proceedings of the GLSVLSI 2003, pp: 157-160, 2003.

[Gottumukkal, 2004] Gottumukkal, R.; Asari, V.K.; "An improved face recognition technique based on modular PCA approach". Pattern Recognition Letters, vol. 25, pp. 429-436, 2004.

[Götze, 1993] Götze, J.; Paul, S.; "An efficient Jacobi-like Algorithm for Parallel Eigenvalue Computation". IEEE Transactions on Computers, vol. 42, no. 9, pp: 1058-1065, 1993.

[Govindarajan, 1998] Govindarajan, S.; Ouaiss, I.; Kaul, M.; Srinivasan, V.; Vemuri, R.; "An effective design system for dynamically reconfigurable architectures". Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM 98), pp: 312 – 313, 1998.

[Guccione, 1999] Guccione, S.; "List of FPGA-based Computing Machines". http://www.io.com/~guccione/HW_list.html 1999.

[Haiqing, 2000] Haiqing W.; Zhihuan S.; Ping L; "Improved PCA with optimized sensor locations for process monitoring and fault diagnosis". Proceedings of the 39th IEEE Conference on Decision and Control, Vol. 5, pp: 4353 – 4358, 2000.

[Hamid, 1994] Hamid, G.; "An FPGA-Bases Coprocessor for Image Processing". IEE Colloquium on Integrated Imaging Sensors and Processing, pp: 6/1 - 6/4, 1994.

[Hampson, 2002] Hampson G.A., Netherlands Foundation for Research in Astronomy. http://www.nfra.nl/~hampson/chap5-8.pdf 2002.

[Hemkumar, 1992] Hemkumar, N.D.; Cavallaro, J.R.; "A systolic VLSI architecture for complex SVD". Proceedings IEEE International Symposium on Circuits and Systems (ISCAS '92), vol. 3, pp: 1061 – 1064, 1992.

[Hernandez, 2004] Hernández, A.; Gardel, A.; Mateos, R.; Bravo, I.; Andrés, A.; "VHDL-Based Hough transform specification using CORDIC". Actas Semanario Anual de Automática e Instrumentación (SAAEI 04), ponencia: 261, 2004.

[Herron, 2003] Herron, T.J.; Reddy, K.M.; Garg, R.; Devanahalli, K.; "Eigen Decompositions of Covariance Matrices on a Fixed Point DSP". National Conference on Communication. Indian Institute of Technology, Madras, India. 2003

[IEEE1284, 2000] IEEE Standard Signalling Method for a Bidirectional Parallel Peripheral. Interfaz for Personal Computer. IEEE-SA Standards Board, 2000.

[Jagadish, 1989] Jagadish, H.V., Kailath, T.K., "A family of new efficient arrays for Matrix", IEEE Trans. on Comput., vol. 38, no. 1, pp 149-155, 1989.

[Jamro, 2001] Jamro, E.; Wiatr, K.; "Convolution Operation Implemented in FPGA Structures for Real-Time Image Processing". Proceedings of the 2nd International Symposium on Image and Signal Processing and Analysis (ISPA 2001), pp: 417-422, 2001.

[Jang, 2002] Jang, J.; Choi, S.; Prassana, V.K.; "Area and Time efficient implementations of Matrix Multiplication on FPGAs." Proceedings of the 2002 IEEE International Conference on Field-Programmable Technology, (FPT), pp: 93 – 100, 2002.

[Jang, 2002b] Jang, J.; Choi, S.; Prassana, V.K.; "Energy-Efficient Matrix Mulitplicacion on FPGAs". Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications (FPL 2002). pp: 534 – 544, 2002.

[Jianwen, 2004] Jianwen, L.; Ching Chuen, J.; "Partially Reconfigurable Matrix Multiplication for Area and Time efficiency on FPGAs". Proceedings of the EUROMICRO Systems on Digital System Design (DSD'04), pp: 204-208, 2004.

[Jiménez, 2005] Jimenez, J.A.; Mazo, M.; Urena, J.; Hernandez, A.; Alvarez, F.; Garcia, J.J.; Santiso, E.; "Using PCA in time-of-flight vectors for reflector recognition and 3-D localization". IEEE Transactions on Robotics. Vol.1, Issue 5, pp:909 – 924. 2005

[Jolliffe, 1972] Jolliffe, I. T.; "Discarding Variables in a Principal Component Analysis I: Artificial Data"; Applied Statistics, no. 21, pp.160-173, 1972.

[Jolliffe, 1973] Jolliffe, I. T.; "Discarding Variables in a Principal Component Analysis II: Real Data"; Applied Statistics, no. 22, pp.21-31, 1973.

[Kaiser, 1960] Kaiser, H. F.; "The application of electronic computers to factor analysis", Educational and Psychological Measurement, Nº 20, pp.141-151, 1960.

[Kesaal, 2003] Kessal, L.; Abel, N.; Demigny, D.; "Real-time image processing with dynamically reconfigurable architecture". Journal real-time imaging, vol. 9, pp: 297-313, 2003.

[Kim, 2002] Kim, M.; Ichigue, K.; Arai, H.; "Design of Jacobi EVD processor based on CORDIC for DOA estimation with MUSIC algorithm". Proceedings of the 13th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, vol. 1, pp: 120 – 124, 2002.

[Kim, 2003] Kim, M.; Ichige, K.; Arai, H.; "Implementation of FPGA based fast DOA estimator using unitary MUSIC algorithm". Proceedings of the IEEE 58th Vehicular Technology Conference (VTC 03), vol.1, pp: 213 – 217, 2003.

[Kittler, 1986] Kittler, J.; Illingworth, J.; "Minimum error thresholding"; Recognition, vol. 19, no. 1, pp: 41-47, 1986.

[Knittel, 1996] Knittel, G.; "A PCI-compatible FPGA-Coprocessor for 2D/3D Image Processing". Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, pp: 136 – 145, 1996.

[Kota, 1993] Kota K., Cavallaro J.R. "Numerical Accuracy and hardare tradeoffs for CORDIC arithmetic for Special-Purpose Processors". IEEE Transactions on Computers, vol. 42, no.7, pp: 769-779. 1993.

[Kung, 1979] Kung, H.T.; Leiserson, C.E.; "Systolic arrays (for VLSI)", Sparse Matrix Proc. Society for Industrial and Applied Mathematics, pp: 256-82, 1979.

[Kung, 1982] Kung, H.T.; "Why systolic architectures?" IEEE Computers, vol. 16, pp: 37-46, 1982.

[Lee, 1988] Lee, H.B.; Grondin, R.O.; "A comparison of Systolic Architectures for Matrix Multiplication". IEEE Journal of Solid-State Circuits, vol. 23, No. 1, pp 285-289, 1988.

[Lee, 1991] Lee, J-A; Lang, T.; "SVD by constant-factor-redundant-CORDIC". Proceedings 10th IEEE Symposium on Computer Arithmetic, pp: 264 – 271, 1991.

[Li, 1996] Li, J.; Du, Y.; Wang, J.; "Design a pocket multi bit multiplier in FPGA". Proceedings of the 2nd International Conference on ASIC, pp: 275 – 279, 1996.

[Lisa, 1998] Lisa, F.; "Configurable Computing For Real-Time Vision". Tesis Doctoral. Departamento de Informática de la Universidad Autónoma de Barcelona. 1998.

[Ma, 1999] Ma, J.; Parhi, K.K.; Deprettere, E.F.; "An algorithm transformation approach to CORDIC based parallel singular value decompositions architectures". Conference on Signals, Systems and Computers, Conference Record of the Thirty-Third Asilomar, vol. 2, pp:1401 – 1405, 1999.

[Manay, 2003] Manay, S; Yezzi, A.; "Anti-geometric diffusion fohresholding and fast segmentation"; IEEE Transactions on image processing, vol. 12, no. 11, 2003.

[Martín, 2005] Martín, J.L.; Zuloaga, A.; Cuadrado, C.; Lázaro, J.; Bidarte, U.; "Hardware implementation of optical flow constraint equation using FPGAs". Computer Vision and Image Understanding, vol. 48, issue 3, pp: 462-490, 2005.

[Mencer, 1998] Mencer, O.; Morf, M.; Flynn, M.; "PAM-Blox: High Performance FPGA Design for Adaptative Computing". Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, pp 167-174, 1998.

[Meribout, 2002] Meribout, M.; Nakanishi, M.; Ogura; T.; "Accurate and real-time image processing on a new PC-Compatible Board". Journal Real-Time imaging, vol. 8, pp: 35-51, 2002.

[Meribout, 2002b] Meribout, M.; Nakanishi, M.; Ogura; T.; "A parallel algorithm for real-time object recognition". Journal Pattern Recognition, vol. 35, pp: 1917-1931, 2002.

[Micron, 2004] Datasheets Synchronous DRAM Module MT8LSDT864A – 64MB, MT16LSDT1664A – 128MB (http://www.micron.com/products/modules/). 2004

[Micron, 2004b] Datasheets 1.3-Megapixel CMOS Active-Pixel digital image sensor MT9M413C36STM (http://www.micron.com/products/partdetail?part=MT9M413C36STM). 2004

[Minami, 1992] Minami, Y.; Iijima, N.; Mitsui, H.; Yoshida, Y.; Sone, M.; "High speed calculation of eigenvalue by digital signal processor" Proceedings of the 35th Midwest Symposium on Circuits and Systems, vol.2, pp:1136 – 1139. 1992.

[Moore, 1965] Moore, G.. "Cramming more components onto integrated circuits". Electronics, Volume 38, 1965.

[Morales, 2003] Morales-Sandoval, M.; Pérez-Gutiérrez, M.; Feregrino-Uribe, C.; Arias-Estrada, M.; "Arquitectura FPGA para un procesador matricial". ENC 2003, IV Congreso Internacional de Ciencias de la Computación, Avances en Ciencias de la Computación, pp: 91-96, 2003.

[Morris, 2003] Morris, K; "Low Cost FPGA". FPGA and Programmable Logic Journal (http://fpgajournal.com/articles/), 2003

[Nakagawa, 1979] Nakagawa, Y.; Rosenfeld, A.; "Some experiments on variable thresholding"; Pattern Recognition. vol. 11, no. 3, pp. 191-204, 1979.

[Otsu, 1979] Otsu, N; "A threshold selection method from gray level histogram"; IEEE Transaction of System, Man and Cybernetics, vol. SMC-8, pp. 62-66, 1979.

[Pan, 1984] Pan, V.; "How can we speed up matrix multiplications?" Society for Industrial and Applied Mathematics (SIAM), Vol. 26, No.3, pp 393 – 415, 1984.

[Pan, 1996] Pan, Y.; Hamdi, M.; "Singular value decomposition on processor arrays with a pipelined bus system". Journal of Network and Computer Applications, vol. 19, pp: 235 - 248, 1996.

[Parhi, 1999] Parhi, K.K.; "VLSI Digital Signal Processing Systems Design and Implementation". Ed. John Wiley & Sons, ISBN: 0-471-24186-5, 1999.

[Parlett, 2000] Parlett B.N.; "The QR Algorithm" Computing in Science & Engineering, vol. 2, Issue: 1 , pp:8 – 42, 2000.

[Paschalakis, 2003] Paschalakis, S.; Lee, P.; "Double precision floating-point arithmetic on FPGAs". Proceedings of IEEE International Conference on Field-Programmable Technology (FPT), pp: 352-358, 2003

[Piacentino, 1999] Piacentino, M.R.; van der Wal, G.S.; Hansen, M.W.; "Reconfigurable elements for a video pipeline processor". Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 99), pp: 82 – 91, 1999.

[Prasanna, 1991] Prasanna, V.K.; Tsai, Y.; "On Synthesizing Optimal Family of linear systolic arrays for Matrix Multiplication". IEEE Transactions of Computer, Vol 40, No. 6, pp: 770-774, 1991.

[Qiu, 2003] Qiu B., Prinet V., Perrier E., Monga O. "Multi-Block PCA Method for Image Change Detection". Proceedings of the 12th International Conference on Image Analysis and Processing (ICIAP'03), pp: 385-390, 2003.

[Quenot, 1994] Quenot, G.M.; Kraljic, I.C.; Serot, J.; Zavidovique, B.; "A reconfigurable compute engine for real-time vision automata prototyping". Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines (FCCM 94), pp: 91 – 100, 1994.

[Ratha, 1999] Ratha, N.K.; Jain, A.K.; "Computer Vision Algorithms on Reconfigurable Logic Arrays". IEEE Transactions on Parallel and Distributed Systems. Vol. 10, No. 1. pp: 29-43, 1999.

[Rutishauser, 1977] Rutishauser H. The Jacobi method for real symmetric matrices. In Handbook for Automatic Computation, Vol. 2 (C. Reinsch, ed.), pp: 202–211. Springer-Verlag. 1977

[Scrofano, 2002] Scrofano, R.; Choi, S.; Prassanna, V.K.; "Energy efficiency of FPGAs and Programmable Processors for Matrix Multiplication". Proceedings of the IEEE International Conference on Field-Programmable Technology, (FPT02), pp: 422-425, 2002.

[Shao, 2004] Shao-Yi, Chien; Yu-Wen, Huang; Bing-Yu, Hsieh; Shyh-Yih, Ma; Liang-Gee, Chen; "Fast video segmentation algorithm with shadow cancellation, global motion compensation and adaptive threshold techniques", IEEE Transactions on multimedia, vol. 6, no. 5, 2004.

[Sleijpen, 2000] Sleijpen G. L. G., Van der Vorst H.A. A Jacobi-Davidson Iteration Method for Linear Eigenvalue Problems. Society for Industrial and Applied Mathematics, vol. 42, number 2, 2000.

[Sweeney, 2001] Sweeney, C.; Blyth, B.; "RC-1000-PP Hardware Reference Manual". Celoxica Inc. Didcot, UK, 2001

[Swets, 1996] Swets, D.L.; Weng, J.J.; "Using discriminant eigenfeatures for image retrieval"; IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 18, No. 8, pp. 831-836, 1996.

[Talu, 2000] Talu, H.M.; Igci, E.; Tekin, M.E.; Sevtekin, H.S.; Genç, B.Ç, Heywood, M.I.; "Reconfigurable computing implementation of binary morphological operators using 4-, 6- and 8- connectivity". Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '00), vol. 6, pp: 3386 – 3389, 2000.

[Tessier, 1999]. Tessier, R.; Burleson, W.; "Reconfigurable Computing for Digital Signal Processing: A survey." Journal of VLSI Signal Processing 28, pp: 7-27, 1999.

[Thornton, 1999] Thornton, M.A.; Gaiche, J.D.; Lemieux, J.V.; "Tradeoff Analysis of Integer Multiplier Circuits Implemented in FPGAs". Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), pp. 301-304, 1999.

[Toledo, 2005] Toledo, A.; Navarro, P.; Soto, F.; Suardíaz, J.; Fernández, C.; "Experiences on developing computer vision hardware algorithms using Xilinx System Generator". Microprocessors and Microsystems, vol. 29, issues: 8-9, pp: 411-419. 2005

[Vázquez, 2004] Vázquez, J.F.; Mazo, M.; Lázaro, J.L.; Luna, C.A.; Ureña, J.; García, J.J.; Cabello, J.; Hierrezuelo, L.; "Detection of moving objects in railway using vision". Proceedings of 2004 Intelligent Vehicles Symposium, pp: 872-875, 2004

[Vázquez, 2006] Vázquez González, J.F. "Sistema de apoyo a la seguridad de vías férreas basado en técnicas de análisis de imágenes". Tesis Doctoral. Departamento de Electrónica. Universidad de Alcalá. 2006.

[Volder, 1959] Volder J.; "The CORDIC trigonometric computing technique. IRE Trans. Electronic Comput, EC-8, 3, pp: 330-344, 1959.

[Walther, 1971] Walther J.S. "A unified algorithm for elementary functions". AFIPS Spring Joint Computer Conf. 1971, pp 379-385. 1971

[Weisstein, 1999] Weisstein E. W.. "Singular Value." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/SingularValue.html 1999.

[Wiatr, 1998] Wiatr, K.; "Pipeline architecture of specialized Reconfigurable Processors in FPGA Structures for Real Time Image Pre-Processing". Proceedings of the 3rd International Conference on Signal Processing, vol. 1, pp: 131-138, 1996.

[Wikipedia, 2006] "Ordenamiento de la burbuja". http://es.wikipedia.org/wiki/Ordenamiento_de_burbuja 2006

[Wilkinson, 1999] Wilkinson, J.H.; "The algebraic eigenvalue problem". Oxford Science Publications, ISBN: 0198534183, 1999.

[Winograd, 1968] Winograd, S.; "A new algorithm for inner product". IEEE Trans. Comput., C-18, pp: 693-694, 1968.

[Xilinx, 2001] "Designing High-Performance Memories and Multipliers". Xcell Journal Q3-2001.

[Xilinx, 2004] "Virtex-4: Breakthrough Performance at the Lowest Cost" Xcell Journal: Issue 51. 2004

[Xilinx, 2005] "CORDIC 3.0. Product Specification (http://www.xilinx.com/bvdocs/ipcenter/data_sheet/cordic.pdf), 2005.

[Xilinx, 2007] Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet. (http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex_ii_pro_fpgas/index.htm) 2007

[XSG, 2003] Xilinx System Generator for DSP v3.1 2003.

[Yang, 2000] Yang, M.-H.; Ahuja, N.; Kriegman, D. "Face recognition using kernel eigenfaces". Proceedings 2000 International Conference on Image Processing, vol. 1, pp:37 – 40, 2000.

[Ye, 2004] Ye, J.; Janardan, R.; Li, Q. "GPCA: An efficient Dimension Reduction Scheme for Image Compression and Retrieval". Proceedings of the 10th International Conference on Knowledge Discovery and Data (KDD'04), pp: 354-363, 2004.

[Zhou, 1995] Zhou, B.B.; Brent, R. P.; Kahn, M. H.; "Efficient one-sided Jacobi algorithms for singular value decomposition and the symmetric eigenproblem". Proceedings of the IEEE First International Conference on Algorithms and Architectures for Parallel Processing (ICAAPP), pp: 256-262, 1995.

[Zhuo, 2004] Zhuo, L.; Prasanna, V.K.; "Scalable and modular algorithms for floating-point matrix multiplication on FPGA". Proceedings 18th International Parallel and Distributed Processing Symposium (IPDPS'04), pp: 92-102, 2004.

[Ziegler, 2002] Ziegler, H.; So Byoungro; Hall, M.; Diniz, P.C.; "Coarse-grain pipelining on multiple FPGA architectures". Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 02), pp: 77 – 86, 2002.