POSGRADO INTERINSTITUCIONAL EN CIENCIA Y TECNOLOGÍA

CENTRO DE INGENIERÍA Y DESARROLLO INDUSTRIAL

Desarrollo de un método de sustracción de fondo basado en ICA para una arquitectura FPGA-SoC

TESIS

QUE PARA OBTENER EL GRADO ACADÉMICO DE:

MAESTRO EN CIENCIA Y TECNOLOGÍA CON ESPECIALIDAD EN MECATRÓNICA

PRESENTA:

Fernando Carrizosa Corral.

ASESORES

Dr. Hugo Jiménez Hernández.

Dr. Jorge Alberto Soto Cajiga.

SANTIAGO DE QUERÉTARO, QUERÉTARO,
24 de abril de 2017

CIENCIA Y TECNOLOGÍA

Director de Posgrado
PICYT – CIDESI
Querétaro

Los abajo firmantes, miembros del Comité Tutorial del alumno **Fernando Carrizosa Corral**, una vez leída y revisada la Tesis titulada **"Desarrollo de un método de sustracción de fondo basado en ICA para una arquitectura FPGA-SoC"**, aceptamos que la referida tesis revisada y corregida sea presentada por el alumno para aspirar al grado de **Maestro en Ciencia y Tecnología** en la opción terminal de **Mecatrónica** durante el Examen de Grado correspondiente.

Y para que así conste firmo la presente a los 25 días del mes de abril del año dos mil diecisiete.

Tutor Académico

Hugo Jiménez Hdz

Dr. Hugo Jiménez Hernández

Director de Posgrado
PICYT – CIDESI
Querétaro

Los abajo firmantes, miembros del Jurado del Examen de Grado del alumno **Fernando Carrizosa Corral**, una vez leída y revisada la Tesis titulada **"Desarrollo de un método de sustracción de fondo basado en ICA para una arquitectura FPGA-SoC"**, aceptamos que la referida tesis revisada y corregida sea presentada por el alumno para aspirar al grado de **Maestro en Ciencia y Tecnología** en la opción terminal de **Mecatrónica** durante el Examen de Grado correspondiente.

Y para que así conste firmamos la presente a los 25 días del mes de abril del año dos mil diecisiete.

Presidente

Secretario

Vocal

Vocal

Vocal

Resumen

Esta tesis presenta el desarrollo de un método para la detección de movimiento basado en el Análisis de Componentes Independientes (ICA por sus siglas en inglés) y su implementación en un Field-Programmable Gate Array (FPGA) System-on-a-Chip (SoC) con procesador embebido. El método de sustracción de fondo basado en ICA del que partió este trabajo presentó mejores resultados que Mezcla de Gaussianas (MOG) cuando el fondo es dinámico. Sin embargo, ese enfoque se desarrolló para una computadora y la manera en la que se emplea ICA es demasiado compleja, requiriendo una computadora de alto rendimiento para su implementación. Con el propósito de que este algoritmo pueda ser aplicado en los sistemas embebidos de visión actuales, se desarrolla una versión que explota la arquitectura del FPGA-SoC con procesador embebido. Ésta es una tecnología reciente que complementa el paralelismo de los FPGA con la capacidad de cómputo de propósito general de los procesadores. En este trabajo, el fondo se actualiza continuamente estimando la intensidad media de cada pixel mediante Esperanza-Maximización (EM), el primer plano se obtiene utilizando FastICA y el movimiento se determina con un umbral basado en desviación estándar. De esta investigación se obtuvo una versión del método de sustracción de fondo basado en ICA adecuada para una arquitectura FPGA-SoC, así como su implementación. Los resultados obtenidos de la implementación en el FPGA-SoC muestran un aumento del 3353 % en la cantidad de cuadros por segundo respecto a una implementación utilizando únicamente el procesador embebido.

Abstract

This thesis presents the development of a background subtraction method based on Independent Component Analysis (ICA) and its implementation on a Field-Programmable Gate Array (FPGA) System-on-a-Chip (SoC) with an embedded processor. The previous ICA-based background subtraction approach, from which this work derived, achieved better results than Mixture of Gaussians (MOG) when the background is dynamic. Nevertheless, that approach was developed for a computer and the mean of using ICA is too complex, requiring a high-end computer. In order to extend the use of this approach to current embedded vision systems, a method that exploits the architecture of an FPGA-SoC with embedded processor is developed. This recent technology complements the parallelism of FPGA with the general purpose computing of processors. In the present work, background is continually updated by estimating the average pixel intensity via Expectation-Maximization (EM), the foreground is retrieved by using FastICA, and movement is selected with a threshold based on standard deviation. This research achieved a version of the ICA-based background subtraction method appropriate for an FPGA-SoC architecture, as well as its implementation. The results obtained from the FPGA-SoC implementation show an increase of 3353 % in the frame rate in comparison with an implementation using only its embedded processor.

Dedicatoria

A mi familia y amigos que constantemente me han apoyado sin importar la distancia.

En particular a mi madre que siempre me motivó a cumplir mis metas.

Índice general

1. Fundamento teórico. **9**

1.1. Detección de movimiento . . . 9

1.2. Análisis de componentes independientes . . . 11

1.3. Arquitecturas de hardware . . . 13

1.4. Paralelismo . . . 16

2. Propuesta. **18**

2.1. Desarrollo del método de sustracción de fondo . . . 18

2.1.1. Modelo de fondo . . . 18

2.1.2. Extracción del primer plano . . . 19

2.1.3. Detección de movimiento . . . 21

2.2. Implementación del método . . . 22

3. Modelo experimental y resultados. **25**

3.1. Diseño experimental . . . 25

3.2. Resultados . . . 26

3.3. Discusión de resultados . . . 29

4. Conclusiones. **36**

A. Información adicional sobre la implementación. **42**

A.1. Pipeline de FastICA . . . 42

A.2. Comunicación FPGA-procesador . . . 46

A.3. Uso de recursos . . . 53

Índice de figuras

1.1. Ejemplo de señales estadísticamente (a) independientes y (b) dependientes. . 12
1.2. Modelo básico de ICA. 12
1.3. Paralelismo: (a) en un caso trivial y (b) en un caso no trivial. 16
1.4. Paralelismo en pipeline. 17
1.5. Paralelismo al procesar imágenes. 17

2.1. Diagrama a bloques del método. 18
2.2. Diagrama de flujo de la implementación propuesta. 23

3.1. Resultados obtenidos de procesar la secuencia 1 en la simulación en Matlab. 27
3.2. Resultados obtenidos de procesar la secuencia 1 en el FPGA-SoC. 28
3.3. Resultados obtenidos de procesar la secuencia 2 en la simulación en Matlab. 29
3.4. Resultados obtenidos de procesar la secuencia 2 en el FPGA-SoC. 30
3.5. Histogramas de la precisión (P) obtenida al procesar la secuencia 1: (a) en la simulación en Matlab y (b) en el FPGA-SoC. 32
3.6. Histogramas de la sensibilidad (R) obtenida al procesar la secuencia 1: (a) en la simulación en Matlab y (b) en el FPGA-SoC. 32
3.7. Histogramas de la precisión (P) obtenida al procesar la secuencia 2: (a) en la simulación en Matlab y (b) en el FPGA-SoC. 33
3.8. Histogramas de la sensibilidad (R) obtenida al procesar la secuencia 2: (a) en la simulación en Matlab y (b) en el FPGA-SoC. 33
3.9. Valores de precisión (P) obtenidos a lo largo de la secuencia 1: (a) en la simulación en Matlab y (b) en el FPGA-SoC. 34

3.10. Valores de sensibilidad (R) obtenidos a lo largo de la secuencia 1: (a) en la simulación en Matlab y (b) en el FPGA-SoC. 34
3.11. Valores de precisión (P) obtenidos a lo largo de la secuencia 2: (a) en la simulación en Matlab y (b) en el FPGA-SoC. 35
3.12. Valores de sensibilidad (R) obtenidos a lo largo de la secuencia 2: (a) en la simulación en Matlab y (b) en el FPGA-SoC. 35

A.1. Diagrama de flujo del pipeline de FastICA. 44
A.2. Diagrama del punto fijo en el pipeline de FastICA. 45

Índice de tablas

1.1. Comparación entre arquitecturas. 14

2.1. Especificaciones técnicas del hardware. 24

3.1. Precisión y sensibilidad obtenidas de procesar la secuencia 1 en la simulación en Matlab. 27

3.2. Precisión y sensibilidad obtenidas de procesar la secuencia 1 en el FPGA-SoC. 28

3.3. Precisión y sensibilidad obtenidas de procesar la secuencia 2 en la simulación en Matlab. 29

3.4. Precisión y sensibilidad obtenidas de procesar la secuencia 2 en el FPGA-SoC. 30

3.5. Tiempo de ejecución promedio por imagen procesada. 31

3.6. Tiempo de ejecución promedio por iteración de FastICA. 31

A.1. Tiempos de ejecución por operación en el pipeline de FastICA. 43

A.2. Canales de comunicación utilizados. 46

A.3. Recursos de hardware del FPGA utilizados en la implementación. 53

Lista de acrónimos

BRIEF - Binary Robust Independent Elementary Features

CPU - Central Processing Unit

DSP - Digital Signal Processor

EM - Expectation-Maximization

FAST - Features from Accelerated Segment Test

FPGA - Field Programmable Gate Array

fps - frames per second

GPU - Graphics Processing Unit

HOG - Histogram of Oriented Gradients

ICA - Independent Component Analysis

IMU - Inertial Measurement Unit

KDE - Kernel Density Estimation

LUT - Look Up Table

MOG - Mixture of Gaussians

PCA - Principal Component Analysis

pdf - Probability Density Function

SLAM - Simultaneous Localization and Mapping

SoC - System-on-a-Chip

SVD - Singular Value Decomposition

UAV - Unmanned Aerial Vehicle

VLSI – Very Large Scale Integration

Introducción.

La detección de movimiento es una de las tareas más investigadas en el área de visión artificial debido a que es un paso crítico para muchos procesos complejos, como clasificación de objetos y monitoreo de actividades [1]. Algunos ejemplos de aplicaciones para esta tarea son vigilancia, análisis de movimiento en humanos, navegación de robots, detección de eventos, detección de anomalías, conferencias de video, análisis de tráfico y compresión de video. El resultado de estas tareas se ve afectado en gran medida por la precisión y robustez del método de detección de movimiento utilizado. Actualmente los investigadores se enfocan en reducir los errores causados por movimientos periódicos en la escena, variaciones en la iluminación, sombras, camuflaje, ruido en la cámara, entre otros [1]. Al necesitar sobrellevar más de estos problemas, los métodos de detección de movimiento se vuelven computacionalmente más demandantes y requieren hardware de alta capacidad computacional.

Uno de los trabajos dedicados a reducir los errores en la detección de movimiento presentó un método de sustracción de fondo basado en ICA [2]. Este método se mostró capaz de detectar el movimiento en interiores y exteriores; también resultó robusto a cambios en la iluminación y a las sombras. Además, en las pruebas mostró una mayor precisión que el método MOG, el cual es uno de los más utilizados. No obstante, la alta complejidad de la manera en que se utilizó ICA sobre imágenes dificulta su implementación en un sistema embebido.

Para seleccionar un método de detección de movimiento, un diseñador debe considerar los requerimientos de memoria, la velocidad y la precisión adecuada para la aplicación [3]. Uno de los dispositivos frecuentemente utilizados en la literatura para realizar aplicaciones de visión artificial es el FPGA, debido a su alta capacidad para paralelizar operaciones a nivel pixel a un bajo costo energético. Sin embargo, es común que estas aplicaciones también se

beneficien de técnicas de programación de alto nivel, las cuales son más sencillas de utilizar en los procesadores. Necesidades similares a ésta han impulsado uno de los recientes avances tecnológicos en el área de sistemas embebidos; la comercialización de SoC que integran uno o múltiples procesadores embebidos con un FPGA, referidos en el resto de este documento simplemente como FPGA-SoC.

Los estudios presentados por Fykse [4] y Struyf *et al.* [5] comparan el uso de FPGA y GPU para aplicaciones de visión artificial. Ambos estudios llegan a conclusiones similares: la implementación en el GPU conlleva menor tiempo de desarrollo y tiene un rendimiento ligeramente mayor al del FPGA. Aunque el FPGA tiene menor consumo eléctrico y tamaño en comparación al GPU.

Rodríguez-Andina *et al.* [6] presentaron una revisión sobre los últimos avances en FPGA que se espera tengan un alto impacto en los sistemas digitales para la industria. Uno de ellos es el uso de procesadores embebidos en SoC basados en FPGA, particularmente los diseñados por Altera y Xilinx que incluyen procesadores de la familia Cortex-A. Una de las aplicaciones para la que consideran que esta tecnología tendrá un alto impacto es en los sistemas de visión. Además, destacan para el caso particular de los procesadores embebidos que, aunque muchas aplicaciones se podrían ver beneficiadas, aún son muy pocos los trabajos que reportan el uso de estas nuevas tecnologías.

Sobre esta misma tecnología, Eberli [7] analizó algunas de sus aplicaciones y concluyó que ayudarán a resolver algunos problemas de sistemas embebidos que no eran factibles hace algunos años. De acuerdo a su análisis, la ventaja principal que tienen los sistemas que usan esta tecnología es que se puede desarrollar código para control y a su vez utilizar la lógica del FPGA para acelerar los procesamientos numéricos. A un algoritmo que se implemente en hardware (FPGA) le llevará menos tiempo obtener resultados que si se implementa sólo en software (CPU). Aunque algunas operaciones o tipo de dato incrementan la complejidad de diseñar el hardware, por lo que se suele buscar un balance entre hardware y software [8]. Por lo tanto, utilizar estos FPGA-SoC puede facilitar el desarrollo de sistemas embebidos de visión utilizando métodos que combinen adecuadamente los procesadores y el paralelismo de los FPGA.

A continuación se describen brevemente algunos de los trabajos relevantes que tratan estas diferentes tecnologías:

- Van, Wu y Chen [9] implementaron ICA en un FPGA para separar señales en un electroencefalograma de ocho canales. Buscaron la eficiencia en términos de consumo energético y lograron una disipación de poder de 16.35mW a 100MHz a 1V.
- Wang y Chen [10] propusieron un diseño de hardware para sustracción de fondo basado en un FPGA. Su diseño paralelizó los cálculos computacionales y eliminó las operaciones de división. Se realizó una simulación con videos de 640x480 y se obtuvieron 51 fps.
- Heo *et al.* [11] desarrollaron un esquema para detección de objetos y lo implementaron en un FPGA con un procesador ARM embebido. Implementaron FAST, BRIEF y distancia de Hamming.
- May y KrougJicof [12] presentaron un método para detectar y evadir obstáculos en UAV y lo implementaron en un FPGA. Sólo se realizaron pruebas con grabaciones de vuelo. No obstante, remarcaron que una implementación en FPGA es apropiada para los UAV por sus requerimientos de tamaño, peso y poder.
- Schmid y Hirschmuller [13] presentaron un sistema para estimar la orientación de un robot móvil combinando visión estéreo y un IMU. El sistema integra varios procesadores y un FPGA. El peso del sistema es adecuado para algunos dispositivos móviles y UAVs, aunque podría mejorarse utilizando un FPGA con procesador embebido.
- Ranjith y Muniraj [14] implementaron ICA en un FPGA para procesar señales de electroencefalograma. Utilizaron la modularidad, jerarquía y paralelismo de una implementación en VLSI para reducir la complejidad del algoritmo y aumentar la velocidad de procesamiento.
- Nikolic *et al.* [15] diseñaron un sensor de SLAM para aplicaciones robóticas utilizando un ARM-FPGA y un IMU. Implementaron Harris y FAST en el FPGA mientras que los algoritmos para realizar SLAM se ejecutaron en el procesador ARM.

- Birem y Berry [16] desarrollaron una cámara inteligente utilizando un FPGA. Utilizaron un diseño modular que transmite las características encontradas mediante Harris y Stephen logrando un sensor versátil.
- Zhou *et al.* [17] elaboraron un esquema para realizar odometría en un micro-UAV utilizando dos cámaras y un IMU y lo implementaron en un FPGA con un procesador ARM embebido. Implementaron FAST como extractor de características y BRIEF como descriptor.
- Szabo y Gontean [18] presentaron un método para detectar la posición de un brazo robótico y lo implementaron en un FPGA.
- Tabkhi, Sabbagh y Schirner [19] implementaron MOG en un FPGA-SoC con procesador ARM embebido, logrando un consumo energético 600 veces menor a una solución basada únicamente en procesadores ARM.
- Han y Oruklu [8] desarrollaron un sistema en un FPGA-SoC con procesador ARM embebido que detecta señales de tráfico en una imagen y las envía a una PC.
- Hsiao *et al.* [20] implementaron un módulo para un sistema de detección de humanos en un FPGA utilizando HOG. Su implementación logró 15 fps, suficiente para considerarlo un sistema en tiempo real.
- Calvo-Gallego, Sánchez-Solano y Jiménez [21] implementaron un método de sustracción de fondo basado en lógica difusa en un FPGA y lo encapsularon en un núcleo de propiedad intelectual. El núcleo se mostró adecuado para procesar video de 640x480 a 60 fps.
- Nikitakis *et al.* [22] implementaron un método de sustracción de fondo para sistemas de visión distribuidos en diferentes FPGAs y procesadores. Los resultados mostraron que los FPGA logran un menor consumo energético por pixel y un mejor desempeño en fps. Por lo que los FPGA son una alternativa para evitar los cuellos de botella de los sistemas centralizados.

Explotando los beneficios del cómputo paralelo y serial, los FPGA-SoC también podrían permitir utilizar en sistemas embebidos algunos métodos de mayor precisión que actualmente sólo son apropiados para computadoras de alto rendimiento. Tal como el método basado en ICA presentado por Jiménez-Hernández [2] que logró mayor precisión que MOG. Sin embargo, no es apropiado para un sistema embebido, no explotó paralelismo y la velocidad resultante es muy lenta para la mayoría de las aplicaciones. En este trabajo se propone explotar la arquitectura de los FPGA-SoC para desarrollar e implementar un método de sustracción de fondo basado en ICA apropiado para un sistema embebido. Esto se logró principalmente examinando algoritmos para encontrar oportunidades de paralelismo y distribuyendo las operaciones requeridas entre el FPGA y el procesador, usando el procesador como una unidad de control y el FPGA como un acelerador de hardware.

Objetivos

Objetivo general

Desarrollar una versión del método de sustracción de fondo basado en ICA que utilice el paralelismo de una arquitectura FPGA-SoC con procesador embebido.

Objetivos específicos:

- Generar una versión del método actual basado en ICA con requerimientos adecuados para un sistema embebido.
- Examinar los algoritmos y paralelizar para una implementación en FPGA.
- Distribuir las operaciones del método entre el FPGA y el procesador.
- Implementar el método utilizando únicamente el procesador del FPGA-SoC.
- Implementar utilizando tanto el FPGA como el procesador.
- Establecer un diseño experimental para probar el método.

Hipótesis

Si se desarrolla una versión del método de sustracción de fondo basado en ICA que utilice el paralelismo de la arquitectura del FPGA-SoC con procesador embebido, entonces su implementación logrará una mayor cadencia de imágenes respecto a una que sólo utilice una ejecución secuencial en el mismo procesador.

Metodología

La metodología creada para desarrollar e implementar un método en una arquitectura FPGA-SoC se resume en el diagrama mostrado en la Figura I.1. Posteriormente se detalla para el caso particular tratado en esta tesis.

Primero se realizó un programa utilizando el enfoque propuesto por Jiménez-Hernández [2] para una PC en Matlab, utilizando funciones disponibles en sus librerías de software que incluyen una implementación de la clásica formulación de FastICA. Posteriormente, se realizaron los cambios propuestos para el modelo de fondo, extracción del primer plano y detección de movimiento a forma de validar la propuesta de reducir la entrada de ICA a dos componentes para lograr que el método sea compatible con la escasa memoria disponible en el FPGA. Después, se trabajó en sustituir las funciones obtenidas de las librerías, analizando los algoritmos para desarrollar funciones propias específicamente pensadas para la arquitectura del FPGA. La principal aportación de esta tarea fue lograr reformular la clásica expresión de FastICA, de manera que la implementación en FPGA se puede realizar directamente utilizando pipelines. El siguiente paso consistió en identificar las operaciones que eran más apropiadas para el FPGA y desarrollar el código en Matlab evitando operadores que no son triviales de representar en VHDL, *e.g.*, las variables matriciales o vectoriales de Matlab. Esto facilita desarrollar el código en VHDL ya que el procedimiento se puede depurar en Matlab, donde las herramientas son más sencillas de utilizar.

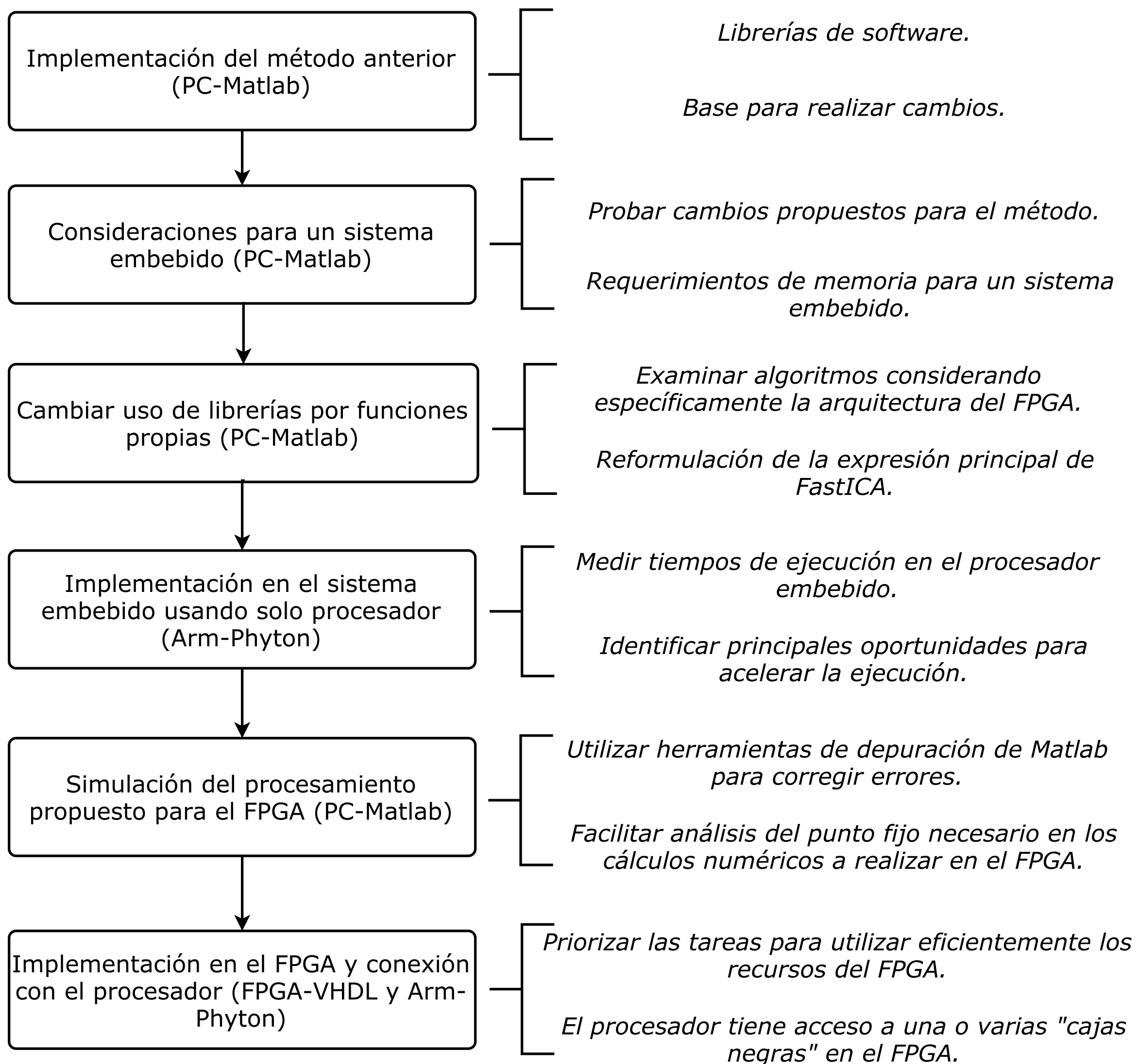


Figura I.1 Metodología.

Alcances

Los alcances considerados para este proyecto son los siguientes:

- La implementación en computadora del método actual se realizará sin buscar eficiencia en el manejo de recursos, velocidad ni precisión.

- El método que se desarrollará no necesariamente contendrá todas las etapas del método actual, de ser necesario para lograr la implementación en el FPGA-SoC se podrán modificar, sustituir o eliminar algunas etapas (mientras se mantenga a ICA como base).

- El método no será completamente paralelo, sólo se paralelizarán las operaciones para las que se considere tendrán mayor beneficio y para las que se encuentre una manera apropiada de hacerlo.

- De ser necesario se podrán incluir otros dispositivos en la implementación como memorias, interfaces para cámara o convertidores de video (como DVI-HDMI).

- Se intentará reducir el número de PCBs a fabricar y de controladores a programar, en lo posible se utilizarán soluciones abiertas o comerciales para disminuir el tiempo empleado en esta etapa y poder enfocarse en desarrollar el método.

- La implementación se realizará sólo para el modelo de FPGA-SoC que se seleccione.

Capítulo 1

Fundamento teórico.

1.1. Detección de movimiento

La detección de movimiento consiste en la tarea de identificar el movimiento físico de un objeto dentro de una determinada región o área. Los enfoques tradicionales para realizar esta tarea se pueden categorizar en tres grupos [23]:

- *Sustracción de fondo.* Este enfoque detecta regiones en movimiento calculando la diferencia a nivel pixel entre el cuadro (imagen) actual y un fondo de referencia. En su versión más simple, se ve afectado principalmente cuando el fondo es dinámico, hay cambios de iluminación o existen sombras. Éste se considera uno de los más confiables y se han desarrollado numerosos métodos para intentar sobrellevar estos problemas.

- *Diferenciación de cuadros.* Considera la diferencia entre dos cuadros consecutivos para detectar el objeto en movimiento. Generalmente, este enfoque no puede obtener todo el contorno del objeto por lo que se realizan otras operaciones antes de tener un resultado usable.

- *Flujo óptico.* Utiliza vectores para detectar regiones en movimiento aun cuando la cámara no está fija. Con este enfoque es posible obtener un conocimiento completo sobre el movimiento del objeto. No obstante, tiene una elevada complejidad computacional y es muy sensible al ruido.

Algunos de los métodos más comunes que utilizan el enfoque de sustracción de fondo son los siguientes [3]:

- *Promedio gaussiano continuo.* Pretende modelar cada pixel del fondo de manera independiente utilizando una pdf. Con cada nuevo cuadro se calcula el promedio y la desviación estándar de la pdf utilizando una media ponderada entre los valores de la

imagen actual y de la pdf anterior. El movimiento se detecta calculando la diferencia entre el promedio de la pdf de cada pixel y el valor actual del pixel.

- *Filtro mediano temporal.* Se diferencia del promedio gaussiano continuo ya que, en lugar de usar una media ponderada, utiliza una mediana de cada pixel de las imágenes más recientes para calcular las nuevas pdfs. Su desventaja es que requiere mantener en memoria las imágenes anteriores más recientes.

- *Mezcla de gaussianos (MOG).* Este método permite la detección en situaciones donde es posible que diferentes objetos formen parte del fondo en diferentes instantes, por ejemplo nieve, lluvia, olas o árboles cuando hay vientos fuertes. Calcula la probabilidad de observar cierto valor en un pixel utilizando entre 3 y 5 distribuciones gaussianas. Cada una de las distribuciones representa a un objeto del fondo o en movimiento. Para distinguir a los objetos, se supone que los objetos del fondo son los que presentan las distribuciones más altas y compactas.

- *Estimación de densidad de kernel (KDE).* Calcula la pdf del fondo utilizando los fondos detectados más recientemente, generalmente se usan alrededor de 100. Cada uno genera un kernel gaussiano y se asume que todos tienen la misma varianza. Una de sus desventajas es que requiere almacenar en memoria la cantidad de fondos a utilizar.

De estos métodos, promedio gaussiano continuo y filtro mediano temporal pueden tener una precisión aceptable con bajos requerimientos de memoria. MOG y KDE tienen una muy buena precisión. Por otro lado, KDE tiene requerimientos de memoria muy elevados.
La evaluación del desempeño de los métodos de sustracción de fondo generalmente se realiza considerando la aplicación para la que se destinan e involucran un grupo de observadores humanos. Existen dos tipos de evaluación dependiendo de si se tiene o no el resultado esperado: la independiente y la relativa. La relativa se utiliza con el resultado esperado por tanto suele dar resultados más confiables. Uno de los enfoques relativos es el basado en pixel que utiliza cuatro cantidades para formar una métrica: verdaderos positivos, falsos positivos, verdaderos negativos y falsos negativos [24].

Uno de los métodos recientes basados en sustracción de fondo que mostró mejor precisión que MOG, siendo capaz de detectar el movimiento tanto en interiores como en exteriores y robusto a cambios en la iluminación y a las sombras, se basa en ICA [2]. ICA es un método que toma un conjunto de datos multivariable y obtiene un conjunto de componentes lineal y estadísticamente independientes, es decir que la función de probabilidad de cada componente no se ve influida por el resto de las componentes. El método se introdujo por primera vez en los ochenta [25] y es una solución al clásico problema de la fiesta de cocteles, un caso de separación ciega de señales; donde se han mezclado varias componentes (señales) y se busca recuperarlas. En el problema de detección de fondo, estas componentes representan las pdfs del fondo, los objetos en movimiento y el ruido que se encuentran mezclados en las imágenes capturadas.

1.2. Análisis de componentes independientes

El modelo básico de ICA [26] supone que se tienen N señales lineal y estadísticamente independientes (véase Figura 1.1) $\mathbf{s}_i$ $(i = 1, \ldots, N)$ y que las señales no se pueden observar directamente. Usando N sensores se obtienen N conjuntos de observaciones $\mathbf{x}_i$ $(i = 1, \ldots, N)$ que corresponden a diferentes mezclas de las señales fuente. Este modelo se expresa como la siguiente multiplicación de matrices:

$$X = AS, \tag{1.1}$$

donde A es una matriz desconocida llamada matriz de mezcla y X y S son las representaciones matriciales de los vectores $\mathbf{x}_i$ y $\mathbf{s}_i$, respectivamente.
Conociendo únicamente las observaciones, es posible calcular un estimado, $\tilde{\mathbf{s}}_i$, de las señales fuente obteniendo la matriz de separación W, donde $W = A^{-1}$, de la siguiente manera:

$$\tilde{S} = WX, \tag{1.2}$$

donde $\tilde{S}$ es la representación matricial de la estimación de las componentes independientes $\tilde{\mathbf{s}}_i$. En ICA esta matriz de separación se obtiene buscando que las componentes de $\tilde{\mathbf{s}}_i$ sean estadísticamente independientes. Las propiedades que se utilizan para cuantificar la

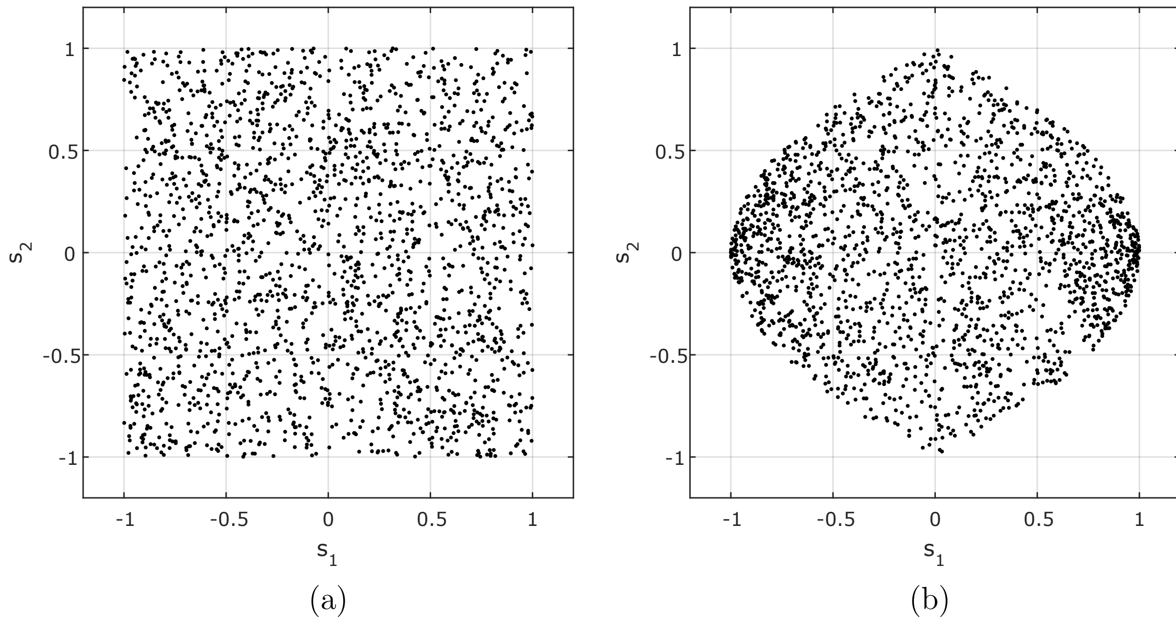


Figura 1.1: Ejemplo de señales estadísticamente (a) independientes y (b) dependientes.

independencia estadística de las componentes no están especificadas en ICA, algunas de las más comunes son curtosis y entropía [26]. Un esquema del modelo básico se muestra en la Figura 1.2.

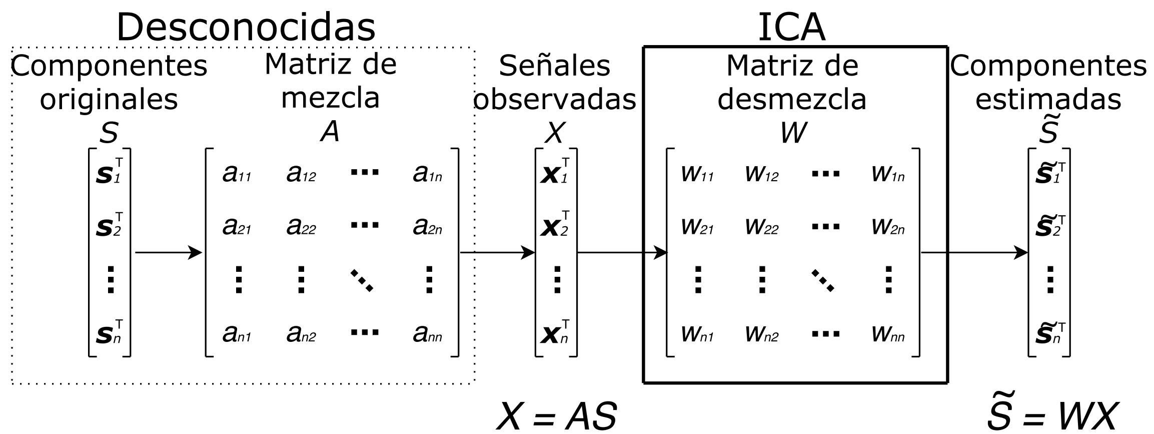


Figura 1.2: Modelo básico de ICA.

Otro método utilizado frecuentemente para separación ciega de señales es PCA. Éste se basa en que si la información es gaussiana, encontrar componentes independientes se reduce a decorrelacionar la información. No obstante, en realidad la información suele no seguir una distribución gaussiana. ICA busca encontrar componentes estadísticamente

independientes cuando la información no es gaussiana. En el caso particular de sustracción de fondo, el método puede aplicarse asumiendo que la pdf del fondo se comporta principalmente de manera gaussiana mientras que las pdfs que representan los objetos en movimiento no son gaussianas.

FastICA es un método iterativo basado en Newthon-Raphson el cual calcula la matriz de separación [25]. La expresión principal de FastICA está dada por

$$\mathbf{w}_i^+ = E\{Z(g(\mathbf{w}_i^T Z))^T\} - E\{\dot{g}(\mathbf{w}_i^T Z)\}\mathbf{w}_i \tag{1.3}$$

donde $\mathbf{w}_i$ es un vector de la matriz de desmezcla el cual estima una de las componentes originales, Z se obtiene decorrelacionando los vectores $\mathbf{x}_i$, y g y $\dot{g}$ son la primer y segunda derivada de una función no lineal y no cuadrática, respectivamente. Inicialmente, cada $\mathbf{w}_i$ se selecciona aleatoriamente y la expresión principal se repite hasta que la dirección de $\mathbf{w}_i$ converge, normalizando $\mathbf{w}_i$ antes de cada iteración.

1.3. Arquitecturas de hardware

Algunas implementaciones de sustracción de fondo propuestas en la literatura logran procesamiento en tiempo real con una alta resolución de video a costa de un alto consumo energético y un gran número de recursos. Otras logran procesar imágenes rápidamente y con alta resolución utilizando métodos muy simples, pero se obtienen resultados pobres [21]. Para acelerar la ejecución de estos métodos se pueden implementar en FPGA, GPU y DSP [10]. El DSP puede utilizar algunas técnicas de paralelización a nivel instrucción, mas su capacidad computacional es limitada. El FPGA y el GPU tienen la capacidad de realizar cómputo paralelo. Sin embargo, el GPU no tiene una buena eficiencia energética. Actualmente, una tendencia en el desarrollo de sistemas de visión es utilizar primero un GPU para realizar una prueba de concepto en una computadora, donde el tiempo de desarrollo es menor, y posteriormente adecuar la aplicación a un FPGA, para aprovechar sus ventajas en cuanto a tamaño y consumo energético. La Tabla 1.1 presenta una comparativa de estas arquitecturas.

Tabla 1.1: Comparación entre arquitecturas.

Característica	**Arquitectura**			
	Procesador	**FPGA**	**GPU**	**DSP**
Dificultad para desarrollar software	Baja (sistema operativo)	Alta	Media	Baja
Paralelismo (no. de hilos)	Fijo (nulo o bajo, dependiente del modelo)	Versatil (alto)	Fijo (alto)	Fijo (nulo o bajo, dependiente de la instrucción)
Frecuencia de reloj	Alta	Baja	Media	Media
Consumo energético	Moderado	Moderado	Alto	Moderado
Embebido	Sí	Sí	En desarrollo	Sí

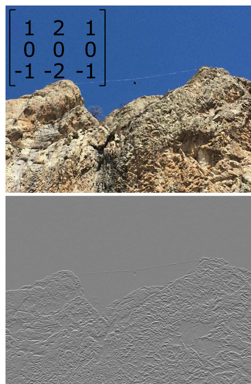
El procesador es el dispositivo responsable por la mayor parte del procesamiento en las computadoras personales. Desarrollar software para esta tecnología es rápido y su popularidad ha llevado a la creación de múltiples sistemas operativos, lenguajes de programación, bibliotecas y kits de desarrollo. Los procesadores con múltiples núcleos permiten un paralelismo limitado y pueden alcanzar frecuencias de reloj elevadas. Por otro lado, estas características normalmente están restringidas en los dispositivos embebidos a cambio de menor consumo energético y un menor tamaño.

Los FPGA son lógica reconfigurable que puede programarse para comportarse como un circuito digital. Los FPGAs suelen incluir recursos tales como LUTs, flip-flops, DSP slices, bloques de RAM, entre otros. Estos dispositivos son muy versátiles, permitiendo a los desarrolladores seleccionar diferentes niveles de paralelismo espacial y temporal. Uno de los mayores desafíos al diseñar en FPGA es que los proyectos complejos requieren un alto tiempo de desarrollo de software y pueden terminar eventualmente con los recursos de hardware del FPGA.

Para desarrollar en FPGA existen herramientas que aceleran el diseño y permiten reutilizar módulos diseñados previamente, esto las convierte en buenas plataformas para el prototipado de algoritmos de procesamiento de imágenes. Dentro de estas herramientas se encuentran los núcleos de propiedad intelectual, los cuales encapsulan código desarrollado por el fabricante u otra empresa, que describe dispositivos comúnmente utilizados al desarrollar en FPGA. El usuario puede utilizar estos núcleos para disminuir el tiempo de desarrollo de sus productos. Algunos ejemplos son analizadores de frecuencia, puentes H e interfaces para periféricos. Los núcleos más ampliamente utilizados son los que describen procesadores [6]. El desarrollo de estos procesadores implementados en software permitió extender el uso del FPGA a una gran cantidad de aplicaciones industriales.

A pesar de que los procesadores se pueden implementar mediante software dentro del mismo FPGA, su velocidad es inferior a la de los procesadores tradicionales. Esto ocurre ya que la versatilidad de la arquitectura de un FPGA se logra a costa de una velocidad relativamente baja en la conmutación de los transistores. Con los avances recientes en las tecnologías utilizadas para la fabricación de SoCs, los fabricantes de FPGA han sido capaces de embeber procesadores dentro del mismo chip. Esta implementación mediante hardware permite utilizar una arquitectura fija para el procesador con lo que se logra una mayor velocidad que en la implementación por software. El enfoque utilizado generalmente en esta arquitectura es utilizar el procesador como una unidad de control y el FPGA como un acelerador de hardware para las operaciones que convengan.

1.4. Paralelismo



Secuencial:

```
sum=0;
for(i=0; i<3; i++)
 for(j=0; j<3;j++)
  sum+=m(i)(j)*im(x-i-1)(y-j-1)
```

Paralelo:

```
m(0)(0)*im(x-1)(y-1);
...
m(2)(2)*im(x+1)(y+1);
```

(a)

Secuencial:

```
meses(0)=costoInicial;
inte=interesMensual;
for(i=1;i<n;i++)
 meses(i)=meses(i-1)*(1+inte);
```

Paralelo:

```
meses(1)=costoInicial*(1+inte)^1;
...
meses(n-1)=costoInicial*(1+inte)^(n-1);
```

(b)

Figura 1.3: Paralelismo: (a) en un caso trivial y (b) en un caso no trivial.

El FPGA funciona como acelerador al explotar el paralelismo que se pueda lograr en un método. No se ha podido generar un proceso automático que a partir de un algoritmo secuencial determine uno paralelo, salvo en los casos triviales. Además, en la práctica se debe considerar si utilizar paralelismo es seguro para la aplicación y si vale la pena paralelizar. La Figura 1.3 (a) ilustra un caso de paralelismo trivial (filtro Sobel) donde sólo se requiere desenrrollar un bucle, por otro lado en la Figura 1.3 (b) se muestra un caso donde se requiere una reformulación del algoritmo (interés compuesto) para lograr el paralelismo. En estos dos casos el paralelismo logrado es del tipo espacial. El paralelismo no sólo se puede aplicar desenrrollando bucles. También se puede lograr en un proceso planteándolo de forma similar a una línea de producción como se muestra en la Figura 1.4. Esto se conoce como un pipeline y corresponde a paralelismo del tipo temporal. Dentro de visión, el paralelismo más sencillo se logra cuando cada pixel de la imagen puede ser procesado de manera independiente. La Figura 1.5 muestra el paralelismo espacial y temporal correspondientes a este caso.

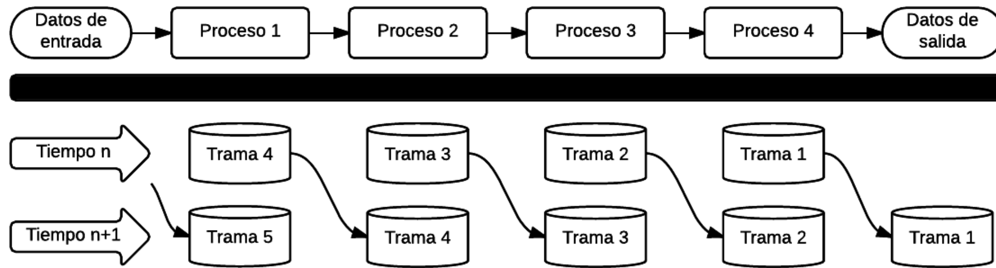


Figura 1.4: Paralelismo en pipeline.

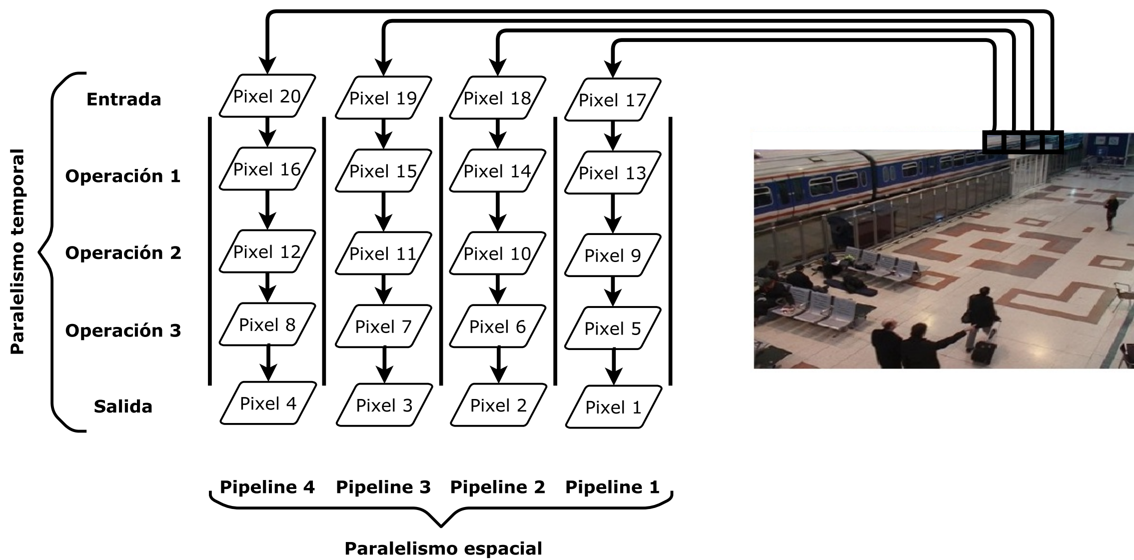


Figura 1.5: Paralelismo al procesar imágenes.

Capítulo 2

Propuesta.

2.1. Desarrollo del método de sustracción de fondo

El método de sustracción de fondo desarrollado consiste de tres bloques principales modelo de fondo, extracción del primer plano y detección de movimiento (véase Figura 2.1).

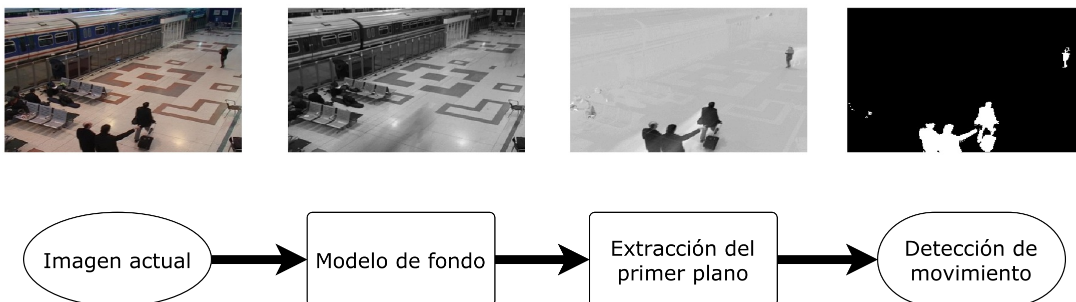


Figura 2.1: Diagrama a bloques del método.

2.1.1. Modelo de fondo

EM es una solución para mantener una media actualizada que no requiere almacenar las entradas anteriores en un buffer. Una estimación simple de la media se puede obtener mediante

$$O^{+} = (1 - \alpha)O + \alpha I, \tag{2.1}$$

donde O^{+} es la media actualizada, O es la media anterior, I es la entrada más reciente y α es el radio de aprendizaje ajustable en el conjunto $(0, 1)$. Esta estimación de la media puede adaptarse a cambios lentos o repentinos dependiendo de la elección de α.

Cada pixel del fondo se determina con una estimación de la media basada en EM. Cuando se ajusta el valor del parámetro α, un valor inicial útil es $1/k$. k siendo el número de cuadros que se espera un objeto en movimiento permanezca en la misma región, lo cual depende tanto

del tamaño y de la velocidad del objeto. Este algoritmo es adecuado para un FPGA-SoC debido a que cada pixel se calcula independientemente, no requiere almacenar los cuadros anteriores en los bloques de RAM y sólo emplea multiplicadores y sumadores, los cuales se implementan fácilmente en el FPGA.

2.1.2. Extracción del primer plano

Este bloque se basa en el enfoque presentado por Jiménez-Hernández [2], en el cual la secuencia de imágenes (cada una como un vector $\mathbf{x}_i$) es usada como la entrada de ICA y la salida consiste de componentes representando el fondo, el primer plano y el ruido. Dichas componentes se categorizan en base a la cantidad de información que contiene, la mayoría contenida en el fondo y la minoría en el ruido. Esta comparación se logra examinando los eigenvalores obtenidos por la SVD de W^{-1}, *i.e.*, un eigenvalor alto corresponde a una gran cantidad de información. Al suprimir los eigenvalores asociados con las componentes del fondo y del ruido se puede recuperar únicamente la información del primer plano.

Por lo tanto, este trabajo propone usar sólo dos imágenes como la entrada de ICA, la primer entrada $\mathbf{x}_1$ es el cuadro más reciente y la segunda $\mathbf{x}_2$ representando un estimado del fondo. Además, para facilitar una implementación en un FPGA-SoC, la expresión principal de FastICA (1.3) se reformuló como

$$w_{i,k}^{+} = \sum_{j=1}^{M} z_{k,j} g(w_{i,1} z_{1,j} + w_{i,2} z_{2,j}) - \dot{g}(w_{i,1} z_{1,j} + w_{i,2} z_{2,j}) w_{i,k}, \tag{2.2}$$

donde M es el número de pixeles, $w_{i,k}$ es un elemento de la matriz W, $z_{k,j}$ es un elemento de la matriz obtenida al decorrelacionar los vectores $\mathbf{x}_i$. Nótese que $w_{i,1}$ y $w_{i,2}$ deben ser actualizados simultáneamente, ya que estos conforman $\mathbf{w}_i$. La matriz de decorrelación Wh puede obtenerse a través de la SVD de la matriz de covarianza Cov. Dada la descomposición $Cov = U\Sigma V^T$, la matriz de decorrelación puede obtenerse como $Wh = \Sigma^{-1/2} V^T$ y su inversa como $Wh^{-1} = V\Sigma^{1/2}$.

Con esta reformulación, la contribución de cada pixel j se puede calcular independientemente y posteriormente realizar una suma acumulada. Estas características son apropiadas para el paralelismo, los pixeles pueden ser divididos en un número predeterminado de grupos,

elegido en base a los recursos disponibles en el FPGA. Adicionalmente, cada grupo puede ser procesado en paralelo empleando una pipeline, logrando paralelismo tanto espacial como temporal.

En este caso particular sólo se tienen dos componentes y por lo tanto únicamente dos vectores $\mathbf{w}_i$. Sabiendo que en el espacio decorrelacionado los vectores $\mathbf{w}_i$ son ortogonales, es posible calcular únicamente $\mathbf{w}_1$ y determinar $\mathbf{w}_2$ mediante una ortogonalización trivial, *i.e.*, $w_{2,1} = w_{1,2}$ y $w_{2,2} = -w_{1,1}$. Esta estrategia es útil para la implementación en FPGA ya que únicamente se consumen los recursos de hardware correspondientes al cálculo de una componente.

En el caso de requerir más componentes, la reformulación para FPGA presentada en la ecuación (2.2) se puede extender a N componentes de la siguiente manera:

$$w_{i,k}^{+} = \sum_{j=1}^{M} z_{k,j} g(\sum_{l=1}^{N} w_{i,l} z_{l,j}) - \dot{g}(\sum_{l=1}^{N} w_{i,l} z_{l,j}) w_{i,k}, \tag{2.3}$$

donde $w_{i,1}...w_{i,N}$ deben ser actualizados simultaneamente. Forzosamente se deben estimar al menos $N-1$ vectores $\mathbf{w}_i$ y no es posible utilizar la ortogonalización trivial que se planteó anteriormente. Se puede optar por estimar los $N-1$ vectores en paralelo y antes de cada iteración realizar una ortogonalización simétrica. Sin embargo, esto requerirá un FPGA con una gran cantidad de recursos de hardware. La opción alternativa es estimar un vector a la vez y antes de cada iteración ortogonalizar respecto a los vectores ya obtenidos. De esta manera se reducen los recursos de hardware necesarios a cambio de incrementar el tiempo de ejecución. Nótese que $\mathbf{w}_N$ puede obtenerse de manera trivial a partir de los otros vectores, de manera similar al caso de dos componentes.

A continuación, se obtiene W al multiplicar Wh^{-1} por la matriz formada con los vectores $\mathbf{w}_i$. Después de calcular W, las componentes independientes son estimadas mediante

$$\tilde{S} = W[(\mathbf{x}_1 - mean\{\mathbf{x}_1\}) \quad (\mathbf{x}_2 - mean\{\mathbf{x}_2\})]^T, \tag{2.4}$$

y la matriz de mezcla A se obtiene a través de W^{-1}. Dado que sólo se tienen dos vectores de entrada, A consistirá de dos eigenvalores. El mayor se asocia al fondo y el otro combina la información del primer plano y el ruido. Por lo tanto, el primer plano puede ser estimado

por

$$X^* = A^* \tilde{S}, \tag{2.5}$$

donde la primer fila de X^* es el fondo (representado como un vector) y A^* se obtiene suprimiendo el mayor eigenvalor de A.

2.1.3. Detección de movimiento

Una vez que se ha obtenido la imagen de primer plano, se debe determinar cuáles pixeles corresponden al movimiento. Si no se regresa la media después de decorrelacionar los vectores $\mathbf{x}_i$ la imagen de primer plano obtenida tendrá valores cercanos a cero en los pixeles que corresponden al fondo. Asumiendo que la pdf de estos pixeles es gaussiana, el problema de clasificar un pixel como movimiento se reduce a determinar si ese pixel corresponde a esta distribución gaussiana.

Una solución inmediata a este problema es usar un umbral basado en la desviación estándar, donde la desviación estándar σ se puede estimar utilizando todos los pixeles en la imagen de primer plano y la media μ es simplemente cero debido a la decorrelación. Así, el movimiento se determina por

$$M(p) = \begin{cases} 0 & \text{if } abs(p) \leq \beta\sigma \\ 1 & \text{en caso contrario,} \end{cases} \tag{2.6}$$

donde p es la intensidad del pixel en la imagen de primer plano y β es un parámetro ajustable. Este algoritmo se divide en dos etapas, primero se calcula la desviación estándar y después se realiza la comparación. Ambas etapas ofrecen oportunidades para paralelizar. Aunado a esto, dado que la media μ es cero, obtener la desviación estándar se reduce a

$$\sigma = \sqrt{\sum_{j=1}^{M} (x^*_{1,j})^2}, \tag{2.7}$$

donde $x^*_{1,j}$ es un elemento de la primer fila de X^*, *i.e.*, un pixel de la imagen de primer plano.

2.2. Implementación del método

Antes de comenzar a desarrollar el código en VHDL, se debe decidir que operaciones se realizarán en el FPGA y cuales en el procesador embebido. Aunque la ejecución del proceso sería más rápida al realizar todas las operaciones en el FPGA, esto requeriría un tiempo de desarrollo muy elevado y un FPGA con una alta cantidad de recursos de hardware. Combinando adecuadamente el FPGA y el procesador, la implementación del método conlleva menos tiempo y se puede utilizar un FPGA de menor costo. De esta manera, se incrementa la factibilidad de integrar el paralelismo de los FPGAs en un sistema de visión embebido.

El enfoque utilizado considera al procesador para controlar el flujo del proceso y realizar aquellas operaciones que son difíciles de implementar en el FPGA o requieren una gran cantidad de recursos de hardware. Mientras que el FPGA es tratado como un acelerador de hardware. Éste se emplea en las etapas donde se procesan grandes bloques de información (como una imagen) y algún nivel de paralelismo está disponible.

No obstante, las etapas que requieren intercambiar entradas o salidas de tamaño considerable requieren una evaluación más profunda, porque la comunicación entre el procesador y el FPGA puede ralentizar el proceso y se consumen recursos de hardware adicionales. Las entradas grandes pueden evitarse almacenando la imagen actual $\mathbf{x}_1$ y la estimación del fondo $\mathbf{x}_2$ en el FPGA y enviando desde el procesador únicamente la información necesaria para calcular la entrada requerida a partir de $\mathbf{x}_1$ y $\mathbf{x}_2$. *E.g.*, al implementar la expresión principal reformulada de FastICA no es necesario enviar la matriz Z (cuya dimensión es $2 \times N$), en su lugar el FPGA puede recibir únicamente la matriz de decorrelación Wh (cuya dimensión es 2×2) para poder calcular Z de la siguiente manera:

$$Z = Wh[\mathbf{x}_1 - mean\{\mathbf{x}_1\} \quad \mathbf{x}_2 - mean\{\mathbf{x}_2\}].^T, \tag{2.8}$$

donde $mean\{\mathbf{x}_1\}$ y $mean\{\mathbf{x}_2\}$ son los valores de intensidad promedio de $\mathbf{x}_1$ y $\mathbf{x}_2$, respectivamente, los cuales pueden ser fácilmente calculados durante un paso anterior. Este planteamiento involucra algunas multiplicaciones y sumas adicionales, sin embargo éstas se pueden implementar dentro del pipeline correspondiente a FastICA y como resultado la

diferencia en el tiempo de procesamiento es despreciable.

La implementación propuesta para el método desarrollado señalando que operaciones se realizan en el procesador y cuales en el FPGA se resume en la Figura 2.2.

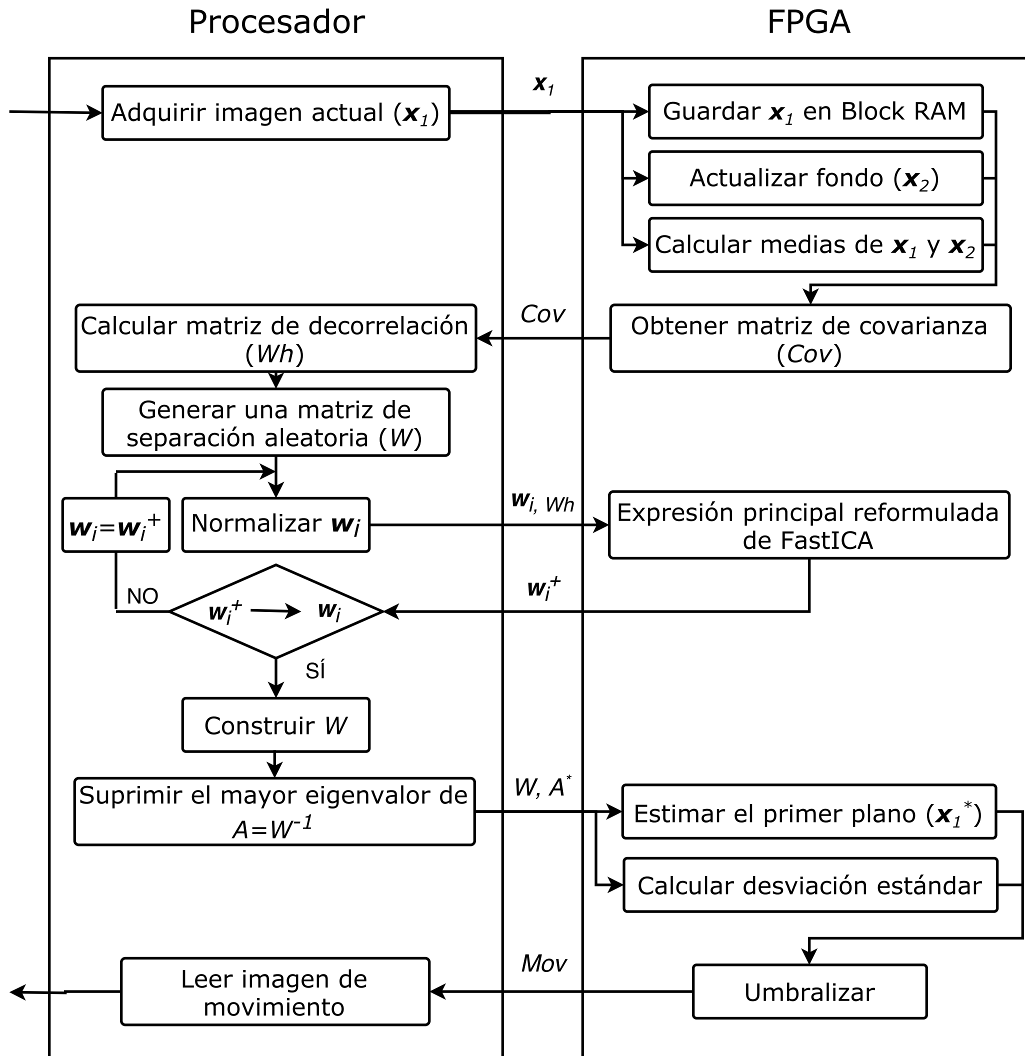


Figura 2.2: Diagrama de flujo de la implementación propuesta.

De acuerdo a los alcances planteados y a los resultados obtenidos, únicamente se utilizó el FPGA para el cálculo de FastICA y de la matriz de covarianza. Además, la estimación del fondo se realizó fuera de línea para facilitar la experimentación. También se optó por cambiar la condición de convergencia para $\mathbf{w}_i$ por un número fijo de iteraciones.

El método se implementó en una tarjeta de desarrollo Zedboard [27] basada en el zynq-7000 de Xilinx. La Tabla 2.1 muestra sus especificaciones técnicas más relevantes. La

comunicación entre el FPGA y el procesador se realizó utilizando Xillybus [28]. Para facilitar el desarrollo de software se instaló una distribución de Linux embebido. Se eligió Xillinux [29] ya que está desarrollada específicamente para manejar Xillybus. Utilizando este núcleo de propiedad intelectual y sus controladores complementarios se pueden definir tantos canales de comunicación como la aplicación requiera. Para el FPGA estos canales de comunicación se manipulan mediante el uso de FIFOs, los cuales pueden ser generados desde el IP Catalog de Vivado [30]. Por otra parte, el procesador puede acceder a la interfaz de estos canales mediante los “devices files” que se crean automáticamente al tener los controladores correspondientes. El envío y recepción de información se logra al manipularlos como simples archivos de escritura y lectura, respectivamente. Se usó Python [31] en el procesador para manejar estos archivos con el paquete “io”, mientras que los cálculos numéricos se realizaron con el paquete “numpy” (*numerical python*).

Tabla 2.1: Especificaciones técnicas del hardware.

<table>
<tr><th colspan="3">Zedboard Zynq-7000</th></tr>
<tr><td rowspan="8">Procesador</td><td>Familia</td><td>Dual core ARM Cortex A9</td></tr>
<tr><td>Frecuencia</td><td>Hasta 667MHz</td></tr>
<tr><td>RAM</td><td>512MB</td></tr>
<tr><td>Disco duro</td><td>Memoria SD</td></tr>
<tr><td rowspan="2">Conectividad</td><td>Ethernet</td></tr>
<tr><td>USB UART</td></tr>
<tr><td rowspan="2">Video</td><td>HDMI</td></tr>
<tr><td>VGA</td></tr>
<tr><td rowspan="5">FPGA</td><td>Familia</td><td>Artix-7</td></tr>
<tr><td>LUTs</td><td>53200</td></tr>
<tr><td>Flip-flops</td><td>106400</td></tr>
<tr><td>Block RAM</td><td>4.9MB</td></tr>
<tr><td>DSP slices</td><td>220</td></tr>
</table>

Capítulo 3

Modelo experimental y resultados.

3.1. Diseño experimental

Se identificaron las siguientes desventajas de utilizar experimentación capturando video en vivo:

- Después de cada prueba se debe realizar la selección imagen por imagen de cuales pixeles corresponden a movimiento.
- En cada prueba la entrada es diferente, dificultando encontrar errores en la implementación.

Con el fin de simplificar el proceso de evaluación de resultados, se optó por utilizar secuencias de imágenes extraídas desde la base de datos PETS [32] para las cuales se cuenta con una referencia, indicando qué pixeles de cada imagen corresponden a movimiento. La desventaja de realizar la experimentación capturando video en vivo es que después de cada prueba se debe realizar la selección imagen por imagen de cuales pixeles corresponden a movimiento. Además, la base de datos PETS es reconocida y frecuentemente se usa para validar los resultados de las investigaciones en el área de visión.

El diseño experimental consistió en procesar dos secuencias de 310 imágenes en escala de grises cada una, con un tamaño de 320 por 200 pixeles, y evaluar únicamente la detección de movimiento, para la cual se cuenta con la referencia. El enfoque usado para la evaluación es a nivel pixel [24] mediante la precisión y la sensibilidad. Primero se obtienen los verdaderos positivos (TP), falsos positivos (FP), verdaderos negativos (TN) y falsos negativos (FN). Después la precisión P se obtiene mediante

$$P = \frac{TP}{TP + FP}, \tag{3.1}$$

y la sensibilidad R se calcula de la siguiente manera:

$$R = \frac{TP}{TP + FN}. \tag{3.2}$$

Además se midieron los tiempos de ejecución de las tareas principales en el caso de la implementación usando sólo el procesador embebido y cuando se usaron tanto el FPGA como el procesador.

3.2. Resultados

Una muestra de las imágenes obtenidas de la secuencia 1 en la simulación en Matlab y en el FPGA-SoC se presentan en las Figuras 3.1 y 3.2, respectivamente. Algunas de las imágenes correspondientes a la secuencia 2 se muestran en las Figuras 3.3 y 3.4 para la simulación y la implementación, respectivamente. Los valores de precisión (P) y sensibilidad (R) calculados con estas muestras de imágenes se exponen en las Tablas 3.1, 3.2, 3.3 y 3.4. En éstas también se presentan los valores promedio de precisión y sensibilidad en la secuencia. Al calcular estos valores promedio, se excluyeron las imágenes que tenían poco o nada de movimiento ya que resultaban en valores extremos. Así, 240 imágenes se consideraron de cada secuencia. En ambas secuencias los parámetros α y β se seleccionaron como 0.02 y 2.00, respectivamente.

La distribución de los valores de precisión y sensibilidad en la secuencia 1 se puede observar en los histogramas presentados en las Figuras 3.5 y 3.6, respectivamente, tanto para la simulación en Matlab como para la implementación en el FPGA-SoC. Los histogramas correspondientes a la precisión y sensibilidad en la secuencia 2 se muestran en las Figuras 3.7 y 3.8, respectivamente.

Las gráficas de los diferentes valores de precisión y sensibilidad obtenidos a lo largo de la secuencia 1 se ilustran en las Figuras 3.9 y 3.10, respectivamente, tanto para la simulación en Matlab como para la implementación en el FPGA-SoC. Las gráficas de precisión y sensibilidad correspondientes a la secuencia 2 se presentan en las Figuras 3.11 y 3.12, respectivamente.

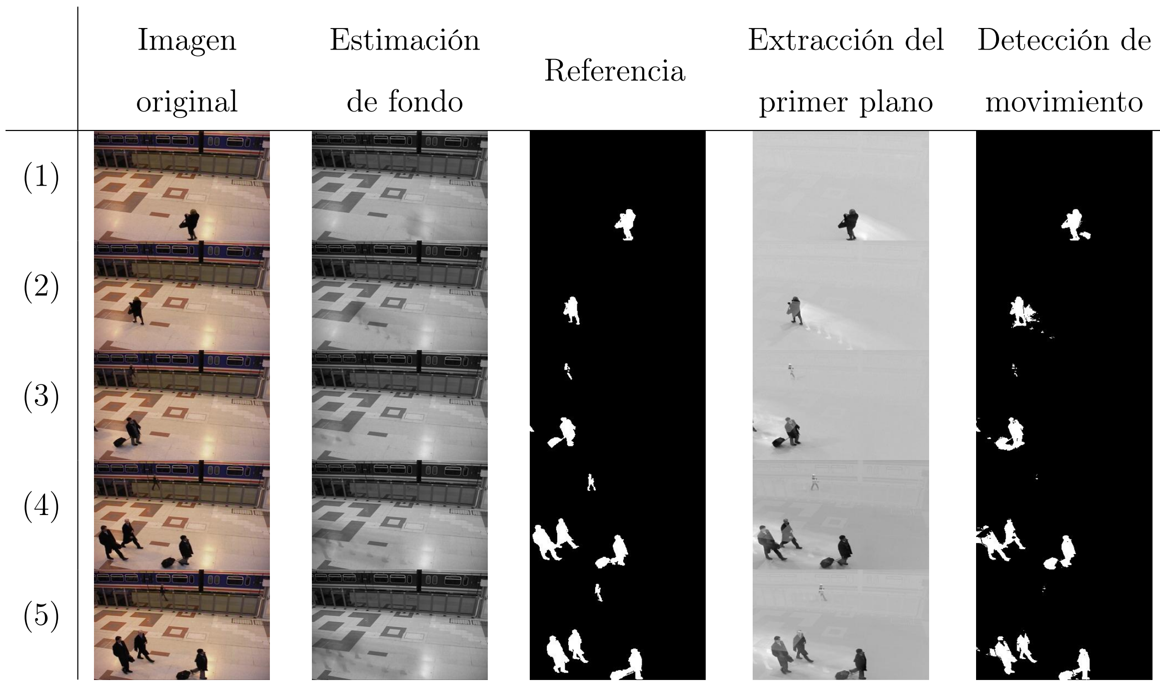


Figura 3.1: Resultados obtenidos de procesar la secuencia 1 en la simulación en Matlab.

Tabla 3.1: Precisión y sensibilidad obtenidas de procesar la secuencia 1 en la simulación en Matlab.

	Precisión (P)	**Sensibilidad** (R)
(1)	0.84	0.94
(2)	0.56	0.94
(3)	0.67	0.79
(4)	0.82	0.70
(5)	0.84	0.72
Promedio de la secuencia	0.73	0.88

Los tiempos de ejecución promedio por imágen para cada una de las tareas principales utilizando únicamente el procesador ARM embebido y utilizando tanto el FPGA como el procesador ARM embebido se presentan en la Tabla 3.5. Además, en la Tabla 3.6 se presentan los tiempos de ejecución promedio por iteración de FastICA para las tareas correspondientes. En el caso de la implementación sólo con el procesador ARM no existen

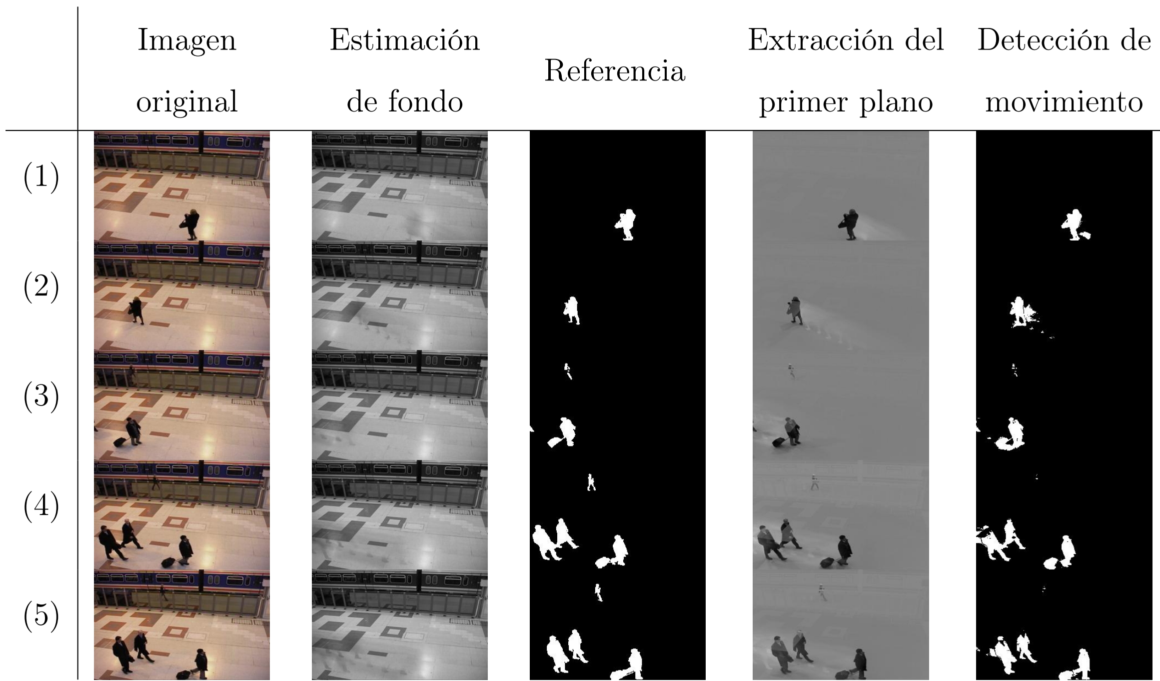


Figura 3.2: Resultados obtenidos de procesar la secuencia 1 en el FPGA-SoC.

Tabla 3.2: Precisión y sensibilidad obtenidas de procesar la secuencia 1 en el FPGA-SoC.

	Precisión (P)	**Sensibilidad** (R)
(1)	0.84	0.94
(2)	0.56	0.94
(3)	0.67	0.79
(4)	0.82	0.70
(5)	0.84	0.72
Promedio de la secuencia	0.73	0.88

tiempos de codificación y decodificación ya que estas tareas están asociadas con la comunicación procesador-FPGA. Además, las iteraciones de FastICA en el procesador se implementaron utilizando la expresión principal clásica ya que esta permite utilizar operaciones vectorizadas para su implementación, las cuales son más apropiadas para un procesador.

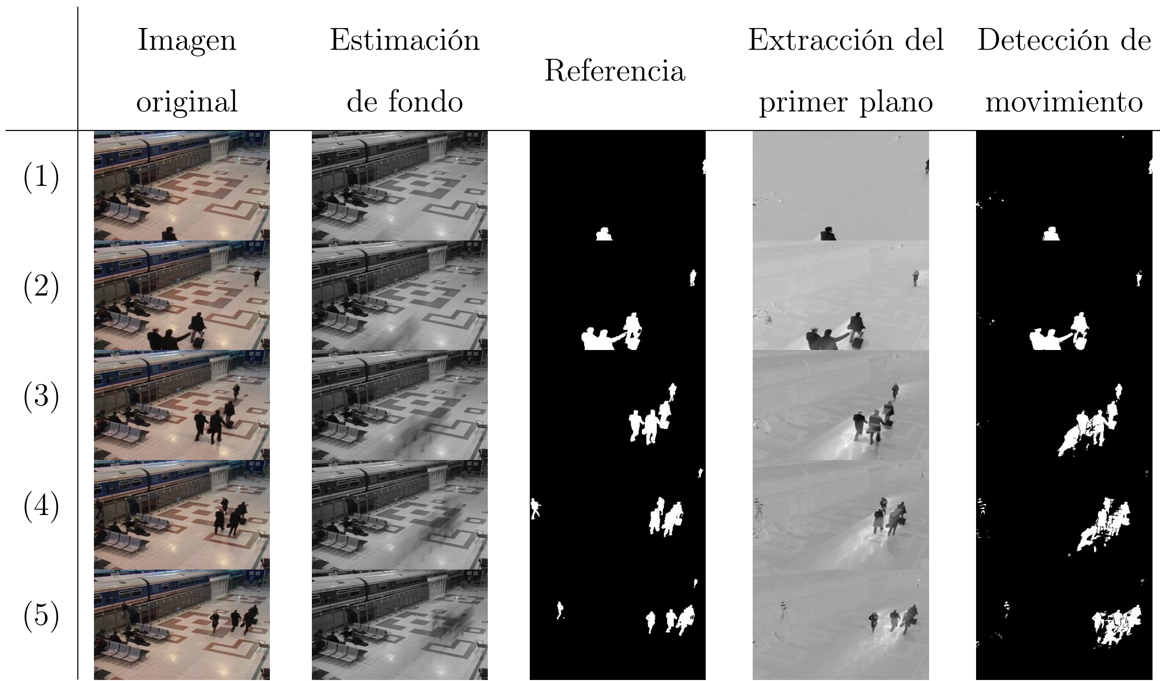


Figura 3.3: Resultados obtenidos de procesar la secuencia 2 en la simulación en Matlab.

Tabla 3.3: Precisión y sensibilidad obtenidas de procesar la secuencia 2 en la simulación en Matlab.

	Precisión (P)	**Sensibilidad** (R)
(1)	0.79	0.94
(2)	0.94	0.84
(3)	0.64	0.81
(4)	0.59	0.75
(5)	0.51	0.78
Promedio de la secuencia	0.65	0.81

3.3. Discusión de resultados

Los resultados muestran que la extracción del primer plano y la detección de movimiento se obtienen correctamente. Examinando los resultados de las Figuras 3.1, 3.2, 3.3 y 3.4, se observa que los valores de precisión y sensibilidad para la segunda secuencia en general fueron menores. Los valores menores se pueden atribuir a que la estimación de fondo retiene

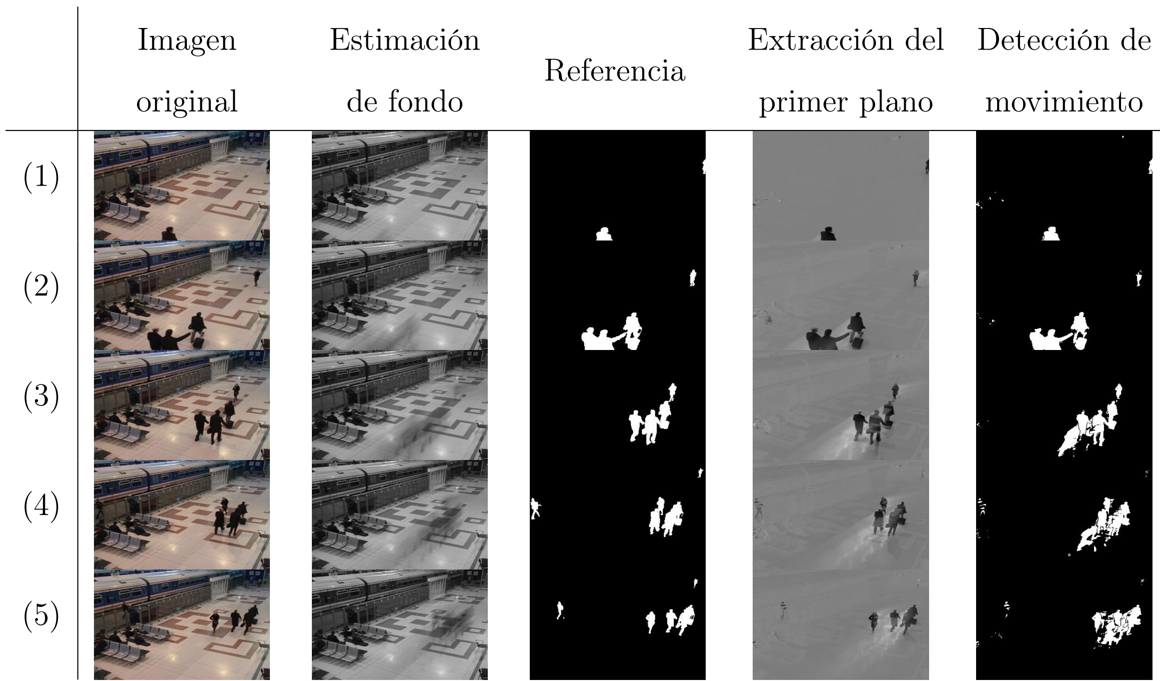


Figura 3.4: Resultados obtenidos de procesar la secuencia 2 en el FPGA-SoC.

Tabla 3.4: Precisión y sensibilidad obtenidas de procesar la secuencia 2 en el FPGA-SoC.

	Precisión (P)	**Sensibilidad** (R)
(1)	0.79	0.93
(2)	0.94	0.84
(3)	0.64	0.81
(4)	0.60	0.75
(5)	0.51	0.78
Promedio de la secuencia	0.65	0.81

demasiada información sobre el movimiento anterior. Esto podría indicar que se necesita un menor factor de aprendizaje α para la secuencia 2, ya que contiene una mayor área en movimiento. Alternativamente, se pueden explorar otros modelos de fondo que proporcionen más robustez contra el tamaño del área en movimiento o se puede utilizar un filtro morfológico para mejorar los resultados, como se discutió en [33] donde se presentaron resultados parciales de esta investigación.

Tabla 3.5: Tiempo de ejecución promedio por imagen procesada.

Tarea	**Tiempo promedio [ms]**	
	Arm	**Arm+FPGA**
Obtener matriz de covarianza	28.29	17.24
Calcular matriz de decorrelación	12.22	1.04
FastICA (100 iteraciones)	4245.74	68.42
Suprimir el mayor eigenvalor	0.69	0.69
Extraer el primer plano	19.69	28.88
Umbralizar el movimiento	8.27	8.66
Total	4314.90	124.93

Tabla 3.6: Tiempo de ejecución promedio por iteración de FastICA.

Tarea	**Tiempo promedio [ms]**	
	Arm	**Arm+FPGA**
Codificar paquete de datos		0.09
Expresión principal	42.20 (clásica)	0.33 (reformulada)
Decodificar resultado		0.06
Normalizar	0.19	0.18
Total	42.39	0.66

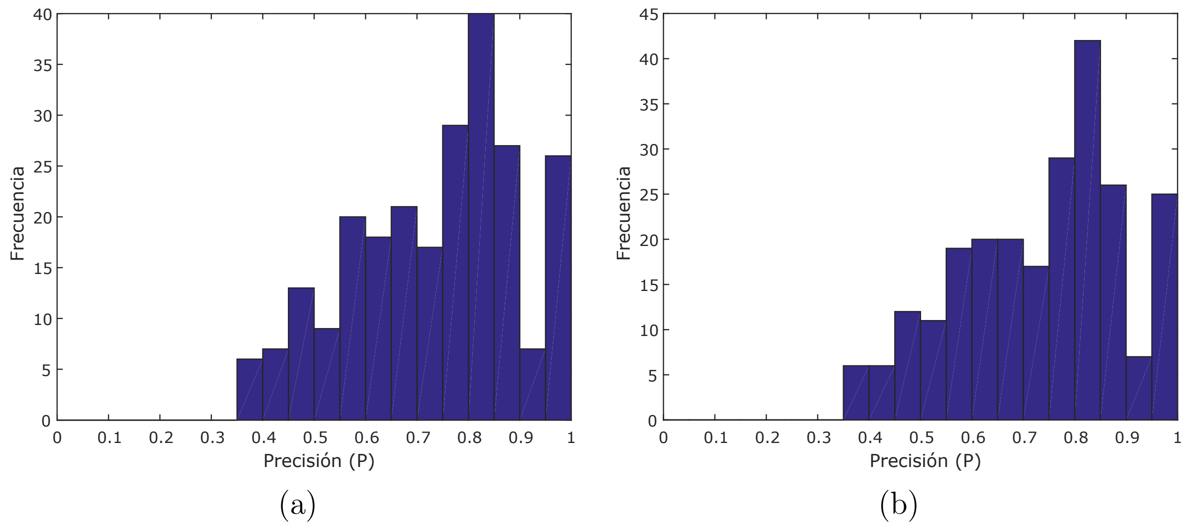


Figura 3.5: Histogramas de la precisión (P) obtenida al procesar la secuencia 1: (a) en la simulación en Matlab y (b) en el FPGA-SoC.

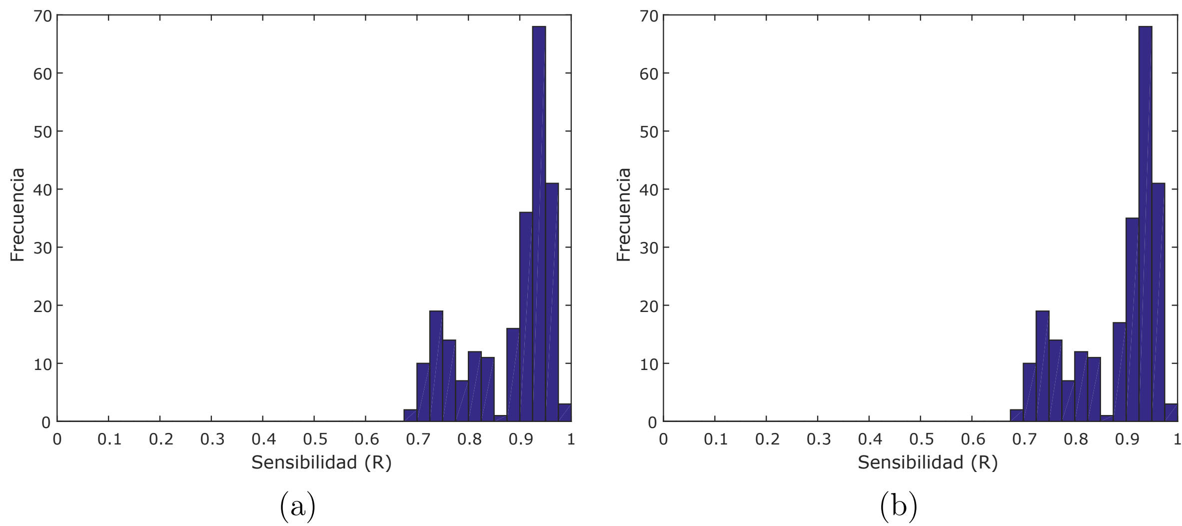


Figura 3.6: Histogramas de la sensibilidad (R) obtenida al procesar la secuencia 1: (a) en la simulación en Matlab y (b) en el FPGA-SoC.

Al comparar los resultados obtenidos en la simulación en Matlab y en la implementación en el FPGA-SoC en las imágenes de las Figuras 3.1, 3.2, 3.3 y 3.4, en los valores de precisión y sensibilidad de las Tablas 3.1, 3.2, 3.3 y 3.4, en las distribuciones de las Figuras 3.5, 3.6, 3.7 y 3.8 y en las gráficas de las Figuras 3.9, 3.10, 3.11 y 3.12 se puede observar que tienen un comportamiento similar, lo cual indica que el método planteado se llevó correctamente de la simulación a la implementación. Realizar una implementación de esta complejidad en VHDL

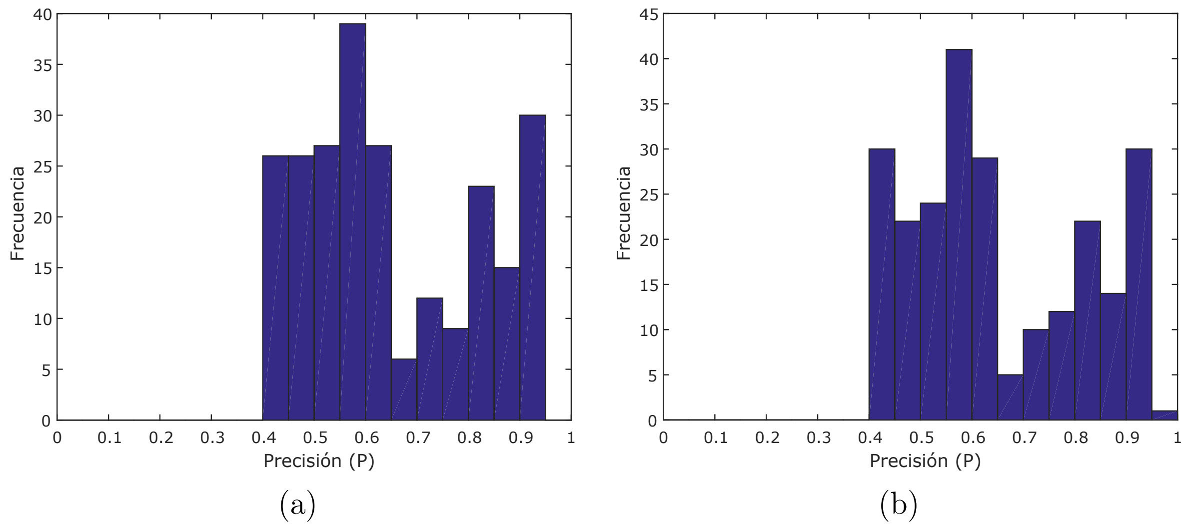


Figura 3.7: Histogramas de la precisión (P) obtenida al procesar la secuencia 2: (a) en la simulación en Matlab y (b) en el FPGA-SoC.

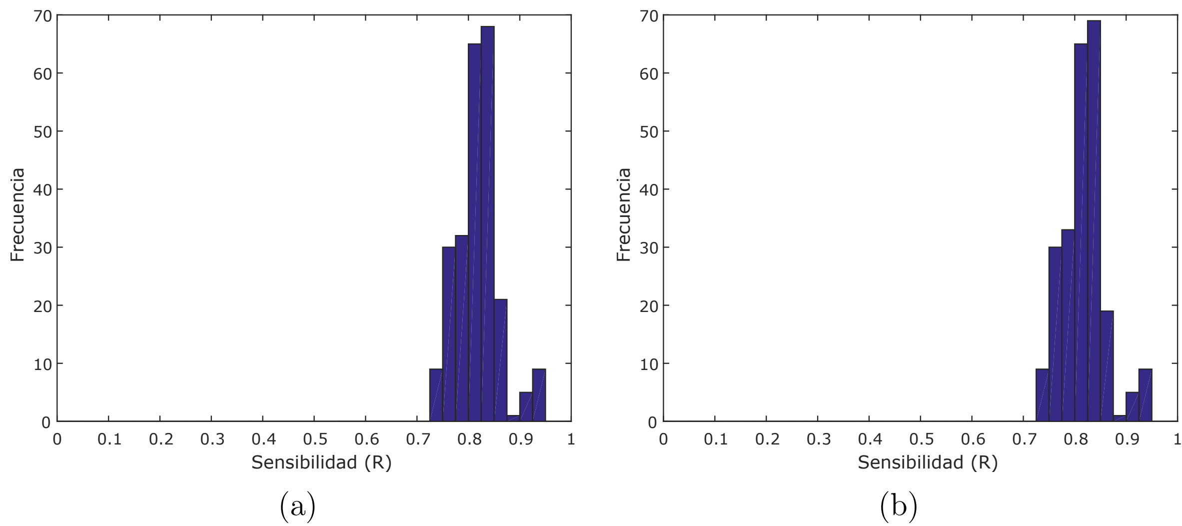


Figura 3.8: Histogramas de la sensibilidad (R) obtenida al procesar la secuencia 2: (a) en la simulación en Matlab y (b) en el FPGA-SoC.

implica diversas consideraciones como lo son definir un flujo en las operaciones del pipeline, evaluar los cambios en el punto fijo y sincronizar las latencias de los diferentes operadores. Por esta razón, es de suma utilidad tener previamente simulaciones que ilustren el resultado esperado y así identificar con mayor facilidad los errores en la implementación.

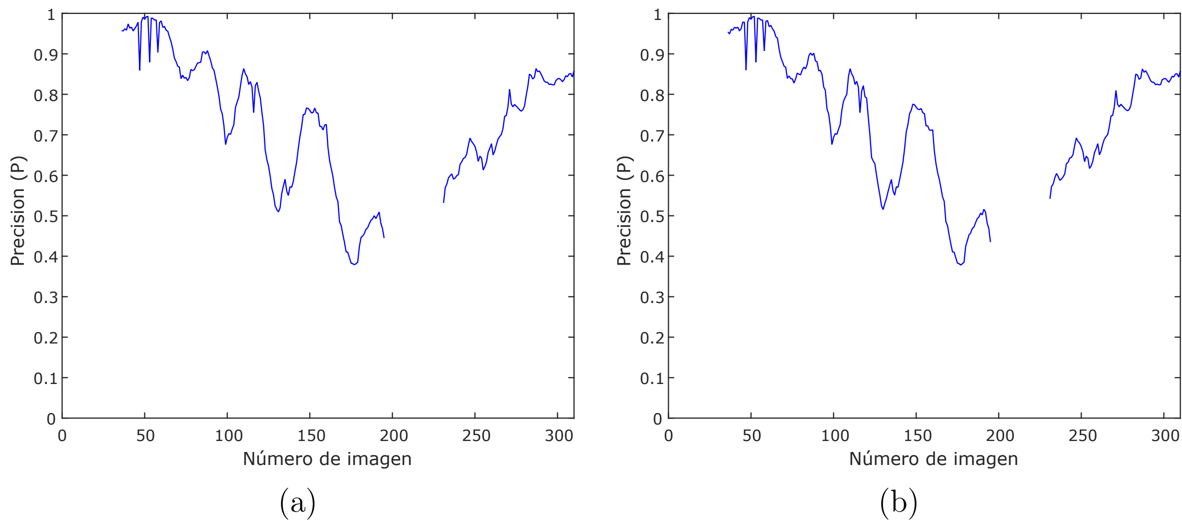


Figura 3.9: Valores de precisión (P) obtenidos a lo largo de la secuencia 1: (a) en la simulación en Matlab y (b) en el FPGA-SoC.

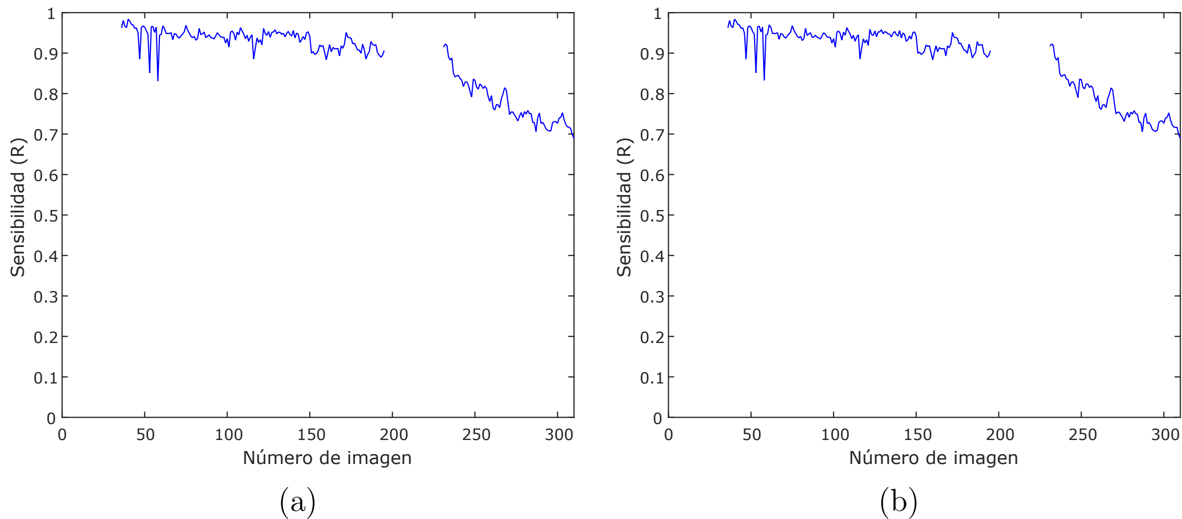


Figura 3.10: Valores de sensibilidad (R) obtenidos a lo largo de la secuencia 1: (a) en la simulación en Matlab y (b) en el FPGA-SoC.

La medición de los tiempos de ejecución en la implementación utilizando únicamente el procesador ARM embebido indica que la tarea con mayor prioridad para acelerar con el FPGA es el cálculo de FastICA. Los resultados de la Tabla 3.5 muestran que el tiempo de ejecución de FastICA se redujo en un 98.3 %, con lo cual se incrementó el número de cuadros por segundo en 3353 %. Los tiempos de ejecución de la implementación acelerando el proceso con el FPGA se obtuvieron utilizando únicamente cuatro pipelines. Con un FPGA con más

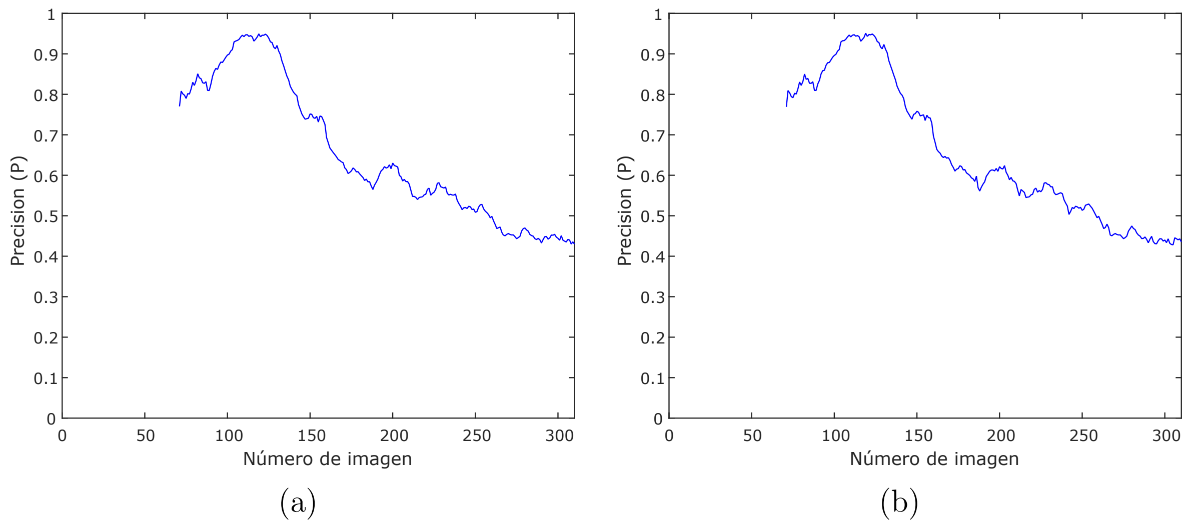


Figura 3.11: Valores de precisión (P) obtenidos a lo largo de la secuencia 2: (a) en la simulación en Matlab y (b) en el FPGA-SoC.

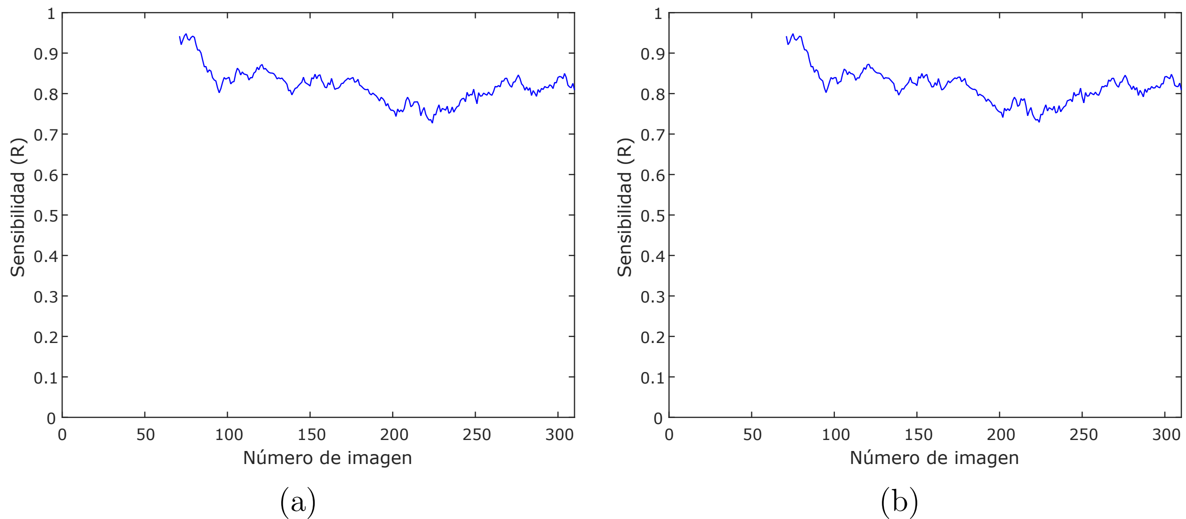


Figura 3.12: Valores de sensibilidad (R) obtenidos a lo largo de la secuencia 2: (a) en la simulación en Matlab y (b) en el FPGA-SoC.

recursos de hardware o reduciendo la precisión numérica, sería posible aumentar el número de pipelines y así reducir aún más el tiempo de ejecución. De manera similar, el tamaño de la imagen a procesar se puede incrementar en función de la cantidad de RAM disponible en el FPGA.

Capítulo 4

Conclusiones.

Esta tesis presentó un método de sustracción de fondo basado en ICA desarrollado e implementado para un FPGA-SoC con procesador embebido. El enfoque propuesto para desarrollar e implementar el método permite explotar la arquitectura del FPGA-SoC, incrementando la factibilidad de utilizar estas tecnologías para acelerar la ejecución de algoritmos de visión en los sistemas embebidos.

El método se dividió en tres bloques principales: actualizar el modelo de fondo estimando la media mediante EM, extraer el primer plano utilizando ICA, y detectar el movimiento con un umbral basado en desviación estándar. Se discutieron las oportunidades de paralelismo para cada bloque. El método se implementó utilizando únicamente el procesador embebido y se realizó un análisis de los tiempos de ejecución para determinar en cuáles tareas enfatizar los esfuerzos de paralelizar para ejecutar en el FPGA. Se determinó que la extracción del primer plano era la tarea crítica y al paralelizar el algoritmo de FastICA e implementarlo en el FPGA se redujo el tiempo de ejecución de 4314.90ms a 124.93ms, correspondiente a un incremento del 3353 % en cuadros por segundo. No se consideró la captura de la imagen ni la estimación de fondo en ninguna de estas mediciones.

En los resultados obtenidos al implementar el método en un FPGA-SoC se observa que tanto la extracción del primer plano como la detección de movimiento se determinaron correctamente. La detección de movimiento se evaluó en dos secuencias de imágenes utilizando una referencia para calcular la precisión y sensibilidad. En la primera secuencia los valores promedio de la precisión y la sensibilidad fueron 0.72 y 0.88, respectivamente. Para la segunda secuencia, se obtuvieron valores promedio de 0.64 y 0.81 para la precisión y sensibilidad, respectivamente. La reducción de los valores en la segunda secuencia se puede explicar por una pobre estimación del fondo, ocasionada por una mayor área en movimiento que requeriría un menor radio de aprendizaje.

El trabajo futuro incluye explorar modelos de fondo alternativos que también puedan beneficiarse del paralelismo, optimizar el código en VHDL para reducir los recursos utilizados en el FPGA, realizar experimentación con cámaras omnidireccionales orientada a una aplicación de vigilancia multicámara y evaluar la robustez del método contra problemas particulares de la detección de movimiento, tales como los cambios en la iluminación.

Bibliografía

[1] J. S. Kulchandani and K. J. Dangarwala, “Moving object detection: Review of recent research trends,” in *Pervasive Computing (ICPC), 2015 International Conference on*, pp. 1–5, IEEE, 2015.

[2] H. Jiménez-Hernández, “Background subtraction approach based on independent component analysis,” *Sensors*, vol. 10, no. 6, pp. 6092–6114, 2010.

[3] M. Piccardi, “Background subtraction techniques: a review,” in *Systems, man and cybernetics, 2004 IEEE international conference on*, vol. 4, pp. 3099–3104, IEEE, 2004.

[4] E. Fykse, “Performance comparison of gpu, dsp and fpga implementations of image processing and computer vision algorithms in embedded systems,” Master’s thesis, Institutt for elektronikk og telekommunikasjon, 2013.

[5] L. Struyf, S. De Beugher, D. H. Van Uytsel, F. Kanters, and T. Goedemé, “The battle of the giants: a case study of gpu vs fpga optimisation for real-time image processing,” in *Proceedings PECCS 2014*, vol. 1, pp. 112–119, VISIGRAPP, 2014.

[6] J. J. Rodríguez-Andina, M. D. Valdes-Pena, and M. J. Moure, “Advanced features and industrial applications of fpgas—a review,” *IEEE Transactions on Industrial Informatics*, vol. 11, no. 4, pp. 853–864, 2015.

[7] F. Eberli, “Next generation fpgas and socs-how embedded systems can profit,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 610–613, 2013.

[8] Y. Han and E. Oruklu, "Real-time traffic sign recognition based on zynq fpga and arm socs," in *Electro/Information Technology (EIT), 2014 IEEE International Conference on*, pp. 373–376, IEEE, 2014.

[9] L.-D. Van, D.-Y. Wu, and C.-S. Chen, "Energy-efficient fastica implementation for biomedical signal separation," *IEEE transactions on neural networks*, vol. 22, no. 11, pp. 1809–1822, 2011.

[10] Y.-K. Wang and H.-Y. Chen, "The design of background subtraction on reconfigurable hardware," in *Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP), 2012 Eighth International Conference on*, pp. 182–185, IEEE, 2012.

[11] H. Heo, J.-y. Lee, K.-y. Lee, and C.-h. Lee, "Fpga based implementation of fast and brief algorithm for object recognition," in *TENCON 2013-2013 IEEE Region 10 Conference (31194)*, pp. 1–4, IEEE, 2013.

[12] K. May and N. Krouglicof, "Moving target detection for sense and avoid using regional phase correlation," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pp. 4767–4772, IEEE, 2013.

[13] K. Schmid and H. Hirschmüller, "Stereo vision and imu based real-time ego-motion and depth image computation on a handheld device," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pp. 4671–4678, IEEE, 2013.

[14] J. Ranjith and N. Muniraj, "Fpga implementation of optimized independent component analysis processor for biomedical application," in *Computer Communication and Informatics (ICCCI), 2013 International Conference on*, pp. 1–5, IEEE, 2013.

[15] J. Nikolic, J. Rehder, M. Burri, P. Gohl, S. Leutenegger, P. T. Furgale, and R. Siegwart, "A synchronized visual-inertial sensor system with fpga pre-processing for accurate real-time slam," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pp. 431–437, IEEE, 2014.

[16] M. Birem and F. Berry, "Dreamcam: A modular fpga-based smart camera architecture," *Journal of Systems Architecture*, vol. 60, no. 6, pp. 519–527, 2014.

[17] G. Zhou, J. Ye, W. Ren, T. Wang, and Z. Li, "On-board inertial-assisted visual odometer on an embedded system," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pp. 2602–2608, IEEE, 2014.

[18] R. Szabo and A. Gontean, "Robotic arm movement using color detection with fpga vision," in *Design and Technology in Electronic Packaging (SIITME), 2014 IEEE 20th International Symposium for*, pp. 117–122, IEEE, 2014.

[19] H. Tabkhi, M. Sabbagh, and G. Schirner, "A power-efficient fpga-based mixture-of-gaussian (mog) background subtraction for full-hd resolution," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pp. 241–241, IEEE, 2014.

[20] P.-Y. Hsiao, S.-Y. Lin, and S.-S. Huang, "An fpga based human detection system with embedded platform," *Microelectronic Engineering*, vol. 138, pp. 42–46, 2015.

[21] E. Calvo-Gallego, S. Sánchez-Solano, and P. B. Jiménez, "Hardware implementation of a background substraction algorithm in fpga-based platforms," in *Industrial Technology (ICIT), 2015 IEEE International Conference on*, pp. 1688–1693, IEEE, 2015.

[22] A. Nikitakis, I. Papaefstathiou, K. Makantasis, and A. Doulamis, "A novel background subtraction scheme for in-camera acceleration in thermal imagery," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pp. 1497–1500, IEEE, 2016.

[23] W. Hu, T. Tan, L. Wang, and S. Maybank, "A survey on visual surveillance of object motion and behaviors," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 34, no. 3, pp. 334–352, 2004.

[24] S. Y. Elhabian, K. M. El-Sayed, and S. H. Ahmed, "Moving object detection in spatial domain using background removal techniques-state-of-art," *Recent patents on computer science*, vol. 1, no. 1, pp. 32–54, 2008.

[25] A. Hyvärinen, J. Karhunen, and E. Oja, *Independent component analysis*, vol. 46. John Wiley & Sons, 2004.

[26] G. R. Naik, "Introduction: Indepent component analysis," in *Independent Component Analysis for Audio and Biosignal Applications* (G. R. Naik, ed.), pp. 3–22, Intech, 2012.

[27] "Zedboard." `http://zedboard.org/product/zedboard`. Accesado: 2017-04-07.

[28] "Xillybus an fpga ip core for easy dma over pcie with windows and linux." `http://xillybus.com/`. Accesado: 2017-04-07.

[29] "Xillinux a linux distribution for zedboard, zybo, microzed and sockit." `http://xillybus.com/xillinux`. Accesado: 2017-04-07.

[30] "Vivado Design Suit." `https://www.xilinx.com/products/design-tools/vivado.html`. Accesado: 2017-04-07.

[31] "Python welcome to python.org." `https://www.python.org/`. Accesado: 2017-04-07.

[32] D. Thirde, J. Ferryman, and J. L. Crowley, "Performance Evaluation of Tracking and Surveillance (PETS)." `http://pets2006.net/`. Accesado: 2010-02-05.

[33] F. Carrizosa-Corral, A. Vázquez-Cervantes, J.-R. Montes, T. Hernández-Díaz, L. Barriga-Rodríguez, J. A. Soto-Cajiga, and H. Jiménez-Hernández, "Ica-based background subtraction method for an fpga-soc," in *Electronic Imaging 2017, IS&T International Symposium on*, Society for Imaging Science and Technology, 2017. En prensa.

Apéndice A

Información adicional sobre la implementación.

A.1. Pipeline de FastICA

La Figura A.1 muestra el flujo de datos correspondiente a la ecuación (2.2). Aquí se puede apreciar la importancia de la reformulación de la expresión principal de FastICA, al permitir que el cálculo de la contribución de cada pixel se realice de forma que se acopla directamente a la estructura de un pipeline.

Antes de llevar esta expresión a VHDL se deben analizar los cambios en el punto fijo que surgen con cada operación. La Figura A.2 se detalla el tamaño y alineación del punto fijo en los diferentes registros utilizados, señalando las partes donde es necesario ajustarlos como preparación para una operación posterior. Inicialmente, se consideró utilizar dos bits fraccionales para la media de $\mathbf{x}_1$ y $\mathbf{x}_2$, 23 para la matriz de decorrelación y 31 para el vector $\mathbf{w}_1$. Posteriormente se desarrolló el código en VHDL buscando conservar la mayor precisión posible considerando estas entradas y a partir de ese punto se redujo hasta que fue posible realizar la implementación con los recursos que tiene el FPGA.

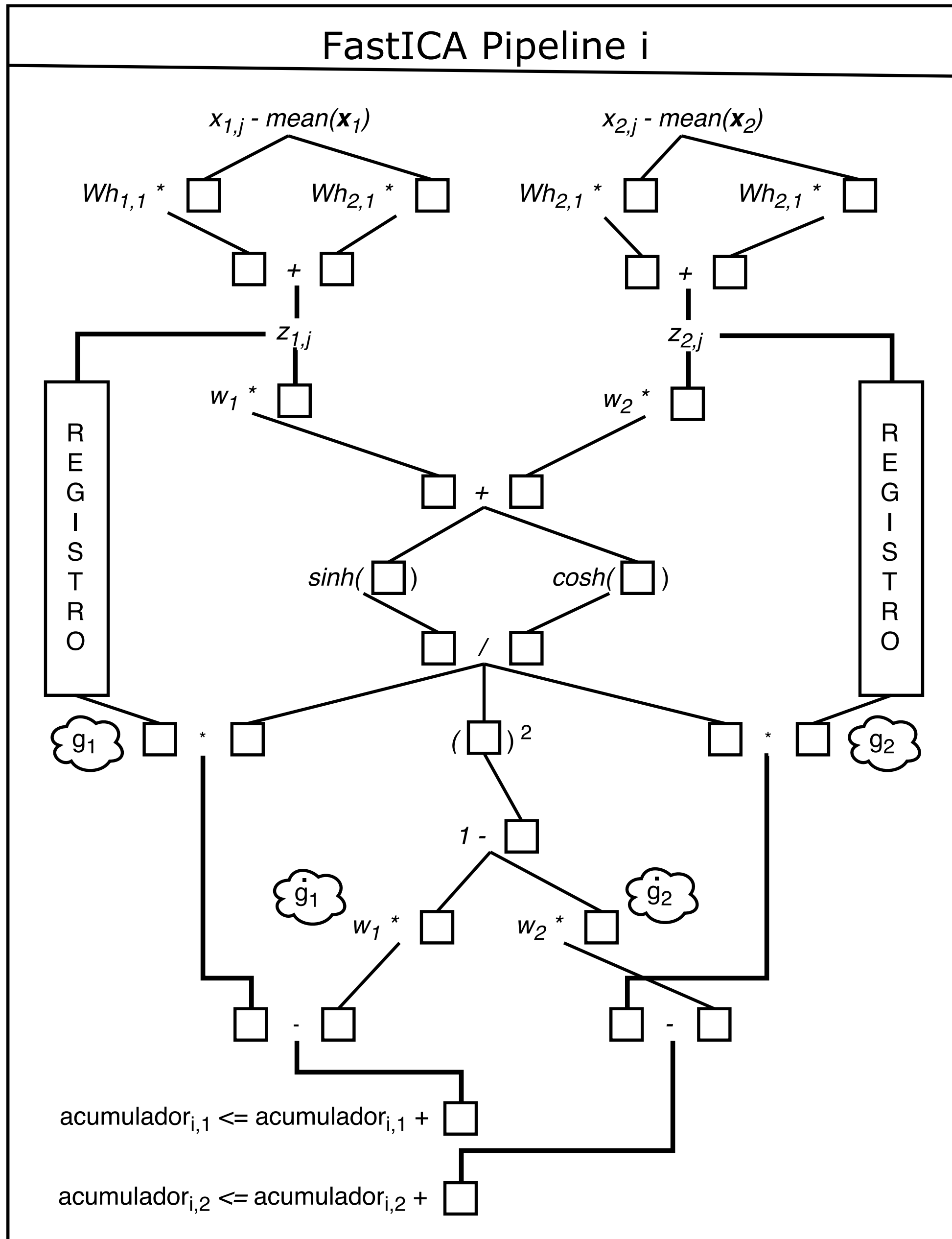
Finalmente se deben analizar los tiempos de ejecución de cada operación para sincronizar el flujo de datos y así garantizar que la información de cada pixel se procese por separado y que no se intente leer el resultado hasta que la operación haya terminado. Los tiempos resultantes de la implementación en el FPGA se resumen en la Tabla A.1.

Tabla A.1: Tiempos de ejecución por operación en el pipeline de FastICA.

# de ciclo	Tareas	
1	Preparar $x_{k,j} - mean\{\mathbf{x}_k\}$	
2	Realizar resta y preparar $y_{l,k,j} = Wh_{l,k} * (x_{k,j} - mean(\mathbf{x}_k))$	
3	Realizar multiplicación	
6	Leer resultado y preparar suma $z_{k,j} = y_{1,k,j} + y_{2,k,j}$	
7	Realizar suma y preparar $w_k * z_{k,j}$	
8	Realizar multiplicación e introducir $z_{k,j}$ al registro de corrimiento	
15	Leer resultado	
16	Suma $theta_j = ((w_1 * z_{1,j}) + (w_2 * z_{2,j}))$	
17	Preparar $sinh(theta_j)$ y $cosh(theta_j)$	
18	Realizar funciones hiperbólicas	
54	Leer resultados y preparar $tanh(theta_j) = sinh(theta_j)/cosh(theta_j)$	
55	Realizar división	
91	Leer resultado y preparar $tanh(theta_j * tanh(theta_j)$ y $z_{k,j} * tanh(theta_j)$	
92	Realizar $tanh2(theta_j) = tanh(theta_j)^2$	Realizar $g_j = z_{k,j} * tanh(theta_j$
99	Leer $tanh2(theta_j)$	
100	Preparar $1 - tanh2(theta_j)$	
101	Realizar y preparar $gp_{k,j} = w_k * (1 - tanh2(theta_j))$	
102	Realizar multiplicación	
110	Leer $gp_{k,j}$	Leer $g_{k,j}$
111	Realizar resta $g_{k,j} - gp_{k,j}$	
112	Leer y acumular $g_{k,j} - gp_{k,j}$	

Entrada 1 [$Wh_{1,1}$ $Wh_{1,2}$ $Wh_{2,1}$ $Wh_{2,2}$]

Entrada 2 [w_1 w_2]

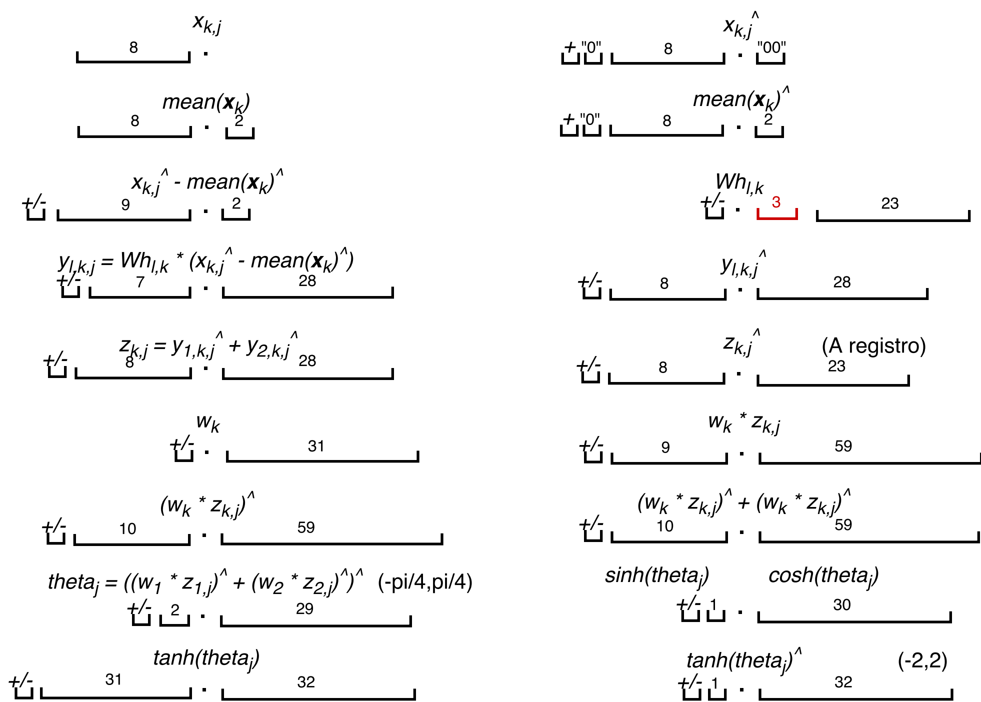


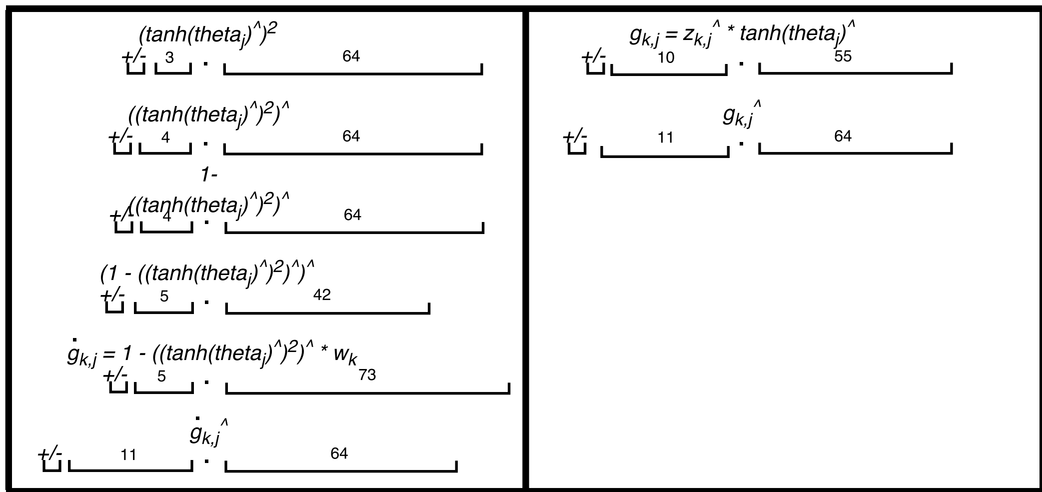
w_1 <= acumulador$_{1,1}$ + acumulador$_{2,1}$ + ...

w_2 <= acumulador$_{1,2}$ + acumulador$_{2,2}$ + ...

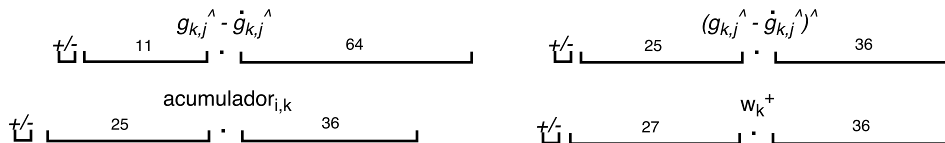
Salida [w_1 w_2]

Figura A.1: Diagrama de flujo del pipeline de FastICA.





(Los siguiente cálculos corresponden a una imagen de 320x200 pixeles y 4 pipelines)



^ - Indica un cambio en el tamaño del registro como preparación para una operación posterior.

rojo - Señala bits se pueden omitir del registro sin pérdida de información.

Figura A.2: Diagrama del punto fijo en el pipeline de FastICA.

Tabla A.2: Canales de comunicación utilizados.

Nombre	Bits	Dirección	Descripción
xillybus_write_x1	32	Procesador a FPGA	Enviar imagen actual
xillybus_read_x1	32	FPGA a procesador	Repetir imagen actual (depurar)
xillybus_write_x2	32	Procesador a FPGA	Enviar estimación del fondo
xillybus_read_x2	32	FPGA a procesador	Repetir estimación (depurar)
xillybus_read_cov	32	FPGA a procesador	Enviar covarianza
xillybus_write_wh	8	Procesador a FPGA	Enviar matriz de decorrelación
xillybus_write_w	32	Procesador a FPGA	Enviar vector $\mathbf{w}_i$
xillybus_read_w	32	FPGA a procesador	Enviar vector $\mathbf{w}_i^+$

A.2. Comunicación FPGA-procesador

En la Tabla A.2 se resumen los canales de comunicación que se utilizaron en la implementación. El Listado A.1 muestra un fragmento del código en Python para ilustrar cómo se interactua con estos canales desde el procesador. Por otro lado, el Listado A.2 muestra un fragmento de código en VHDL en referencia a cómo realizar una lectura mientras que el Listado A.3 trata con la escritura.

Listado A.1: Interacción con los canales de comunicación en Python.

```
import io
import struct


# Codificar mensaje
WFP = w*2147483648.0 # Ajustar punto fijo
WFP_0=long(WFP[0])
WFP_1=long(WFP[1])


WFPwrite = ''
A=''
```

```
t=struct.pack('l', WFP_0)
A=''.join((A, t ))
t=struct.pack('l', WFP_1)
A=''.join((A, t ))

WFPwrite=A

#Enviar w
fifo_w_W=io.open('/dev/xillybus_write_w','wb')
fifo_w_W.write(WFPwrite)
fifo_w_W.flush()
fifo_w_W.close()

#Leer nueva w
fifo_r_W=io.open('/dev/xillybus_read_w','rb',16)
newWMsg=fifo_r_W.read(16) # Bytes a leer
fifo_r_W.close()

# Decodificar w
w_1=struct.unpack('q',newWMsg[0:8])
w_1=float(w_1[0])
w_2=struct.unpack('q',newWMsg[8:16])
w_2=float(w_2[0])
```

Listado A.2: Lectura de un canal de comunicación en VHDL.

```
-- Recibir matriz Wh
process (bus_clk)
  begin
  if (bus_clk'event and bus_clk = '1') then
  case fifo_read_white_state_reg is
```

```
    when idle =>
      fifo_read_white_delay_reg <='0';
      fifo_read_white_cont_reg <= (others => '0');
      u_r_wh_en_reg <= '0';
      fifo_read_white_state_reg <= idle;
      if(u_r_wh_empty = '0') then
        fifo_read_white_buff_cont_reg <=
        fifo_read_white_buff_cont_reg + 1;
      end if;
      if(unsigned(fifo_read_white_buff_cont_reg) =
      TO_UNSIGNED(100,14)) then
        fifo_read_white_buff_cont_reg <= (others => '0');
        fifo_read_white_state_reg <= dumping;
      end if;
    when dumping =>
      fifo_read_white_state_reg <= dumping;

  if(u_r_wh_empty = '0') then
    u_r_wh_en_reg <= '1';
    fifo_read_white_cont_reg <= fifo_read_white_cont_reg +1;
  else
    u_r_wh_en_reg <= '0';
  end if;

  if(u_r_wh_en_reg = '1') then
    fifo_white_msg_reg(95 downto 88) <= u_r_wh_data(7 downto 0);
    fifo_white_msg_reg(87 downto 0) <=
    fifo_white_msg_reg(95 downto 8);
  end if;
```

```
      if(unsigned(fifo_read_white_cont_reg) =
      TO_UNSIGNED(12,14)) then
        fifo_read_white_cont_reg <= (others => '0');
        fifo_read_white_delay_reg <= '1';
      end if;

      if(fifo_read_white_delay_reg = '1') then
        fifo_read_white_state_reg <= idle;
      end if;
      end case;

      white_1_1_reg <= signed(fifo_white_msg_reg(23 downto 0));
      white_1_2_reg <= signed(fifo_white_msg_reg(47 downto 24));
      white_2_1_reg <= signed(fifo_white_msg_reg(71 downto 48));
      white_2_2_reg <= signed(fifo_white_msg_reg(95 downto 72));

      end if;
    end process;

  u_r_wh_en  <= u_r_wh_en_reg;

  fifo_wh_instance : fifo_8bit_ipcore
  port map(
    rst          => reset_r_wh,
    wr_clk        => bus_clk,
    rd_clk        => bus_clk,
    din           => user_w_write_wh_data,
    wr_en         => user_w_write_wh_wren,
```

```
  rd_en        => u_r_wh_en,
  dout         => u_r_wh_data,
  full         => user_w_write_wh_full,
  empty        => u_r_wh_empty
  );

reset_r_wh <= not user_w_write_wh_open;
```

Listado A.3: Escritura a un canal de comunicación en VHDL.

```
  process (bus_clk)
    variable temp_msg:  std_logic_vector(127 downto 0);
    variable temp_payload:  std_logic_vector(31 downto 0);
    begin
    if (bus_clk'event and bus_clk = '1') then
    case fifo_cov_state_reg is
      when idle =>
        fifo_cov_state_reg <= idle;
        fifo_cov_cont_reg <= (others => '0');
        fifo_cov_en_reg <= '0';
        fifo_cov_data_reg <= (others => '0');

        if begin_send_cov_reg='1' then
          fifo_cov_state_reg <= dumping;

          -- Media de x1
          temp_msg(39 downto 32) := std_logic_vector(mean_x1);
          -- Varianza de x1
          temp_msg(31 downto 0) :=
          std_logic_vector(sum_xx_reg(37 downto 6));
```

```
        -- Media de x2
        temp_msg(79 downto 72) := std_logic_vector(mean_x2);
        -- Varianza de x2
        temp_msg(71 downto 40) :=
        std_logic_vector(sum_yy_reg(37 downto 6));
        -- Covarianza de x1 y x2
        temp_msg(111 downto 80) :=
        std_logic_vector(sum_xy_reg(37 downto 6));

        fifo_cov_msg_reg <= temp_msg;
      end if;
    when dumping =>
      fifo_cov_state_reg <= dumping;
      if (fifo_cov_cont_reg= TO_UNSIGNED(4,8)) then
        fifo_cov_state_reg <= idle;
        fifo_cov_cont_reg <= (others => '0');
        fifo_cov_en_reg <= '0';
        fifo_cov_data_reg <= (others => '0');
      else
        fifo_cov_cont_reg <= fifo_cov_cont_reg +1;
        temp_payload := fifo_cov_msg_reg(31 downto 0);
        fifo_cov_en_reg <= '1';
        fifo_cov_data_reg <= temp_payload;
        fifo_cov_msg_reg(127 downto 96) <= (others => '0');
        fifo_cov_msg_reg(95 downto 0) <=
        fifo_cov_msg_reg(127 downto 32);
      end if;
  end case;
  end if;
```

```
    end process;

    u_w_cov_data <= fifo_cov_data_reg;
    u_w_cov_en   <= fifo_cov_en_reg;

    fifo_cov_instance : fifo_32bit_ipcore
    port map(
      rst         => reset_w_cov,
      wr_clk      => bus_clk,
      rd_clk      => bus_clk,
      din         => u_w_cov_data,
      wr_en       => u_w_cov_en,
      rd_en       => user_r_read_cov_rden,
      dout        => user_r_read_cov_data,
      full        => u_w_cov_full,
      empty       => user_r_read_cov_empty
      );

  reset_w_cov <= not user_r_read_cov_open;
```

Tabla A.3: Recursos de hardware del FPGA utilizados en la implementación.

Recurso	**Usado**	**Disponible**	**%**
Look up tables	47808	53200	89
Flip flops	64040	106400	60
Bloques RAM	69.5	140	49
DSP	80	220	36

A.3. Uso de recursos

En la Tabla A.3 se muestra el porcentaje de uso de los recursos de hardware más relevantes del FPGA. Las dos principales medidas que se pueden tomar para reducir el uso de recursos son realizar una optimización del código en VHDL y analizar el impacto en la precisión del punto fijo para determinar cuáles registros se pueden reducir en tamaño sin afectar drásticamente la calidad de los resultados obtenidos. Para esta última medida se puede utilizar la simulación en Matlab, modificando la resolución de las variables correspondientes a los registros luego de cada operación. Por ejemplo, el Listado A.4 muestra como simular un registro con ocho bits para la parte fraccional.

Listado A.4: Simulación de un registro con punto fijo.

```
puntoFijo=8;
registroEjemplo=fix(registroEjemplo*(2^puntoFijo))/(2^puntoFijo);
```