ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Grado

CO-DISEÑO HARDWARE-SOFTWARE DE UN SISTEMA DE POSICIONAMIENTO BASADO EN PROCESADO DE VÍDEO

(Hardware-Software Codesign of a Positioning System based on Video Processing)

Para acceder al Título de

Graduado en Ingeniería de Tecnologías de Telecomunicación

Autor: Ángel Álvarez Ruiz

Julio - 2016

E.T.S. DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACION

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE GRADO

Realizado por: **Ángel Álvarez Ruiz**

Director del TFG: **Víctor Fernández Solórzano**
Íñigo Ugarte Olano

Título: "Co-diseño hardware-software de un sistema de posicionamiento basado en procesado de vídeo"

Title: **"Hardware-Software Codesign of a Positioning System based on Video Processing"**

Presentado a examen el día: **22 de Julio de 2016**

para acceder al Título de

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Composición del Tribunal:

Presidente (Apellidos, Nombre):	Villar Bonet, Eugenio
Secretario (Apellidos, Nombre):	Fernández Solórzano, Víctor
Vocal (Apellidos, Nombre):	Mediavilla Bolado, Elena

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente	Fdo.: El Secretario
Fdo.: El Vocal	Fdo.: El Director del TFG (sólo si es distinto del Secretario)
Vº Bº del Subdirector	Trabajo Fin de Grado Nº (a asignar por Secretaría)

Dedicado a
mis padres
Yolanda y Ángel

Agradecimientos

Deseo manifestar mi más sincero agradecimiento a Víctor Fernández e Íñigo Ugarte, directores de este trabajo, por su interés, sus consejos y su inestimable ayuda en la realización del mismo.

A los integrantes del Grupo de Ingeniería Microelectrónica de la Universidad de Cantabria, en especial a Patricia Martínez, Álvaro Díaz, Alejandro Nicolás y Héctor Posadas, por estar siempre dispuestos a echar una mano y conseguir que el ambiente de trabajo fuera agradable desde el primer día.

Por último, a mi familia, a Marta y a mis amigos, que han formado parte de mi día a día y me han animado y apoyado en todo momento.

Palabras clave

Co-diseño HW-SW, Zynq, HLS (Síntesis de alto nivel), OpenCV, Linux

Keywords

HW-SW Codesign, Zynq, HLS (High Level Synthesis), OpenCV, Linux

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Contexto

Durante los últimos años, ha habido un creciente interés en sistemas relacionados con el posicionamiento de objetos en entornos tridimensionales y aplicaciones de realidad virtual y realidad aumentada. Su uso se ha extendido en sectores muy diversos, como robótica, medicina y ocio. Para obtener la posición del usuario, estos sistemas se basan en el uso de diferentes elementos, como cámaras, sensores ópticos, acelerómetros, giroscopios, GPS, etc.

En concreto, resulta de interés en relación con este trabajo la realidad aumentada, que consiste en ofrecer una visión del entorno físico a través de un dispositivo electrónico (gafas, pantalla) combinándola con elementos virtuales que permiten ampliar información sobre el mundo real alrededor del usuario e interactuar con ellos. De esta forma se crea una realidad mixta en tiempo real. En estos sistemas es fundamental el conocimiento de la posición del usuario; como integran un sistema de visión (compuesto por una o más cámaras) puede aprovecharse obtener la posición mediante sistemas de visión por computador que detectan elementos del entorno como bordes o marcadores de referencia.

El sistema de posicionamiento que se implementa en este trabajo consigue localizar un objetivo (usuario, robot, etc) equipado con una cámara en un entorno tridimensional que comprende una serie de marcadores luminosos LED, llamados marcadores de referencia. Su funcionamiento está basado en procesado de imagen de la región donde se encuentra el individuo usando algoritmos para detectar marcadores de referencia. Después, a través de triangulación y conociendo la distancia real entre marcadores, es posible obtener los parámetros para calcular la profundidad de la cámara y posicionarla en el entorno.

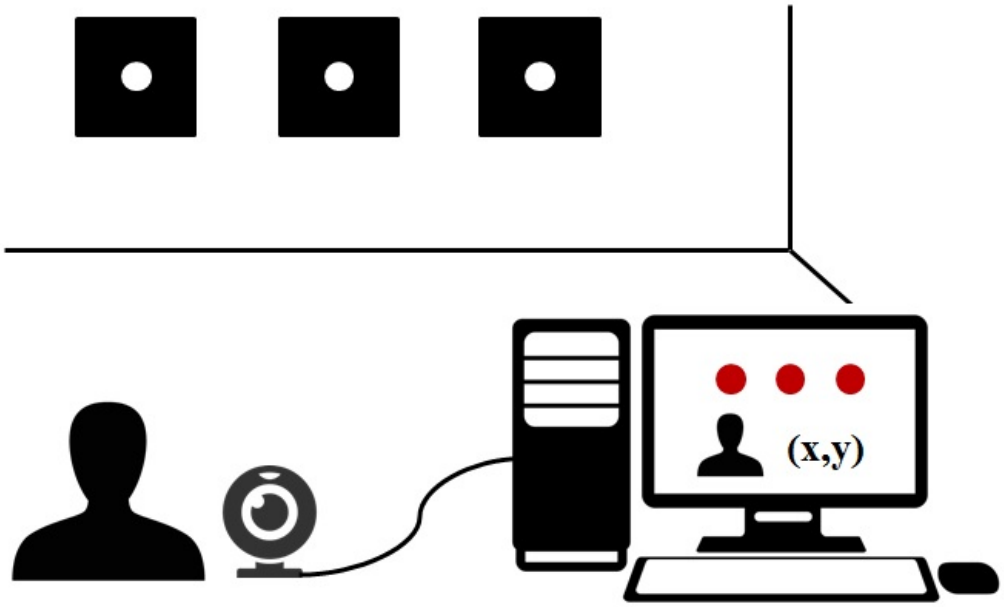

Figura 1.1: Entorno de posicionamiento que comprende 3 marcadores luminosos LED y el equipo necesario (PC) para ejecutar el algoritmo de posicionamiento y localizar a un usuario equipado con una cámara, mediante procesado de imagen.

Este método es objeto de una patente de la Universidad de Cantabria [2] y ha sido desarrollado en C++, tratándose de una versión puramente software y probada en un PC de escritorio con sistema operativo Linux. En la figura 1.1 se muestra el entorno de posicionamiento y el sistema de detección de marcadores (una cámara y un PC ejecutando el algoritmo) para ubicar al objetivo.

1.2. Motivación

Los sistemas descritos anteriormente se basan en procesado de vídeo de altas prestaciones y tienen una alta carga computacional, especialmente en las fases de análisis de imagen. Son aplicaciones que deben funcionar en tiempo real, de manera que posicionen al usuario de forma instantánea y con una respuesta lo más rápida posible, por lo que necesitan ejecutarse en equipos de alto rendimiento.

El problema que se plantea es que el producto final no puede depender de un PC para ejecutar los algoritmos de posicionamiento basados en procesado

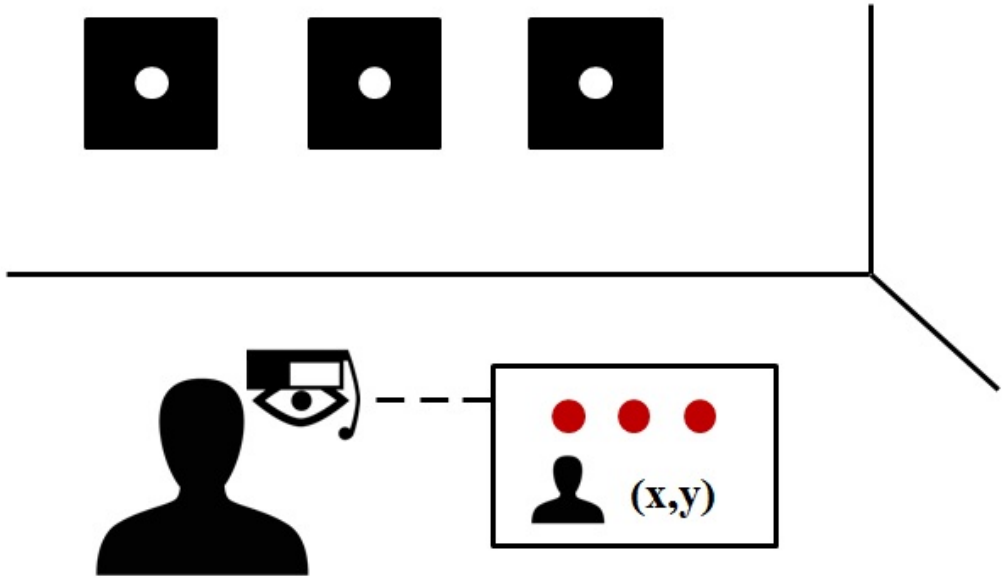

Figura 1.2: Entorno de posicionamiento que comprende 3 marcadores luminosos LED para localizar a un usuario equipado con una cámara y un sistema embebido (por ejemplo, unas gafas de realidad virtual), mediante procesado de imagen.

de vídeo, porque las aplicaciones que se persiguen requieren una plataforma portátil que aporte al usuario movilidad e independencia de una toma de corriente. Por ejemplo, las aplicaciones de realidad virtual y aumentada están pensadas para ejecutarse en plataformas como gafas/cascos de realidad virtual, smartphones o tablets, etc.

Por tanto, la utilidad de estos sistemas pasa por la posibilidad de implementarlos en una plataforma embebida que forme parte de un equipo pequeño y de bajo consumo. Para conseguir un equilibrio entre prestaciones, tamaño y consumo de energía, la elección de la plataforma y el diseño hardware a medida de ciertas partes del sistema son fundamentales. En la figura 1.2 se representa el entorno y sistema de posicionamiento explicado en el apartado anterior ejecutándose en una plataforma embebida como unas gafas de realidad virtual, independizando al usuario de la utilización de un ordenador.

La solución que se presenta en este trabajo para permitir una implementación como la que se ha comentado consiste en portar el sistema de posicionamiento basado en detección de marcadores, que se ejecuta en un PC, a una placa de desarrollo (Digilent ZedBoard) equipada con un Zynq-7000 All

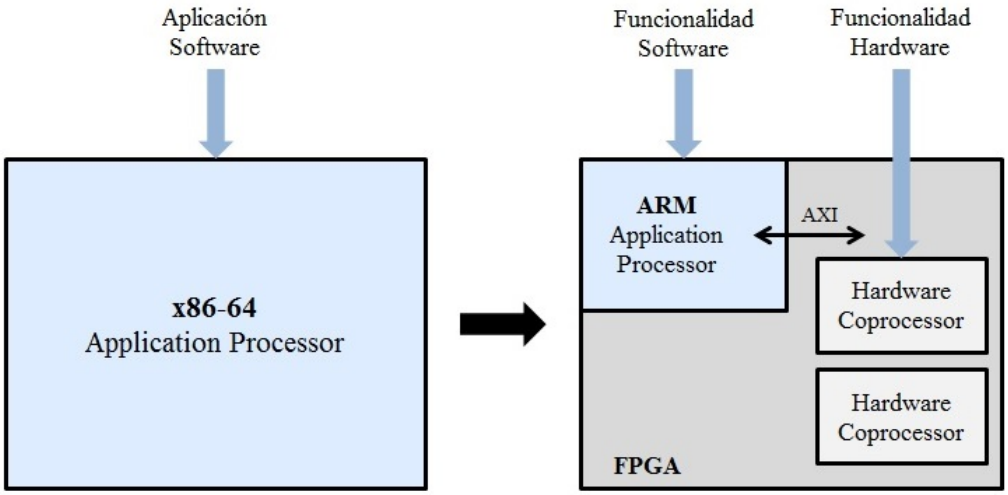

Figura 1.3: El sistema de posicionamiento se porta desde la arquitectura original (procesador x86-64 convencional de PC) a una arquitectura destinada a un sistema embebido de bajo consumo (Xilinx Zynq-7000 AP SoC).

Programmable SoC de Xilinx (figura 1.3). Este dispositivo integra un microprocesador de propósito general basado en ARM capaz de ejecutar sistemas operativos completos y lógica programable equivalente a la de una FPGA convencional. A través de interfaces AXI estándar, proporciona conexiones de alto ancho de banda y baja latencia entre ambas partes del dispositivo. Esto significa que el procesador y la lógica programable pueden usarse para lo que cada uno hace mejor, sin la sobrecarga de interconectar dos dispositivos separados físicamente. Además, los beneficios de integrar el sistema en un único chip incluyen reducciones de tamaño, consumo energético y coste total.

Para llevarlo a cabo, es necesario aprovechar las capacidades hardware y software que proporciona el SoC Zynq, ya que el ARM Cortex A9 que incorpora proporciona un rendimiento demasiado bajo para ejecutar una versión del sistema completamente software sobre Linux. Se comienza realizando un análisis software para evaluar las prestaciones y detectar los cuellos de botella del algoritmo de detección de marcadores de referencia. Después, es posible conseguir una aceleración de las partes con mayor carga computacional a través del diseño hardware a medida de subsistemas que realicen estas tareas. Por último, se integran las funcionalidades hardware y software para utilizarlas de manera conjunta en el sistema y conseguir un aprovechamiento óptimo de la arquitectura final.

1.3. Objetivos

Este trabajo presenta el proceso de co-diseño hardware-software e implementación de un sistema de posicionamiento basado en procesado de vídeo, partiendo de una versión puramente software desarrollada en lenguaje C++. La plataforma final es el SoC Zynq-7000 de Xilinx, en la placa de desarrollo Digilent ZedBoard.

A partir de un análisis detallado del rendimiento del sistema, se lleva a cabo una aceleración de las partes más lentas o cuellos de botella, combinando las capacidades hardware y software de la plataforma final (Zynq), para permitir su implementación en un sistema embebido de bajo consumo y con unas prestaciones de tiempo real aceptables.

Por tanto, en cuanto a objetivos del trabajo, pueden considerarse:

- Como finalidad principal, conseguir mediante aceleración hardware una mejora significativa en el throughput global del sistema de posicionamiento sobre la plataforma final. Dicha plataforma debe ser adecuada para un dispositivo portátil de bajo consumo.
- En general, presentar y probar una metodología para la aceleración de aplicaciones de procesado de vídeo que utilizan funcionalidad de la librería de visión por computador OpenCV.
- Diseñar hardware dedicado de altas prestaciones para el procesado de vídeo utilizando síntesis de alto nivel.
- Integrar funcionalidades hardware y software en una plataforma embebida sobre el sistema operativo Linux.

1.4. Estructura del documento

En este primer capítulo, se presenta la importancia del proyecto exponiendo su contexto actual y la necesidad de diseñar sistemas que utilizan hardware dedicado y aprovechan al máximo la arquitectura sobre la que funcionan. Además, se hace un breve resumen del trabajo realizado y los objetivos que se han cubierto.

En el segundo capítulo, se explican algunos fundamentos teóricos necesarios para la comprensión del trabajo y que pueden ser consultados por el lector en caso de necesidad.

Los sucesivos apartados exponen, por orden, la metodología seguida para conseguir la implementación final del sistema. En el tercer capítulo se explica

el algoritmo de posicionamiento y su ejecución en la plataforma de destino, incluyendo el funcionamiento y puesta en marcha del sistema operativo Linux sobre Zynq y el análisis del rendimiento (profiling) del software, para llevar a cabo una partición de la funcionalidad entre hardware y software. En el cuarto capítulo se aborda el diseño de los bloques IP hardware que implementan la funcionalidad elegida, utilizando síntesis de alto nivel (Vivado High Level Synthesis). En el quinto capítulo se explica la integración de los módulos diseñados en el resto de la plataforma hardware y su manejo desde software en Linux. Finalmente, en el sexto capítulo se presentan los resultados de la ejecución del sistema acelerado por hardware en comparación con el sistema software original. El séptimo capítulo expone las conclusiones del trabajo.

Capítulo 2

Conceptos previos

2.1. Zynq-7000 All Programmable SoC

Un System-on-Chip (SoC) es un único circuito integrado de silicio que implementa la funcionalidad de un sistema completo, en lugar de necesitarse varios chips físicos diferentes interconectados entre sí. Hablando de sistemas digitales, se puede combinar procesado, lógica de propósito específico, interfaces, memorias, etc.

Los dispositivos Zynq-7000 All Programmable SoC de Xilinx comprenden dos partes principales integradas en un único chip: un Sistema de Procesado (PS) basado en un microprocesador Dual-Core ARM Cortex A9 y Lógica Programable (PL) basada en la arquitectura de las FPGA 7-series de Xilinx. La comunicación entre el PS y la PL se realiza mediante interfaces AXI (Advanced eXtensible Interface) estándar, que proporcionan conexiones de alta velocidad de transmisión y baja latencia [3].

La parte de PL es ideal para implementar lógica de alta velocidad (para acelerar tareas con una alta carga computacional, como sistemas aritméticos, procesado de vídeo en tiempo real, etc) y el PS soporta sistemas operativos completos y rutinas software, de manera que la funcionalidad total de cualquier sistema puede ser adecuadamente particionada entre hardware y software.

Entre las potenciales aplicaciones de Zynq se incluyen las comunicaciones cableadas y wireless, automoción, procesado de imagen y vídeo, computación de altas prestaciones, medicina, control industrial y muchas más. La arquitectura puede ser empleada para satisfacer las demandas de aplicaciones con requerimientos tanto de computación paralela de alta velocidad como de funcionalidad secuencial con uso intensivo del procesador.

A continuación se detalla la arquitectura del Zynq-7000 All Programmable SoC. Un esquema del dispositivo se muestra en la figura 2.1, detallando los componentes más importantes del PS y la PL. Puede encontrarse más información en [4].

Sistema de Procesado (PS)

La base del sistema de procesado de todos los dispositivos Zynq es un microprocesador Dual-Core ARM Cortex A9 que puede operar a frecuencia de reloj de hasta 1 GHz, dependiendo del modelo Zynq concreto. Es un procesador "hard", es decir, existe dentro del SoC como un elemento de silicio dedicado y optimizado (en contraposición a los procesadores "soft" que se implementan utilizando la lógica programable de la FPGA, como el Xilinx MicroBlaze).

La Application Processing Unit (APU) está formada por los dos núcleos del procesador ARM, cada uno con una serie de elementos adicionales: un coprocesador NEON y una unidad de punto flotante (FPU), una MMU (Memory Management Unit) y memoria caché L1 y L2.

Además de la APU, el sistema de procesado incluye interfaces para periféricos de E/S (USB, GigE, UART, I2C, SPI...), memoria cache, interfaces de memoria, hardware de interconexión y circuitería de generación de reloj.

Lógica Programable (PL)

La lógica programable de Zynq está basada en la arquitectura de las FPGA Artix-7 y Kintex-7 de Xilinx. Está predominantemente compuesta de CLBs (Configurable Logic Blocks), que contienen dos *slices* cada uno. Los *slices* están formados por 4 LUTs (Look-Up Tables), 8 Flip-Flops y lógica adicional. Para proporcionar una interfaz entre la lógica y los pines físicos del dispositivo, existen IOBs (Input/Output Blocks) alrededor del perímetro de la PL.

Adicionalmente a la lógica general, hay dos tipos de recursos de propósito especial: memorias Block RAM dedicadas de 36 kB o 18 kB (capaces de implementar RAMs, ROMs y bufferes FIFO) y DSPs (Digital Signal Processors) de tipo DSP48E1 para aritmética de alta velocidad.

La PL recibe cuatro entradas separadas de reloj desde el PS y, adicionalmente, tiene la posibilidad de generar y distribuir sus propias señales de reloj independientemente del PS.

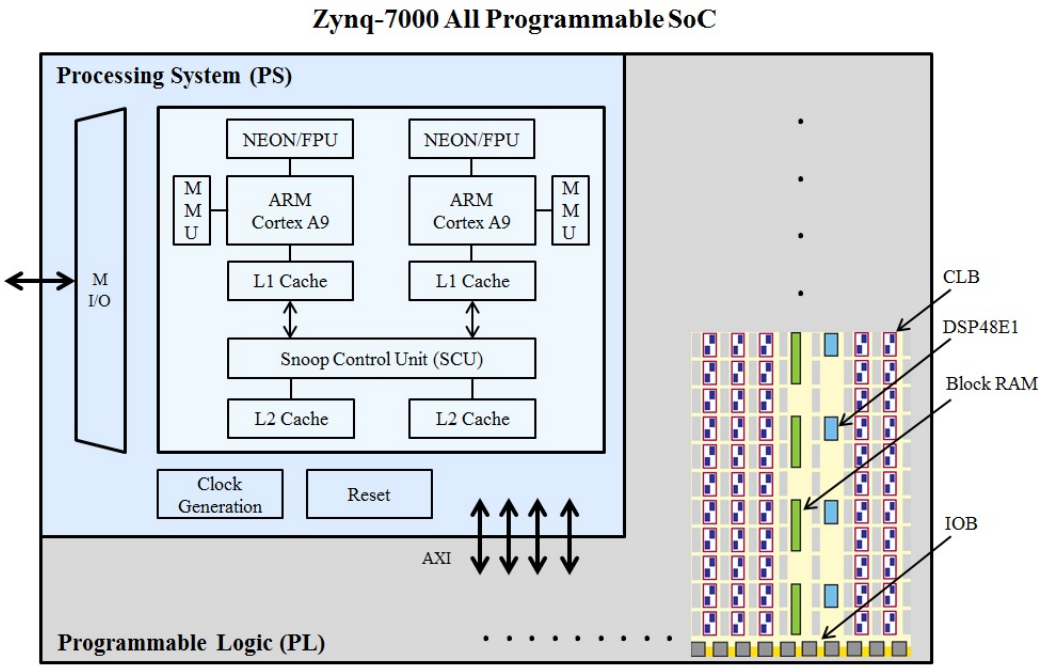

Figura 2.1: Arquitectura del Zynq-7000 All Programmable SoC, mostrando los componentes más importantes del Sistema de Procesado (PS) y la Lógica Programable (PL).

2.2. Placa de desarrollo ZedBoard

Una de las placas de desarrollo más populares que integran un SoC Zynq-7000 es la ZedBoard [5]. Es una placa de bajo coste y con una gran comunidad de usuarios que incluye un dispositivo Zynq XC7Z020, fruto de una asociación entre Xilinx, Avnet (el distribuidor) y Digilent (el fabricante de la placa). La implementación y ejecución final del presente trabajo se han realizado sobre una ZedBoard. La placa puede verse en la figura 2.2.

La ZedBoard cuenta con numerosas interfaces, entre ellas salida de vídeo HDMI y VGA, entradas y salidas de audio, conector Ethernet y slot para tarjetas SD. Además, dispone de un codec de audio y un transmisor de HDMI de Analog Devices. Tiene 3 interfaces USB: un USB-OTG para conectar periféricos, un USB-JTAG para programar el dispositivo Zynq desde el PC y un USB-UART para permitir la comunicación por puerto serie con un PC.

Incluye 512 MB de memoria DDR3 y dos osciladores para generar señales de reloj (uno de 100 MHz y otro de 33.3333 MHz). El usuario puede especificar el método de programación y arranque de la placa por medio de una serie de *jumpers* posicionados justo encima del logo de Digilent.

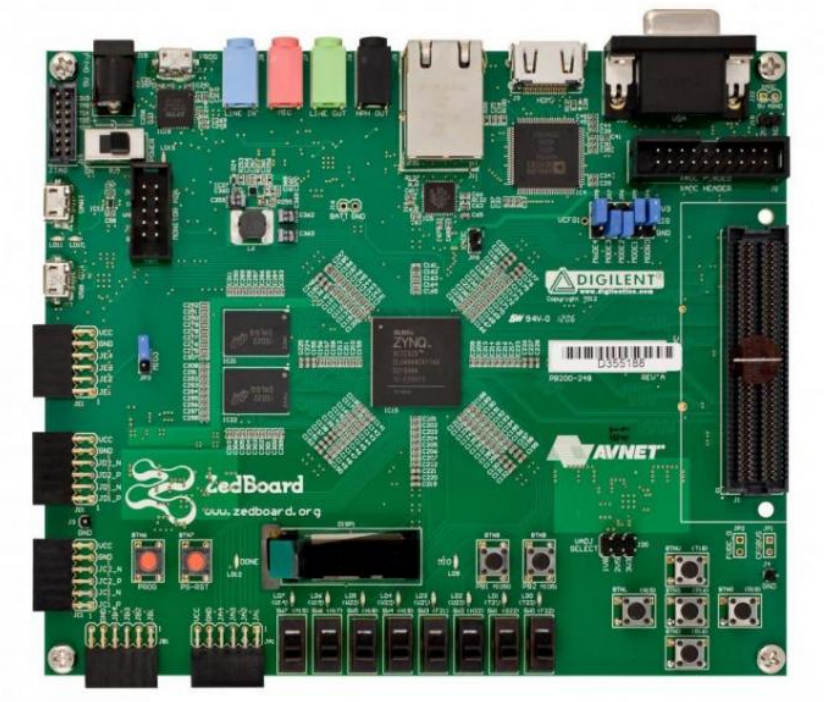

Figura 2.2: Placa de desarrollo Digilent ZedBoard (utilizada en este trabajo), que integra un dispositivo Zynq XC7Z020 (situado en el centro de la PCB).

2.3. Fundamentos de imagen digital

Las imágenes digitales consisten en una matriz de datos representando una cuadrícula de píxeles o puntos de color. Cada elemento de la matriz almacena el valor con el que se codifica el color del píxel correspondiente. En los sistemas electrónicos, estos valores se representan con un determinado número de bits.

La resolución de imagen se refiere al número de píxeles en una imagen digital; más resolución implica mejor calidad. Por ejemplo, una imagen de tamaño 1600x1200 píxeles tiene más resolución que otra de 640x480. En este caso se ha utilizado una relación de aspecto de 4:3, pero no es necesario adoptar esta proporción.

La profundidad de color es la cantidad de bits de información necesarios para representar el color de un píxel en una imagen digital. Es frecuente representar cada píxel con 8 bits (2^8 colores) en imágenes en escala de grises y con 24 bits (2^{24} colores) en imágenes en color, utilizando 8 bits para cada uno de los 3 canales de color.

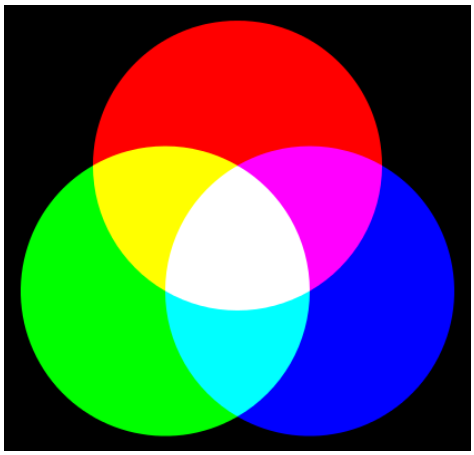

Figura 2.3: Modelo aditivo de colores RGB (rojo, verde y azul). Es posible formar un color añadiendo colores primarios en diferentes proporciones al negro.

El modelo de color más usado en imagen digital es el RGB (Red, Green, Blue), que hace referencia a la composición del color en términos de la intensidad de tres los colores primarios de la luz (figura 2.3). RGB es un modelo basado en la síntesis aditiva, con el que es posible representar cualquier color mediante la mezcla por adición de los tres colores de luz primarios (añadiendo colores al negro o ausencia de luz).

Considerando una imagen en escala de grises con profundidad de color de 8 bits, esto significa que cada píxel tiene un valor de gris entre 0 (negro) y 255 (blanco). Por ejemplo, un píxel oscuro puede tener un valor de 15 y uno brillante puede tener un valor de 240. Cada píxel se almacena como un byte, así que una imagen en escala de grises de tamaño 640x480 requiere 300 kB de almacenamiento (640 x 480 = 307200 bytes).

En una imagen en color con profundidad de color de 24 bits, cada píxel se representa por 3 bytes, generalmente 1 byte representando cada canal de color: R (rojo), G (verde) y B (azul). Un ejemplo puede verse en la figura 2.4. Como cada valor esta en el rango 0-255, este formato soporta 256 x 256 x 256 combinaciones diferentes, lo que hace un total de 16 777 216 colores. Sin embargo, esta flexibilidad conlleva mayor necesidad de almacenamiento; una imagen de 640x480 con color de 24 bits requiere 921.6 kB de almacenamiento sin ninguna compresión. Es importante conocer que muchas imágenes con color de 24 bits son almacenadas realmente como imágenes de 32 bits, con un byte extra de información para un canal adicional llamado canal alfa, que define la transparencia u opacidad del píxel en la imagen.

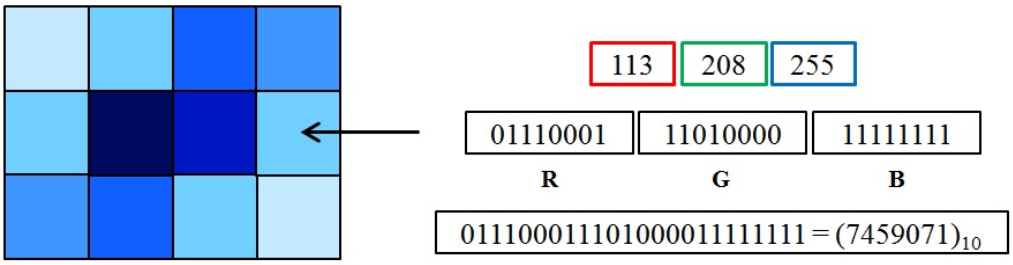

Figura 2.4: Ejemplo de codificación de un píxel en una imagen con 3 canales de color (RGB) y profundidad de color de 24 bits (8 bits por canal, por lo que el rango de valores es 0-255 para cada uno).

La mayoría de los formatos de archivo para almacenar imágenes incorporan alguna técnica de compresión debido al gran tamaño de almacenamiento que ocupan los datos de una imagen. El estándar más importante actualmente para compresión de imagen es JPEG, que utiliza un método de compresión de imagen con pérdidas basado en la DCT (Transformada Discreta del Coseno) para reducir el contenido de frecuencia espacial alta de la imagen (la información de las regiones en las que ocurren muchos cambios en solamente unos pocos píxeles).

Más información sobre teoría y modelos de color, técnicas de representación de gráficos y codificación y compresión de imagen puede encontrarse en [6].

2.4. OpenCV

OpenCV (Open Source Computer Vision) es una librería open source de visión por computador originalmente desarrollada por Intel. La librería está escrita en C y C++ y corre bajo Linux, Windows, Mac OS X, iOS y Android. Uno de los objetivos de OpenCV es proporcionar una infraestructura de visión artificial sencilla de usar para poder desarrollar aplicaciones bastante sofisticadas rápidamente. Contiene más de 500 funciones que abarcan muchas áreas, incluyendo control industrial, medicina, seguridad, interfaz de usuario, calibración de cámaras, visión estéreo y robótica [7]. La licencia open source BSD de OpenCV ha sido estructurada de manera que se puede construir un producto comercial usando OpenCV, sin obligación de que dicho producto sea open source o destinar una parte de los beneficios al dominio público (quedando a decisión del propietario). En parte, es debido a estos términos de licencia liberales que exista una gran comunidad de usuarios que incluye

personal de importantes compañías (Google, IBM, Intel, Nvidia...) y centros de investigación (Standford, MIT...).

La visión por computador o visión artificial es la transformación de datos desde imágenes o vídeos 2D/3D en una conclusión o una nueva representación. Estas transformaciones se realizan para alcanzar un cierto objetivo. Por ejemplo, las conclusiones pueden ser "hay una persona en esta escena" o "se han detectado 2 marcadores de referencia". Una nueva representación puede ser pasar una imagen en color a escala de grises o eliminar el movimiento de la cámara de una secuencia de imágenes.

OpenCV tiene una estructura modular, lo que significa que el paquete incluye varias librerías dinámicas o estáticas. Los módulos disponibles más importantes son:

- **core:** un módulo compacto que define las estructuras básicas de datos, incluyendo el array multidimensional Mat (utilizada por OpenCV para manejar todo tipo de imágenes) y las funciones básicas usadas por los demás módulos.
- **imgproc:** un módulo de procesado de imagen que incluye filtros lineales y no lineales, transformaciones geométricas, conversión de espacio de color, histogramas, etc.
- **video:** un módulo de análisis de vídeo que incluye estimación de movimiento, obtención del primer plano y algoritmos de seguimiento de objetos.
- **calib3d:** calibración de cámaras simples y estéreo, algoritmos de correspondencias estéreo y elementos de reconstrucción 3D.
- **features2d:** algoritmos de reconocimiento de características, como detectores de bordes, esquinas, puntos de interés, etc.
- **objdetect:** detección de objetos y elementos de las clases predefinidas (por ejemplo, caras, ojos, personas, coches, etc).
- **highgui:** funciones de interfaz de usuario que pueden ser usadas para mostrar imágenes/vídeo en una sencilla ventana de UI, para capturar imágenes o vídeo por una cámara o desde un soporte de almacenamiento y para escribir las imágenes/vídeo en un archivo.
- **gpu:** implementaciones de la mayoría del resto de funciones optimizadas para ser ejecutadas en GPUs CUDA. También hay algunas funciones que solo están implementadas para GPU..

Toda la documentación relacionada con el uso e interfaz de las funciones de OpenCV está disponible online en [8]. Aunque la cantidad de información es enorme y muy útil, el propósito de la documentación de OpenCV no es explicar los algoritmos implementados.

También se proporcionan instrucciones sobre cómo compilar los archivos fuente de OpenCV, instalarlo y utilizarlo sobre diferentes sistemas operativos y plataformas. Esta información puede consultarse en [9].

2.5. High Level Synthesis

Vivado Design Suite es la suite de software desarrollada por Xilinx para diseñar y trabajar con sus nuevos dispositivos, como Zynq. Entre los productos que incluye se encuentran:

- **Vivado:** es un IDE (Entorno de Desarrollo Integrado) para diseñar y sintetizar el hardware que se implementa en la lógica programable de las FPGAs o del SoC Zynq. Además de trabajar con diseños RTL (VHDL/Verilog), cuenta con la posibilidad de trabajar a más alto nivel formando e integrando bloques IP hardware, lo que aumenta la capacidad de diseño basado en reutilización.
- **Xilinx SDK (Software Development Kit):** es un IDE para el diseño de software basado en Eclipse que incluye soporte para los procesadores embebidos en los dispositivos de Xilinx, compilador de GNU (gcc y g++) para desarrollar en C y C++ para ARM, herramientas de debug y una serie de utilidades adicionales.
- **Vivado HLS (High Level Synthesis):** es un IDE para acelerar la creación de bloques IP hardware a partir de especificaciones de alto nivel en C, C++ o System C sin necesidad de crear manualmente el diseño RTL (VHDL/Verilog), siendo este generado automáticamente por la herramienta (síntesis de alto nivel).

Este apartado pretende centrarse en Vivado HLS [10]. Como se ha mencionado, esta herramienta sintetiza hardware digital directamente desde una descripción de alto nivel desarrollada en C, C++ o System C, eliminando la necesidad de generar manualmente una descripción RTL (VHDL/Verilog). Así, la funcionalidad del diseño y su implementación hardware se mantienen separadas, lo que proporciona una gran flexibilidad.

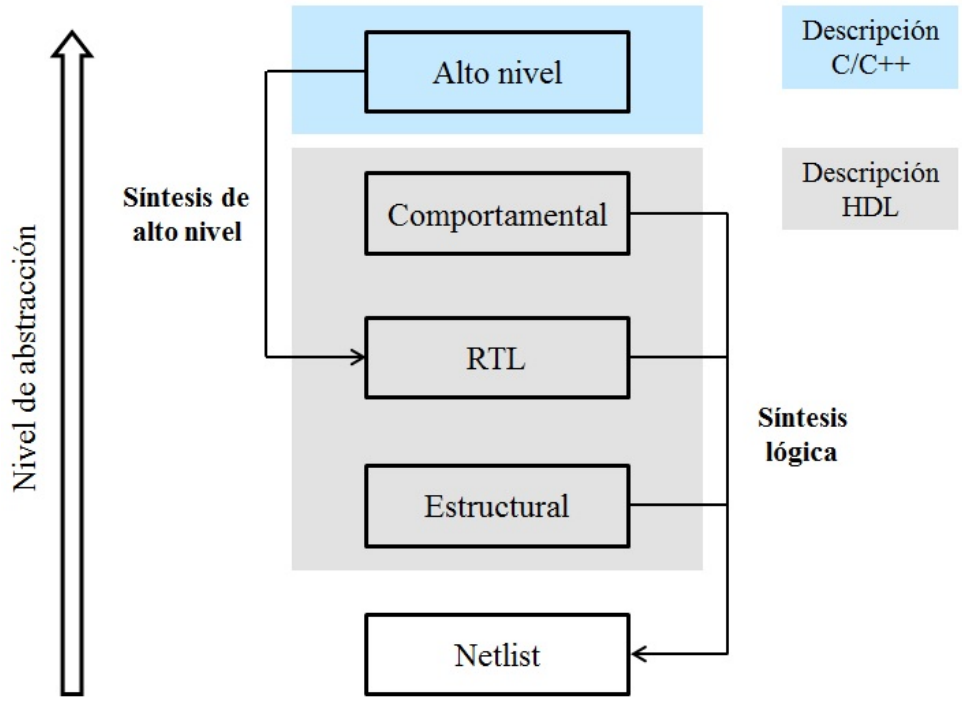

Figura 2.5: Diferentes niveles de abstracción en el diseño hardware, indicando los procesos de síntesis de alto nivel y síntesis lógica.

En el diseño tradicional de hardware para FPGAs, "síntesis" generalmente se refiere a síntesis lógica (es decir, el proceso de analizar e interpretar código HDL y generar una netlist). Sin embargo, síntesis de alto nivel significa sintetizar el código basado en C de alto nivel para obtener una descripción HDL, que después servirá para realizar una síntesis lógica y formar una netlist. Dicho de otra manera, la síntesis de alto nivel y la síntesis lógica se aplican las dos, una después de la otra, para llevar un diseño desde alto nivel en HLS hasta la implementación hardware. Este proceso se muestra en la figura 2.5.

Motivación para el uso de síntesis de alto nivel

La separación entre la funcionalidad y la implementación del diseño significa que el código fuente no describe explícitamente la arquitectura. Pueden crearse variaciones en la arquitectura rápidamente aplicando directivas para la síntesis en HLS, en lugar de básicamente reescribir el código fuente como sería necesario a nivel RTL.

No obstante, la mayor ventaja es una reducción significativa del tiempo de diseño. Al abstraerse de los detalles de bajo nivel, la descripción del circuito se vuelve más simple. Esto no elimina la necesidad de especificar aspectos como tamaños de datos, paralelismos, etc. pero, a grandes rasgos, el diseñador dirige el proceso y la herramienta implementa los detalles.

Obviamente, la implicación de renunciar a un control total sobre el hardware exige que el diseñador confíe en que la herramienta de síntesis implemente la funcionalidad de más bajo nivel correcta y eficientemente, además de que conozca cómo ejercer influencia sobre este proceso.

Flujo de diseño con Vivado HLS

La entrada para HLS son los archivos con el código fuente C/C++ del diseño. También es posible introducir un testbench, escrito en lenguaje de alto nivel (C/C++). Antes de sintetizar nada, se puede ejecutar una simulación C para realizar una verificación funcional y comprobar que las salidas de la función que se desea sintetizar son las esperadas, utilizando el testbench.

Después, se realiza la síntesis C (esto es, la síntesis de alto nivel propiamente dicha para generar el diseño RTL). Esto incluye al análisis y procesado del código C junto con las restricciones (tales como el periodo de reloj) y directivas (para indicar detalles de la arquitectura hardware) introducidas por el usuario.

Una vez que se ha producido el modelo RTL para el diseño, puede ser comprobado mediante la cosimulación RTL. Este proceso reutiliza el testbench C/C++ para generar las entradas de la versión RTL y comprobar las salidas, por lo que ahorra el trabajo de generar un nuevo testbench RTL.

Además de funcionalmente, la implementación RTL puede ser evaluada en términos de área y recursos ocupados en la lógica programable del dispositivo de destino y en términos de rendimiento (el informe de síntesis aporta datos como latencia del circuito, número de ciclos de reloj necesarios para la generación de la salida y máxima frecuencia de reloj conseguida).

Cuando el diseño ha sido validado y la implementación ha cumplido los objetivos de área/rendimiento deseados, puede utilizarse usando directamente los archivos RTL (VHDL/Verilog) generados por HLS. Sin embargo, para integrarlo en un diseño más grande, lo más recomendable es utilizar la opción de HLS para empaquetarlo y generar un bloque IP destinado a ser utilizado en otras herramientas de Xilinx. Por ejemplo, se puede introducir y conectar de manera muy cómoda en el IP Integrator de Vivado.

Un esquema con el flujo de diseño con Vivado HLS explicado en esta sección se muestra en la figura 2.6.

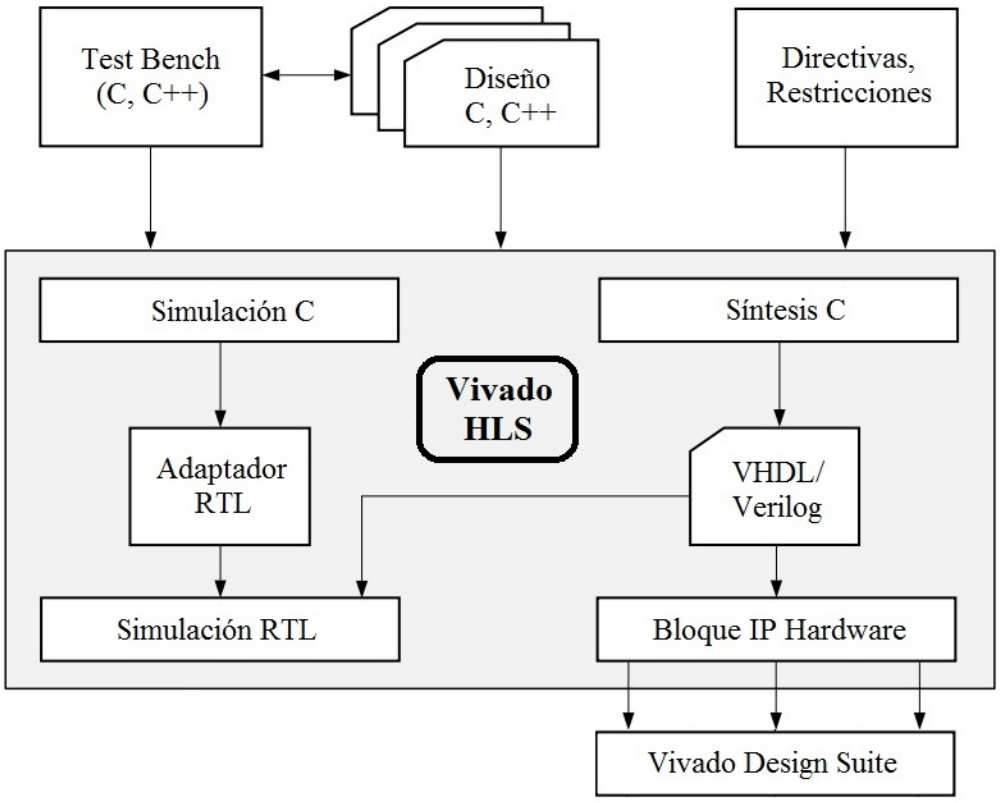

Figura 2.6: Flujo de diseño con Vivado HLS. Partiendo de un diseño descrito en un lenguaje de alto nivel (C/C++), se obtiene un diseño RTL (VHDL/ Verilog) que puede empaquetarse como un bloque IP preparado para ser utilizado con facilidad en otras herramientas de Xilinx. El proceso admite verificación funcional a partir de un testbench, escrito también en C/C++

Capítulo 3

Sistema a diseñar y análisis Software

3.1. Sistema de posicionamiento

El sistema implementado en este trabajo es objeto de una patente de la Universidad de Cantabria [2]. El punto de partida es una versión del mismo completamente en software, desarrollada en C++ y probada en un PC de escritorio con sistema operativo Linux (presentada en la figura 1.1 de la introducción).

Se trata de un método y sistema de localización espacial de un objetivo (usuario, robot...) en un entorno tridimensional que comprende al menos un marcador luminoso, utilizando una cámara y un procesador de señales con acceso a una memoria, como se muestra en la figura 3.1.

Cuando solo se visualiza un marcador de referencia, es necesario utilizar

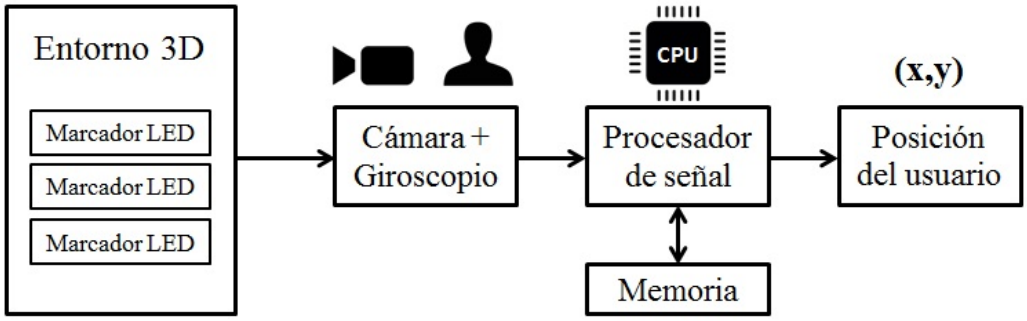

Figura 3.1: Funcionamiento del sistema de posicionamiento.

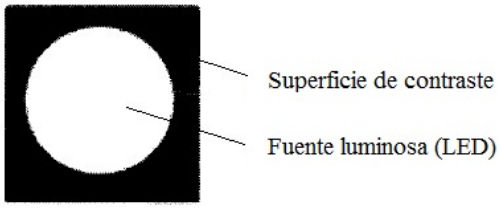

Figura 3.2: Representación de un tipo de marcador luminoso.

una cámara estéreo para obtener la posición del usuario o punto a caracterizar. Para ello, se aplican correspondencias estéreo (buscar un mismo punto en las dos imágenes) y se utiliza geometría proyectiva o epipolar (describe la relación existente entre los planos de la imagen de las cámaras y el punto). En caso de visualizar más de un marcador de referencia, es posible obtener la posición del individuo mediante una sola cámara, a través de triangulación, conociendo la distancia entre marcadores.

Una de las ventajas de este método es que se puede implementar en sistemas de realidad aumentada, porque las cámaras muestran lo que ve el usuario. Este sería un posible campo de aplicación; por ejemplo, utilizando unas gafas de realidad virtual con cámara estéreo. También lo sería la localización de robots o máquinas en entornos industriales.

Un tipo de marcador de referencia como los utilizados puede verse en la figura 3.2. Consta de una fuente de luz (preferiblemente un diodo LED que emite en el rango visible, 400-700 nm) y una superficie de contraste.

En la figura 3.3 se muestra un caso en el que se detectan dos o más marcadores en la imagen captada por la cámara. Un posible escenario de aplicación del sistema sería un entorno en el que el usuario se encuentre completamente rodeado de marcadores de referencia (por ejemplo, en las paredes de una habitación).

En este trabajo se considera la parte del sistema encargada de la detección de los marcadores de referencia, que es la que contiene mayor carga computacional. Además, se ha implementado utilizando una sola cámara, ya que esta parte se aplica por igual e independientemente a ambas imágenes en el caso de utilizar una cámara estéreo, por lo que la extensión al uso de dos cámaras es directa.

El algoritmo para detectar marcadores de referencia se puede dividir en seis etapas:

1. Captura de una imagen RGB desde la cámara. Para ilustrar el proceso,

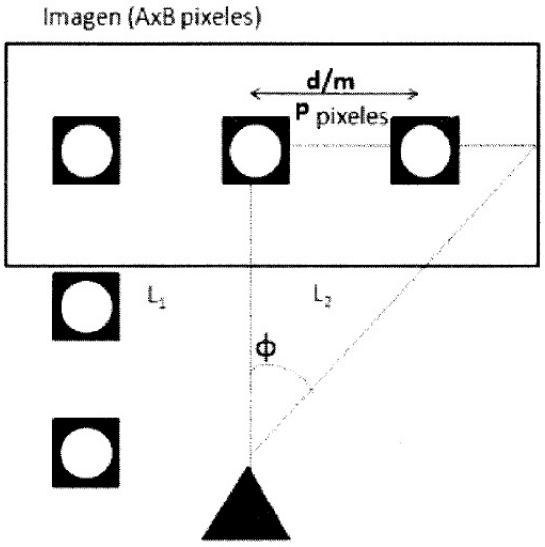

Figura 3.3: Esquema de los marcadores y parámetros usados por el sistema.

se supone que se captura la imagen mostrada en la figura 3.4.

2. Buscar las zonas más brillantes de la imagen. En concreto, encontrar píxeles con colores en el rango desde [205,205,205] a [255,255,255].

3. Filtrado de la imagen. Se suaviza la imagen para eliminar píxeles blancos no significativos, reduciendo el numero de contornos a examinar. El resultado se muestra en la figura 3.5.

4. Detectar el contorno de las zonas brillantes. En cada imagen se detectan muchos contornos (superficies blancas cerradas), pero no todos ellos se corresponden con un marcador. La imagen de la figura 3.6 indica todos los contornos encontrados.

5. Eliminar contornos considerando la forma del marcador, para eliminar falsos positivos. Se excluyen los rectángulos cuya base y altura no tienen dimensiones similares.

6. Eliminar contornos considerando la diferencia de intensidad entre la fuente de luz y la superficie de contraste, para eliminar falsos positivos. Se conservan únicamente aquellos contornos con un contraste importante entre la zona brillante y la región que la rodea.

La imagen final, señalando el único marcador de referencia verdadero encontrado por el algoritmo, se puede ver en la figura 3.7.

Figura 3.4: Imagen RGB original capturada por la cámara.

Figura 3.5: Imagen procesada mostrando únicamente la información de regiones brillantes.

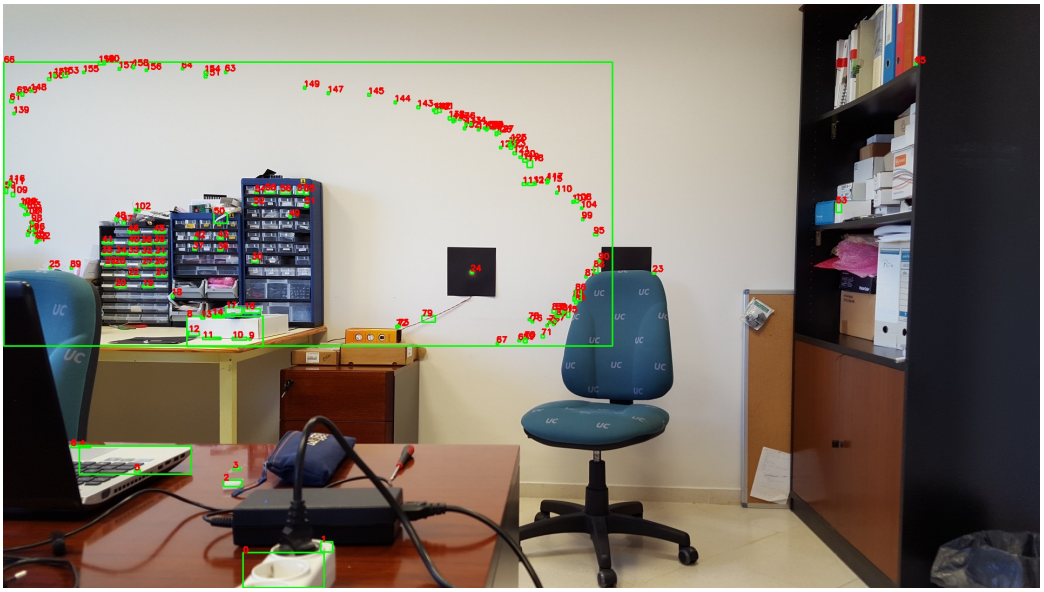

Figura 3.6: Imagen con todos los contornos encontrados por el algoritmo.

Figura 3.7: Imagen final tras la detección del marcador de referencia correcto.

3.2. Sistema operativo Linux en ZedBoard

3.2.1. Razón de utilizar sistema operativo

Para llevar a cabo la implementación del sistema de posicionamiento en la placa es necesario el uso de un sistema operativo.

En primer lugar, debido a que un software desarrollado para un OS es muy fácil de portar a una nueva arquitectura que use el mismo OS (más que si ha sido desarrollado para un dispositivo en concreto). Por ejemplo, el algoritmo ha sido desarrollado en una versión de escritorio de Linux, por lo que es relativamente sencillo ejecutarlo en un sistema Linux embebido.

En segundo lugar, porque el sistema operativo proporciona soporte para utilizar características como salida hacia pantallas externas vía HDMI o reconocimiento de cámaras con entrada USB, entre otros. De otro modo, estas características tendrían que ser desarrolladas por el diseñador.

Por tanto, al tratarse de un sistema complejo y hacer uso de funciones incluidas en el sistema operativo y en librerías de terceros, parte de la funcionalidad permanecerá en software - sobre el sistema operativo Linux - en la versión final. La figura 3.8 muestra la arquitectura general de alto nivel de un sistema GNU/Linux.

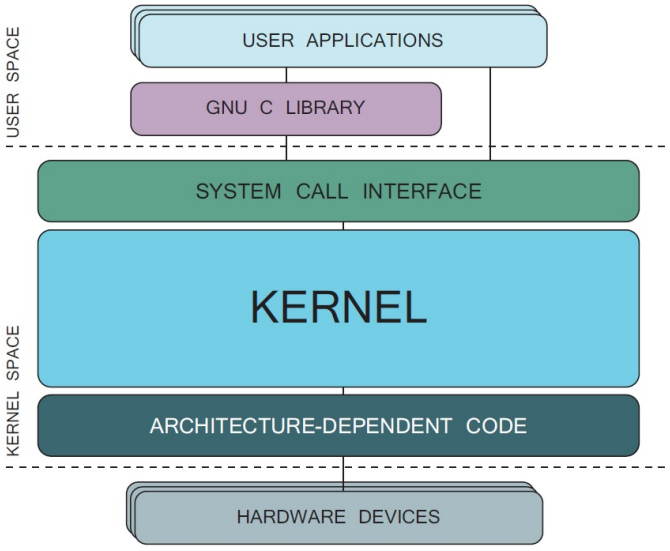

Figura 3.8: Arquitectura de alto nivel de un sistema GNU/Linux.

3.2.2. Arranque de Linux en ZedBoard

Es posible arrancar el kernel de Linux en Zynq y utilizar el sistema de ficheros de una distribución de Linux completamente funcional (en este caso Linaro, que es prácticamente igual a una versión de escritorio de Ubuntu). De este modo, se puede conseguir que la ZedBoard funcione como un PC, utilizando un teclado y un ratón USB y una pantalla HDMI. En este entorno se pueden instalar y ejecutar una gran variedad de aplicaciones y desarrollar software de forma nativa para ARM en la placa con las herramientas de GNU.

El arranque de Linux en un sistema embebido como Zynq presenta algunas diferencias frente al arranque en un PC convencional [3]. En este caso, mediante los jumpers presentes en la placa, se configura la ZedBoard para arrancar desde la tarjeta SD. Por lo tanto, todos los ficheros necesarios para el arranque y la carga del sistema operativo deben estar copiados en la tarjeta de memoria de una manera determinada.

- **Etapa 0 (Boot ROM):** Cuando se enciende o se resetea la placa, se ejecuta el código de la ROM de arranque en el procesador primario. Su función es cargar la imagen del FSBL (First Stage Bootloader) desde la tarjeta SD y copiarlo en la memoria interna del chip (On-Chip Memory, OCM).

- **Etapa 1 (First Stage Bootloader):** El FSBL (First Stage Bootloader) se encarga de inicializar la CPU con los datos de configuración del procesador, configurar la FPGA utilizando un bitstream y cargar el SSBL (Second Stage Bootloader) o aplicación inicial del usuario en la memoria RAM de la placa, así como iniciar su ejecución.

- **Etapa 2 (Second Stage Bootloader):** En el caso de una aplicación sin sistema operativo, el código se cargaría y ejecutaría en esta etapa. En el caso de estar cargando un sistema operativo como Linux o Android, se ejecuta un SSBL (Second Stage Bootloader), como U-Boot. U-Boot es un bootloader open source universal en el entorno de Linux usado por Xilinx para el microprocesador que incorpora Zynq. Este bootloader carga en memoria RAM el device tree (un archivo que proporciona al kernel información sobre el hardware del sistema) y la imagen del kernel de Linux e inicia su ejecución.

Las diferentes etapas del proceso de arranque de Linux en Zynq se muestran en la figura 3.9.

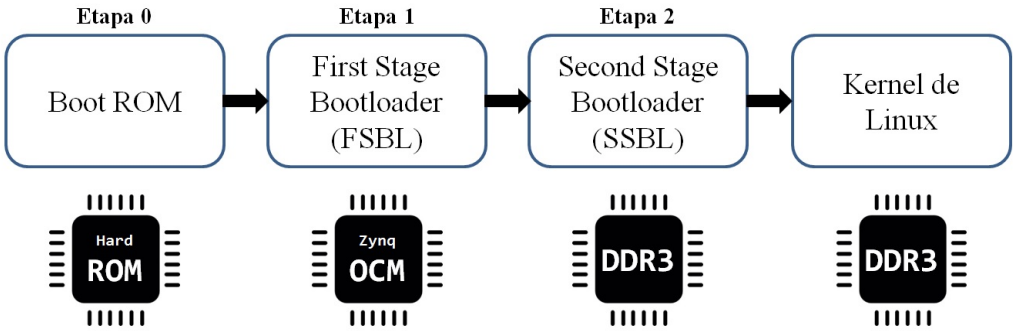

Figura 3.9: Proceso de arranque de Linux en Zynq.

3.2.3. Generación de los archivos de arranque

Si se desea una guía paso a paso para generar los archivos de arranque y poner en marcha el sistema, se recomienda consultar [11].

Particionado de la tarjeta SD

Todos los archivos necesarios para arrancar Linux deben encontrarse en la tarjeta SD en las particiones correspondientes, como se muestra en la figura 3.10.

En primer lugar, debe haber una partición con formato FAT32 con los primeros archivos necesarios para ejecutar el kernel y configurar la FPGA: los dos bootloaders, el devicetree, el propio kernel de Linux y el bitstream. En segundo lugar, se necesita una partición con formato EXT4 donde se carga el sistema raíz de ficheros de la distribución deseada.

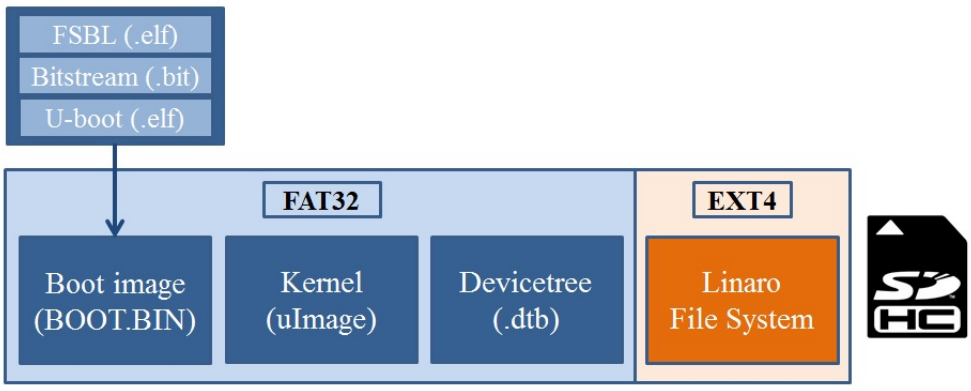

Figura 3.10: Archivos de arranque de Linux en Zynq.

Generación del Second Stage Bootloader (U-Boot)

Es posible obtener el código fuente de U-Boot preparado para Zynq en el repositorio Git de Xilinx [12]. Después, es necesario configurarlo no solo para Zynq, sino para la propia placa de desarrollo (ZedBoard). Además, es preciso incluir (o modificar) en el código las instrucciones que ejecutará el U-Boot cuando sea lanzado en el arranque. En este caso, tiene que leer la tarjeta SD y cargar en determinadas direcciones de memoria RAM el devicetree, la imagen del kernel de Linux y lanzar su ejecución:

```
mmcinfo
fatload mmc 0 0x3000000 ${kernel_image}
fatload mmc 0 0x2A00000 ${devicetree_image}
bootm 0x3000000 - 0x2A00000
```

La instrucción *bootm* funciona con formatos de imagen de kernel sin comprimir (uImage), como la usada en este trabajo, mientras que la instrucción *run* funciona con imagenes comprimidas (zImage).

Una vez preparado, el U-Boot puede ser generado utilizando la herramienta de compilación *Make*.

Generación del Kernel de Linux

Analog Devices mantiene y distribuye una versión del kernel de Linux que funciona en placas de desarrollo con el SoC Zynq-7000 y que puede obtenerse desde su repositorio Git [13]. Tras obtener el código fuente, es necesario configurarlo para Zynq y compilarlo para la arquitectura ARM utilizando la herramienta de compilación *Make*. Para generar la imagen del kernel que leerá la placa (el archivo uImage) es necesario una herramienta que se crea al compilar el U-Boot. De este modo se obtiene una imagen preparada para ser cargada por el U-Boot en el arranque del sistema.

Generación del Devicetree

El devicetree es un archivo que contiene una estructura de datos describiendo el hardware presente en el sistema, de manera que el kernel lo pueda conocer y utilizar. En un PC, esta información es suministrada por la BIOS, pero los procesadores ARM no tienen un equivalente.

Es preciso compilar los archivos fuente del devicetree para Zynq. Se encuentran, al igual que el kernel, en el repositorio Git de Analog Devices [13]. Incluyen una capa común para cualquier dispositivo Zynq y otra capa con

información sobre algunas placas de desarrollo, entre las que se encuentra la ZedBoard. Estos archivos incluyen toda la información y relaciones necesarias para que el kernel de Linux utilice el hardware de Analog Devices incluido en la ZedBoard, como son los controladores de audio y de salida de vídeo por HDMI. Al realizar cambios en el hardware programable del SoC Zynq, es posible que sea necesario modificar y recompilar el devicetree si se han producido cambios en las direcciones de memoria asociadas a módulos hardware, en las señales de interrupción, etc.

Obtención del sistema de ficheros de la distribución Linaro

El sistema raíz de ficheros (Root File System, RFS) es el sistema de ficheros contenido en la misma partición en la que se encuentra el directorio raíz usado por el kernel de Linux, y es el sistema de ficheros en el que se montan todos los demás (es decir, se adjuntan lógicamente al sistema).

El RFS está separado del kernel de Linux, pero es requerido por el kernel para completar el proceso de arranque. Esta separación añade flexibilidad a las distribuciones de Linux; generalmente usan un kernel de Linux común (o con pequeñas variaciones) y un sistema raíz de ficheros con todo el contenido que las caracteriza.

El RFS de Linaro puede ser descargado desde la página web oficial [14]. Se trata de una distribución que proporciona una experiencia de escritorio similar a Ubuntu, con interfaz gráfica Unity, especializada en plataformas ARM.

Generación del Hardware (bitstream)

La plataforma hardware para configurar la FPGA del SoC Zynq se genera con el software Vivado de Xilinx. Para ejecutar un sistema operativo, es necesaria una plataforma base que incluye varios módulos hardware y controladores y sus conexiones con el bus AXI y con el microprocesador. Esta plataforma base es proporcionada por Analog Devices e integra los periféricos necesarios para utilizar la salida de vídeo HDMI (el transmisor de HDMI ADV7511 de Analog Devices se conecta al procesador a través de bloques IP proporcionados por la compañía), la salida de audio, la memoria DDR3 y periféricos I2C. El diseño constituye un punto de partida funcional para cualquier sistema que requiera vídeo por HDMI y audio, con o sin sistema operativo [15].

La plataforma proporcionada por Analog Devices está implementada con la versión 2013.4 de Vivado Design Suite. La utilizada en este trabajo ha

sido portada a la versión 2015.3 y, debido a cambios por la actualización de bloques IP, el devicetree ha sido modificado en consecuencia.

Generación del First Stage Bootloader (FSBL)

El First Stage Bootloader puede generarse utilizando el software SDK (Software Development Kit) de Xilinx. Para ello, simplemente hay que importar la plataforma hardware generada con Vivado y crear un FSBL para Zynq.

Generación de la imagen de arranque (BOOT.BIN)

La imagen de arranque es un archivo de nombre BOOT.BIN que contiene, en este orden, tres particiones con los siguientes archivos:

1. FSBL (First Stage Bootloader)
2. Bitstream (hardware para configurar la FPGA)
3. U-Boot (Second Stage Bootloader)

La imagen de arranque se genera a partir de los archivos que contiene con el software SDK de Xilinx, utilizando la herramienta *Create Zynq Boot Image*.

3.3. Profiling del Software en Zedboard

Una vez que se dispone de Linux en la plataforma embebida, es posible compilar el software de forma nativa para ARM en la placa y ejecutar en ella la versión software del sistema de posicionamiento.

El primer paso para llevar a cabo una aceleración hardware de un sistema es realizar un análisis detallado del rendimiento del algoritmo (profiling). Este análisis puede revelar si merece la pena diseñar módulos hardware, lo cual generalmente requiere una mayor cantidad de tiempo que el desarrollo de software.

Es útil comenzar presentando un árbol de llamadas con la estructura del programa, mostrando las principales funciones que utiliza (figura 3.11). El objetivo del análisis es obtener, para cada función, datos de tiempo absolutos y en porcentaje respecto al tiempo de ejecución total del programa.

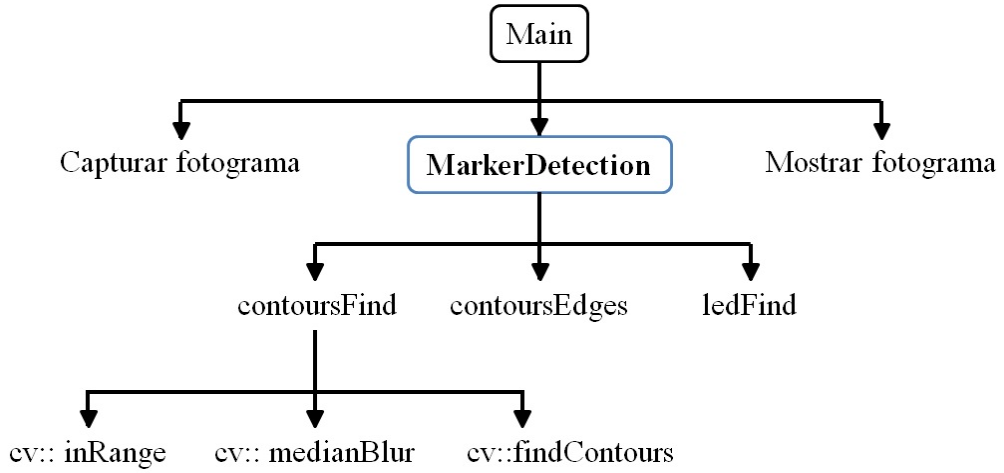

Figura 3.11: Árbol de llamadas simplificado del programa (se muestran solo las funciones con tiempos de ejecución más significativos).

Compilación de OpenCV para profiling

El software que se quiere analizar utiliza varias funciones de la librería de visión artificial OpenCV (Open Source Computer Vision). Esto supone que la mayor parte del tiempo de ejecución pasa en funciones que están dentro de la librería. Si se desea que las herramientas de profiling funcionen correctamente en estos casos, es necesario hacer dos cosas antes de compilar los archivos fuente de OpenCV.

Por una parte, editar el archivo *CMakeLists.txt* para activar el soporte para profiling. En este caso, se trata de activar los siguientes flags:

```
OCV_OPTION(ENABLE_PROFILING ON)
OCV_OPTION(ENABLE_OMIT_FRAME_POINTER ON)
```

La primera de ellas sirve para compilar OpenCV con los flags -g -pg, necesarios para realizar el profiling con las Binutils de GNU.

Por otra parte, las herramientas de profiling no pueden entrar en las librerías dinámicas o (Shared Object, *.so en Linux), por lo que OpenCV se debe compilar de manera especial para generar librerías estáticas (*.a en Linux):

```
cmake -DCMAKE_BUILD_TYPE=RELEASE -DBUILD_SHARED_LIBS=OFF
make install
```

Profiling utilizando Valgrind

Se han utilizado varias estrategias para estudiar el rendimiento del algoritmo sobre la plataforma final (sobre la ZedBoard). En primer lugar, utilizando las herramientas de Valgrind, un conocido software de análisis para Linux. En concreto debe utilizarse *Callgrind*, una herramienta de profiling que permite visualizar los resultados gráficamente a través de la interfaz *KCachegrind*. Es necesario instalar todos estos paquetes:

```
sudo apt-get install valgrind kcachegrind graphviz
```

Para analizar un ejecutable, se lanza *Callgrind* de la siguiente forma (los ficheros generados se abren con *KCachegrind*):

```
valgrind --tool=callgrind ./[executable]
```

Aunque estos programas han funcionado correctamente analizando el algoritmo sobre el PC (x86), no han conseguido analizar correctamente la ejecución del software en Zynq (ARMv7).

Profiling utilizando gprof (GNU)

En segundo lugar, se ha utilizado *gprof*, la utilidad de profiling de GNU. Para ello, se debe compilar y enlazar el programa con opciones de profiling, añadiendo la opción -pg. De este modo se genera código adicional para obtener información que puede interpretar *gprof*. Después se ejecuta el programa y automáticamente se genera un archivo llamado gmon.out. Por último se lanza *gprof*, redirigiendo la salida a un archivo de texto:

```
$  gprof [executable] gmon.out > analysis.txt
```

La información de profiling generada está dividida en dos partes: análisis plano y árbol de llamadas. En la figura 3.12 se muestra una parte del fichero de resultados generado, tras ejecutar el software utilizando una cámara USB con resolución VGA (640x480) para un número alto de fotogramas (tiempo de ejecución suficiente como para obtener datos, ya que la precisión de *gprof* es de 0.01 segundos). Se muestran todas las funciones del programa, ordenadas por porcentaje de tiempo usado por la función sobre el tiempo total de ejecución del programa.

Se observa que la función de OpenCV medianBlur es la que más tiempo tarda en ejecutarse, con casi un 61 % del tiempo total del algoritmo, seguida por la función de OpenCV inRange, con un 13 % del tiempo.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 60.97     51.20    51.20                             cv::medianBlur(cv:
 12.99     62.11    10.91                             cv::inRange8u(unsi
 12.02     72.20    10.09                             icvCvt_BGR2RGB_8u_
  4.31     75.82     3.62                             cvFindNextContour
  3.85     79.05     3.23                             cv::inRange(cv::_I
  2.12     80.83     1.78                             cv::ThresholdRunne
  0.50     81.25     0.42                             cvApproxPoly
  0.32     81.52     0.27                             cv::sum8u(unsigned
  0.13     81.63     0.11   511143     0.00     0.00  void std::_Constru
  0.13     81.74     0.11    64894     0.00     0.01  cv::Point_<int>* s
  0.13     81.85     0.11                             cvSetSeqReaderPos
  0.11     81.94     0.10   576037     0.00     0.00  bool __gnu_cxx::op
  0.11     82.04     0.10                             cv::NAryMatIterato
```

Figura 3.12: Resultados del profiling realizado con *gprof*.

Profiling manual utilizando el timer del sistema

Por último, se ha realizado un profiling insertando manualmente en el código funciones definidas en la cabecera `<time.h>` para obtener el tiempo del reloj del sistema en puntos estratégicos.

En concreto, se ha utilizado la función `clock_gettime()`, que obtiene el tiempo del reloj especificado con precisión de un nanosegundo. En este caso, se obtiene el tiempo del reloj `CLOCK_MONOTONIC`, que representa el tiempo absoluto (wall-clock) transcurrido desde un punto fijo y arbitrario en el pasado. No se ve afectado por cambios en el reloj de la hora del sistema (`CLOCK_REALTIME`) y es la mejor opción para obtener el tiempo transcurrido entre dos eventos que han ocurrido en la máquina sin un reinicio de por medio. Los datos temporales obtenidos son estructuras de tipo `timespec`, como se especifica en `<time.h>`:

```
struct timespec {
time_t   tv_sec;        /* seconds */
long     tv_nsec;       /* nanoseconds */
};
```

El método que se ha utilizado para medir el tiempo requerido por una función consiste en obtener el valor del reloj del sistema antes y después de su ejecución y calcular la diferencia entre ambos. Para que la medida sea más rigurosa, esta diferencia acumula en una variable durante la ejecución del algoritmo para un número alto de fotogramas y, al finalizar, se divide la suma total entre el número de fotogramas procesados para obtener un valor medio.

Por ejemplo, para obtener el tiempo de ejecución medio de la función `funcion_A()` durante 100 fotogramas:

```
#include <time.h>
#define BILLION 1000000000L
#define NUM_FRAMES 100

void main(){

struct timespec start, end;
uint_64_t diff;
uint_64_t time_funcion_A = 0;

for(i=0; i<NUM_FRAMES; i++){
  clock_gettime(CLOCK_MONOTONIC, &start);
  funcion_A();
  clock_gettime(CLOCK_MONOTONIC, &end);
  diff = BILLION*(end.tv_sec-start.tv_sec)+end.tv_nsec-
     start.tv_nsec;
  time_funcion_A += diff;
}
time_funcion_A = time_funcion_A/NUM_FRAMES;

}
```

De esta forma se han medido los tiempos de ejecución de todas las funciones, probando el sistema durante un tiempo de funcionamiento largo con imágenes VGA (640x480) procedentes de una cámara y con imágenes Full HD (1920x1080) desde archivo. Los resultados se muestran en las tablas 3.1 y 3.2. El porcentaje del total se refiere explícitamente al porcentaje respecto del tiempo total del algoritmo de detección de marcadores, excluyendo las fases de captura de imagen desde la cámara USB y de muestra del vídeo en pantalla, que son bastante lentas.

Detección de marcadores VGA (640x480) SW		
Función	Tiempo de ejecución	Porcentaje del total
MarkerDetection	89200965 ns	-
cv::medianBlur	53854717 ns	60.4 %
cv::inRange	15792675 ns	17.7 %
cv::findContours	7982231 ns	8.9 %

Tabla 3.1: Tiempo de ejecución de las funciones con mayor carga computacional del algoritmo, para imágenes VGA (640x480).

Detección de marcadores FHD (1920x1080) SW		
Función	Tiempo de ejecución	Porcentaje del total
MarkerDetection	502750078 ns	-
cv::medianBlur	338926879 ns	67.4 %
cv::inRange	76811420 ns	15.3 %
cv::findContours	41110169 ns	8.2 %

Tabla 3.2: Tiempo de ejecución de las funciones con mayor carga computacional del algoritmo, para imágenes FHD (1920x1080).

3.4. Partición HW-SW

A la vista de los resultados del profiling, existen dos funciones de OpenCV que consumen la mayor parte del tiempo de ejecución del algoritmo de detección de marcadores: *cv::medianBlur* y *cv::inRange*, como se ve en la figura 3.13.

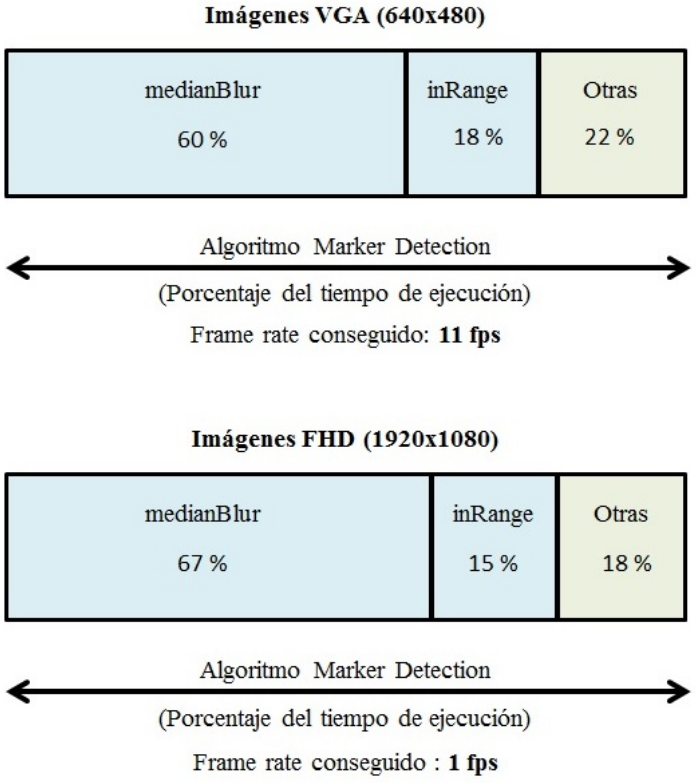

Figura 3.13: Tiempo de ejecución del algoritmo mostrando el porcentaje de las funciones con mayor carga computacional, para imágenes VGA y FHD.

Para diferentes resoluciones de imagen, los porcentajes sobre el total del algoritmo son similares, mientras que el tiempo de ejecución varía notablemente.

La ejecución del algoritmo sobre el ARM Cortex A9 del SoC Zynq resulta demasiado lenta para ser usada en un sistema de tiempo real que necesite responder rápidamente. Procesando imágenes VGA (640x480) se alcanza una tasa de 11 fps (fotogramas por segundo). Si se utilizan imágenes Full HD (1920x1080), el *framerate* desciende a tan solo 1 fps, lo cual es claramente insatisfactorio. Por tanto, el bajo throughput justifica una aceleración hardware del sistema.

Estudiando las características de las dos funciones que más tiempo gastan, se constata que es posible llevar a cabo una síntesis hardware de las mismas. Además, son usadas consecutivamente, de forma que los datos de salida de la primera son los datos de entrada de la segunda. Esto hace que la transferencia de datos entre software y hardware solamente tiene que ocurrir una vez para ejecutar las dos funciones. Teniendo todo esto en cuenta, una síntesis hardware de las funciones de OpenCV *inRange* y *medianBlur* es apropiada para llevar a cabo una buena aceleración del algoritmo.

Capítulo 4

Generación del Hardware

Teniendo en cuenta que el SoC Zynq-7000 incluye un sistema de procesado (PS) basado en ARM y lógica programable (PL), el objetivo es aprovechar la parte de FPGA para integrar bloques IP hardware que implementen la funcionalidad elegida en el anterior capítulo.

Para diseñar los bloques IP, se ha utilizado Xilinx Vivado HLS (High Level Synthesis). Esta herramienta es capaz de convertir diseños basados en C (C/C++) a archivos de diseño RTL (VHDL/Verilog). Sin embargo, para obtener un diseño de altas prestaciones, es necesario comprender bien cómo funciona la herramienta y modificar el código de acuerdo al hardware que se desea generar. Un esquema general del flujo de diseño seguido se muestra en la figura 4.1.

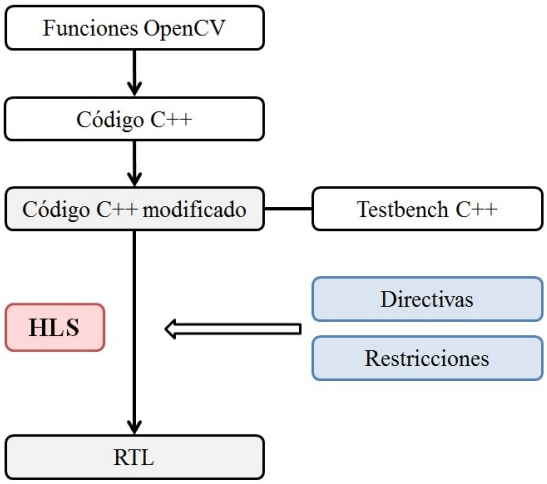

Figura 4.1: Flujo de diseño para generar los bloques IP hardware usando síntesis de alto nivel.

Vivado HLS proporciona una librería de vídeo que incluye algunas funciones de procesado con comportamiento e interfaz equivalente a las funciones correspondientes de OpenCV [16]. Sin embargo, en el repertorio disponible no se encuentran las funciones deseadas.

El primer paso es estudiar y entender cómo trabajan las funciones de OpenCV requeridas y desarrollar una versión C/C++ propia. Después, esta versión es comprobada para confirmar que proporciona el mismo resultado que la original. Posteriormente, el código es modificado, sintetizado por HLS y comprobado de nuevo. El código sintetizable los IPs hardware cuyo diseño explica este capítulo se puede ver en el apéndice A.

Se trabaja con imágenes codificadas con profundidad de color de 8 bits, por lo que el rango de valores es de 0 a 255 para R, G y B (en el caso de imágenes en color) o para la escala de grises (en el caso de imágenes en blanco y negro).

4.1. Bloque IP para inRange

La función de OpenCV *inRange* [17] recibe como entrada una imagen de 3 canales (color RGB) y devuelve como salida una imagen de un canal (blanco y negro). Su cometido es leer cada píxel de la imagen de entrada y, si sus valores RGB están dentro de un cierto rango, fija el pixel correspondiente de la imagen de salida al máximo valor, 255 (blanco). En caso contrario, fija el pixel de la imagen de salida al mínimo valor, 0 (negro).

Es la función encargada de buscar las zonas más brillantes de la imagen. En este caso, se buscan píxeles con colores en el rango desde [205,205,205] a [255,255,255]. Si todas las componentes de color (R, G y B) tienen valores altos, entonces el píxel es muy brillante (cercano al blanco). Un esquema del funcionamiento de *inRange* se muestra en la figura 4.2.

En primer lugar, se ha desarrollado una versión C++ que funciona igual que la función original de la librería de OpenCV. Esto se ha comprobado aplicando ambas funciones a la misma imagen y comparando los archivos binarios resultantes. En esta versión, los límites superior e inferior del rango a chequear están parametrizados; no es así en la versión hardware, ya que es una implementación final que debe desempeñar una función concreta y ocupar el mínimo área.

Las interfaces de entrada y salida de los IP hardware son de tipo AXI Stream. Esto significa que los píxeles son recibidos y transmitidos serializados, uno a uno. Un algoritmo como el empleado en la función *inRange* puede recibir un píxel (x,y) e inmediatamente producir el píxel (x,y) de salida; no

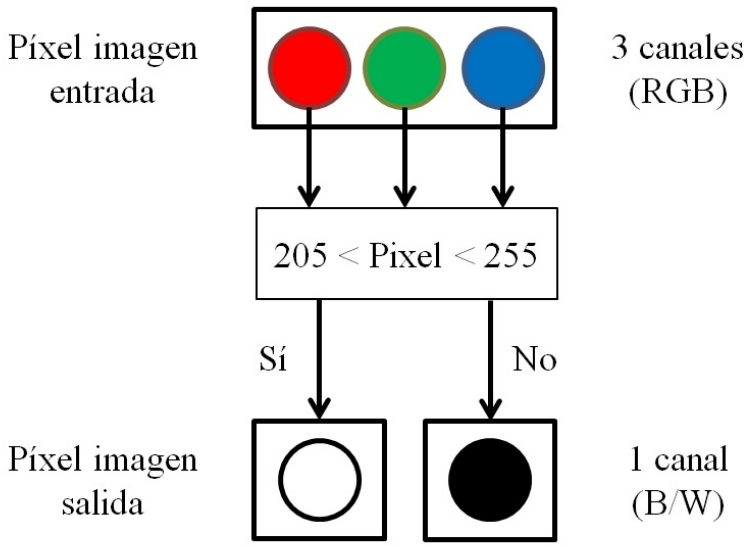

Figura 4.2: Comportamiento de la función *inRange*.

necesita almacenar o disponer de otros píxeles para computar.

A continuación se muestra un pseudocódigo para *inRange*:

```
for filas{
  for columnas{
    Leer pixel de entrada
    Comprobar los valores RGB
    Escribir pixel de salida
  }
}
```

En un caso de este tipo, los resultados de la síntesis de HLS son suficientemente buenos introduciendo directamente el código C++ original (más las instrucciones necesarias para describir las interfaces del IP, fijar algunas señales de protocolo, etc). Sin embargo, el throughput del IP puede mejorarse aún más introduciendo una directiva de pipeline al lazo interno con intervalo de inicialización de 1 (II=1). Así, se produce un píxel de salida con cada ciclo de reloj. Las prestaciones temporales estimadas del IP (sin integrar el IP en el sistema final), basadas en el informe de la síntesis realizado por Vivado HLS, se muestran en la tabla 4.1.

El tamaño de imagen que puede procesar el IP está parametrizado; el periférico cuenta con dos registros en la interfaz al bus, donde se puede escribir por software el tamaño de la imagen a tratar (hasta Full HD, 1920x1080).

inRange HW (1920x1080)			
Entrada HLS	Periodo de reloj	Ciclos de reloj	Throughput
Código C++ original	3.37 ns	4149362	71.51 fps
Código C++ original y directivas de síntesis	3.89 ns	2077922	**123.71 fps**

Tabla 4.1: Prestaciones temporales del IP diseñado para *inRange*.

4.2. Bloque IP para medianBlur

La función de OpenCV *medianBlur* [18] se utiliza en este caso con imágenes de un solo canal (blanco y negro). Su misión es reducir el ruido de la imagen, mediante la aplicación de un filtro de mediana. De esta forma, se suaviza la imagen eliminando zonas brillantes no significativas de muy pocos píxeles, reduciendo en gran cantidad el número de contornos (superficies blancas cerradas) a examinar en fases posteriores del algoritmo.

Los filtros de mediana son filtros no lineales muy utilizados en procesado de imagen y de vídeo, porque proporcionan una gran reducción de ruido mientras que preservan los bordes de la imagen mejor que los filtros de suavizado lineales, como el filtro de media.

En este caso, se utiliza un filtro de mediana en 2D con una ventana de tamaño 3x3 píxeles. Su funcionamiento consiste en reemplazar el valor de cada píxel de la imagen por la mediana de los píxeles vecinos (es decir, aquellos contenidos en la ventana de 3x3 centrada alrededor del píxel considerado). Un esquema del funcionamiento de un filtro de mediana sobre un píxel se muestra en la figura 4.3.

A continuación se muestra un pseudocódigo para *medianBlur* (es decir, para un filtro de mediana 2D), suponiendo que se lee toda la imagen antes de empezar a computar:

```
Leer todos los pixeles de entrada

for filas{
  for columnas{
    Cargar una ventana de 3x3 pixeles
    Calcular la mediana de la ventana
    Escribir pixel de salida
  }
}
```

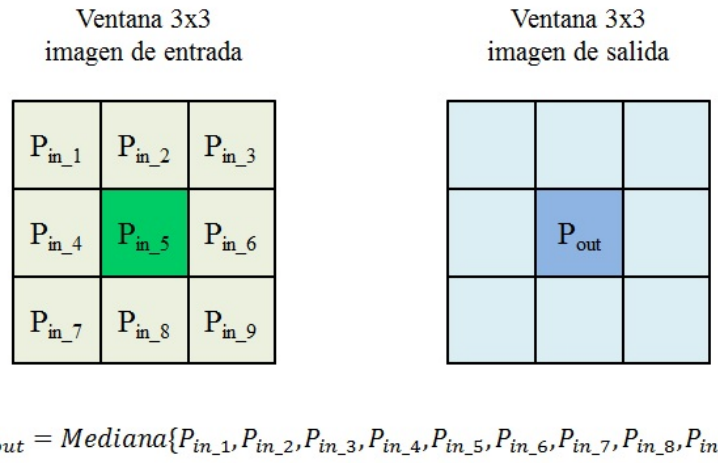

Figura 4.3: Comportamiento de la función *medianBlur* (filtro de mediana) sobre cada píxel de la imagen.

La mediana de un grupo impar de valores puede calcularse ordenándolos numéricamente y tomando el valor que ocupa la posición central en la lista ordenada. Por ejemplo, considerando la siguiente ventana de 3x3 centrada en torno a un píxel de valor 52:

42	128	3
208	52	60
74	87	255

La lista con los valores de los píxeles ordenados numéricamente es:

3 42 52 60 74 87 128 208 255

Y la mediana es el valor central, es decir, 74.

En relación a una versión hardware de este filtro, es preciso recordar que la interfaz AXI Stream utilizada transmite los datos (píxeles) uno a uno, serializados. Desafortunadamente, un filtro de este tipo necesita varios píxeles para producir un píxel de salida (en este caso, una ventana de 3x3 píxeles). Por ello, cuando se recibe un píxel, se necesitan algunos recibidos anteriormente y otros que se recibirán posteriormente. La manera más simple de solucionar esto consiste en leer toda la imagen de entrada y almacenarla completa en un buffer local. Después, se puede aplicar el código C++ normal de un filtro de mediana. Cuando se considera un determinado píxel y se necesitan los que lo rodean, estos están disponibles para que el hardware

compute. De este modo, se puede hacer una síntesis hardware rápidamente desde código C++ como el utilizado en una versión software, usando síntesis de alto nivel. Sin embargo, esto no es una buena manera para que HLS sintetice hardware de alta velocidad. El throughput del IP es muy bajo (en concreto 0.42 fps).

4.2.1. Arquitectura de bufferes de memoria

La razón del bajo rendimiento del diseño es la ausencia de una arquitectura de bufferes de memoria adecuada para procesado de vídeo. La herramienta de síntesis Vivado HLS garantiza que no se crean automáticamente estructuras de memoria, sino que el diseñador debe describir explícitamente estas estructuras en el código para sean implementadas en el diseño RTL. Por ello, se ha modificado el código para generar y manejar una arquitectura de memoria de doble buffer, como se recomienda en [19].

La primera memoria donde se almacenan los píxeles entrantes es un buffer de tres líneas, que actúa como un registro de desplazamiento multidimensional capaz de almacenar los datos de tres líneas de píxeles. Típicamente, los bufferes de línea son implementados como block RAMs dentro de la FPGA, para evitar la latencia de las comunicaciones con las memorias DRAM que se encuentran fuera del chip. Además, requieren acceso simultáneo de lectura y escritura, sacando provecho a las block RAM de doble puerto. Por otra parte, se utiliza un segundo buffer para la ventana de 3x3 elementos, que es un subconjunto del buffer de tres líneas que proporciona capacidad de acceso a todos sus elementos de manera totalmente simultánea. El kernel de cómputo del algoritmo para un determinado píxel se aplica a los elementos almacenados en el buffer de la ventana. El camino que siguen los datos en este tipo de arquitectura puede verse en la figura 4.4.

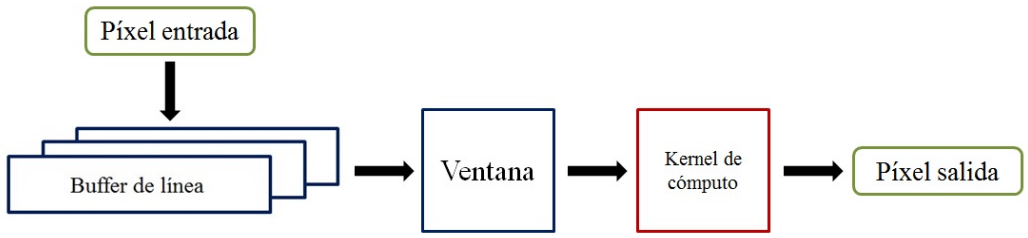

Figura 4.4: Camino de un píxel entrante hasta el kernel de cómputo del algoritmo a través de los dos bufferes (buffer de línea y ventana).

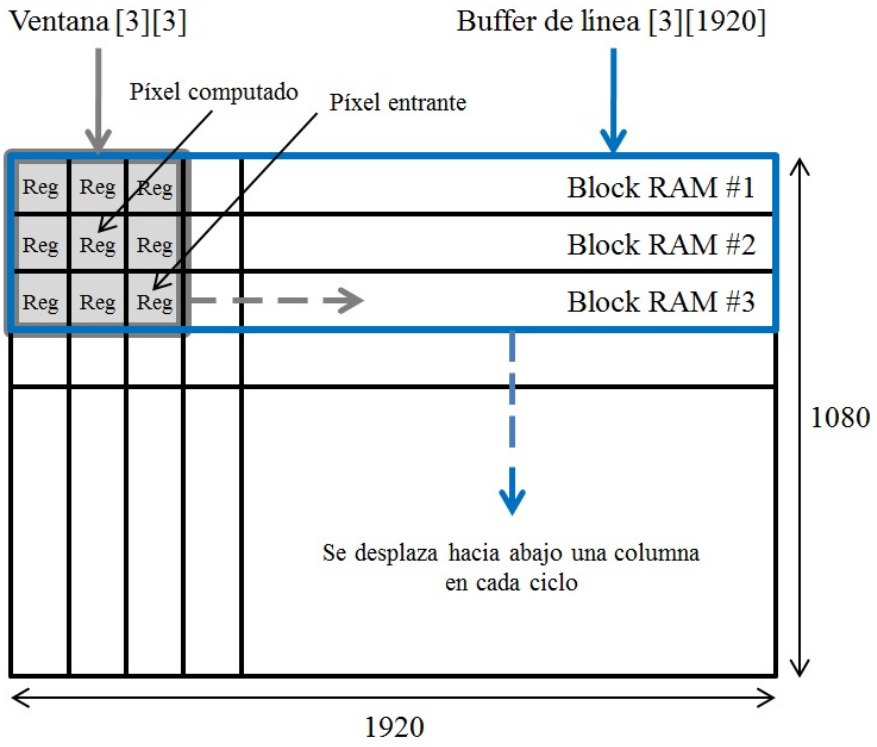

Figura 4.5: Comportamiento de los bufferes de línea y de la ventana (computando con 3x3 píxeles) para procesado de vídeo de alta velocidad.

Cada ciclo de reloj, cuando se recibe un píxel, es cargado en la columna correspondiente de la tercera línea del buffer de línea (la de más abajo en la imagen), avanzando una columna en cada ciclo hasta llegar a la última, rellenando así la línea en horizontal. Simultáneamente, los píxeles previamente almacenados en la columna en cuestión son movidos a la misma posición en el buffer de la línea superior (la de más arriba en la imagen) y el de más arriba es desechado.

Además, cada ciclo de reloj, se introducen en la ventana 3 nuevos píxeles procedentes de cada uno de los 3 bufferes de línea y se desplazan de posición los 6 píxeles que deben seguir mantenidos en la ventana. El movimiento de los bufferes de línea y de la ventana se muestra en la figura 4.5.

El buffer de línea es declarado en C++ como un array multidimensional, de altura igual a la de la ventana y longitud igual al ancho de la imagen. Por ejemplo, para una imagen Full HD y ventana de 3x3 píxeles, el buffer de línea es de tamaño [3][1920].

Efecto de los bufferes en el algoritmo

El uso de estas estructuras de memoria tiene algunas consecuencias en el algoritmo. En primer lugar, es necesario un tiempo al inicio para llenar el buffer de línea con datos suficientes para computar el primer dato de salida. En segundo lugar, el algoritmo debe ejecutarse durante un número de iteraciones mayor que el número de datos de entrada disponibles para poder generar todos los datos de salida, debido al tiempo inicial en el que se reciben datos de entrada pero no se generan salidas.

Debido a este offset entre entradas y salidas, es necesario extender los bucles del cómputo del algoritmo, desde su forma original para una imagen de dimensiones [HEIGHT][WIDTH] (como puede ser [1080][1920]):

```
for(row=0; row<HEIGHT; row++){
  for(col=0; col<WIDTH, col++){
  ...
  }
}
```

A añadir una iteración más en cada bucle, filas y columnas:

```
for(row=0; row<HEIGHT+1; row++){
  for(col=0; col<WIDTH+1, col++){
  ...
  }
}
```

Ahora, es necesario añadir una restricción para realizar el número correcto de lecturas de píxeles de entrada (leer tantos como el tamaño de la imagen):

```
if(row<HEIGHT && col<WIDTH){
  line_buffer[2][col] = input_pixel;
  Manejo de los bufferes de linea y ventana
}
```

Para generar píxeles de salida a la vez que llegan píxeles de entrada, el dato de salida debe ir retrasado una línea y un píxel respecto de la entrada, tanto al comienzo como al final (cuando ya no llegan entradas):

```
if(row>0 && col>0){
  output_pixel = median_of_window;
}
```

4.2.2. Kernel de cómputo del algoritmo

Se entiende por kernel de cómputo de un algoritmo una subrutina que realiza una operación numérica común, particularmente diseñada para aceleradores hardware de altas prestaciones (como GPUs, DSPs o FPGAs).

En el caso de un filtro de mediana, consiste en el cálculo de la mediana de los píxeles contenidos en la ventana para cada posición en la imagen. Como se ha detallado anteriormente, la mediana puede ser obtenida ordenando numéricamente los elementos de la ventana y tomando aquel que ocupa la posición central en la lista. El ordenamiento de arrays es una de las operaciones más críticas en sistemas embebidos, además contar con numerosas aplicaciones. La teoría de los algoritmos de ordenamiento es profunda y compleja. Optimizar este cómputo es uno de los aspectos (junto a la estructura de bufferes) que permitirá dotar al algoritmo de un buen rendimiento.

Algoritmos y redes de ordenamiento

El algoritmo de la burbuja, usado en la versión software, es uno de los métodos más sencillos de implementar para ordenar arrays. Esencialmente es un algoritmo de fuerza bruta lógica que se ejecuta de forma secuencial [20].

Sin embargo, el rendimiento puede mejorarse considerablemente en FPGA describiendo una red de ordenamiento [21]. Estas estructuras ordenan un número fijo de valores, pudiendo aprovechar la ejecución paralela de varias comparaciones en hardware. La secuencia de comparaciones se conoce de antemano, por lo que tienen un comportamiento determinista en cuanto al tiempo que tardan en ordenar el conjunto de datos. Esto permite que los resultados se produzcan con un throughput conocido y constante.

Las redes de ordenamiento se construyen con comparadores de dos entradas y registros. Los comparadores pueden intercambiar los valores de su entrada si es necesario y tienen una latencia de un ciclo de reloj. Los registros son necesarios para retrasar un ciclo los elementos que no necesitan ser comparados en una de las etapas y para introducir pipeline.

Red de ordenamiento impar-par

En este caso, para ordenar los 9 valores almacenados en la ventana, se ha utilizado una red construida mediante el algoritmo de ordenamiento impar-par. Un esquema simplificado de dicha red puede verse en la figura 4.6.

Se comparan los elementos adyacentes de dos en dos y, si están en el orden incorrecto (el primero es más grande que el segundo) son intercambiados.

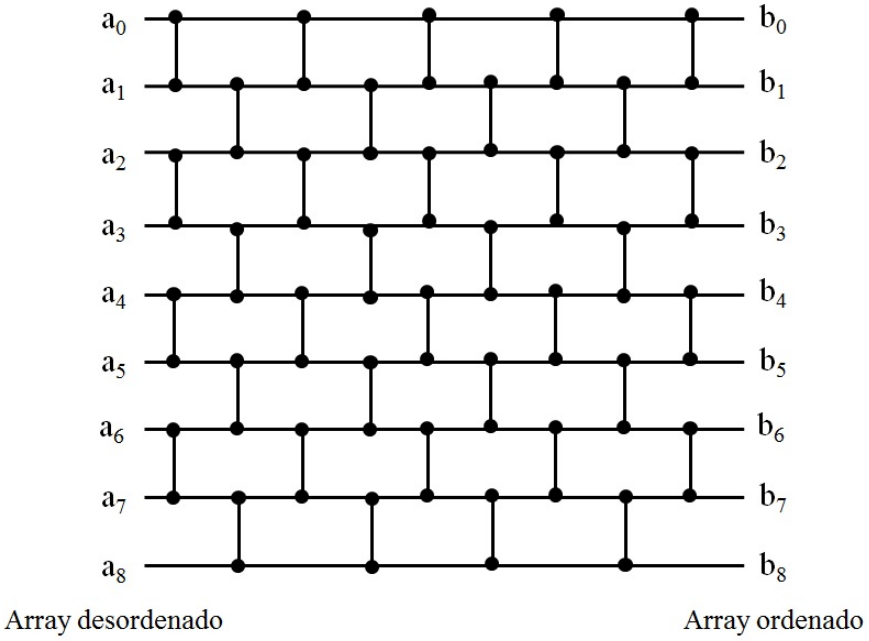

Figura 4.6: Red de ordenamiento impar-par para arrays de 9 elementos. Las interconexiones verticales representan comparadores.

En una etapa se comparan elementos con índice par-impar y en la siguiente elementos con índice impar-par, durante tantas etapas como elementos, hasta que la lista se encuentre ordenada.

El algoritmo de ordenamiento se ha descrito en C++ (sintetizable por HLS), para N=9 elementos, de la siguiente manera:

```
//Odd-Even Sorting Network

for (int stage = 0 ; stage < N ; stage++){
  //stage is odd
  if (stage & 1){
    for(i=2; i<N; i+=2){
      if (local[i] < local[i-1]){

      //Swap elements
      temp = local[i];
      local[i] = local[i-1];
      local[i-1] = temp;
      }
    }
  }
```

```
    //stage is even
    else{
      for(i=1; i<N; i+=2){
        if (local[i] < local[i-1]){

        //Swap elements
        temp = local[i];
        local[i] = local[i-1];
        local[i-1] = temp;
        }
      }
    }
  }
```

4.2.3. Síntesis final

Sintetizar el código C++ modificado para incluir los bufferes de memoria no proporciona unas altas prestaciones por sí solo, pero el código está preparado para sacar provecho de algunas directivas de síntesis de HLS.

Al igual que en el otro IP, el tamaño de la imagen a tratar está parametrizado; el módulo cuenta con dos registros en la interfaz al bus, donde se puede escribir por software el tamaño de la imagen a filtrar (hasta Full HD, 1920x1080).

Pipeline

Al algoritmo de ordenamiento se le aplica una directiva de pipeline con II=1, que permite sintetizar la red de ordenamiento registrando cada etapa y realizando todas las comparaciones de una etapa en paralelo.

El lazo interno del filtro espacial (el lazo que recorre las columnas de la imagen), que incluye el manejo de los bufferes y la red de ordenación para aplicar el kernel a cada píxel, también se configura con pipeline II=1.

De esta forma, la red de ordenamiento tiene una latencia de 9 ciclos de reloj. Después, se produce un array ordenado en cada ciclo. Además, un píxel de salida se produce en cada ciclo de reloj.

Elementos de memoria

Respecto a las estructuras de memoria generadas, el buffer de 3 líneas es particionado en 3 arrays de línea separados, de manera que cada uno de ellos se mapea en una block RAM de doble puerto diferente. Así, se puede acceder (lectura y escritura) a elementos de las 3 líneas en el mismo ciclo de reloj.

Por otra parte, el buffer de la ventana se particiona completamente, de forma que HLS lo mapea en 9 registros. El mapeado en memoria de los bufferes también se indica en la figura 4.5.

Resultados

Combinando las modificaciones del código con las directivas de síntesis, el throughput del IP supera los 74 fps para imagenes en Full HD (1920x1080). Todos los resultados temporales indicados en los informes de síntesis de HLS se muestran en la tabla 4.2.

medianBlur HW (1920x1080)			
Entrada HLS	Periodo de reloj	Ciclos de reloj	Throughput
Código C++ original	4.31 ns	553656603	0.42 fps
Código C++ modificado	4.26 ns	442318177	0.53 fps
Código C++ modificado y directivas de síntesis	6.39 ns	2091737	**74.82 fps**

Tabla 4.2: Prestaciones temporales del IP diseñado para *medianBlur*.

Capítulo 5

Integración HW-SW

La aplicación final se debe ejecutar sobre Linux en un SoC Zynq con un procesador basado en ARM. Como se concluyó en el capítulo 3, parte de la funcionalidad permanecerá en software, como la captura de vídeo utilizando la librería OpenCV, parte del procesado (incluyendo funciones OpenCV) y la presentación de resultados (coordenadas de los marcadores de referencia detectados). Otra parte de la funcionalidad del algoritmo es acelerada en hardware a través de los bloques IP diseñados en el capítulo 4.

Por tanto, es necesario integrar los IP generados en la plataforma hardware que se implementa en la lógica programable de Zynq y que permite la ejecución de Linux sobre este dispositivo. Por otra parte, es preciso manejarlos desde software para hacer uso de ellos en una aplicación de usuario bajo el control del sistema operativo.

5.1. Plataforma Hardware

Vivado IP Integrator

La plataforma hardware que configura la parte de FPGA de Zynq se construye con el software Vivado de Xilinx, en concreto utilizando el IP Integrator, una herramienta de Vivado para construir diseños grandes a partir de un diagrama de bloques IP que proporciona un nivel de abstracción más alto que un diseño RTL. Los bloques IP pueden provenir del usuario (ya sea desde HLS o VHDL/Verilog), de Xilinx o de terceros y se comunican entre sí usando interfaces estándar (como AXI).

Los productos de Vivado Design Suite han sido diseñados para el máximo aprovechamiento de las FPGAs de Xilinx. Además, no solo tienen soporte para estos dispositivos sino para plataformas completas, entre ellas todos los

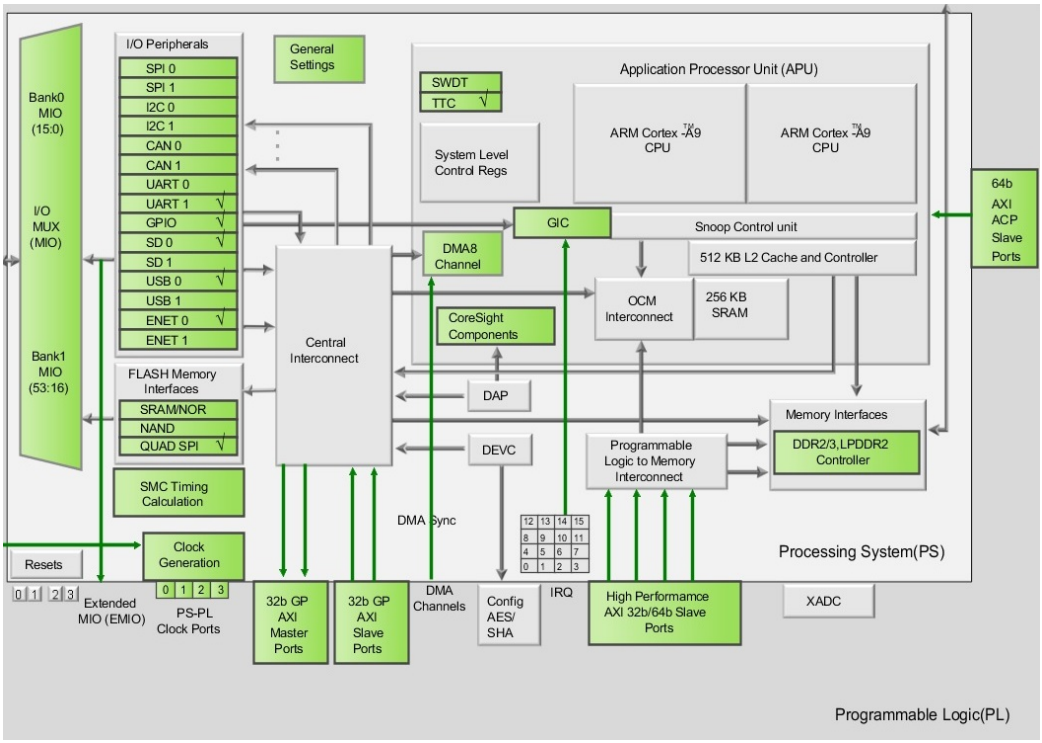

Figura 5.1: Sistema de procesado (PS) de Zynq preconfigurado para ZedBoard Zynq Evaluation and Development Kit en Vivado IP Integrator.

Zynq All Programmable SoC y varias placas de desarrollo, entre las que se encuentra la ZedBoard.

Por ejemplo, al elegir como plataforma de destino la ZedBoard e instanciar en IP Integrator un sistema de procesado (PS) de Zynq, Vivado preconfigura el PS con los periféricos, drivers y mapeado de memoria adecuado para soportar la placa (ver figura 5.1). Después, se especifica la interfaz entre el sistema de procesado y la lógica programable (PL) y se pueden añadir los bloques IP del usuario o de terceros para completar el diseño, siendo las interfaces automáticamente generadas [23].

Sistema hardware completo

En este caso se utiliza una plataforma hardware de partida, que es la mencionada en el capítulo 3, proporcionada por Analog Devices para permitir la ejecución de Linux sobre Zynq utilizando salida de vídeo y audio. A esta base es necesario añadir los IPs de procesado de vídeo diseñados. Los IPs generados se conectan al sistema utilizando dos interfaces AXI Stream

(entrada y salida), una interfaz AXI, una entrada de reloj y otra de reset [24].

Para acceder a la memoria principal y leer las imágenes capturadas por el software y escribir las imágenes producidas, los IPs hardware se conectan a un IP AXI VDMA (Video Direct Memory Access) proporcionado por Xilinx. Usar un DMA (Direct Memory Access) permite leer y escribir en la memoria principal del sistema (la memoria RAM DDR3 de 512 MB situada fuera del SoC Zynq) sin utilizar la CPU.

En la figura 5.2 se presenta un esquema simplificado del sistema completo, en el que se muestran la lógica programable (PL), el sistema de procesado (PS) y la memoria principal DDR3. Dentro de la PL se encuentran los IPs de procesado de vídeo, junto a dos módulos de entrada y salida adicionales cuya utilidad se explicará en apartados posteriores, el VDMA y los componentes necesarios para llevar a cabo las comunicaciones PL-PS.

A partir de esta figura se desarrollan los siguientes apartados del capítulo, atendiendo a las siguientes comunicaciones hardware-software:

- **Comunicación IPs - Memoria.** Se realiza utilizando DMA (Direct Memory Access). Se ha dividido en dos partes:
  - Comunicación VDMA - Memoria.
  - Comunicación VDMA - IPs.
- **Comunicación IPs - CPU.** Se realiza utilizando el método de IPs hardware asignados al espacio de memoria (memory-mapped).

En la figura 5.3 se muestra la vista del diagrama de bloques real en Vivado IP Integrator, con un nivel de zoom que permite visualizar el diseño completo.

Señales de reloj y reset

La línea de reloj de los IPs se conecta al generador de reloj interno de la lógica programable, configurado a 100 MHz. Por tanto, los módulos funcionan con un periodo de reloj de 10 ns. Este reloj no es el mismo que utiliza la CPU (el reloj del dual-core ARM está configurado a 666.67 MHz) ni la memoria DDR3 (el reloj de la memoria RAM está configurado a 533.33 MHz), como se ve en la figura 5.4.

La señal de reset de los IPs se conecta al bloque Processor System Reset, que se encarga de generar la señal de reset de todos los periféricos y está controlado por el procesador, por lo que puede ser utilizada por software.

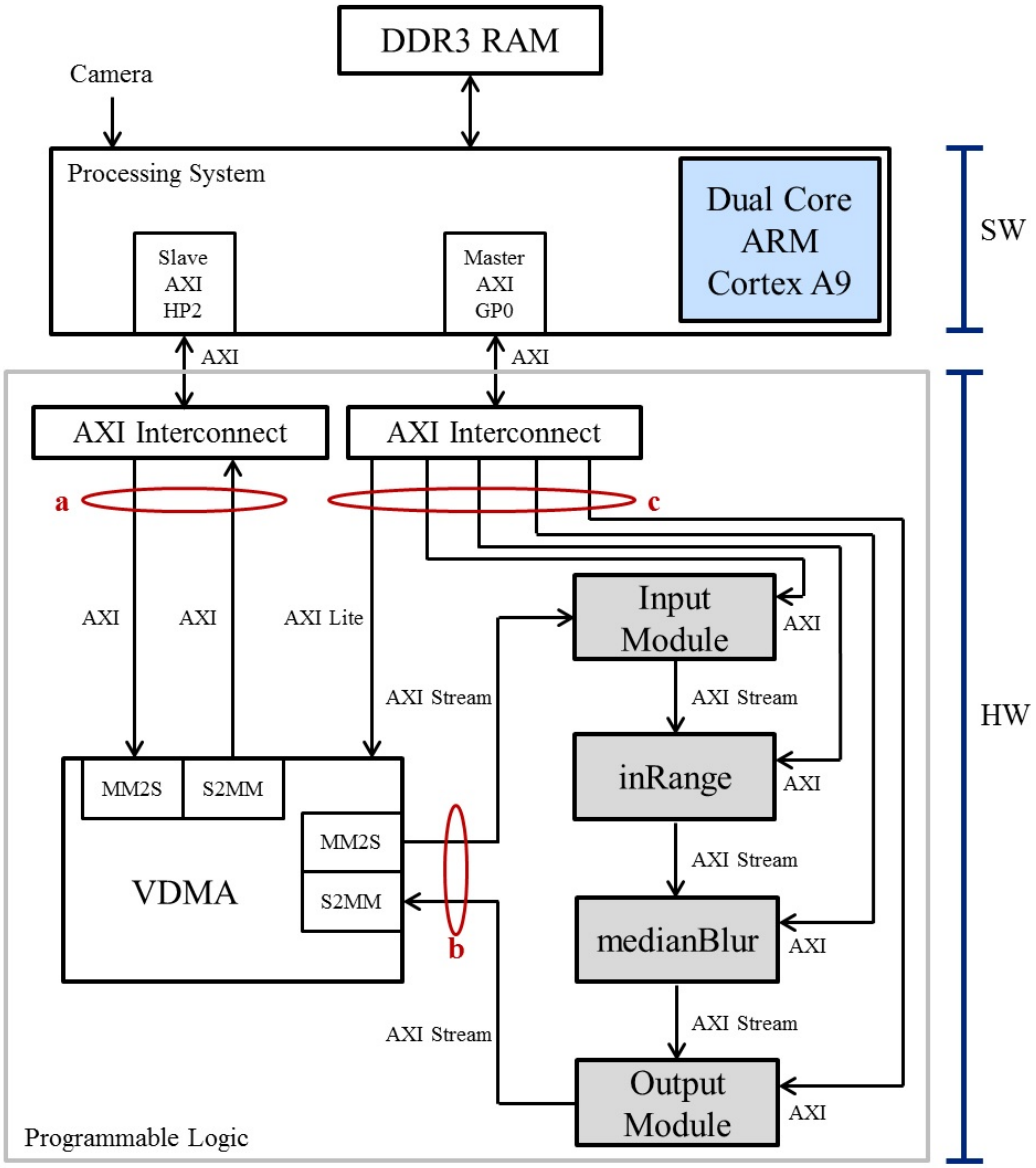

Figura 5.2: Diagrama de bloques simplificado del hardware utilizado. Comprende el sistema de aceleración hardware completo, el sistema de procesado y la memoria RAM DDR3. Se señalan las comunicaciones entre elementos del sistema a partir de las cuales se desarrollan los siguientes apartados del capítulo: Comunicación VDMA - Memoria (a), Comunicación VDMA - IPs (b) y Comunicación IPs - CPU (c).

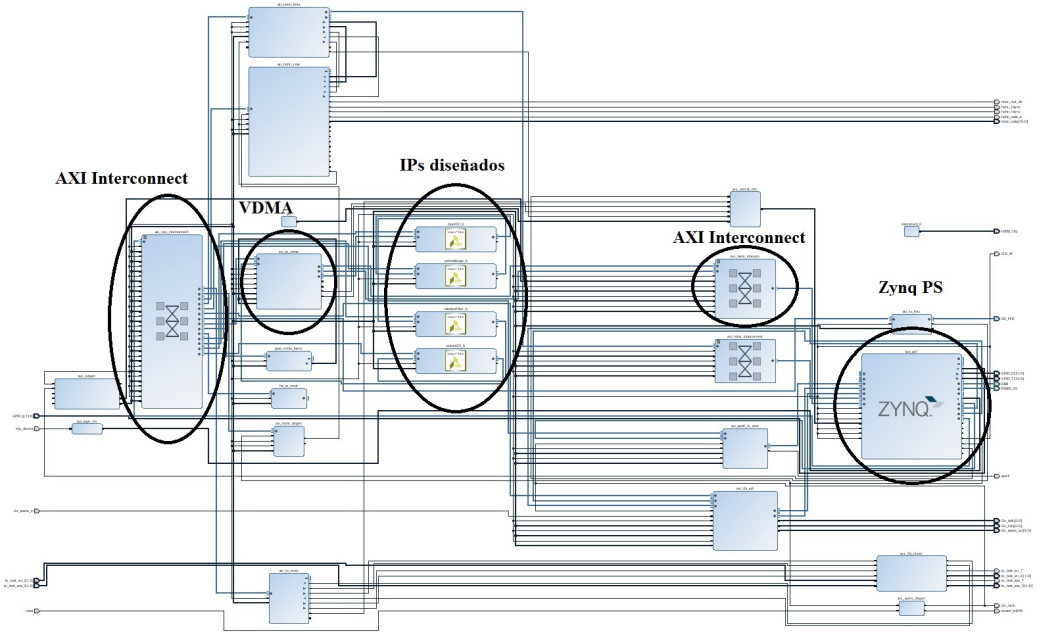

Figura 5.3: Diagrama de bloques de la plataforma hardware completa en Vivado IP Integrator.

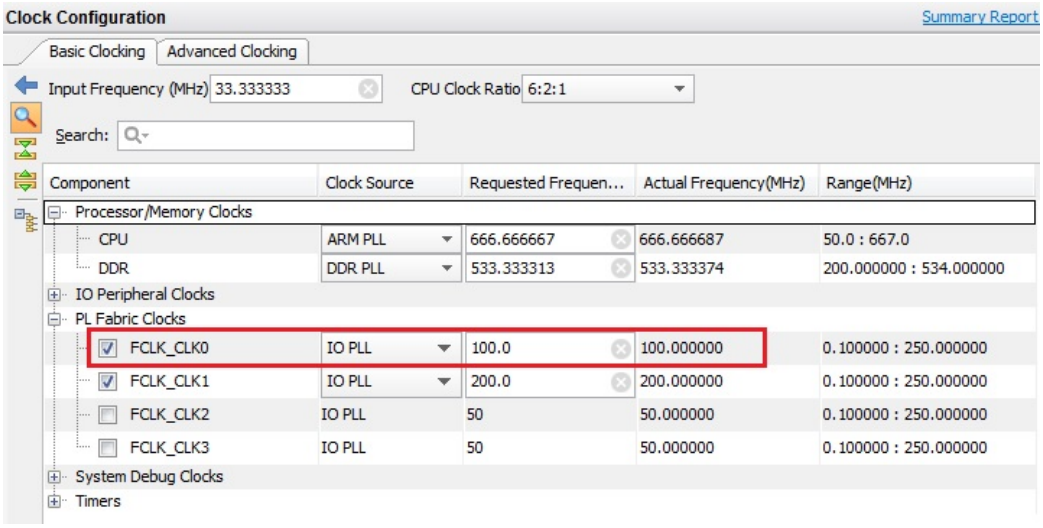

Figura 5.4: Configuración de los diferentes relojes del sistema (FPGA, CPU y DDR3) en el bloque Zynq PS en Vivado IP Integrator.

5.2. Comunicación IPs - Memoria

5.2.1. Comunicación VDMA - Memoria

Utilizando DMA (Direct Memory Access) se pueden realizar transferencias independientemente de la CPU. La CPU configura e inicia el DMA, realiza otras operaciones mientras está en progreso y finalmente recibe una interrupción del controlador DMA cuando la operación ha terminado. Así, se evita sobrecargar la CPU en aplicaciones que realicen un uso intensivo de la memoria. No obstante, aunque la CPU no sea necesaria para la transferencia, sí se necesita el bus del sistema [25].

Las interfaces AXI (maestro y esclavo, MM2S y S2MM) del VDMA se conectan a un bloque AXI Interconnect, que a su vez las conecta al puerto S_AXI_HP2 (Slave High Performance AXI) del sistema de procesado. Este puerto es una de las interfaces AXI de alto rendimiento entre los elementos de memoria del sistema de procesado (como la RAM DDR3) y la lógica programable, formada por un bus de datos de 32 o 64 bits. La parte de PL actúa siempre como maestro y el PS como esclavo. Esta conexión incluye bufferes FIFO para acomodar la lectura y escritura de ráfagas de datos y soportar comunicaciones de alta velocidad.

5.2.2. Comunicación VDMA - IPs

Con el objetivo de adaptar la lectura/escritura de datos desde memoria por el VDMA a la interfaz usada por los IPs de procesado de vídeo, se introducen dos módulos hardware adicionales (uno a la entrada y otro a la salida de la cadena de procesado) cuyo diseño se aborda en este apartado.

Almacenamiento de las imágenes en memoria

Es importante entender cómo almacena OpenCV las imágenes en memoria. Las imágenes a procesar son capturadas con una cámara digital utilizando funciones de la librería OpenCV y almacenadas en memoria en formato *Mat*. *Mat* es básicamente una clase con dos partes de datos: la cabecera de la matriz (que contiene información como su tamaño, el tipo de imagen en cuanto a canales y profundidad de color, etc) y un puntero a la matriz que contiene los datos de los píxeles.

El tamaño de la matriz de píxeles del formato *Mat* depende del número de canales y de la profundidad de color (lo más común es utilizar 8 bits por canal, de manera que un píxel en escala de grises ocupa 1 byte y un píxel en

	Column 0	Column 1	Column ...	Column m
Row 0	0,0	0,1	...	0, m
Row 1	1,0	1,1	...	1, m
Row ...	...,0	...,1	...	..., m
Row n	n,0	n,1	n,...	n, m

Figura 5.5: Matriz de píxeles para una imagen en escala de grises almacenada en memoria utilizando la clase *Mat* de OpenCV para C++.

	Column 0			Column 1			Column ...			Column m		
Row 0	0,0	0,0	0,0	0,1	0,1	0,1	...	...	...	0, m	0, m	0, m
Row 1	1,0	1,0	1,0	1,1	1,1	1,1	...	...	...	1, m	1, m	1, m
Row ...	...,0	...,0	...,0	...,1	...,1	...,1	...	...	...	..., m	..., m	..., m
Row n	n,0	n,0	n,0	n,1	n,1	n,1	n,...	n,...	n,...	n, m	n, m	n, m

Figura 5.6: Matriz de píxeles para una imagen en color BGR almacenada en memoria utilizando la clase *Mat* de OpenCV para C++.

color ocupa 3 bytes). Las matrices de imágenes en escala de grises (un solo canal) son almacenadas en memoria como se muestra en la figura 5.5. Las matrices de imágenes en color (con tres canales, RGB) son almacenadas en memoria utilizando una subcolumna para cada canal, como se muestra en la figura 5.6. OpenCV utiliza el sistema de color BGR. Como en la mayoría de los casos el tamaño de la memoria es lo suficientemente grande, las filas se almacenan una tras otra formando una única larga fila, lo cual facilita el proceso de lectura [22].

Para adaptarse a un bus estándar con tamaño de palabra de datos de 32 bits, los bloques hardware diseñados en el capítulo 4 trabajan con interfaces de entrada y salida de 32 bits (recibiendo o transmitiendo un píxel, ya sea en color o escala de grises, en cada palabra). En la figura 5.7 se describen estas interfaces.

Como se ha comentado anteriormente, para acceder a los datos de la imagen capturada por el software, los IPs hardware se conectan a un módulo AXI VDMA (Video Direct Memory Access) que permite al hardware leer y escribir en la memoria principal del sistema. Como el VDMA lee de memoria palabras de 32 bits y OpenCV utiliza la memoria de manera continua (esto es, en 32 bits almacena datos de 4 canales, lo que significa 4 píxeles en imágenes en escala de grises o bien un píxel y un canal del siguiente en imágenes en color), cada palabra de memoria que lee el VDMA no se corresponde con un píxel como requiere el hardware.

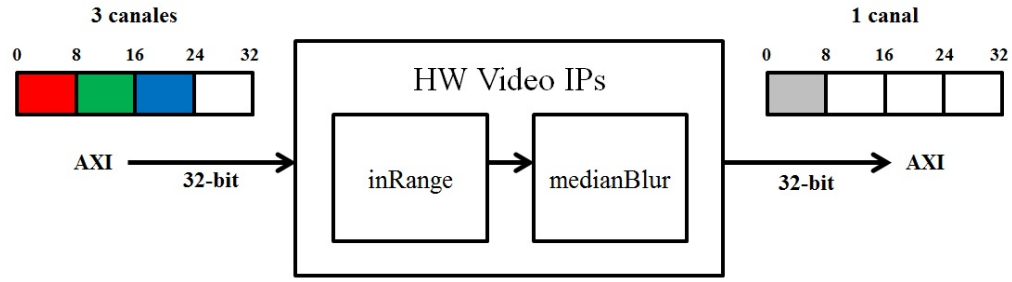

Figura 5.7: Interfaces de entrada y salida de los bloques hardware diseñados (los píxeles están contenidos en palabras de 32 bits).

Solución hardware para el paso de datos

Para solucionar esto, es necesario añadir a la entrada y a la salida del hardware dos módulos para adaptar las interfaces: de las palabras que el DMA lee de memoria (más de un píxel por palabra) a las que utiliza el hardware (un píxel por palabra). Para ilustrar todo este proceso, en la figura 5.8 se muestra el camino que siguen los datos de la imagen al leerse de memoria, procesarse en hardware y guardarse de nuevo en memoria, prestando atención a las interfaces entre bloques para adaptar el paso de datos.

De esta forma, el VDMA lee una palabra de memoria que contiene un píxel completo y el canal azul del siguiente píxel. El módulo de entrada pasa un píxel a los IPs de procesado, y almacena el canal azul sobrante. Cuando el VDMA lee la siguiente palabra, el módulo junta el canal azul almacenado con el verde y rojo del mismo píxel y lo envía, guardando los canales correspondientes al tercer píxel. Siguiendo este procedimiento, cuando el VDMA ha leído 3 palabras, el módulo completa la transferencia de 4 píxeles completos, que han sido transmitidos a los IPs de procesado en 4 palabras de 32 bits.

En cuanto a la salida del hardware, los IPs de vídeo producen un píxel en blanco y negro y lo transmiten en una palabra de 32 bits. El módulo de adaptación de salida almacena el byte que contiene la información y, cuando ha juntado 4 píxeles, los transmite a memoria en una palabra de 32 bits a través del VDMA.

Los módulos de adaptación son bloques IP hardware diseñados con Vivado HLS que funcionan en pipeline con el resto del sistema. Su comportamiento es el de una máquina de 4 estados (ya que la lectura de 3 palabras de memoria se corresponde a 4 píxeles RGB completos y la escritura en memoria de una palabra se corresponde a 4 píxeles blanco y negro completos), como se describe en la figura 5.9.

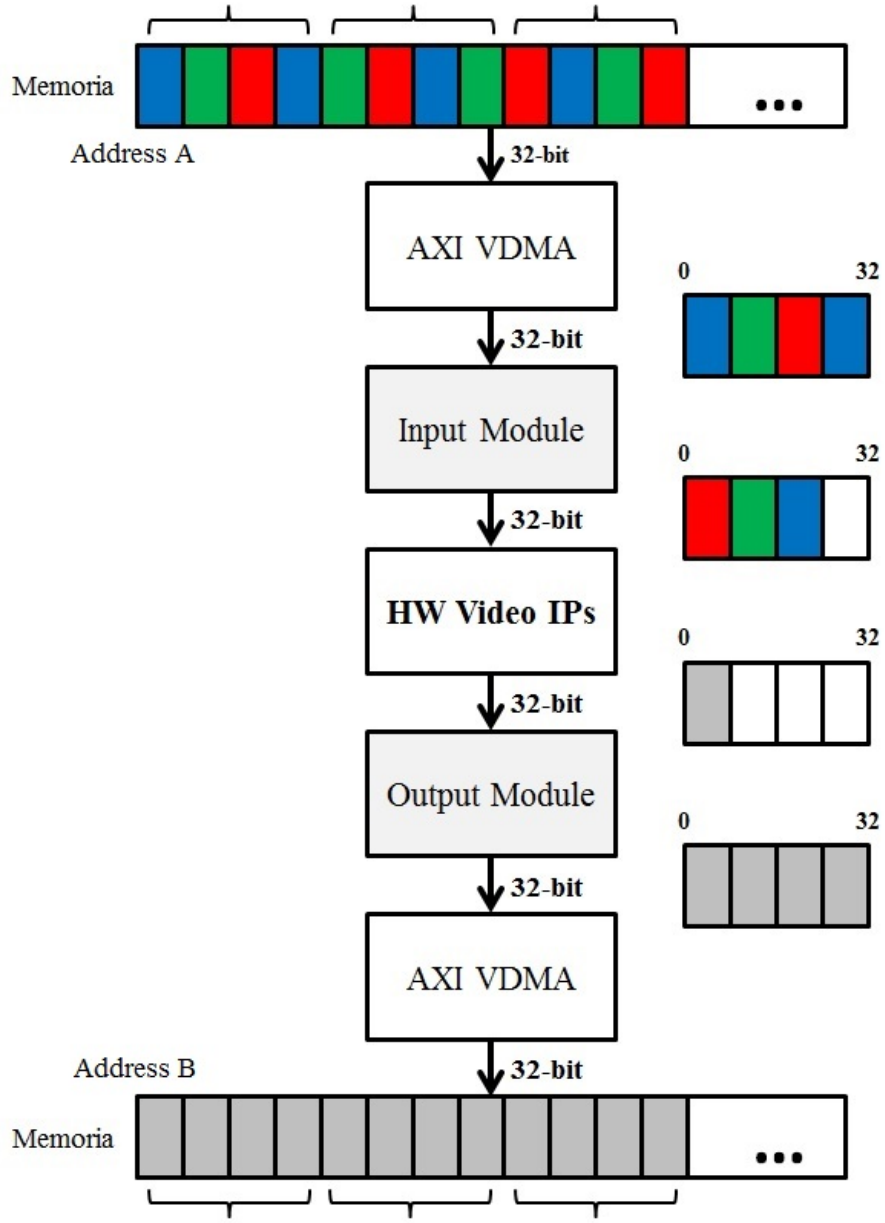

Figura 5.8: Camino seguido por los datos de los píxeles cuando se leen de memoria, se procesan en hardware y regresan a memoria, destacando la forma de transmitir los datos entre los diferentes elementos hardware.

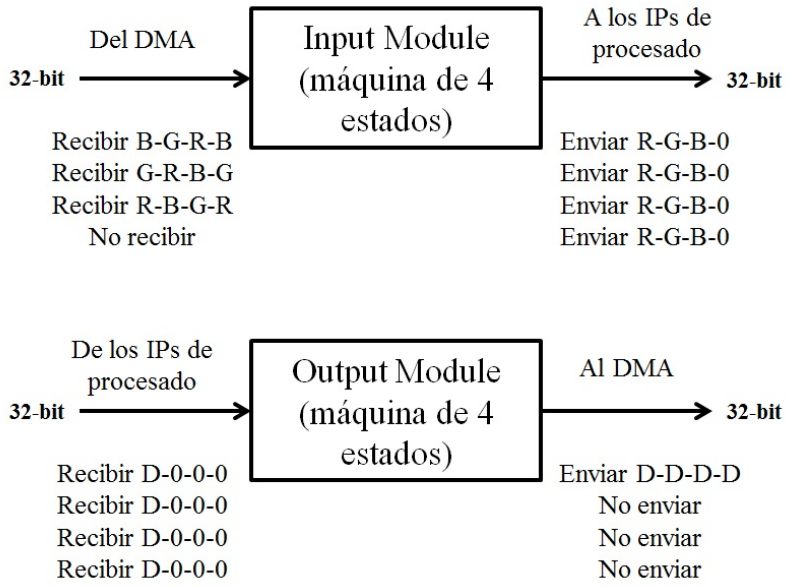

Figura 5.9: Descripción del funcionamiento de los IPs de adaptación de interfaces: de almacenamiento software en bytes de forma continua a una palabra de 32 bits por cada píxel y viceversa.

Conexiones hardware entre el VDMA y los IPs

Los datos pasan de unos bloques a otros serializados o "en chorro" utilizando la interfaz AXI Stream y todos los bloques funcionan en pipeline. Por tanto, la interfaz AXI Stream maestro (MM2S) del VDMA, encargada de enviar a los IPs los datos sacados de memoria, se conecta a la entrada AXI Stream del módulo de adaptación de entrada. La interfaz AXI Stream esclavo (S2MM) del VDMA, encargada de recibir datos de los IPs para enviarlos a memoria, se conecta a la salida AXI Stream del módulo de adaptación de salida.

Conexiones hardware entre los IPs de procesado

Las entradas y salidas AXI Stream restantes de los IPs se conectan entre ellas siguiendo el orden de ejecución (esto es, la salida de datos de un IP se conecta a la entrada del siguiente en la cadena de procesado), como se ve en la figura 5.10. Cabe recordar que las conexiones AXI Stream no se limitan al bus de datos de 32 bits, sino que incluyen todas las señales de protocolo necesarias para la correcta comunicación entre IPs.

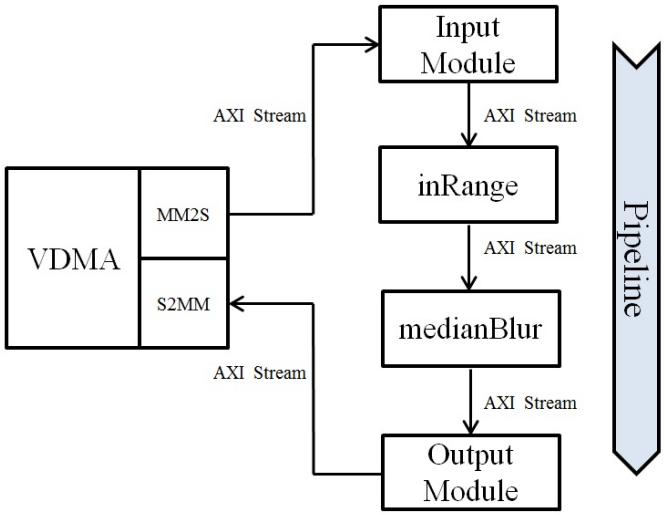

Figura 5.10: Conexionado de las interfaces de datos AXI Stream del VDMA y los IPs de procesado de vídeo.

5.3. Comunicación IPs - CPU

Conexiones hardware entre los IPs y la CPU

La interfaz AXI de los IPs se conecta al bloque AXI Interconnect, que a su vez los conecta al puerto M_AXI_GP0 (Master General Purpose AXI) del sistema de procesado. Dicho puerto es una de las interfaces AXI de propósito general entre el sistema de procesado y la lógica programable, formada por un bus de datos de 32 bits adecuado para comunicaciones de baja o media velocidad, directa y sin buffer. En esta conexión, el PS actúa como maestro (como indica la letra M) y la PL como esclavo. Esta conexión es usada por la CPU para configurar los parámetros de los bloques IP (como la resolución de las imágenes a procesar) desde software en tiempo de ejecución, escribiendo en los registros del bloque hardware.

La interfaz AXI Lite del VDMA se conecta de la misma manera al módulo AXI Interconnect, para ser controlado desde la CPU a través del puerto M_AXI_GP0. La CPU se encarga de configurar e iniciar la transferencia por DMA entre la memoria y los IPs cuando sea necesario, desde la aplicación de usuario. La configuración del controlador de acceso directo a memoria incluye las direcciones de memoria fuente y destino de los datos que se transferirán y el número de bytes a transferir.

Comunicación desde software entre los IPs y la CPU

En cuanto a la comunicación entre la CPU y los IPs, se utiliza el método de mapear los periféricos hardware en memoria. Este esquema consiste en utilizar el mismo bus de direcciones para la memoria y los módulos hardware, de forma que las instrucciones de la CPU para acceder a memoria se utilizan también para acceder a los periféricos.

Los periféricos cuentan con una serie de registros hardware (y en ocasiones memorias) que constituyen lo que se denomina interfaz al bus. Estos registros de los dispositivos comparten el espacio de direcciones con la memoria principal del sistema (por lo que pueden leerse y escribirse como si se tratara de memoria convencional) y se utilizan para recibir o enviar información a los periféricos (ver figuras 5.11 y 5.12).

Las direcciones de memoria asignadas a todos los IPs hardware son proporcionadas por Vivado IP Integrator, tras ejecutar la opción de asignar direcciones, como se ve en la figura 5.13.

La interfaz al bus de los IPs diseñados es automáticamente generada por Vivado HLS. Por ejemplo, en la figura 5.14 se muestra la interfaz generada para el IP de *medianBlur*. Por otro lado, la interfaz al bus del AXI VDMA puede encontrarse en la guía del IP proporcionada por Xilinx [26].

Si, por ejemplo, se quiere configurar el IP de *medianBlur* para procesar imágenes de tamaño 1920x1080, se obtiene la dirección virtual (la que maneja el sistema operativo) correspondiente a la dirección física base del IP (que es 0x43C00000), se suma el offset necesario para los registros *height* y *width* (es decir, 0x14 y 0x1C respectivamente) y se escribe en las direcciones resultantes el valor deseado utilizando punteros. En este caso, se escribiría 1080 en la dirección del registro *height* y 1920 en la del registro *width*. Se procede de la misma manera para utilizar las señales del registro de control, pero en este caso es necesario utilizar máscaras que permitan escribir o leer a nivel de bit (ya que el bus solo permite lecturas o escrituras de 32 bits).

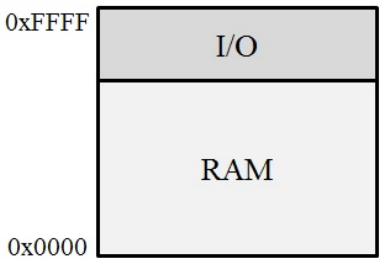

Figura 5.11: Espacio de memoria virtual al utilizar I/O mapeada en memoria.

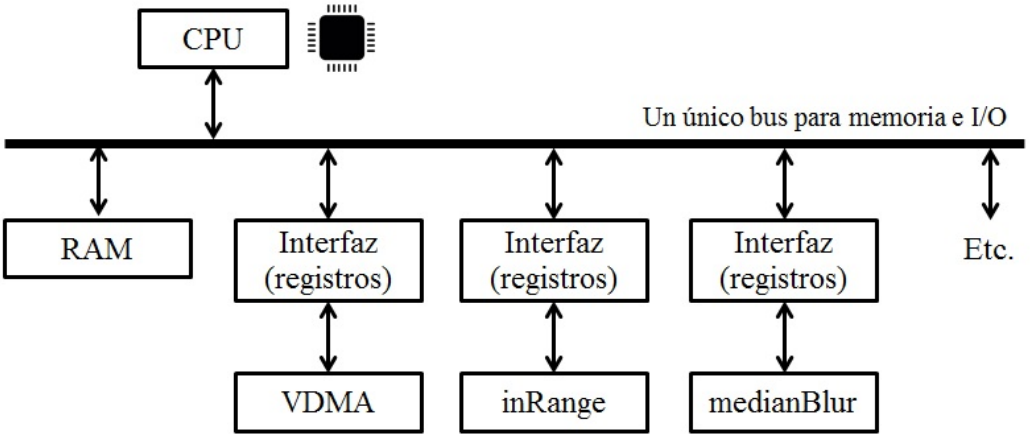

Figura 5.12: Arquitectura del sistema cuando el control de periféricos hardware desde la CPU se realiza utilizando I/O mapeada en memoria.

axi_hdmi_dma					
Data_MM2S (32 address bits : 4G)					
sys_ps7	S_AXI_HP0	HP0_DDR...	0x0000_0000	512M	0x1FFF_FFFF
sys_ps7					
Data (32 address bits : 0x40000000 [1G])					
axi_hdmi_clkgen	s_axi	axi_lite	0x7900_0000	64K	0x7900_FFFF
axi_hdmi_core	s_axi	axi_lite	0x70E0_0000	64K	0x70E0_FFFF
axi_spdif_tx_core	S_AXI	reg0	0x75C0_0000	64K	0x75C0_FFFF
axi_i2s_adi	S_AXI	reg0	0x7760_0000	64K	0x7760_FFFF
axi_hdmi_dma	S_AXI_LITE	Reg	0x4300_0000	64K	0x4300_FFFF
axi_iic_fmc	S_AXI	Reg	0x4162_0000	64K	0x4162_FFFF
axi_iic_main	S_AXI	Reg	0x4160_0000	64K	0x4160_FFFF
axi_ip_vdma	S_AXI_LITE	Reg	0x4301_0000	64K	0x4301_FFFF
hls_ip_reset	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
gpio_vdma_fsync	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
medianFilter_0	S_AXI_CONTROL_BUS	Reg	0x43C0_0000	64K	0x43C0_FFFF
withinRange_0	S_AXI_CONTROL_BUS	Reg	0x43C1_0000	64K	0x43C1_FFFF
inputCV_0	S_AXI_CONTROL_BUS	Reg	0x43C2_0000	64K	0x43C2_FFFF
outputCV_0	S_AXI_CONTROL_BUS	Reg	0x43C3_0000	64K	0x43C3_FFFF
axi_ip_vdma					
Data_MM2S (32 address bits : 4G)					
sys_ps7	S_AXI_HP2	HP2_DDR...	0x0000_0000	512M	0x1FFF_FFFF
Data_S2MM (32 address bits : 4G)					
sys_ps7	S_AXI_HP2	HP2_DDR...	0x0000_0000	512M	0x1FFF_FFFF

Figura 5.13: Direcciones de memoria asignadas a los IPs hardware del diseño por la herramienta IP Integrator de Vivado.

```
// File generated by Vivado(TM) HLS - High-Level Synthesis

// CONTROL_BUS
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/COH)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x10 : reserved
// 0x14 : Data signal of height
//        bit 31~0 - height[31:0] (Read/Write)
// 0x18 : reserved
// 0x1c : Data signal of width
//        bit 31~0 - width[31:0] (Read/Write)
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on W

#define XMEDIANFILTER_CONTROL_BUS_ADDR_AP_CTRL     0x00
#define XMEDIANFILTER_CONTROL_BUS_ADDR_GIE         0x04
#define XMEDIANFILTER_CONTROL_BUS_ADDR_IER         0x08
#define XMEDIANFILTER_CONTROL_BUS_ADDR_ISR         0x0c
#define XMEDIANFILTER_CONTROL_BUS_ADDR_HEIGHT_DATA 0x14
#define XMEDIANFILTER_CONTROL_BUS_BITS_HEIGHT_DATA 32
#define XMEDIANFILTER_CONTROL_BUS_ADDR_WIDTH_DATA  0x1c
#define XMEDIANFILTER_CONTROL_BUS_BITS_WIDTH_DATA  32
```

Figura 5.14: Interfaz al bus para el IP de *medianBlur* generada por Vivado HLS, detallando los registros hardware, su dirección (offset) y sus bits.

Entrando en detalle con el ejemplo mencionado, a continuación se presentan las partes más importantes del código C++ para el control del IP de *medianBlur* (llamado *medianFilter* para evitar la confusión con la función software).

Se puede obtener una dirección virtual para el IP, conocida su dirección física, utilizando la instrucción *mmap()*, que mapea dispositivos o archivos en el espacio de direcciones virtuales [27]. Para obtener un puntero `ptrIPMedian` a la dirección base virtual del IP, se procede de la siguiente forma:

```
#define XMEDIANFILTER_BASEADDR 0x43C00000

int fd;

long sz = sysconf(_SC_PAGESIZE);  // _SC_PAGESIZE = 4096

printf("Opening the dev memory %d\n", sysconf(_SC_PAGESIZE)
    );

fd = open("/dev/mem", O_RDWR);

if(fd < 1) {
  perror("Opening /dev/mem \n");
  exit(-1);
}else
  printf("File open with %d index\n",fd);

ptrIPMedian = (int *) mmap(NULL,sz, (PROT_READ|PROT_WRITE),
     MAP_SHARED, fd, XMEDIANFILTER_BASEADDR);

if(ptrIPMedian == MAP_FAILED) {
  perror("Error with mmap");
  exit(-1);
}
```

Para facilitar la escritura en los registros hardware del IP, se definen dos macros (una para configurar la altura y otra para el ancho de la imagen). Consisten en dos instrucciones que escriben estos valores en los registros correspondientes, a partir de la dirección base del IP. Notar que para sumar a la dirección base el offset correspondiente a los registros, dicho offset se divide entre 4 (desplazamiento de dos bits a la derecha) debido a la aritmética de punteros del lenguaje C/C++ (incrementar un puntero a entero en 0x4 bytes equivale a sumar 0x1 en C/C++, ya que el tamaño de un entero son 4 bytes):

```
#define XMEDIANFILTER_SetHeight(InstancePtr, Data)  ( *(
    InstancePtr + (
    XMEDIANFILTER_CONTROL_BUS_ADDR_HEIGHT_DATA>>2)) ) = Data
    ;

#define XMEDIANFILTER_SetWidth(InstancePtr, Data)   ( *(
    InstancePtr + (XMEDIANFILTER_CONTROL_BUS_ADDR_WIDTH_DATA
    >>2)) )  = Data;
```

Por último, se pueden utilizar estas macros para configurar las dimensiones de la imagen a procesar en el IP de forma sencilla en una función de inicialización que se utiliza en la aplicación de usuario:

```
int initIPMedian(int height, int width){

*ptrHLSReset = 0; //reset IPs
*ptrHLSReset = 1; //release reset
printf("Initializing IP MedianFilter\n");

XMEDIANFILTER_SetHeight(ptrIPMedian, height);
XMEDIANFILTER_SetWidth(ptrIPMedian, width);
XMEDIANFILTER_DisableAutoRestart(ptrIPMedian);

printf("IP MedianFilter init successful\n");

return 0;
}
```

Uso de la memoria con sistema operativo y MMU

Uno de los problemas que surgen al utilizar sistema operativo es que la MMU (Memory Management Unit) es la responsable del manejo de los accesos a memoria por parte de la CPU. La función principal de la MMU es gestionar la separación entre direcciones de memoria físicas y lógicas (o virtuales), realizando la traducción entre ambas. Su funcionamiento se muestra en el esquemático de la figura 5.15.

La instrucción *mmap()* permite al usuario obtener una dirección virtual correspondiente a una dirección física. Por otra parte, no es posible que el usuario obtenga o conozca una dirección física correspondiente a una dirección virtual dada por el sistema operativo.

El DMA necesita ser configurado con las direcciones físicas de la memo-

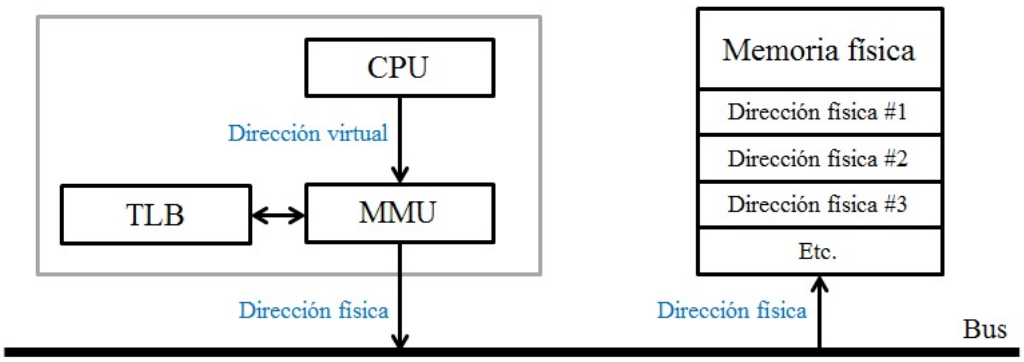

Figura 5.15: Funcionamiento de la MMU, traduciendo direcciones virtuales manejadas por la CPU a direcciones físicas de la memoria.

ria DDR3 donde los datos deben leerse y escribirse. Usando *mmap()*, las direcciones lógicas correspondientes pueden ser obtenidas. Sin embargo, las declaraciones de imagen en OpenCV retornan direcciones virtuales para el buffer de la imagen que el usuario no puede elegir, es decir, realizan la operación de *malloc* internamente. Para hacer posible la transferencia de datos, es necesario hacer coincidir las direcciones virtuales que asignan las funciones de la librería OpenCV y las direcciones virtuales correspondientes a la parte de memoria física a la que accede el DMA.

Para encontrar una solución a este problema, se ha observado que la función de OpenCV *Videocapture()* lee el puntero a la matriz de datos de la imagen capturada previamente para asignar el mismo buffer de memoria cuando se utiliza en un bucle. Entonces, cuando el primer fotograma de vídeo es capturado por la cámara, el puntero a los datos de la imagen proporcionado por OpenCV es reemplazado con el valor de la dirección virtual donde el DMA ha sido configurado para leer los datos y después este puntero es leído y mantenido por la función *Videocapture()* durante los fotogramas siguientes.

En cuanto a la imagen producida por el hardware, el puntero a la matriz de datos de la imagen proporcionado por OpenCV es reemplazado por el valor de la dirección virtual donde el DMA ha sido configurado para escribir los datos de la imagen procesada. En la figura 5.16 se resume la estrategia explicada para la transferencia de las imágenes.

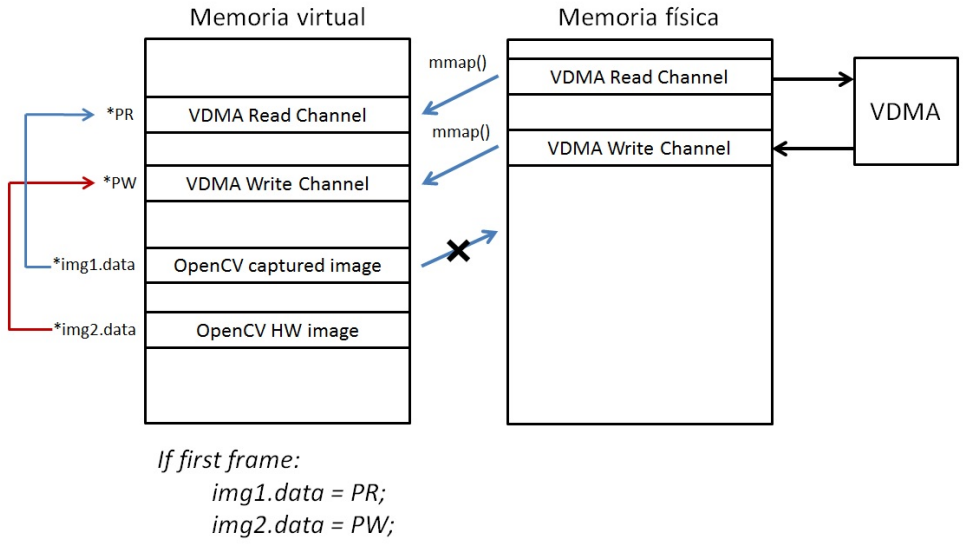

Figura 5.16: Solución software para hacer corresponder las direcciones virtuales dadas por OpenCV para las imágenes y las direcciones virtuales mapeadas desde la memoria física usadas por el VDMA.

Capítulo 6

Resultados

Una vez que se han integrado los bloques hardware diseñados en la plataforma final y se ha conseguido utilizar las comunicaciones hardware-software necesarias desde la aplicación de usuario en Linux, es momento de estudiar las prestaciones del sistema acelerado por hardware en comparación a las de la versión original puramente software.

En definitiva, en este capítulo se analiza el rendimiento de la aplicación ejecutándose sobre la plataforma final (el SoC Zynq, haciendo uso tanto del microprocesador ARM Cortex A9 como de la lógica programable). Además de presentar los tiempos de ejecución de las partes más significativas del algoritmo, que pueden compararse con los expuestos en el capítulo 3.3 para la versión software, se constata la significativa mejora en el throughput global probando así el éxito de la implementación.

6.1. Imágenes VGA (640x480)

Procesando imágenes VGA (640x480), la ejecución del algoritmo en software (la versión preliminar completamente en C++) tiene un throughput de 11 fps (fotogramas por segundo). Usando el sistema hardware diseñado (aceleración hardware), se alcanza una tasa de fotogramas de 57 fps. Esto significa una ejecución alrededor de 5.2 veces más rápida.

En la tabla 6.1 aparecen los tiempos de ejecución del algoritmo completo, de la parte hardware y de la función software más representativa (al reducirse el porcentaje de tiempo sobre el total del algoritmo de las funciones implementadas en hardware, el porcentaje de las funciones software aumenta). La figura 6.1 muestra gráficamente el tiempo de ejecución del algoritmo para imágenes VGA, tanto en software como utilizando los IPs hardware.

6.2. Imágenes FHD (1920x1080)

Procesando imágenes en resolución FHD (1920x1080), la versión software del algoritmo solamente puede procesar 1 fps (fotograma por segundo). Usando aceleración hardware, se consigue un throughput de 8 fps. Esto se traduce en una mejora significativa de las prestaciones (8 veces más rápido).

En la tabla 6.2 aparecen los tiempos de ejecución del algoritmo completo, de la parte hardware y de la función software más representativa. La figura 6.2 muestra gráficamente el tiempo de ejecución del algoritmo para imágenes FHD, tanto en software como utilizando los IPs hardware.

Detección de marcadores VGA (640x480) HW-SW		
Función	Tiempo de ejecución	Porcentaje del total
MarkerDetection	17459126 ns	-
Hardware (medianBlur + inRange)	3639798 ns	20.8 %
cv::findContours	6580649 ns	37.7 %

Tabla 6.1: Tiempo de ejecución de las funciones con mayor carga computacional del algoritmo, para imágenes VGA (640x480), utilizando aceleración hardware.

Detección de marcadores FHD (1920x1080) HW-SW		
Función	Tiempo de ejecución	Porcentaje del total
MarkerDetection	117574604 ns	-
Hardware (medianBlur + inRange)	21889572 ns	18.6 %
cv::findContours	48384421 ns	41.2 %

Tabla 6.2: Tiempo de ejecución de las funciones con mayor carga computacional del algoritmo, para imágenes FHD (1920x1080), utilizando aceleración hardware.

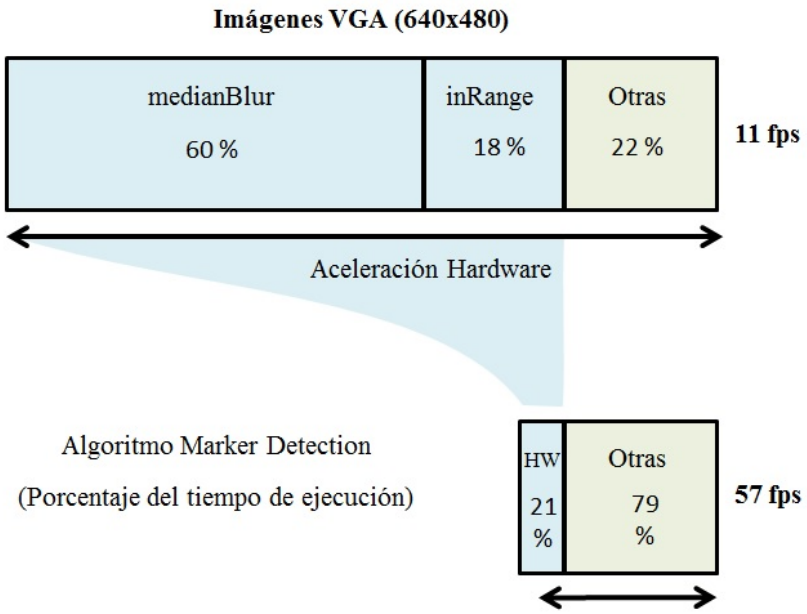

Figura 6.1: Tiempo de ejecución del algoritmo para imágenes VGA, comparando el rendimiento del software y el hardware diseñado.

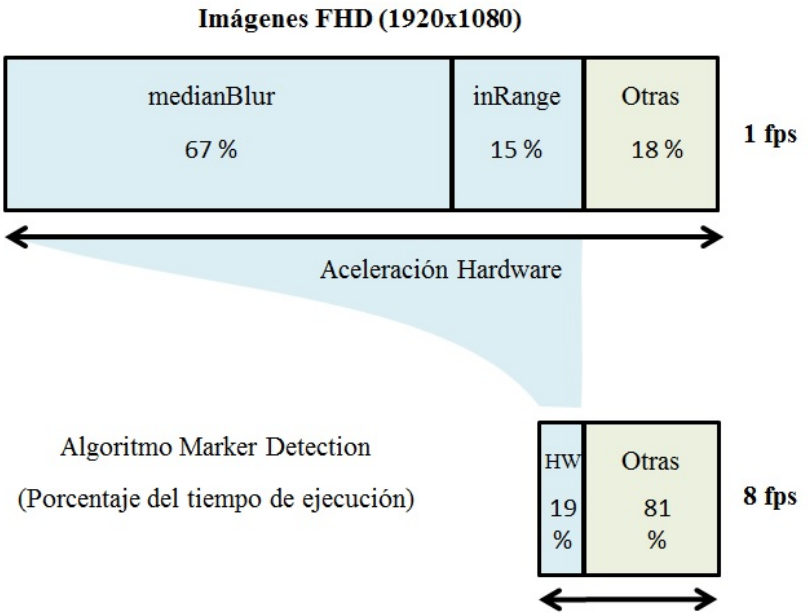

Figura 6.2: Tiempo de ejecución del algoritmo para imágenes FHD, comparando el rendimiento del software y el hardware diseñado.

Capítulo 7

Conclusiones

Este trabajo presenta el co-diseño hardware-software e implementación de un sistema de posicionamiento basado en procesado de vídeo sobre la plataforma Xilinx Zynq-7000, aprovechando sus capacidades hardware (FPGA) y software (ARM Cortex-A9). A partir de un análisis realizado a la versión preliminar del sistema, completamente software, fueron detectados algunos cuellos de botella. Las partes con mayor carga computacional fueron derivadas a hardware. La implementación final sobre este dispositivo es capaz de proporcionar buenas prestaciones en un sistema portátil de bajo consumo.

En relación a los objetivos planteados en la introducción, puede concluirse lo siguiente:

- El objetivo principal consistía en conseguir mediante aceleración hardware un incremento de las prestaciones del algoritmo de posicionamiento sobre la plataforma Zynq. Los resultados finales muestran una mejora significativa del throughput global del sistema, probando el éxito de la implementación realizada.
- La metodología seguida es una prueba del desarrollo de un sistema complejo en un tiempo de diseño relativamente rápido, utilizando herramientas cuyo grado de automatización va más allá del nivel RTL. El flujo de diseño incluye aspectos automáticos que permiten realizar aceleración HW desde C++ utilizando síntesis de alto nivel. No obstante, se ha visto que el conjunto de funciones OpenCV que pueden ser sintetizadas de forma automática en HLS es limitado y, por lo tanto, la síntesis directa de OpenCV a hardware no está completamente solucionada. Por ello, fue necesario llevar a cabo un estudio y diseño en alto nivel de las funciones requeridas.

- En cuanto a la generación de hardware utilizando síntesis de alto nivel, se ha comprobado que el tiempo de diseño es mucho más rápido que trabajando a nivel RTL y es posible evaluar diferentes arquitecturas rápidamente. Sin embargo, una síntesis de altas prestaciones desde C++ requiere un conocimiento profundo de la herramienta HLS. Esto significa que no conduce a los resultados esperados si el usuario no conoce muy bien el proceso de síntesis y la plataforma hardware. El código fue transformado para sacar partido de algunas directivas de síntesis y generar bloques IP de alto throughput.

- La integración de los bloques IP diseñados hizo necesario incluir hardware adicional y el conocimiento de las interfaces físicas disponibles en la arquitectura del Zynq-7000 con el objetivo de maximizar la tasa de transferencia de datos entre hardware y software. Estas comunicaciones fueron controladas desde la aplicación de usuario teniendo en cuenta el manejo que realiza el sistema operativo sobre la memoria y el hardware.

Líneas futuras de investigación

En cuanto a líneas futuras de investigación, puede ser interesante integrar un sistema que acelere la captura de imagen si la intención es realizar una implementación final en la ZedBoard. Esto se debe a que la entrada de cámara por USB 2.0 tiene un ancho de banda bastante limitado, por lo que se ha alcanzado una capacidad de procesamiento superior al *framerate* proporcionado por la cámara. Una posible solución sería implementar una entrada de cámara por hardware directamente a la FPGA.

Por otro lado, resultaría de interés analizar el coste o compromiso en términos de velocidad/área/potencia para cada una de las implementaciones, incluso considerando una versión puramente software sobre un microprocesador ARM de mayores prestaciones.

También se podría escalar el enfoque a vídeos con resolución 4K, cada vez más frecuentes en la electrónica de consumo. A medida que aumenta la resolución, la tendencia es un mayor aprovechamiento del hardware y un mayor cuello de botella en la CPU. Por ello, al mezclar funcionalidades hardware y software y subir en resolución de imagen, sería conveniente utilizar microprocesadores más potentes. En definitiva, el hardware escala bien (no solo en prestaciones sino también en cuanto a diseño, ya que al generarse desde alto nivel el cambio de resolución es directo) pero mantener el uso de microprocesadores poco potentes no resulta adecuado.

Publicaciones asociadas

El presente trabajo ha supuesto la realización de un artículo científico remitido a Design of Circuits and Integrated Systems Conference (DCIS) y aceptado a fecha de julio de 2016 para ser presentado en la conferencia [1].

Apéndice A

Código C++ de los IPs Hardware para HLS

A.1. IP *inRange*

```
//Angel Alvarez Ruiz
//University of Cantabria

//Synthesizable functions of HLS video library
#include <hls_video.h>

#define MAX_WIDTH  1920
#define MAX_HEIGHT 1080

typedef ap_axiu<8, 1, 1, 1> PIXEL_BW;
typedef hls::stream<PIXEL_BW> AXI_STREAM_BW;

typedef ap_axiu<24, 1, 1, 1> PIXEL_RGB;
typedef hls::stream<PIXEL_RGB> AXI_STREAM_RGB;

////////////////////////////////////////////////////////
//Checks if image pixel values lie within the specified
   limits - in this case 205 to 255
//Processes three (RGB) color channels and outputs black &
   white image
////////////////////////////////////////////////////////

void withinRange(AXI_STREAM_RGB& src_axi, AXI_STREAM_BW&
   dst_axi, int height, int width){

  //Create AXI streaming interfaces for the core
  #pragma HLS RESOURCE core=AXIS      variable=src_axi
     metadata="-bus_bundle INPUT_STREAM"
```

```
  #pragma HLS RESOURCE core=AXIS      variable=dst_axi
     metadata="-bus_bundle OUTPUT_STREAM"
  #pragma HLS RESOURCE core=AXI_SLAVE variable=height
     metadata="-bus_bundle CONTROL_BUS"
  #pragma HLS RESOURCE core=AXI_SLAVE variable=width
     metadata="-bus_bundle CONTROL_BUS"
  #pragma HLS RESOURCE core=AXI_SLAVE variable=return
     metadata="-bus_bundle CONTROL_BUS"

  short int r, c;
  PIXEL_RGB pixelIn;
  PIXEL_BW pixelOut;
  unsigned char dataRED, dataGREEN, dataBLUE;

  L1:for(r=0; r<height; r++){
    #pragma HLS LOOP_TRIPCOUNT min=600 max=1080 avg=720

    L2:for(c = 0; c < width; c++){
      #pragma HLS LOOP_TRIPCOUNT min=800 max=1920 avg=1280
      #pragma HLS PIPELINE II=1

      //Read incoming AXI data
      pixelIn = src_axi.read();

      //Get input pixel RGB values
      dataRED   = pixelIn.data.range(23,16);
      dataGREEN = pixelIn.data.range(15,8);
      dataBLUE  = pixelIn.data.range(7,0);

      if(dataRED>=205 && dataGREEN>=205 && dataBLUE>=205){
        //Set output pixel white
        pixelOut.data = 255;
      }else{
        //Set output pixel black
        pixelOut.data = 0;
      }

      //Set AXI protocol signals
      pixelOut.strb = 15;
      pixelOut.user = 1;
      pixelOut.dest = 1;
      if(c == width-1)  {pixelOut.last = 1;}
      else              {pixelOut.last = 0;}

      //Write output AXI data
      dst_axi.write(pixelOut);
    }
  }
}
```

A.2. IP *medianBlur*

Función principal

```
//Angel Alvarez Ruiz
//University of Cantabria

//Synthesizable functions of HLS video library
#include <hls_video.h>

#define SIZE 3
#define EDGE 1
#define MAX_WIDTH  1920
#define MAX_HEIGHT 1080

typedef ap_axiu<32, 1, 1, 1> PIXEL;
typedef hls::stream<PIXEL> AXI_STREAM;

unsigned char median(unsigned char window[SIZE*SIZE]);

//////////////////////////////////////////////////////////
//2D Median Filter for images
//Parameter 'size' must be an odd value
//Processes one color channel (Black & White images)
//Uses 2 buffers for high speed video processing: 3-line
   buffer & 9-pixel window
//////////////////////////////////////////////////////////

void medianFilter(AXI_STREAM& src_axi, AXI_STREAM& dst_axi,
   int height, int width){

  //Create AXI streaming interfaces for the core
  #pragma HLS RESOURCE core=AXIS      variable=src_axi
     metadata="-bus_bundle INPUT_STREAM"
  #pragma HLS RESOURCE core=AXIS      variable=dst_axi
     metadata="-bus_bundle OUTPUT_STREAM"
  #pragma HLS RESOURCE core=AXI_SLAVE variable=height
     metadata="-bus_bundle CONTROL_BUS"
  #pragma HLS RESOURCE core=AXI_SLAVE variable=width
     metadata="-bus_bundle CONTROL_BUS"
  #pragma HLS RESOURCE core=AXI_SLAVE variable=return
     metadata="-bus_bundle CONTROL_BUS"

  short int r, c;
  unsigned char pixel, med, window[SIZE*SIZE];
  static unsigned char line_buffer[SIZE][MAX_WIDTH] = {0};
  PIXEL pixelIn, pixelOut;
```

```
#pragma HLS ARRAY_PARTITION variable=line_buffer complete
   dim=1

rows:for(r = 0; r < height+1; r++){
  #pragma HLS LOOP_TRIPCOUNT min=601 max=1081 avg=721

  cols:for(c = 0; c < width+1; c++){
    #pragma HLS LOOP_TRIPCOUNT min=801 max=1921 avg=1281
    #pragma HLS PIPELINE II=1

    //------------------------------------------
    //-------Memory structures (buffers) management
    //------------------------------------------
    if(c<width && r<height){
      //Slide line buffer down (one pixel each time)
      for(int i = 0; i < SIZE-1; i++){
        line_buffer[i][c] = line_buffer[i+1][c];
      }

      //Read incoming AXI data and save it into line buffer
      pixelIn = src_axi.read();
      line_buffer[SIZE-1][c] = pixelIn.data.range(7,0);

      //Slide window
      for(int i = 0; i < SIZE; i++){
        for(int j = 0; j < SIZE-1; j++){
          window[i*SIZE+j] = window[i*SIZE+j+1];
        }
      }
      for(int i = 0; i < SIZE; i++){
        window[i*SIZE+SIZE-1] = line_buffer[i][c];
      }
    }
    //------------------------------------------

    //Median Filter
    med = median(window);

    //Pixel is the median (and black if it's on the edge of
        image)
    if ( (r>=SIZE-1)&&(r<height)&&(c>=SIZE-1)&&(c<=width) )
       {
      pixel = med;
    }else{
      pixel = 0;
    }

    //Write output pixel (when required first pixels are
       available)
```

```
      if(r>0 && c>0){
        pixelOut.data.range(7,0)  = pixel;
        pixelOut.data.range(31,8) = 0;
        //Set AXI protocol signals
        pixelOut.strb = 15;
        pixelOut.user = 1;
        pixelOut.dest = 1;
        if(c == width)  {pixelOut.last = 1;}
        else      {pixelOut.last = 0;}
        //Write output AXI data
        dst_axi.write(pixelOut);
      }
    }
  }
}
```

Función para el cálculo de la mediana

```
//Angel Alvarez Ruiz
//University of Cantabria

#define SIZE 3
#define N SIZE*SIZE

unsigned char median(unsigned char window[SIZE*SIZE])
{
  #pragma HLS ARRAY_PARTITION variable=window complete dim=1
  #pragma HLS PIPELINE II=1

  unsigned char i, stage, temp;
  unsigned char local[N];

  //Save input data into local array
  for (i=0; i<SIZE*SIZE; i++){
    local[i] = window[i];
  }

  //---------------------------------------------
  //Odd-Even Sorting Network

    for (int stage = 0 ; stage < N ; stage++){
      //stage is odd
        if (stage & 1){
            for(i=2; i<N; i+=2){
                if (local[i] < local[i-1]){

                  //Swap elements
                  temp = local[i];
                  local[i] = local[i-1];
```

```
                    local[i-1] = temp;
                }
            }
        }
        //stage is even
        else{
            for(i=1; i<N; i+=2){
              if (local[i] < local[i-1]){

                //Swap elements
                temp = local[i];
                local[i] = local[i-1];
                local[i-1] = temp;
                }
            }
        }
    }
    //-----------------------------------------------

  //The median is in the middle location of the sorted array
  return local[N/2];
}
```

Referencias

[1] A. Álvarez, I. Ugarte, P. Martínez and V. Fernández, *HW-SW Codesign of a Positioning System. UML to Implementation Case Study*, DCIS 2016 (aceptada)

[2] E. Villar, P. Martínez, F. Alcalá, P. Sánchez and V. Fernández, *Método y sistema de localización espacial mediante marcadores luminosos para cualquier ambiente*, P.N.ES-2543038-B2, 2014

[3] L. H. Crockett, R. A. Elliot, M. A. Enderwitz and R. W. Stewart, *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*, First Edition, Strathclyde Academic Media, 2014

[4] Xilinx, *Zynq-7000 All Programmable SoC Technical Reference Manual*, UG585 (v1.10), February 2015

[5] ZedBoard.org Documentation Website, http://zedboard.org/support/documentation/1521

[6] Ze-Nian Li and Mark S. Drew, *Fundamentals of Multimedia*, Pearson Education International, 2004

[7] Adrian Kaehler and Gary Bradski, *Learning OpenCV: Computer Vision in C++ with the OpenCV Library*, 2nd Edition, O'Reilly Media, 2015

[8] OpenCV Documentation, http://docs.opencv.org/

[9] OpenCV Tutorials, http://docs.opencv.org/2.4/doc/tutorials/introduction/table_of_content_introduction/table_of_content_introduction.html

[10] Xilinx, *Vivado Design Suite User Guide: High-Level Synthesis*, UG902 (v2015.3), September 2015

[11] Avnet Reference Materials, *Ubuntu on Zynq-7000 All Programmable SoC Tutorial For ZedBoard and Zynq Mini-ITX Development Kits*, Version 3, July 2014

[12] Repositorio Git de Xilinx,
https://github.com/xilinx

[13] Repositorio Git de Analog Devices Inc (ADI),
https://github.com/analogdevicesinc/linux

[14] Web oficial de Linaro,
http://www.linaro.org/

[15] Avnet Reference Materials, *Bare Metal HDMI for ZedBoard with ADI IP and ADV7511*, Version 1.0, June 2014

[16] HLS Video Library,
http://www.wiki.xilinx.com/HLS+Video+Library

[17] OpenCV, Operations on Arrays, inRange,
http://docs.opencv.org/2.4/modules/core/doc/
operations_on_arrays.html

[18] OpenCV, Image Filtering, medianBlur,
http://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html

[19] F. Martínez Vallina, *Implementing Memory Structures for Video Processing in the Vivado HLS Tool*, XAPP793 (v1.0), September 20 2012

[20] Wikipedia, Bubble sort
https://en.wikipedia.org/wiki/Bubble_sort

[21] Wikipedia, Sorting network
https://en.wikipedia.org/wiki/Sorting_network

[22] OpenCV, How to scan images, lookup tables and time measurement with OpenCV
http://docs.opencv.org/2.4/doc/tutorials/core/how_to_scan_images/
how_to_scan_images.html

[23] Xilinx Backgrounder, *Vivado IP Integrator: Accelerated Time to IP Creation and Integration*, 2013

[24] F. Martínez Vallina, Christian Kohn and Pallav Joshi, *Zynq All Programmable SoC Sobel Filter Implementation Using the Vivado HLS Tool*, XAPP890 (v1.0), September 25 2012

[25] Wikipedia, Direct memory access
https://en.wikipedia.org/wiki/Direct_memory_access

[26] Xilinx, *AXI Video Direct Memory Access v6.2 LogiCORE IP Product Guide*, November 2015

[27] Linux Programmer's Manual, mmap(2)
http://man7.org/linux/man-pages/man2/mmap.2.html