Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación
Intensificación en Sistemas Electrónicos

Co-diseño de un sistema de detección de señales de tráfico con SDSoC

Autor: Álvaro Arjona Gamito
Tutor: Hipólito Guzmán Miranda

Departamento de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Co-diseño de un sistema de detección de señales de tráfico con SDSoC

Autor:
Álvaro Arjona Gamito

Tutor:
Hipólito Guzmán Miranda
Profesor contratado doctor

Dep. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Trabajo Fin de Grado: Co-diseño de un sistema de detección de señales de tráfico con SDSoC

Autor: Álvaro Arjona Gamito

Tutor: Hipólito Guzmán Miranda

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A mi familia

A mis amigos

A mis maestros

Agradecimientos

Quería agradecer a mi familia por todo el apoyo que me ha dado siempre. A mis padres, por todo su esfuerzo y hacer posible que haya llegado hasta aquí. También quería agradecer a Natividad, por hacerme mejor persona cada día.

Por supuesto, también doy las gracias a todos mis compañeros y amigos que han hecho que mi etapa universitaria sea inolvidable.

Álvaro Arjona Gamito

Sevilla, 2017

Resumen

La visión artificial es una disciplina científica muy estudiada hoy en día y uno de los campos más prometedores para el futuro. Una de las áreas donde cada vez está más introducida es en el ámbito automovilístico, siendo uno de los pilares fundamentales del coche autónomo.

En este trabajo se pretende aplicar la visión artificial para crear un detector de señales de tráfico. Se hará un co-diseño HW/SW con la idea de combinar todos los beneficios de un SO que corre en un microprocesador con los de la capacidad de procesamiento en paralelo de una FPGA. Esto se desarrollará con la herramienta SDSoC de Xilinx y se implementará sobre una plataforma Zedboard.

Se analizará también el rendimiento obtenido en velocidad al emplear sistemas heterogéneos en este tipo de aplicaciones y la eficiencia de las técnicas de visión artificial implementadas.

Abstract

Computer vision is a scientific discipline that is very studied nowadays and one of the most promising sectors for the future. One of the areas where it is increasingly introduced is in the automobile field, being one of the fundamental parts of the autonomous car.

In this dissertation, the intention is to apply computer visión to the creation of a road signs detector. We will do a HW/SW co-design with the idea of combining all the benefits of an OS running on a microprocessor with those of the parallel processing capacity of an FPGA. This will be developed with the Xilinx SDSoC tool and it will be implemented on a Zedboard platform.

It will also analyze the performance obtained in speed using heterogeneous systems in this type of applications and the efficiency of the artificial vision techniques implemented.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

AMS	**A**gile **M**ixed **S**ignal
APU	**A**pplication **P**rocessor **U**nit
ASIC	**A**pplication-**S**pecific **I**ntegrated **C**ircuit
ASSP	**A**pplication **S**pecific **S**tandard **P**roduct
AXI	**A**dvanced e**X**tensible **I**nterface
BSD	**B**erkeley **S**oftware **D**istribution
BSP	**B**oard **S**upport **P**ackages
CLB	**C**onfigurable **L**ogic **B**locks
CPU	**C**entral **P**rocessing **U**nit
DDR	**D**ouble **D**ata **R**ate
DMA	**D**irect **M**emory **A**ccess
DoG	**D**ifference **o**f **G**aussians
EPP	**E**mbedded **P**rocessing **P**latform
EMIO	**E**xtended **M**ultiplexed **I**/**O**s
FF	**F**lips **F**lops
FIFO	**F**irst **I**n **F**irst **O**ut
HDL	**H**ardware **D**escription **L**anguage
HDMI	**H**igh **D**efinition **M**ultimedia **I**nterface
HLS	**H**igh **L**evel **S**ynthesis
HOG	**H**istogram **o**f **G**radient
HW	**H**ardware
IP core	**I**ntellectual **P**roperty core
JTAG	**J**oint **T**est **A**ction **G**roup
LPC	**L**ow **P**in **C**ount
LUT	**L**ook-**U**p **T**ables
PL	**P**rogrammable **L**ogic
PS	**P**rocessing **S**ystem
QSPI	**Q**ueued **S**erial **P**eripheral **I**nterface
RoI	**R**egion **o**f **I**nterest
RoS	**R**egion **o**f **S**earch
RTL	**R**egistrer **T**ransfer **L**anguaje
SDSoC	**S**oftware-**D**efined **S**ystem **O**n **C**hip

SoC	**S**ystem **o**n **C**hip
Soft IP	**Sotf I**ntellectual **P**roperty
SVM	**S**upport **V**ector **M**achine
SW	**S**of**w**are
UART	**U**niversal **A**synchronous **R**eceiver-**T**ransmitter
USB	**U**niversal **S**erial **B**us
VGA	**V**ideo **G**raphics **A**rray
VHDL	**V**HSIC **H**ardware **D**escription **L**anguage

1 INTRODUCCIÓN

1.1 Motivación

El desarrollo del vehículo autónomo es uno de los campos de investigación con más actividad hoy en día. Desde hace varios años, multitud de grupos de investigación y de empresas, principalmente del automóvil, trabajan en ello para que en el futuro puedan tener un lugar privilegiado en el sector de la automoción.

Según varios expertos, se espera que el coche autónomo se pueda comprar libremente a partir del año 2025. Los vehículos autónomos serán vehículos conectados al medio y compartirán grandes cantidades de información entre sí y con el resto de la infraestructura. Es por esto, que el vehículo autónomo forma una importante parte del internet de las cosas (IoT). Varios son los factores que hacen que se alargue la fecha de lanzamiento al mercado y que van más allá del estado del arte de la tecnología: El marco legal, tener que construir costosas infraestructuras que son necesarias, las aseguradoras o la seguridad de estos nuevos sistemas. Los coches autónomos serán también coches conectados, por lo que estarían expuestos a ciberataques.

Los nuevos coches autónomos reducirían el consumo, disminuirían o harían desaparecer los atascos, mejorarían el medio ambiente. Pero uno de los objetivos más importantes serían sin duda la reducción de la mortalidad en las carreteras. Según la comisión europea [1], más de 40 mil personas mueren al año en las carreteras y más de 1 millón y medio resultan heridas.

Antes de llegar al vehículo completamente autónomo, existen híbridos entre coche autónomo y coche completamente manual. Muchos de los coches actuales disponen de asistentes al conductor, como frenado automático en caso de detección de colisión o atropello, aparcamiento automático, detección de obstáculos en la carretera, de peatones o de señales, y otros sistemas.

El campo de la visión artificial está muy presente en la mayoría de estos sistemas que complementan al coche de hoy en día y que esperan mejorarlo hasta llegar al coche autónomo. Sensores de imagen, algoritmos de procesado de la imagen digital y algoritmos de aprendizaje son la parte fundamental de los sistemas de detección de objetos y conducción automática. La mayoría de estas aplicaciones tienen una característica en común: son aplicaciones de tiempo real. Es por esto, que la latencia de estas aplicaciones se convierte en algo crítico. Existen dos formas de reducir el tiempo de ejecución de estas aplicaciones: haciendo un algoritmo más eficiente o mejorando el hardware que lo procesa. De aquí surge el empleo de sistemas hardware heterogéneos capaces de procesar datos a gran velocidad gracias al paralelismo de sus circuitos.

1.2 Objetivos

El objetivo de este trabajo es el del diseño de un sistema de detección de señales de tráfico. Este diseño será en realidad un co-diseño HW/SW que correrá sobre un dispositivo Zynq-7000 AP SoC de Xilinx. Para el desarrollo, se estudiarán distintos métodos de segmentación de la imagen y de reconocimiento de objetos.

Parte del algoritmo del sistema irá implementado en hardware con el objetivo de mejorar las prestaciones del sistema en cuanto a velocidad. El tiempo de ejecución de la parte acelerada será comparada con la misma implementada sobre el microprocesador.

Como resultado, el sistema será capaz de cargar desde una tarjeta de memoria fotografías reales de carreteras y de ciudad que contienen señales de tráfico, detectar las señales y reconocerlas.

1.3 Organización del documento

A lo largo de esta memoria, se hará una introducción teórica sobre las herramientas y tecnologías utilizadas para la realización de este trabajo, se mostrarán los detalles del sistema implementado y de los resultados finalmente obtenidos.

La memoria está compuesta de un total de 7 capítulos. Tras esta introducción, en el segundo capítulo se hablará sobre los dispositivos de la familia Zynq-7000 y la Zedboard, que es la plataforma donde será probado el detector desarrollado. En el tercer capítulo, se hablará sobre las herramientas necesarias para desarrollar el sistema, tanto del entorno de desarrollo utilizado, como de las librerías y tecnologías necesarias. En el cuarto capítulo, se presentarán algunas técnicas de desarrollo hardware cuando se utiliza la síntesis de alto nivel y se hablará de buenas prácticas para mejorar el rendimiento del sistema. Después, en el quinto capítulo, se hablará de la estructura general de un detector de señales de tráfico. En el capítulo seis, se hablará del sistema finalmente implementado, hablando de sus detalles y de los resultados de eficiencia y latencia obtenidos tras su implementación sobre una placa Zedboard. La memoria termina con un capítulo de conclusiones y trabajos futuros.

2 DISPOSITIVOS DE LA FAMILIA ZYNQ – 7000 Y LA ZEDBOARD

En este capítulo se hará una presentación sobre los dispositivos *System-on-chip* (SoC) o sistema en chip. Explicaremos qué son y en qué consisten. Así, presentaremos la familia de dispositivos Zynq®-7000 de la compañía Xilinx®. Por último, presentaremos la placa de desarrollo Zedboard (creada en conjunto por Xilinx, Digilent® y Avnet®) que contiene uno de los dispositivos chip de la familia Zynq-7000 y es sobre la cual está desarrollado este trabajo.

2.1 Dispositivos System-on-chip (SoC)

Los dispositivos SoC son unos circuitos integrados cada vez más utilizados hoy en día. Estos circuitos incluyen, dentro del mismo silicio, distintos módulos o tecnologías con el fin de poder utilizarlos de manera conjunta y aprovechar lo mejor de cada uno en un mismo proyecto. Los SoC permiten reducir el consumo, mejorar el rendimiento y la escalabilidad de los sistemas, y están presentes en multitud de aparatos de uso cotidiano como portátiles, móviles, tablets y otros aparatos.

Los SoC suelen incluir en su interior: CPUs, GPUs, memorias, controladores (de sistema, de memoria, de datos, de interfaces), osciladores, dispositivos de lógica programable, conectividad y otros circuitos.

2.2 La familia Zynq-7000

Los dispositivos de la familia Zynq-7000 se basan en la arquitectura *All Programmable SoC* creada por Xilinx [2]. Estos dispositivos combinan un sistema de procesamiento (PS) dual-core ARM® Cortex™-A9 MPCore™ y una lógica programable (PL) de Xilinx fabricada con tecnología de 28nm dentro del mismo dispositivo. El sistema PS incluye, entre otras cosas, memoria *on-chip*, interfaces de memoria externa y un elevado conjunto de periféricos de entrada/salida.

La familia Zynq-7000 ofrece todas las ventajas de flexibilidad y escalabilidad de una FPGA, y el rendimiento, potencia y la facilidad de uso típicamente asociado a los ASIC y ASSPs. La familia Zynq-7000 de Xilinx incluye un amplio rango de dispositivos pensados para abarcar un amplio rango de aplicaciones. Todos los dispositivos de la familia Zynq-7000 comparten la misma PS, variando entre ellos la parte de PL y los periféricos.

Según la compañía Xilinx, entre el elevado rango de aplicaciones que se pueden desarrollar con la familia Zynq-7000 se incluyen las siguientes [2]:

- Asistencia a la conducción de automóviles, información al conductor y sistema de infotainment.
- Visión artificial, redes industriales y control de motores industriales.
- Radio LTE.
- Diagnóstico e imagen médica.
- Cámara inteligente.

La principal ventaja de los SoC, como los de Zynq-7000, es que la intercomunicación entre sus componentes es mucho más rápida que cuando estos están en chips diferentes. La integración, en este caso de la PS junto con la PL, proporciona altos niveles de rendimientos no alcanzables en soluciones con dos chips debido a la velocidad limitada y a la latencia de interconexión entre chips separados. Esto es crítico, sobre todo, cuando se tienen que pasar grandes cantidades de datos.

Xilinx y otros asociados proporcionan un gran número de módulos soft IP para la familia Zynq-7000. También se proporcionan drivers para los periféricos de PS y PL, además de paquetes de soporte para las tarjetas BSPs. Ya que el procesador que se incluye es de ARM, también están disponibles muchas aplicaciones y herramientas de desarrollo de terceros.

Los procesadores del PS siempre inician primero, permitiendo iniciar la PL desde la PS en cualquier momento y permitiendo la configuración total o parcial de la PL.

2.2.2 Arquitectura del Zynq-7000

La figura 2-1 muestra los bloques funcionales de la arquitectura Zynq-7000 AP SoC. La PS y la PL tienen una alimentación separada, lo que permite apagar la parte de PL si no resulta necesario.

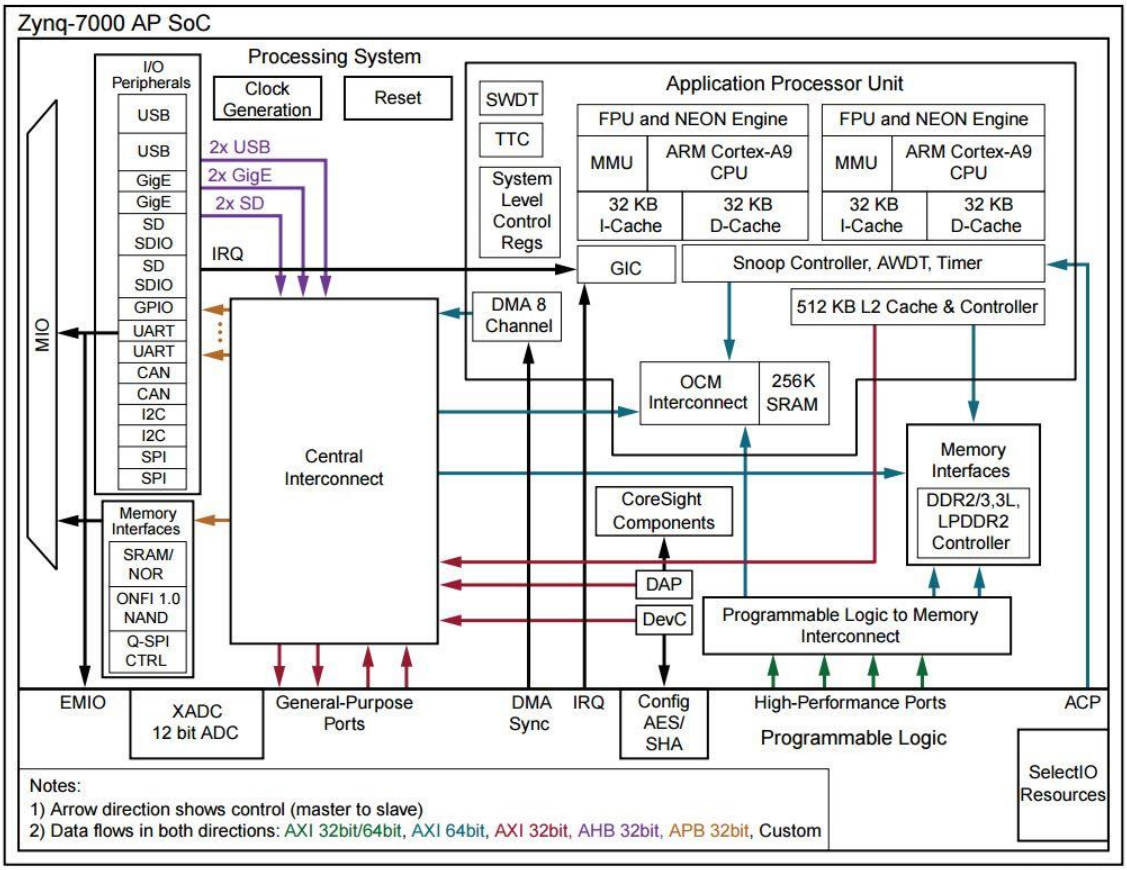


Figura 2-1: Esquema interno de un dispositivo Zynq-7000 (Fuente: Xilinx)

Como ya hemos dicho, los dispositivos Zynq-7000 se dividen en dos partes fundamentales: un sistema de procesamiento y un dispositivo de lógica programable. A continuación, se indica qué contiene fundamentalmente cada una de estas dos partes

- Sistema de procesamiento (PS)
  - Application Processing Unit (APU) – Contiene al microprocesador.
  - Interfaces de memoria – Interfaces a memorias externas.
  - Periféricos de entrada/salida (IOP) – Conexiones a periféricos externos.
  - Interconexiones – Conexiones internas.

- Lógica programable (PL)
  - Look-up table (LUT) – Elemento que realiza operaciones lógicas.
  - Flip-Flop (FF) – Registro que almacena el resultado de la LUT.
  - Pistas – Lo que conecta unos elementos con otros.
  - Input/Output (I/O) pads – Los puertos físicos para los datos de entrada y salida.

Las combinaciones de las LUT con los FF forman lo que se llaman celdas lógicas, que a su vez conforman los bloques lógicos configurables (CLB). Estos bloques lógicos son los elementos básicos de una FPGA. En la figura 2-2 podemos ver cómo es la arquitectura básica de una FPGA.

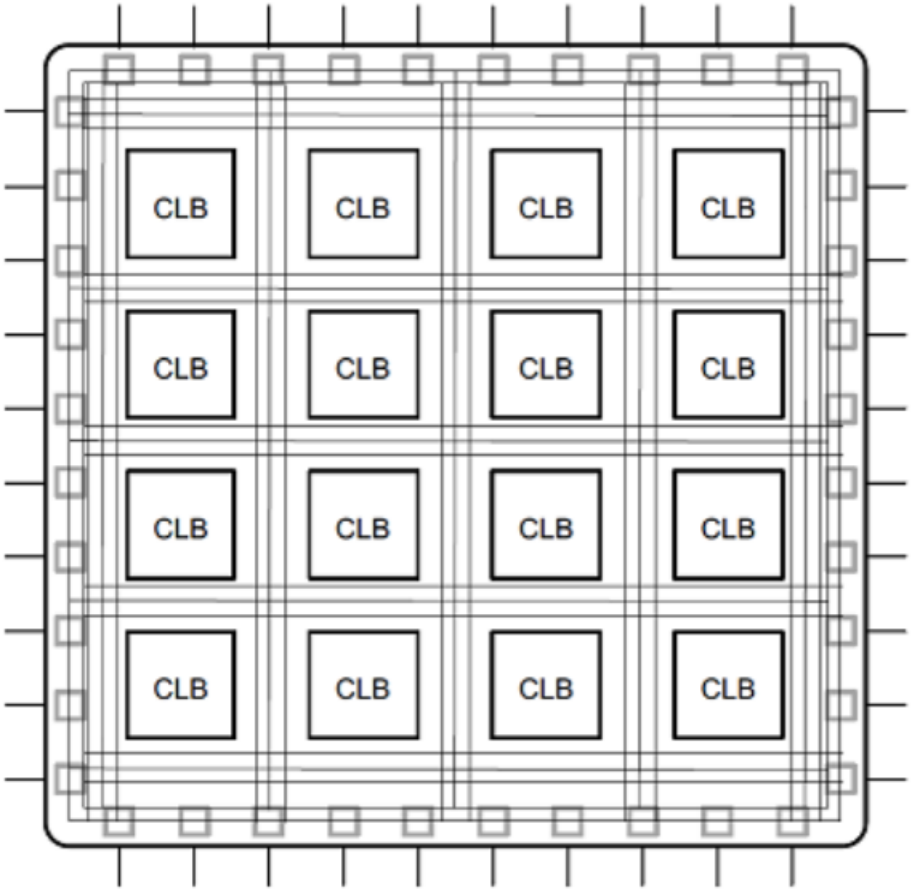


Figura 2-2: Esquema simple de una FPGA (Fuente:Xilinx)

La familia Zynq-7000 está compuesta por distintos modelos que, como ya se ha dicho, comparten el mismo sistema de procesamiento con el mismo procesador ARM de doble núcleo. La diferencia entre ellos está en la lógica programable. En la siguiente tabla se resume las características de cada modelo:

	Z-7010	Z-7015	Z-7020	Z-7030	Z-7045	Z-7100
Procesador	Dual core ARM Cortex-A9 con extensiones NEON y FPU					
Frecuencia máxima del procesador	866MHz			1GHz		
Lógica Programable	Artix-7			Kintex-7		
Biestables	35200	96400	106400	157200	437200	554800
LUTs	17600	46200	53200	78600	218600	277400
Bloques RAMs de 36Kb	60	95	140	265	545	755
DSP slices	80	160	220	400	900	2020

Tabla 1: Modelos de la familia Zynq-7000 (Fuente: Xilinx)

2.2.3 Memorias

Para el desarrollo de aplicaciones basados en dispositivos Zynq-7000 resulta de interés conocer bien qué tipo de memorias están contenidas en el chip y qué tipo de memorias externas son accesibles. Tanto la forma en la que utilicemos estas memorias y la forma en la que transferimos los datos entre la PL y la PS van a ser cruciales en el rendimiento final de nuestro diseño. El acceso a memorias externas es el que tiene más latencia y, por lo tanto, el que más reduce el rendimiento. Por esto, hay que maximizar el uso de las memorias internas. Para ello, existen ciertas técnicas de utilización de las memorias *on-chip* y técnicas de patrones de acceso a datos que serán estudiados en profundidad en capítulos posteriores. Vemos a continuación cuáles son las memorias disponibles [3]:

Memorias contenidas en la APU:

- OCM: Memoria *on-chip* SRAM de doble puerto de 256KB. Acceso rápido para el microprocesador, accesible desde la lógica programable y DMA.
- Memoria cache de dos niveles L1 y L2.

Memorias en la lógica programable:

- Block RAMs
- Registros de desplazamiento
- LUT (memoria ROM)

Memorias externas al chip:

- DDR
- Memoria Flash SD

2.2.4 Interfaces de comunicación entre PS-PL

Como iremos viendo a lo largo del trabajo, nuestro co-diseño contendrá partes del sistema implementado sobre la PS y parte que irá implementado sobre la PL. Para que esto sea posible, la comunicación PS-PL

(tanto para pasar órdenes como para pasar datos a procesar) toma un papel considerable. A continuación, se presentan las interconexiones principales PS-PL, sus protocolos y sus buses. Aunque en los dispositivos Zynq-7000 existen una gran variedad de tipos de conexiones internas [3], aquí nos centraremos solo en las conexiones específicas entre PS-PL. Existen también conexiones internas a cada uno de ellos y conexiones a dispositivos externos al chip. Se puede obtener mucha más información sobre las conexiones PS-PL en [3].

2.2.4.1 Interconexiones AXI4

AXI (Advanced eXtensible Interface) es un protocolo de interconexión basado en la especificación AMBA3.0 que fue creado por ARM en 2003. Se trata de un protocolo estandarizado, lo que beneficia a Xilinx en productividad, flexibilidad y disponibilidad debido a que los desarrolladores no tienen que aprender varios protocolos de comunicación y a que los desarrolladores se pueden beneficiar del uso de una gran cantidad de IP cores disponibles para esta conectividad.

En 2010 se lanzó la especificación AMBA4.0 la cual incluye la segunda versión de AXI, AXI4. Las AXI4 son las principales interfaces de comunicación entre PS y PL de los dispositivos Zynq-7000. Además, las conexiones internas en PS y PL también son del tipo AXI4.

Los protocolos de interfaz AXI4 se dividen en 3 tipos [3]:

- AXI4 estándar: Para transferencias mapeadas en memoria con un alto rendimiento.
- AXI4-lite: Para transferencias mapeadas en memoria pero que requieren menos rendimiento.
- AXI4-stream: Para transferencias de tipo streaming unidireccionales de maestro a esclavo de alta velocidad y sin mapeo en memoria. Sirven para transferir grandes cantidades de datos.

En los dispositivos de la familia Zynq-7000 existen un total de 9 interfaces AXI que comunican PS con PL. Se dividen en tres tipos [3]:

- Interfaces AXI de propósito general: Tienen un bus de datos de 32 bit, permite comunicaciones de pequeña y media tasa de transferencia entre PS-PL. La interfaz es directa y no incluye buffering. En total existen 4 interfaces de propósito general: 2 en la que el master es PS y 2 en el que el master es PL.
- Puertos de alto rendimiento: Se tienen 4 puertos de alto rendimiento que incluyen buffers FIFO de lectura y escritura para traspaso de grandes cantidades de datos. La conexión es directa entre PL y los puertos de memoria DDR o la memoria OCM incluidos en PS. El ancho de los datos puede ser 32 o 64 bits y PL es el master de las 4 interfaces.
- Puerto de coherencia del acelerador: Es una conexión asíncrona entre PL y la APU de PS con un bus de datos de 64 bits. El objetivo es conseguir coherencia entre las cachés L1 y L2 de la APU y los elementos en la PL, siendo PL el master.

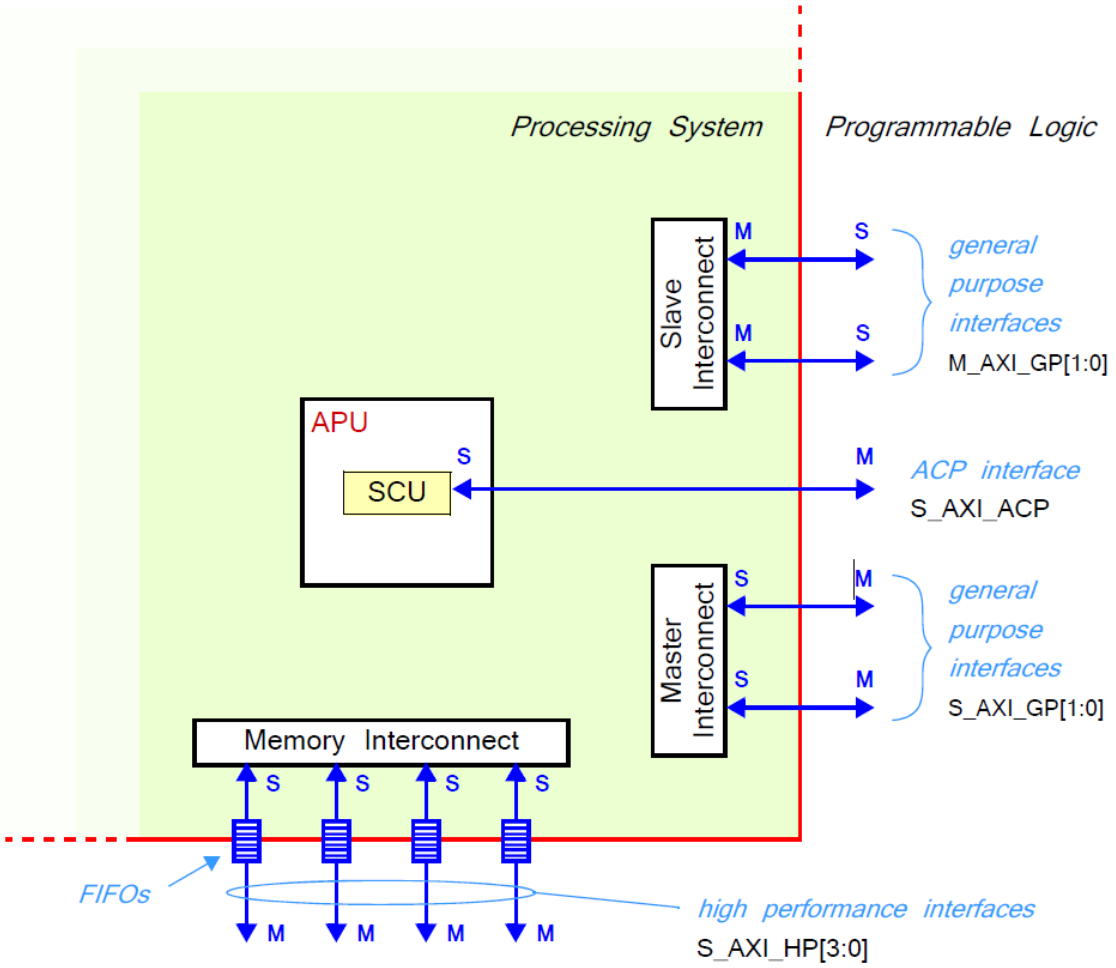


Figura 2-3: Esquema de las interfaces AXI de los dispositivos Zynq-7000 [3]

Todas estas interfaces AXI están compuestas de las señales necesarias para cumplir con los protocolos AXI4.

2.2.4.2 EMIO

Existen pines de conexión entre PS y las interfaces externas como pueden las memorias externas, los puertos usb, puerto I2C, etc. Se trata de pines multiplexados de entrada y salida llamados MIO. Cuando estas conexiones no son directas entre PS y las interfaces externas, sino que se hacen mediante PL. Estas conexiones se hacen a través de interfaces extended MIO, EMIO.

2.2.4.3 Otras señales e interfaces

Otras señales de interconexión entre PS-PL son: los watchdog timers, las señales de reset, las señales de interrupción, así como las señales de interfaz DMA (Direct Memory Access).

2.3 Zedboard

La placa que vamos a utilizar para desarrollar nuestro trabajo será la Zedboard. La Zedboad es una placa de evaluación y desarrollo basada en la plataforma de procesamiento embebido (EPP) Zynq-7000 de Xilinx, en concreto el modelo Z-7020 [4]. Este modelo combina un dual-core ARM Cortex-A9 MPCore (PS) y una

lógica programable Artix-7 FPGA de Xilinx de 85.000 celdas lógicas, 220 slices dsp y 140 BlockRAMs. La Zedboard fue creada en conjunto por Xilinx, Digilent y Avnet. Equipa un importante número de periféricos que, añadido a su capacidad de expansión, la convierte en una placa ideal en un gran número de aplicaciones tanto para desarrolladores novatos como experimentados. A continuación, se resume las características de la Zedboard:

- Xilinx XC7Z020-1CSG484CES EPP
  - Configuración principal: Flash QSPI.
  - Configuraciones auxiliares opcionales:
    - JTAG
    - Tarjeta SD
- Memoria
  - 512 MB DDR3 (128Mx32)
  - 256 Mb QSPI Flash
- Interfaces
  - USB-JTAG
    - Acceso a PL
    - Acceso a PS a través de PS Pmod
  - 10/100/1G Ethernet
  - USB OTG 2.0
  - Tarjeta SD
  - USB 2.0 FS USB-UART bridge
  - 5 conectores compatibles con Digilent Pmod™ (2x6) (1 PS, 4 PL)
  - 1 LPC FMC
  - 1 conector AMS
  - 2 botones de reset (1PS, 1PL)
  - 7 botones pulsadores (2 PS, 5 PL)
  - 8 puentes dip/slide (PL)
  - 9 LEDs de usuario (1 PS, 8 PL)
  - LED de “terminado” (PL)
- Osciladores on-board
  - 33.33 MHz (PS)
  - 100 MHz (PL)
- Imagen/audio
  - Salida HDMI
  - VGA (Color 12-bit)
  - Pantalla OLED 128x32
  - Audio Line-in, Line-out, auriculares y micrófono
- Alimentación
  - Switch On/Off
  - Regulador AC/DC 12 V @5A
- Software
  - ISE® WebPACK Design software
  - Licensia ChipScope™ para XC7Z020

En la figura 2.4 podemos ver una perspectiva desde arriba de la placa de desarrollo Zedboard, con las partes más importantes remarcadas.

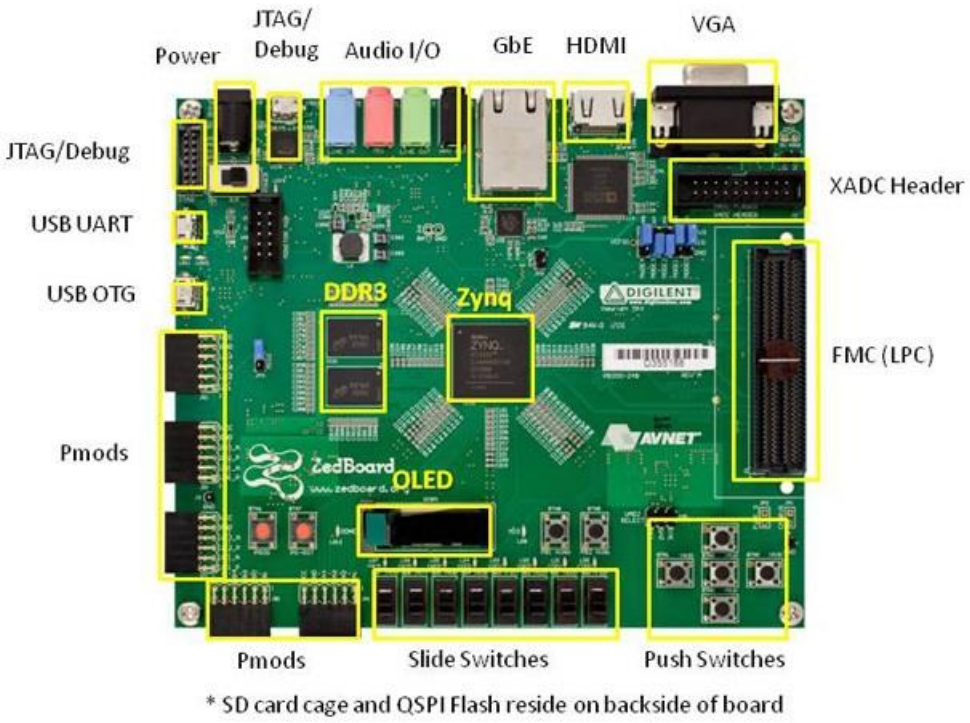


Figura 2-4: Placa Zedboard [4]

3 HERRAMIENTAS PARA EL DESARROLLO

En este capítulo se hablará de aquellas herramientas y entornos de desarrollo que empleamos para el diseño de este trabajo. Se hablará también de la síntesis de alto nivel para el diseño en lógica programable y se introducirán las librerías que se emplean en este trabajo.

3.1 Vivado HLS y la síntesis de alto nivel

La síntesis de alto nivel (HLS) es una técnica que permite desarrollar hardware a partir de código software, tal como C o C++ [5]. HLS aumenta la abstracción del diseño hardware y proporciona un diseño optimizado con las características de rendimiento, área y consumo requeridos, sin la necesidad de utilizar lenguajes de descripción de hardware (HDL). El código fuente de un proyecto realizado con lenguaje de alto nivel debe estar compuesto por un código que describa la funcionalidad del sistema, y un conjunto de directivas y restricciones que ayudarán al diseñador a indicarle al compilador como quiere que sea implementado el sistema (temas de arquitectura del sistema final).

Los requisitos a cumplir de nuestro diseño pueden ser restrictivos en velocidad o en área hardware utilizada. Las directivas que empleemos en nuestro código dependerán directamente de qué índole son estas restricciones. Si queremos reducir la latencia de la aplicación, emplearemos directivas que aumenten el paralelismo aumentando los recursos necesarios. Por otro lado, si los requisitos a cumplir son de área máxima empleada (como podría ser, por ejemplo, para aplicaciones espaciales donde el tamaño y el peso de los dispositivos está limitado) las directivas indicarán que se debe reducir el paralelismo. De esta manera se reducen los recursos empleados a cambio de aumentar la latencia.

Existen dos ideas fundamentales en las herramientas de síntesis de alto nivel: el scheduling y el binding.

- El **scheduling** es el proceso en el cual se decide en qué momento se ejecuta cada parte de un algoritmo. Esto es, cada tarea del algoritmo se divide en partes y se decide en qué ciclos de reloj se realiza cada una.
- El **binding** consiste en decidir cuáles son los componentes de bajo nivel que se van a encargar de ejecutar cada parte del algoritmo.

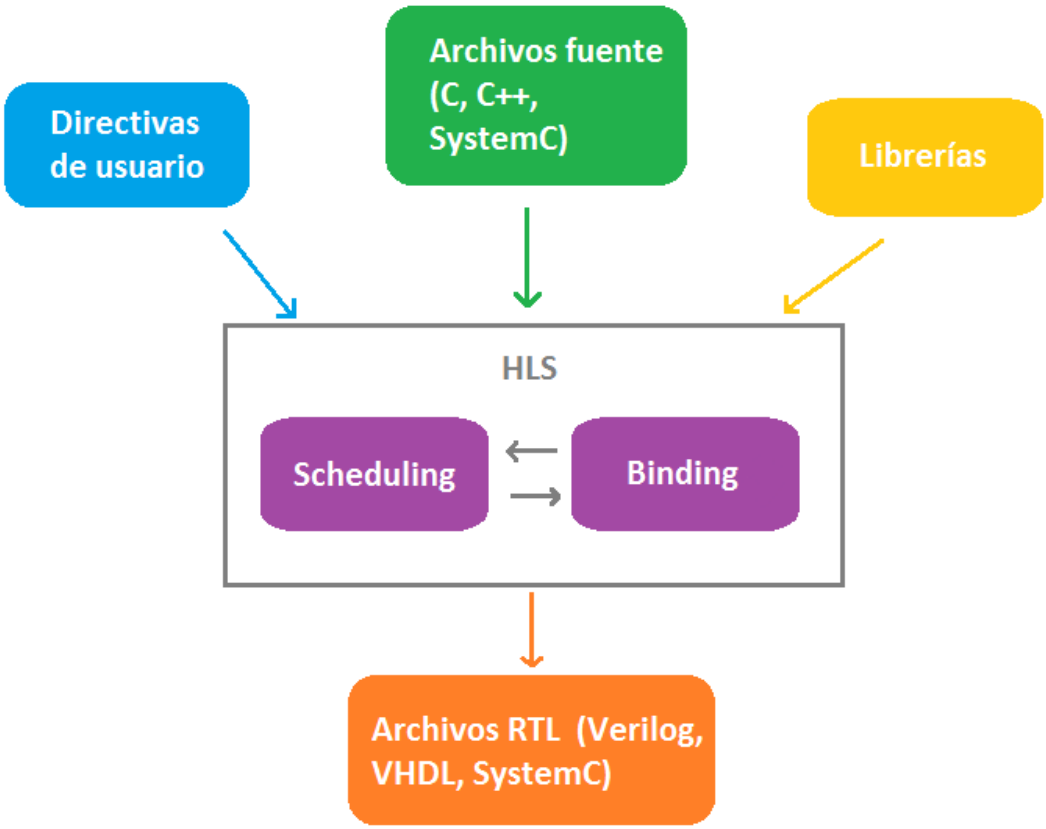


Figura 3-1: Procesos de Scheduling y Binding

El scheduling y el binding dependerán de las directivas utilizadas en nuestro código. Por ejemplo, si con nuestras directivas queremos aumentar la velocidad de ejecución de nuestro código habrá más partes que se ejecuten en el mismo ciclo de reloj y más recursos empleados a cada una. Estas decisiones las hacen las herramientas de HLS de forma automática. En un proceso de diseño hardware con lenguaje de bajo nivel HDL estas decisiones se habría que codificarlas manualmente.

Las dos principales ventajas que ofrece HLS son: aumentar la productividad y reducir el coste de desarrollo. A cambio, obtenemos un diseño que se ve rebajado en rendimiento pero que puede ser válido si cumple con los requisitos mínimos requeridos. A pesar de que el diseño hardware mediante HLS aumenta la abstracción, no deja de ser necesario tener unos conocimientos básicos del dispositivo, de su funcionamiento y de su arquitectura.

La herramienta que ha desarrollado Xilinx para el diseño en HLS es Vivado High-Level Synthesis (HLS). Vivado HLS transforma un código C (como puede ser código C, C++, SystemC u OpenCL) junto con una serie de directivas en una implementación RTL. Después, esta implementación RTL se transforma en un bloque IP que puede ser introducido dentro de otro diseño mediante otra herramienta llamada IP Integrator. En la siguiente figura podemos ver cómo es este proceso.

3.1.1 Arquitectura de un Proyecto de Vivado HLS

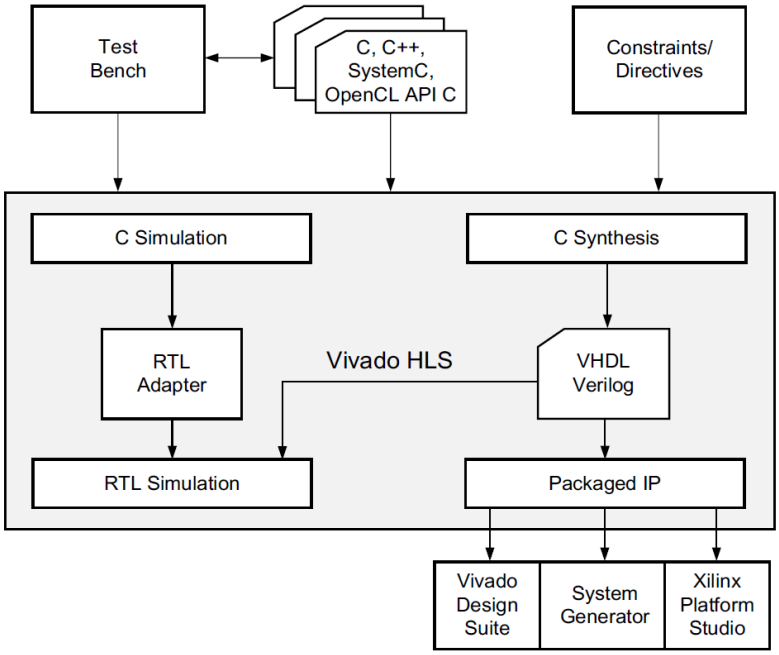


Figura 3-2: Flujo de desarrollo de un proyecto en Vivado HLS [6]

Como se indica en [7] y como podemos observar en la figura anterior, en un proyecto de Vivado HLS tenemos las siguientes entradas:

- Archivos C, C++ o SystemC: Estos archivos contienen las funciones que tienen que ser sintetizadas. Puede tratarse de un único archivo con una única función o, en caso de proyectos más complejos, de varios ficheros y una función principal con una jerarquía de sub-funciones.
- Archivos testbenches: Son testbenches escritos en lenguaje C, C++ o SystemC que permiten verificar el funcionamiento del diseño tanto del modelo en C como del modelo RTL generado.
- Restricciones: Restricciones al diseño, como pueden ser restricciones de timing (establecer un periodo de reloj deseado).
- Directivas: Las directivas que indican al compilador las instrucciones necesarias de cómo tiene que ser la arquitectura final del diseño. Se pueden controlar términos de pipeline, paralelismo y, de forma general, la optimización del diseño.

Igualmente, como se indica en [7] y como podemos ver en la figura anterior, en un proyecto de Vivado HLS tenemos las siguientes salidas:

- Modelo SystemC: modelo del diseño a nivel RTL utilizado para verificación.
- Archivos VHDL o Verilog: Modelo RTL en lenguaje VHDL o Verilog. Se trata de código sintetizable que puede ser introducido en un proyecto y usado para generar un bitstream (archivo .bit) para programar una FPGA o un dispositivo Zynq.
- Paquete IP para Vivado, System Generator, o XPS: Paquete que contiene el modelo para ser implementado en proyectos de Vivado, diseños de System Generator o proyectos XPS. Es también, por lo tanto, código sintetizable.

3.1.2 Interfaz de usuario de Vivado HLS

El entorno de trabajo de Vivado HLS está basado en *Eclipse*. Incluye 3 perspectivas principales [6]:

- Perspectiva de síntesis. La perspectiva de síntesis es la que se utiliza para desarrollar el código de

los bloques IP y de los testbenches para validarlos y verificarlos, de la verificación se hablará más adelante. Se trata de la perspectiva por defecto.

- Perspectiva de debug. La perspectiva de debug permite depurar el código que hemos creado permitiendo simular línea a línea el código y comprobar el valor de las variables en cada momento.
- Perspectiva de análisis. Esta permite ver los detalles de síntesis de los bloques IPs generados. Estos detalles se refieren a la cantidad de recursos necesarios para la creación de cada bloque, la latencia, qué se hace en cada ciclo de reloj, etc. Para ello, primero es necesario realizar la síntesis del código, es decir, generar el bloque IP desde el código C.

En la imagen de la figura 3-3 podemos ver la vista general de Vivado HLS en su perspectiva de síntesis. La perspectiva puede ser cambiada a través de las pestañas de arriba a la derecha.

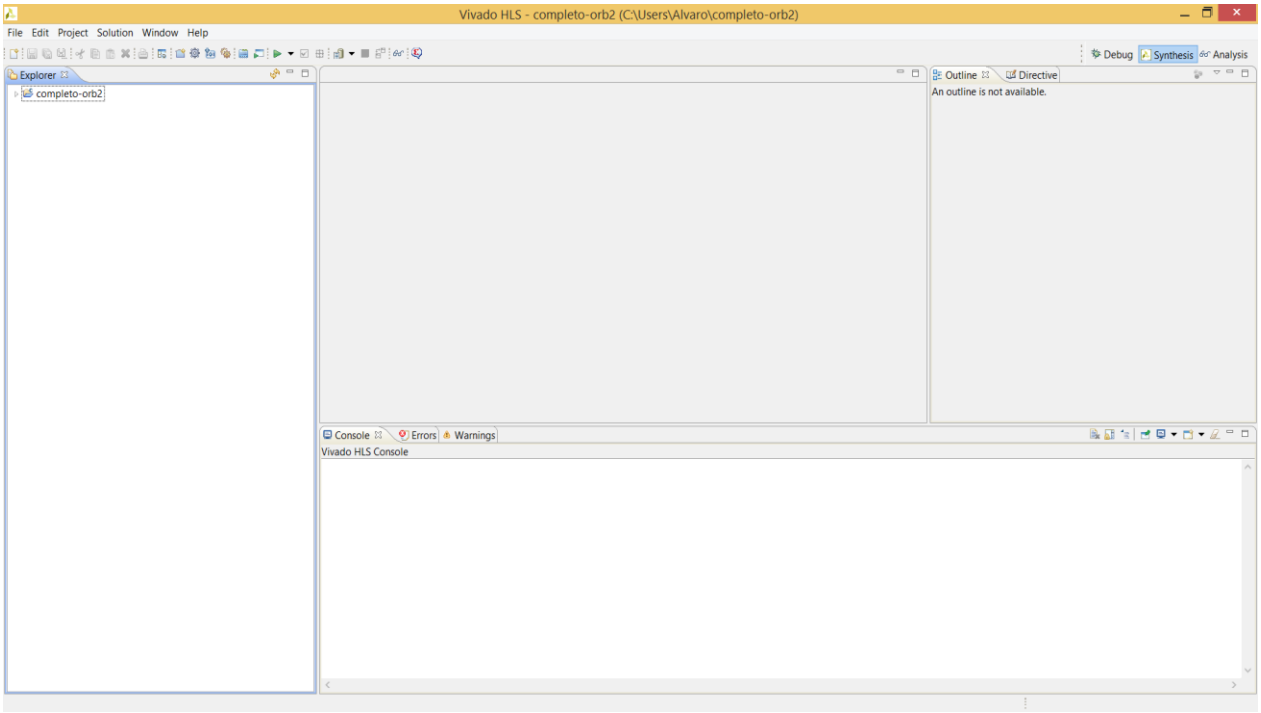


Figura 3-3: Vista del entorno gráfico de Vivado HLS

3.1.3 Simulación y verificación en Vivado HLS

Vivado HLS permite, además de tomar código C y convertirlo en código RTL, comprobar que los bloques IP generados cumplen con el funcionamiento esperado. Para ello, podemos simular el comportamiento de un nuevo bloque a través de testbenches creados también con C (a lo que Xilinx llama como validación del código). A esto se suma que Vivado HLS permite utilizar estos testbenches, los mismos de validación, para la verificación del bloque RTL una vez que este ha sido creado. Vivado HLS permite simular directamente nuestro circuito con el código testbench, como también permite simular paso a paso el código en modo debug. En este modo, se permite ver el valor de cada variable en el paso a paso de nuestro código, así como poner puntos de parada, entre otras técnicas de depuración de código.

Las directivas que aplicamos a nuestro código también permiten optimizar el rendimiento del sistema generado. Existen varias formas de optimizar un sistema, de lo que se hablará más detalladamente en el capítulo 4.

3.2 El entorno de desarrollo SDSoC

SDSoC es un entorno de desarrollo de Xilinx basado en *Eclipse* que se utiliza para la implementación de sistemas completos hardware/software que se basen en los chips de Zynq-7000 o los chips de la familia UltraScale de Xilinx. Los compiladores de sistema (sdscc/sds++) transforman código C/C++ en un sistema

hardware/software completo. Primeramente, se tiene que especificar una plataforma de trabajo, donde quedará definido el hardware que se va a emplear. Se especifican los periféricos, las tarjetas de expansión, etc, que se van a utilizar y las funciones que serán aceleradas en la lógica programable. Al elegir la plataforma también van incluidos archivos de arranque (boot loaders) y archivos del sistema de archivos raíz [8].

En SDSoC todo el código que se escribe es en C/C++. Para el diseño de funciones en hardware se utiliza la herramienta Vivado High-Level Synthesis (HLS), que se encarga de pasar el código C/C++ a código RTL.

3.2.1 Aspecto del entorno de desarrollo

El aspecto general del entorno SDSoC es el que podemos ver en la figura 3-4. De la ventana principal se pueden destacar las siguientes partes más importantes:

- En el lado izquierdo suele estar la ventana exploradora de proyectos, donde podemos navegar a través de todos los archivos correspondientes al proyecto con el que estamos trabajando. Estos pueden ser los ficheros con código del proyecto o los archivos generados tras la compilación, entre otros.
- La ventana del centro es la ventana que se utiliza para abrir los archivos, para programar o para abrir el archivo resumen de un proyecto.
- En la ventana consola se puede ver toda la información de compilación de los proyectos.

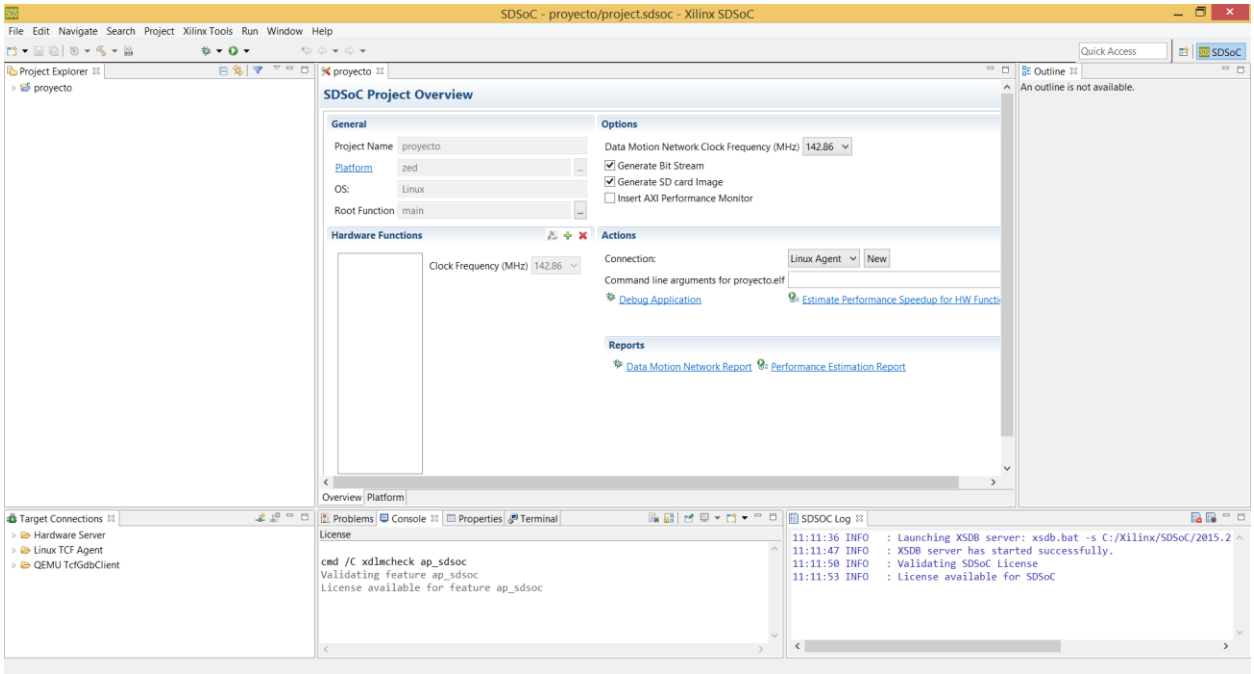


Figura 3-4: Vista del entorno gráfico de SDSoC

3.2.2 Creación de un nuevo proyecto

Para crear un nuevo proyecto se debe pulsar sobre la pestaña *File*. A continuación, se pulsa sobre *New>SDSoC proyect* y aparecerá una nueva ventana donde se configuran las características de nuestro nuevo proyecto. Las opciones más importantes entre las que tenemos que elegir cuando creamos un nuevo proyecto son: el sistema operativo del sistema y la plataforma del sistema. En la figura de abajo podemos ver cómo es la elección de estas opciones.

SDSoC permite crear diseños que contenga el sistema operativo Linux, sistema FreeRTOS o Standalone

(sin sistema operativo). La distribución Linux que se puede incluir a los proyectos SDSoC es Petalinux. Esta distribución de Linux ya viene compilada en la instalación de SDSoC, lo que permite agilizar el proceso de generación de un sistema.

Con las herramientas de Vivado Design Suite se permite crear nuevas plataformas hardware sobre las que diseñar en SDSoC.

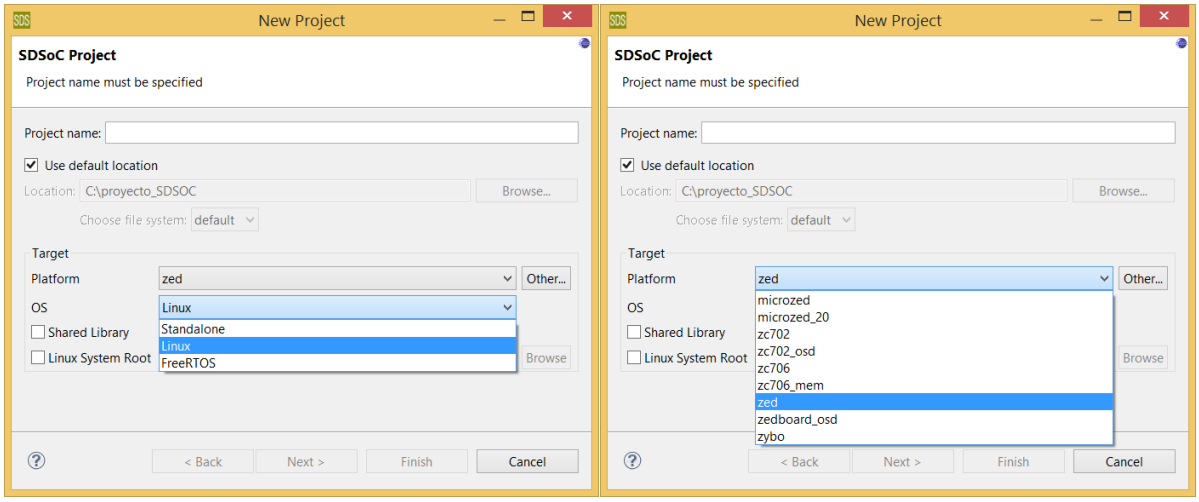


Figura 3-5: Ventana donde se selecciona OS (izquierda) y plataforma (derecha) para un nuevo proyecto en SDSoC

Tras generar un nuevo proyecto, ya se pueden codificar los ficheros fuente donde queda programado el funcionamiento de nuestro sistema.

Las funciones que se quieran acelerar en la lógica programable serán seleccionadas desde la ventana *Hardware Functions*, la cual aparece cuando abrimos el archivo resumen del proyecto. Cuando queramos implementar una nueva función, esta debe cumplir los requisitos necesarios para que sea sintetizable.

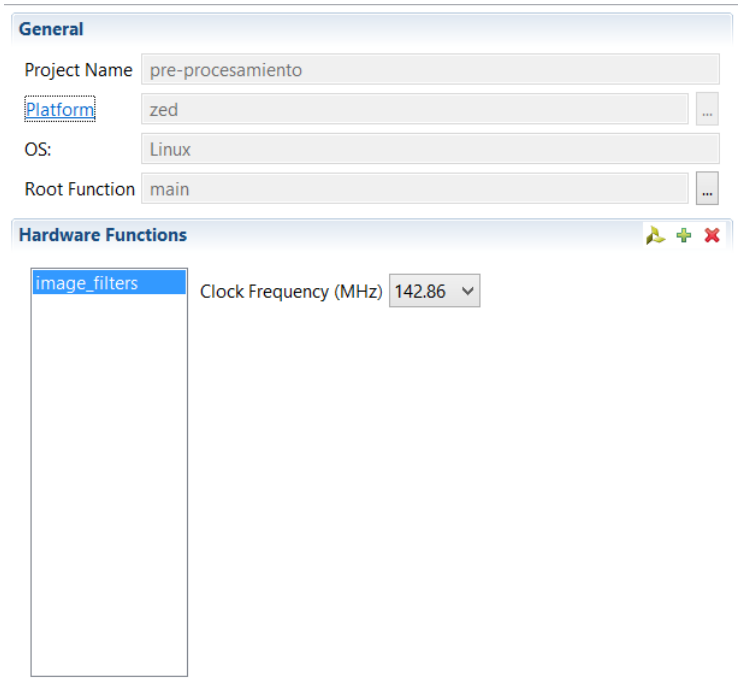


Figura 3-6: Ventana donde se seleccionan las funciones que serán aceleradas en la PL

3.2.3 Plataformas para SDSoC

Las plataformas para SDSoC pueden ser creadas con las herramientas de Vivado Design Suite e IP integrator.

En las dos siguientes imágenes se puede ver el diseño por bloques de la plataforma utilizada en este proyecto y las características del dispositivo Zynq-7000 incluido.

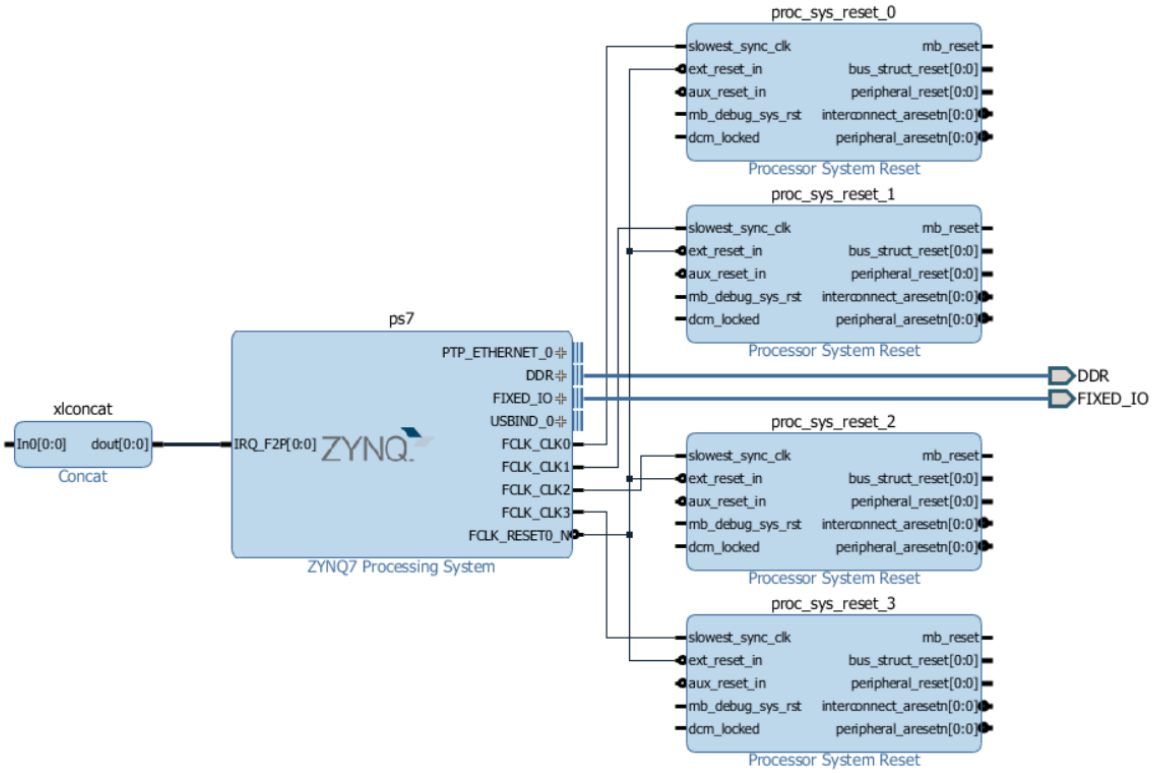


Figura 3-7: Diseño por bloques de la plataforma Zed incluida en SDSoC

Platform Clock Frequencies

Clock	Frequency (MHz)
CPU	666,67
PL 0	166.67
PL 1	142.86
PL 2	100.00
PL 3	200.00

Resources

Resource	Total
BRAM	140
DSP	220
FF	106400
LUT	53200

Figura 3-8: Frecuencias de reloj y recursos disponibles en la plataforma Zed

3.3 OpenCV

OpenCV es una biblioteca libre de visión artificial originalmente desarrollada por Intel que cuenta con una enorme comunidad de usuarios. Al estar bajo la licencia de BSD, puede ser utilizada gratis tanto para uso académico como comercial.

OpenCV es multiplataforma, por lo que existen versiones para distintos sistemas operativos como GNU/Linux, MAC OS, Windows, iOS y Android. Además, puede ser compilada para distintas arquitecturas de máquina. Esta librería se puede emplear con múltiples lenguajes como C++, C, Python y Java.

OpenCV fue desarrollada para ser eficiente y fuertemente enfocada en aplicaciones de tiempo real, como diseñar sistemas de seguridad basados en detección de movimiento, reconocimiento de objetos, monitorización de sistemas industriales, sistemas de ayuda al conductor, como es nuestro proyecto, y otras aplicaciones.

La versión que se emplea para el diseño de este trabajo es la 2.4.5. No es la última versión lanzada, pero es la última versión que incluye Xilinx ya pre-compilada para sistemas ARM, como el que está incluido en la placa de desarrollo Zedboard que se utiliza en este proyecto.

Toda la información sobre OpenCV puede encontrarse en su página web oficial [9].

3.4 Librería HLS Video

Uno de los objetivos de este proyecto es el de acelerar algoritmos de procesamiento de imágenes en la FPGA para mejorar las prestaciones del sistema. Para ello, sería útil poder utilizar las librerías de OpenCV en la FPGA. El problema es que estas funciones no son sintetizables en ella, es decir, que no pueden ser implementadas en una FPGA. Xilinx ha creado la librería HLS_VIDEO que contiene un conjunto de funciones de OpenCV creadas a partir de código C++ sintetizable. Esta contiene funciones dedicadas al procesamiento de imágenes y al traspaso de imágenes entre PS-P a través de las interfaces AXI.

Según se indica en [10] el flujo normal de trabajo para diseñar un sistema acelerado mediante lógica programable es como sigue. Primero, diseñar el algoritmo completo empleando funciones OpenCV. En segundo lugar, averiguar cuáles de estas funciones serían sintetizables en una FPGA. Por último, pasar estas partes del algoritmo a la FPGA empleando la librería sintetizable HLS_VIDEO.

Solo algunas de las funciones incluidas en la librería OpenCV están también incluidas en la librería de Xilinx. Algunas de ellas no son sintetizables o, sencillamente, no están incluidas.

4 OPTIMIZACIÓN DE ALGORITMOS ACELERADOS EN UNA FPGA

Una FPGA es un dispositivo programable compuesto por bloques de lógica configurable (CLB), interconexiones y bloques. Estas interconexiones pueden ser reconfiguradas para poder conseguir la funcionalidad del circuito deseada.

Debido a la estructura que compone la FPGA, esta proporciona un alto grado de capacidad de procesamiento en paralelo. Es capaz de proporcionar un alto grado de rendimiento en aplicaciones que requieren una gran cantidad de operaciones computacionales. Esta es la gran ventaja de una FPGA frente a un microprocesador, el cual descompone cada algoritmo en pequeñas operaciones que realiza de forma secuencial.

Las aplicaciones que suelen ser buenas candidatas para ser implementadas sobre lógica programable como la FPGA son los siguientes [3]:

- Aplicaciones de comunicaciones: Las FPGAs tienen una gran capacidad para manejar grandes cantidades de datos y de información, lo que las convierten en grandes herramientas en tareas de codificación y encriptación empleadas, por ejemplo, en las comunicaciones. Además, las FPGAs permiten velocidades de transferencia más altas ya que, al procesar la información más rápido que un procesador, es capaz de vaciar los buffers más rápido.
- Aplicaciones de visión artificial: De manera resumida, la visión artificial consiste en recibir información visual del exterior, procesarla y actuar en consecuencia. En la visión artificial, la capacidad de procesamiento es crítica ya que se suelen manejar una gran cantidad de datos que deben de ser procesados en poco tiempo. Esta información está en forma de píxeles que pueden ser procesados en paralelo. Es por esto que la FPGA proporciona un gran rendimiento en el campo de la visión artificial.
- Automoción: Sistemas de ayuda al conductor, posicionamiento, control automático, detectores de señales, peatones o colisión. Existen una gran cantidad de aplicaciones dedicadas a mejorar la automoción y que requieren un gran procesamiento de datos en tiempo real.
- Aeronáutica y espacio: Debido a la gran capacidad de procesamiento de la FPGA y a su reducido consumo, tamaño y peso resultan muy útiles en aplicaciones espaciales y aeronáuticas donde el área empleada y el consumo resultan críticos.
- Otras aplicaciones industriales.

Figura 4-1:Aplicaciones para los sistemas heterogéneos con lógica programable (Fuente: Xilinx)

La ventaja principal de los dispositivos heterogéneos, como el Zynq-7000 que se emplea para el desarrollo de este proyecto, es que se pueden acelerar algoritmos que contengan una alta carga de procesamiento en la lógica programable sin renunciar a las ventajas que proporciona un microprocesador (como puede ser utilizar un sistema operativo Linux embebido). Además, no todos los algoritmos son fácilmente implementables y sintetizables sobre una FPGA. Una de las competencias de un co-diseñador es la de decidir qué algoritmos pueden o interesan ser acelerados en la lógica programable.

El co-diseño realizado en este proyecto está completamente desarrollado en lenguaje de alto nivel C++, un lenguaje que fue creado para programar software. Para el desarrollo de las aplicaciones que serán aceleradas en la lógica programable se empleará Vivado HLS, que transforma lenguaje de alto nivel en una descripción hardware y después en lenguaje RTL. La desventaja es que, a cambio de aumentar el nivel de abstracción y la velocidad de desarrollo, el rendimiento conseguido en la aceleración se ve reducido y el área empleada en hardware para la implementación se ve aumentado.

El problema planteado para la síntesis de alto nivel es: ¿Cómo empleamos las técnicas de optimización disponibles en una implementación hardware cuando empleemos lenguaje de alto nivel para su diseño? La solución está en aplicar directivas.

Las directivas permiten al diseñador dar indicaciones al compilador de cómo queremos que sea la arquitectura de nuestro diseño. Con las directivas, el diseñador puede indicar al compilador que genere tal tipo de memoria, que implemente tal tipo de interfaz para la transferencia de datos, que haga paralelismo en un bucle o, por el contrario, que sea procesado de forma secuencial ahorrando recursos. Esto quiere decir que, para un mismo código, es posible obtener implementaciones diferentes según las directivas empleadas.

4.2 Optimización de funciones hardware con HLS

Existen multitud de directivas para los compiladores de Vivado HLS y SDSoC. Pueden revisarse en [8] y [6]. En este capítulo vamos a ver algunas de ellas de forma resumida y basándose en dichas referencias, y veremos cómo ayudan a mejorar el rendimiento del sistema. Las técnicas de optimización HLS para funciones hardware que vamos a ver en este apartado son: funciones inline, pipelining de bucles y funciones, desenrollado de bucles, incremento del ancho de banda de la memoria local, y streaming de datos entre bucles y funciones.

4.2.1 Función inline

Función inline es una técnica de optimización que también se emplea en funciones software. Esta técnica consiste en reemplazar la llamada a una función por copiar el cuerpo de esta y pegarla en la misma línea. Lo que se consigue con esto es hacer que desaparezcan los niveles de jerarquía. Así se permite mejorar la optimización de la función inlined y mejorar la latencia global.

La función inline se aplica en Vivado HLS poniendo la directiva *#pragma HLS inline* al comienzo del cuerpo de la función deseada. Puede verse en el ejemplo siguiente [8] cómo se aplica esta directiva a una función.

```
void mmult_kernel(float in_A[A_NROWS][A_NCOLS],
                  float in_B[A_NCOLS][B_NCOLS],
                  float out_C[A_NROWS][B_NCOLS])
{
#pragma HLS INLINE
    int index_a, index_b, index_d
    //resto del código omitido
}
```

4.2.2 Pipelining y desenrollado de bucles

4.2.2.1 Pipeline

En la ejecución normal de un bucle, como en uno de tipo *for* o de tipo *while*, todas sus iteraciones se producen de manera secuencial. Esto quiere decir que cada iteración en el bucle solo se produce cuando termina completamente la anterior.

Con la técnica de pipeline en bucles no es necesario esperar a que termine la iteración completa de un bucle para comenzar a ejecutar la siguiente. Con el objetivo de entender mejor el pipeline, se muestra en la siguiente figura gráficamente un ejemplo.

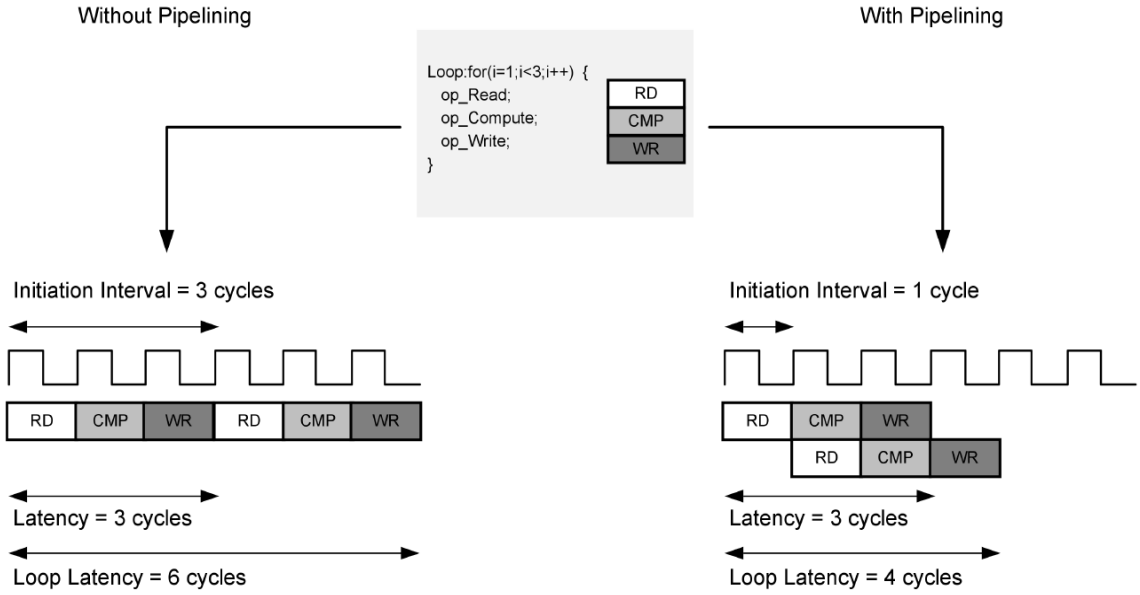


Figura 4-2: Ejemplo de un bucle ejecutado sin pipeline(iquierda) y con pipeline(derecha) [8].

Como podemos ver, el bucle *for* contiene 3 operaciones básicas: leer en memoria, calcular un valor y escribir en memoria. Ejecutando el bucle sin pipeline la latencia total del bucle es de 6 ciclos. Sin embargo, con pipeline la latencia total es de solo 4, ya que no necesita esperar a que la iteración anterior sea completa antes de comenzar la siguiente. La latencia de una sola iteración es de ciclos, como se puede observar.

4.2.2.2 Desenrollado

El desenrollado es otra técnica que permite emplear el paralelismo en los bucles. Esto consiste en realizar más operaciones en cada iteración de un bucle, de modo que se realizan más operaciones a la vez y se disminuye el número de iteraciones. Al disminuir iteraciones, estamos disminuyendo la latencia total del bucle. En el código siguiente podemos ver como se realiza la técnica desenrollado.

```
int sum=0;
for (int i = 0; i<10;i++){
    sum += a[i];
}
```

```
int sum=0;
for (int i = 0; i<10;i=i+2){
    sum += a[i];
    sum += a[i+1]
}
```

Como podemos ver, el desenrollado se realiza repitiendo el cuerpo de un bucle, de modo que las operaciones del bucle se realizan más veces dentro de la misma iteración. El factor de desenrollado del bucle, comúnmente llamado N, indica el número de veces que su cuerpo será copiado en el mismo bucle. Si el factor N es igual al número de iteraciones, se dice que el desenrollado es completo. En el caso de este ejemplo, el factor de desenrollado es $N = 2$.

Para utilizar el desenrollado de bucles en Vivado HLS, tenemos que colocar la directiva *#pragma HLS unroll[factor N]* al comienzo del bucle que queremos desenrollar. En el código siguiente podemos ver un ejemplo de cómo aplicar esta directiva.

```
int sum=0;
for (int i = 0; i<10;i++){
#pragma HLS unroll[factor = 2]
    sum += a[i];
}
```

4.2.2.3 Limitaciones del pipeline y desenrollado

Tanto el pipeline como el desenrollado de bucles explotan el paralelismo de la lógica programable. Existen dos principales factores que limitan este uso: la dependencia de datos entre iteraciones y el número de recursos hardware disponibles.

La dependencia de datos ocurre cuando es necesario conocer el resultado de una iteración antes de comenzar la siguiente. Es por esto, que es obvio que no podrá ejecutarse una iteración sin haber finalizado la anterior y se hayan actualizado todos sus datos.

En cuanto a las limitaciones en recursos hardware, dado que cuanto más paralelismo y velocidad queramos dar a nuestro diseño más recursos deberemos dedicar. Serán las limitaciones en recursos del diseño las que dictaminen si podremos usar paralelismo o no. Otro caso delimitante en recursos es, por ejemplo, cuando se intenta hacer dos operaciones a la vez sobre una memoria de un solo puerto.

4.2.3 Arrays

En Vivado HLS, los arrays siempre son sintetizados como memorias. Estas memorias son mapeadas en la PL como Blocks RAM o memorias RAM distribuidas con slices. La forma en que es reservada esta memoria en PL es importante y conviene saber cómo indicar al compilador el modo en que queremos hacerlo. Las optimizaciones Array disponibles serían las siguientes [8]:

- Mapeo de arrays: Es posible indicar al compilador de que queremos almacenar varios arrays en un solo array más grande. La ventaja que esto produce es que podemos reducir el número de recursos de memoria utilizados.

  Para usar el mapeo de Arrays en nuestro diseño se debe usar la directiva: *#pragma HLS array_map.*

- Partición de arrays: Al contrario que en mapeo de arrays, la partición de arrays consiste en dividir un array en arrays más pequeños o, incluso, en registros individuales que permiten el acceso múltiple en los datos. El objetivo es ampliar el ancho de banda de acceso a datos, es decir, aumentar la tasa de accesos de lectura/escritura.

  Tenemos el ejemplo de una memoria RAM de doble puerto que permite 2 transacciones por ciclo de reloj. Si separamos el array en 4 memorias tipo RAM de doble puerto, obtendríamos hasta 8 transacciones por ciclo.

  Para usar la partición de arrays en nuestro diseño, se usa la directiva: *#pragma HLS array_partition.*

- Reestructuración de arrays: La reestructuración de arrays permite reconstruir un array con palabras pequeñas en otro array con menos palabras y más largas. La motivación de aplicar esta directiva es la de minimizar el número de accesos a memoria.

  La directiva que se debe usar para aplicar la reestructuración de arrays es: *#pragma HLS array_reshape.*

- Streams: Aplicando la directiva *stream* sobre un array permite almacenar un array en una FIFO en lugar de en una RAM.

4.2.4 Pipeline en funciones y en bucles

El pipeline no solo se puede hacer a nivel de bucle, como ya hemos visto, sino que también es posible solapar distintas funciones o bucles en el tiempo cuando lo permite el flujo de datos entre ellos [8]. Es posible realizar pipelining entre funciones, entre funciones y bucles, y entre bucles. Vemos a continuación cómo funciona el pipeline de funciones y de bucles. Para el uso de pipelining en funciones y bucles debe aplicarse la directiva de Vivado HLS *Dataflow.*

4.2.4.1 Pipeline en funciones

Como se puede ver en la siguiente figura, el comportamiento normal de un diseño después de la síntesis es la ejecución secuencial de las funciones que componen el sistema implementado sobre la PL. Utilizando la directiva de Vivado HLS *Dataflow* se puede pedir al compilador que las funciones no esperen a que termine la anterior para que se ejecute, sino que se ejecuten tan pronto como los datos de entrada de la función estén disponibles. Así, en este ejemplo, si no hay pipelining la *func_B* se ejecutará después de que termine la *func_A;* y la *func_C,* después de la *func_B.* En el ejemplo podemos ver que las tres funciones, si no tenemos pipelining, tardarían en ejecutarse completamente 8 ciclos de reloj, mientras que usando pipelining tardarían en ejecutarse 5 ciclos.

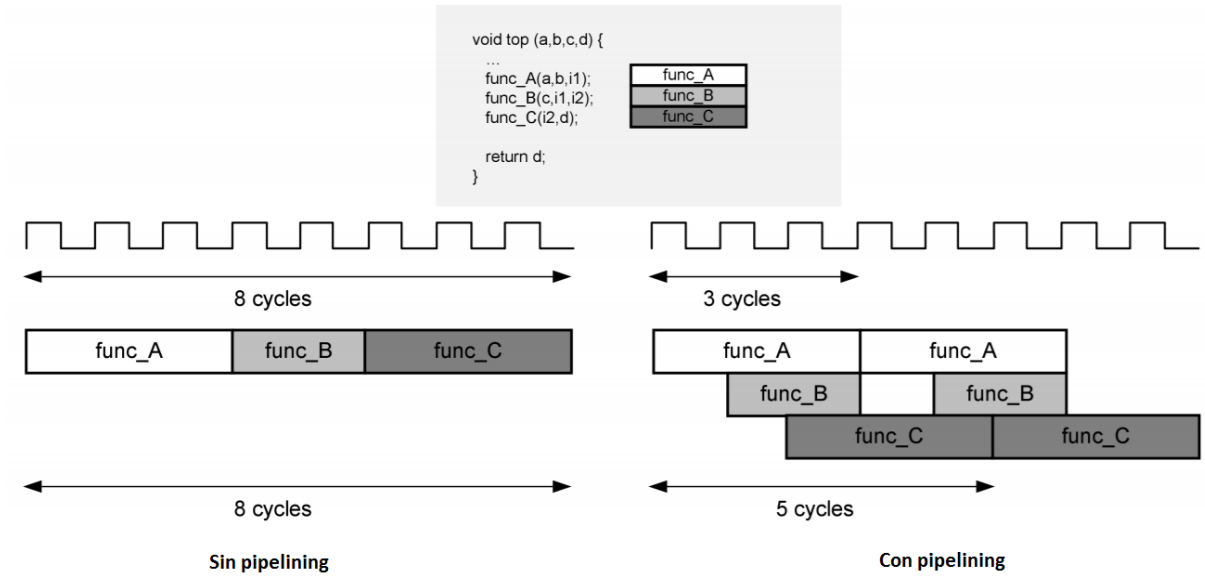


Figura 4-3: Ejemplo de tres funciones ejecutándose sin pipeline(iquierda) y con pipeline(derecha) [8].

4.2.4.2 Pipeline en bucles

De forma similar al pipeline de funciones se puede aplicar el pipelining sobre los bucles, permitiendo la ejecución de varios bucles de forma paralela en lugar de forma secuencial. Para ello, cada bucle se ejecutará desde el momento de que disponga de los datos de entrada necesarios. En el ejemplo de la siguiente figura,

3 bucles se ejecutan de manera secuencial cuando lo hacen sin pipelining tomando para ello hasta 8 ciclos de reloj. En cambio, aplicando la técnica de pipelining, los 3 bucles tardarían en ejecutarse completamente tan solo 5 ciclos de reloj.

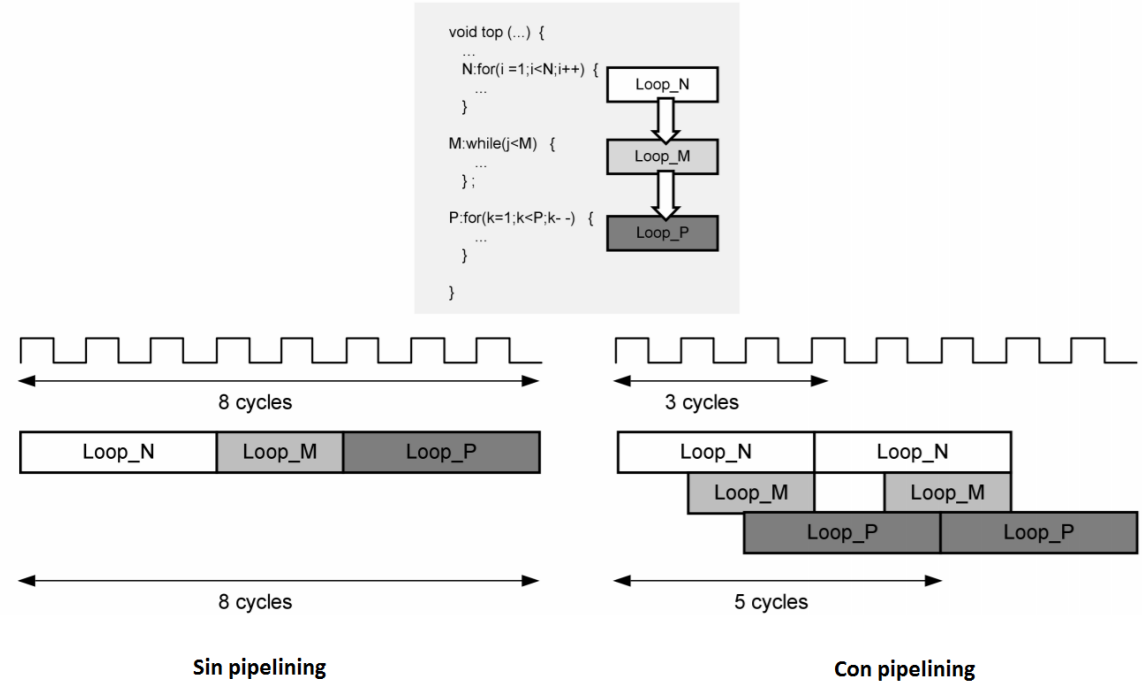


Figura 4-4: Ejemplo de tres bucles ejecutándose sin pipeline (izquierda) y con pipeline (derecha) [8].

4.3 Optimización de la transferencia de datos entre PS-PL con herramientas de HLS

Uno de los puntos clave en la optimización de diseños implementados sobre dispositivos heterogéneos está en la transferencia eficiente de datos entre los diferentes componentes del chip. Para agilizar las transferencias y mejorar el rendimiento en estos sistemas, existen una serie de buenas prácticas que se pueden aplicar como, por ejemplo: almacenar los datos que queremos transmitir en celdas de memoria contiguas, utilizar las interfaces DMA para transferencias de gran tamaño sin interrumpir el sistema, o estructurar correctamente las llamadas a las funciones aceleradas. Este tipo de prácticas suelen ser aplicadas a través de directivas cuando se desarrolla una aplicación con lenguajes de alto nivel.

La herramienta SDSoC empleada en este trabajo proporciona diversas directivas para mejorar el rendimiento de estas transferencias. Además, proporciona también un conjunto de bloques IP prediseñados con el fin de facilitar al diseñador el manejo de las interfaces PS-PL en los dispositivos Zynq-7000 de Xilinx sin que este deba tener demasiados conocimientos técnicos a nivel de hardware. En este apartado, se presentarán estos bloques IP y algunas de estas directivas más interesantes.

No queda de más recordar que, como vimos en el capítulo 2, existen 3 tipos de interfaces en los dispositivos Zynq-7000: las interfaces AXI de propósito general, los puertos de alto rendimiento y el puerto de coherencia del acelerador.

4.3.1 Los SDSoC Data Movers

Con el fin de facilitar el trabajo al diseñador, SDSoC genera, de forma automática, unos bloques IP implementados sobre la PL por cada conexión entre PS-PL que se quiera codificar en el diseño. Estos bloques son conocidos como *data movers* y su funcionalidad es la de arbitrar y facilitar estas conexiones haciendo de bloque intermediario.

Existen distintos tipos de *data movers* y será el propio compilador de SDSoC el que se encargue de elegir el más adecuado a cada conexión. Para la elección del más adecuado, SDSoC tendrá en cuenta las

características de las transferencias, del tamaño de sus datos y de cómo estos están almacenados en memoria. En la siguiente tabla se presentan los distintos *data movers* que se utilizan en SDSoC, se puede encontrar información sobre ellos en la documentación de Xilinx [8].

SDSoC Data Mover	Tamaño de la transferencia	Solo memoria físicamente continua	Descripción
axi_lite			Para transferencias de variables escalares a través de buses AXI4-lite.
axi_dma_simple	$< 8MB$	Sí	Para transferencias de arrays a alta velocidad. Es el *data mover* más eficiente.
axi_dma_sg			Transferencias de arrays a alta velocidad sin límite de tamaño.
axi_dma_2d		Sí	Transferencias de arrays a alta velocidad sin límite de tamaño pero solo para datos almacenados en memoria físicamente continua.
axi_fifo	$\leq 300B$		Transferencias de arrays con menos recursos hardware, a cambio de una menor tasa.
zero_copy		Sí	Para cuando se especifica que los datos sean explícitamente copiados desde PS hasta el acelerador a través del *data mover*.

Tabla 2: SDSoC Data Movers

Como aparece indicado en la tabla, el *data mover* más rápido para transferir arrays es el de axi_dma_simple. El compilador siempre elegirá éste cuando se cumpla con los requisitos necesarios, los cuales son: que los datos a transmitir no pueden superar los $8MB$ de tamaño y que deben de estar almacenados en memoria físicamente continua. En caso de no cumplir con estos requisitos, el compilador se decantará por el bloque axi_dma_sg o axi_dma_2d según las circunstancias. Estos *data mover* utilizan los puertos de alto rendimiento para las transferencias junto con el protocolo de transferencia AXI4-stream.

Para transferencias de pequeños arrays se empleará el *data mover* axi_fifo el cuál, aunque es más lento, depende de menos recursos hardware que los anteriores. Para la transferencia de arrays con estos bloques se suelen utilizar las interfaces AXI de propósito general y el protocolo AXI4.

Cuando se trata de transferir variables escalares, SDSoC elige el bloque axi_lite. Este bloque utiliza la interfaz AXI de propósito general para la transferencia de los datos y el protocolo de transferencia AXI4-lite.

4.3.2 Almacenamiento de los datos en memoria físicamente continua

Cuando se habla de que unos datos están almacenados en memoria físicamente continua, a lo que se refiere es a que todas las celdas de memoria necesarias para almacenar los datos están juntas, es decir, si un puntero de memoria quisiera recorrer todos los datos no tendría que dar saltos en memoria, sino que bastaría con ir incrementando el valor del puntero uno a uno. Esto permite transferencias más rápidas, ya que no resulta necesario calcular saltos en memoria. Esto permite, además, que se puedan emplear bloques DMA para transacciones PL con la memoria en PS sin la intervención del microprocesador.

Almacenar los datos que van a ser transmitidos en memoria físicamente continua tiene claramente sus ventajas como, por ejemplo, poder emplear *data movers* más eficientes para transferirlos.

En SDSoC, podemos asegurarnos de que un array de datos está almacenado en memoria físicamente continua empleando la función *sds_alloc* de la librería *sds_lib* [8]. Para indicar al compilador de que un array está almacenado en memoria físicamente continua, en el caso de que no lo detectara de manera automática, es posible emplear la directiva:

#pragma SDS data mem_attribute (A:PHYSICAL CONTIGUOUS)

4.3.3 Acceso a los datos de forma secuencial o aleatoria.

Las directivas de código también deben indicar qué patrón queremos seguir a la hora de trasmitir datos a través de las interfaces AXI. Generalmente existen dos opciones [8]:

- Cuando una función hardware transmite un array en forma de streaming de datos (es decir, siguiendo el orden índice de la lista) se debe de emplear la directiva:

  #pragma SDS data access_pattern(A:SEQUENTIAL)

- En el caso de que el streaming no sea posible, sino que los valores de la lista serán transmitidos de forma distinta a la de su índice, la directiva que deberá indicarse es la de:

  #pragma SDS data mem_attribute (A:RANDOM)

El streaming es el método más eficiente de transmitir datos ya que se reduce el número de accesos a la memoria externa. Debe de elegirse esta forma por defecto y la aleatoria en caso contrario.

4.4 Procesamiento de imágenes en FPGA con herramientas de HLS

Los algoritmos de procesamiento de imagen y video son algoritmos que contienen una alta carga de procesamiento de datos. El tratamiento que se hace con cada uno de los píxeles suele ser repetitivo y suele admitir el procesamiento en paralelo. Esto quiere decir que estos algoritmos son candidatos perfectos para ser acelerados en la lógica programable. Para conseguir el mayor rendimiento posible, existen una serie de buenas prácticas que se deben de llevar a cabo. Estas buenas prácticas tienen que ver en el empleo de la memoria y cómo no generar cuellos de botella ni dejar a un bloque a la espera de datos. Xilinx ha redactado una nota de aplicación donde habla de estas buenas prácticas [11]. A continuación, se resumirán estos métodos que serán empleados en el proyecto para el procesamiento de imágenes de entrada antes del reconocimiento de las señales de tráfico. Puede consultarse la referencia mencionada para más información.

4.4.1 Empleo de estructuras de memoria

Las imágenes pasan a la lógica programable como un streaming de píxeles. Debido a las limitaciones de latencia y de espacio, no resulta viable almacenar una imagen completa en la memoria de la PL, aunque la mayoría de algoritmos de procesamiento de imágenes necesitan conocer en cada momento el valor de los píxeles vecinos del píxel sobre el que estamos trabajando. ¿Cómo se puede conocer el valor de los pixeles vecinos sin la necesidad de almacenar la imagen completa en memoria?

Existen 3 estructuras de memoria muy utilizadas en el procesamiento digital de imágenes en una lógica programable:

- Registros de desplazamiento: Los registros de desplazamiento permiten almacenar de manera temporal un buffer de una dimensión de datos. Estos datos llegan como un streaming y van almacenándose en el registro de desplazamiento de forma que se pueda acceder a cualquiera de los píxeles en cualquier momento.
- Ventanas de memoria: Una ventana de memoria permite almacenar un conjunto de N píxeles vecinos que rodean a un pixel P central. Puede accederse a cualquiera de ellos en cualquier momento.

- Buffers de línea: Un buffer de línea es un registro de desplazamiento multidimensional capaz de almacenar varias líneas de píxeles de una imagen. Los bufferes de línea son implementados mediante block RAMs.

Puede consultarse más información al respecto en la referencia [12].

4.4.2 Evitar los cuellos de botella en los accesos a datos

Un patrón de acceso a datos incorrecto puede hacer a nuestra aplicación tener que esperar por la lectura y escritura de datos. A continuación, se resumen los detalles que hay que tener en cuenta:

- Evitar releer datos de la memoria externa por parte de la lógica programable. Reutilizar los datos ya leídos almacenándolos en cualquiera de las estructuras de memorias vistas en el apartado anterior.
- Minimizar el acceso a arrays, especialmente si son grandes arrays. Los arrays están implementos en block RAMs, los cuales solo tienen un número limitado de puertos de entrada/salida. Los arrays pueden ser particionados en arrays más pequeños o incluso en registros individuales para permitir el acceso en paralelo a más datos y así evitar los cuellos de botella. Los métodos de partición de arrays han sido vistos en el apartado anterior.
- Minimizar las veces de escritura de datos en memorias externas ya que, de la misma manera que en la lectura de datos, los accesos a memorias externas son las que producen más latencia.

5 DISEÑO DE UN SISTEMA DE DETECCIÓN DE SEÑALES DE TRÁFICO

"Knowing a great deal is not the same as being smart; intelligence is not information alone but also judgement, the manner in which information is coordinated and used.".

- Carl Sagan -

En este capítulo se va a presentar cómo es el desarrollo de un sistema de detección de señales de tráfico. En primer lugar, se plantean los principales problemas a los que hay que enfrentarse a la hora de reconocer un objeto en una imagen y, concretamente, a la hora de reconocer una señal de tráfico en su contexto normal. Después, se analizarán distintas arquitecturas e implementaciones posibles y estudiadas en otros *papers* o trabajos.

5.1 El problema de la detección de señales de tráfico

Los entornos en los que existen las señales de tráfico, como pueden ser las carreteras o la ciudad, suelen ser escenarios complejos, con multitud de objetos que pueden dificultar bastante la detección de señales de tráfico para un sistema de visión artificial. Los factores que dificultan la detección de señales de tráfico pueden ser:

- Oclusiones por objetos que tapan a las señales, como por ejemplo: peatones, coches, arboles u otras señales.
- Señales que aparecen rotadas o giradas, lo cual dificulta su reconocimiento.
- Distintas condiciones de luz que pueden alterar la percepción de los colores por la cámara. Esto puede pasar en situaciones como: en el ocaso del día, de noche o en el amanecer; o con condiciones meteorológicas adversas, como lluvia o nieve.
- El vandalismo, que puede afectar a la forma y al color de la señal.
- Objetos parecidos en forma o en color, los cuales pueden ser reconocidos como una señal de tráfico si se utilizan métodos de reconocimiento poco robustos.
- La gran variedad de señales que existen también es una dificultad añadida importante. No solo se trata de determinar si una imagen contiene una señal de tráfico, sino que también hay que determinar de cuál se trata. La mayoría de las señales presentan características parecidas de color y de forma, y casi siempre la principal dificultad para el clasificador está en diferenciarlas.

5.2 Esquema de un detector de señales de tráfico

Como se ha visto, el entorno de una señal de tráfico es un entorno complejo. Para que sea efectivo un

sistema de detección de señales resulta necesario eliminar el exceso de información de la imagen sin perder la que resulta de interés. Pero esto es algo que ocurre prácticamente en todos los campos de reconocimiento de objetos y de visión artificial. Así pues, cualquier sistema de procesamiento de imágenes se suele dividir en dos niveles: procesamiento de nivel bajo y procesamiento de nivel alto [13].

- Procesamiento de nivel bajo: Es el primero que se aplica a la imagen. Antes del procesamiento de nivel bajo tenemos la imagen completa y original, y tenemos poco conocimiento sobre ella. Los 4 procesos principales que se hacen en el nivel bajo son: adquisición de la imagen, pre-procesamiento, segmentación de la imagen, descripción y clasificación de objetos.
- Procesamiento de nivel alto: Es el siguiente paso tras el procesamiento de nivel bajo. En el procesamiento de nivel alto tenemos suficiente información como para tomar decisiones respecto al contenido de la imagen.

En este trabajo, al igual que en la mayoría de investigaciones sobre detectores de señales de tráfico, al procesamiento de nivel bajo se le suele llamar *etapa de detección* y al procesamiento de nivel alto, *etapa de reconocimiento.* A continuación, se explicarán, por separado y en detalle, las dos etapas.

Figura 5-1: Esquema de un detector de señales de tráfico

5.3 Etapa de detección

El objetivo de la etapa de detección, como se acaba de comentar, es el de reducir la información de la imagen de entrada todo lo posible para facilitar la tarea de reconocimiento a la siguiente etapa. La etapa de detección debe determinar, de la imagen de entrada, qué regiones tienen probabilidad de contener una señal de tráfico y qué regiones deben de ser descartadas. De las primeras, debemos calcular las coordenadas para utilizarlas en la etapa de reconocimiento. Veamos de qué tipo de operaciones está compuesta esta etapa.

La etapa de detección suele incluir las siguientes tareas, que no tienen por qué ser todas:

- Lectura de la imagen. Evidentemente, el primer paso para la etapa de detección es cargar el primer *frame*. Las imágenes pueden venir como imágenes individuales o ser un video, que es una composición de imágenes igualmente. Estas imágenes son leídas en un formato de color determinado, normalmente en el espacio de color RGB, y debe de ser adaptado al sistema. Las imágenes pueden proceder de una cámara o de la memoria.

- Preprocesamiento de la imagen. Consiste en el tratamiento de la imagen para mejorarla. Se pueden aplicar filtros, como el *filtro Gaussiano* o el *filtro de mediana,* para suavizar la imagen y eliminar ruido. También, se suelen emplear operaciones morfológicas para eliminar pequeños objetos que son demasiado pequeños como para reconocerlos.
- Segmentación. Se procesan los píxeles de la imagen en busca de sus características. El objetivo es poder clasificar los píxeles según algún criterio. Normalmente, en los sistemas de detección de señales se suele analizar el color y la textura de los píxeles. De esta manera, se filtrarán aquellos píxeles de un color específico o el conjunto de píxeles que tenga una forma geométrica específica.
- Descripción y clasificación de objetos. Se trata de clasificar, según algún criterio, a aquellos píxeles que han pasado el umbral en la segmentación. Por ejemplo, separar píxeles rojos de azules o separar píxeles que forman una figura circular de los que forman una cuadrada.

5.3.1 Lectura de la imagen

No será objeto de este trabajo la lectura de imágenes desde una cámara, sino que las imágenes serán leídas desde la memoria flash. El hecho de añadir una cámara implicaría el diseño de bloques IP, implementaciones de interfaces DMA, drivers, etc. Las imágenes serán leídas de la memoria empleando funciones definidas en la librería OpenCV. Después, las imágenes se adaptarán al formato empleado en el sistema. Esta parte será mejor explicada en segmentación.

5.3.2 Preprocesamiento de la imagen

Como ya se ha dicho, el preprocesamiento de la imagen sirve para mejorarla de cara a las etapas de segmentación, clasificación y reconocimiento. Existen multitud de técnicas de procesamiento de la imagen, como por ejemplo técnicas de suavizado de la imagen mediante filtros o eliminación de pequeños puntos de la imagen mediante operaciones morfológicas.

5.3.3 Segmentación

5.3.3.1 Segmentación por regiones de búsqueda

Uno de los métodos más simples para segmentar la imagen consiste en separar el marco de la imagen en partes. El criterio de esta separación es considerar que, si una cámara está grabando la carretera o la calle por la que se conduce, las señales de tráfico solo aparecerán en algunas zonas de las imágenes grabadas y no en cualquiera. Estas zonas pueden ser, por ejemplo, los lados de la acera o colgadas de un semáforo, las cuales suelen quedar en los extremos del marco. A estas zonas, que son las únicas de la imagen original que son procesadas, se les llama comúnmente **regiones de búsqueda (RoS)**. En [14] se aplica esta técnica de segmentación. En la siguiente imagen podemos ver un ejemplo de cómo se pueden establecer las RoS:

Figura 5-2: Ejemplo de regiones de búsqueda en una imagen

Como podemos observar en el contexto de esta imagen, gracias al establecimiento de unas RoS adecuadas se evitaría el tratamiento de la parte central de la imagen, que en la mayoría de las veces no contendrá señales de tráfico y sí otros objetos que podrían pasar por otros umbrales y acabar clasificándose erróneamente como una señal.

5.3.3.2 Segmentación por color

Una de las ventajas de las señales de tráfico es que están compuestas de formas y colores muy definidos. En la mayoría de países, las señales de tráfico suelen estar compuestas de los siguientes colores: rojo, azul, amarillo y blanco. Por lo tanto, este es uno de los puntos fuertes que hay que tener en cuenta para filtrar las señales de tráfico. En el método de segmentación por color se hace uso de los colores característicos de las señales para tratar de diferenciarlas del resto de la imagen.

Según este método, la imagen es recorrida píxel a píxel de forma que de cada uno de ellos se extrae el color y se hace pasar por un umbral para determinar si es del color que buscamos. El resultado de esta etapa es una imagen binaria, es decir, una imagen donde sus píxeles pueden valer '1' (en caso de que ese píxel haya pasado el umbral por color) o '0' (en caso de que no lo haya pasado y, por lo tanto, contiene fondo).

La desventaja de los métodos basados en el color es que este puede verse afectado por diversos factores, de los que ya se ha hablado al comienzo del capítulo.

5.3.3.2.1 Espacio de color

Cuando se emplea la segmentación por color hay que tener en cuenta qué espacio de color se va a utilizar. La mayoría de las veces, las imágenes son obtenidas en formato RGB. Este espacio tiene la ventaja de que es el sistema más fácil de implementar ya que suele ser el formato que se obtiene por defecto. Lo único que habría que hacer es extraer de cada píxel los 3 canales (R, G y B) y comparar esos valores con un umbral.

Para la umbralización del color en RGB, en [15] se estudian varios métodos y ecuaciones. Tras varias pruebas, se llega a la conclusión de que las ecuaciones con las que mejores resultados se obtienen para umbralizar el color rojo y el color azul son las siguientes:

- Para el color rojo:

$$R'(i,j) = k_1 R(i,j) \qquad k_1 \epsilon [1 - 1.492] \tag{1}$$

$$pixel(i,j) = rojo\ si \begin{cases} R'(i,j) > 2G(i,j) \\ y \\ R'(i,j) > 2B(i,j) \end{cases} \tag{2}$$

- Para el color azul:

$$R'(i,j) = k_2 R(i,j) \qquad k_2 \epsilon [0.5 - 0.992] \tag{3}$$

$$pixel(i,j) = azul\ si \begin{cases} B(i,j) > 2R'(i,j) \\ y \\ B(i,j) > G + offset\_diff \end{cases} \tag{4}$$

donde $offset\_diff = 16$ es un umbral que se utiliza para mejorar los resultados. Los valores de k_1 y k_2 dependen de los intereses del diseñador y sirven para compensar las diferentes condiciones de luz.

El problema del RGB es que resulta muy sensible a los cambios de luminosidad de la imagen. Los cambios de las condiciones de luminosidad, debido a zonas oscuras o deslumbramientos, afectan de forma aleatoria a la captación del color.

Uno de los espacios de color más utilizados es el HSI. HSI junto con HSV o HSL tienen la ventaja de que son invariantes a los cambios de luz de la imagen. La desventaja es que su implementación es más compleja, sobre todo cuando quieren ser diseñados sobre lógica programable.

Para implementar la segmentación por color con alguno de estos formatos debe calcularse primero la componente de tono (H), cuya fórmula es la siguiente [16]:

$$H = \begin{Bmatrix} \theta\ si\ B \leq G \\ 360° - \theta\ si\ B \geq G \end{Bmatrix} \tag{5}$$

con:

$$\theta = cos^{-1} \left\{ \frac{\frac{1}{2}[(E-G) + (R-B)]}{\left[(R-G)^2 + (R-B)(G-B)^{\frac{1}{2}}\right]} \right\} \tag{6}$$

Como puede verse, calcular H puede ser muy complejo si queremos sintetizar el cálculo en una FPGA. Sin embargo, se puede calcular de forma más rápida empleando la aproximación dada por [17]:

$$H = \begin{cases} 0 & si\ R = G = B \\ \dfrac{(G-B)60°}{\max(R,G,B) - \min(R,G,B)} mod\ 360°, R \geq G, B \\ \dfrac{(B-R)60°}{\max(R,G,B) - \min(R,G,B)} + 120°, G \geq R, B \\ \dfrac{(R-G)60°}{\max(R,G,B) - \min(R,G,B)} + 240°, B \geq R, G \end{cases} \quad (7)$$

Si quisiéramos ahora calcular los valores de saturación (S) e intensidad (I) para el espacio HSI, podemos aplicar las siguientes ecuaciones [18]:

$$S = 1 - \frac{3}{(R+G+B)}[\min(R,G,B)] \quad (8)$$

$$I = \frac{1}{3}(R+G+B) \quad (9)$$

Podemos clasificar un píxel por color, cuando se tiene en formato HSI, con el siguiente criterio [16]:

$$pixel(i,j) \begin{cases} Rojo, & si\ 10 \leq H(i,j)\ \text{ó}\ H(i,j) \leq 300 \\ Azul, & si\ H(i,j) \geq 190\ y\ H(i,j) \leq 270 \\ Amarillo, & si\ 20 \leq H(i,j) \leq 60\ y\ S(i,j) \\ Blanco, & si\ S(i,j) \leq 48\ y\ I(i,j) \geq 60 \end{cases} \quad (10)$$

Para tener más información sobre los espacios de color puede consultarse [19].

5.3.3.2.2 Segmentación por color en otros trabajos

Otros trabajos de investigación donde se ha empleado la segmentación por color son:

- En [16] se extrae la crominancia de cada píxel para clasificarlo por umbralización. En el estudio, se establecen de manera experimental diferentes valores de umbral para diferentes condiciones de luz y meteorológicas. Para más información sobre la crominancia de una imagen y del espacio de color YCbCr (basado en la crominancia) puede consultarse [20].
- En [21] y [22] se utiliza el espacio de color RGB, haciendo uso de umbrales adaptativos.
- En [23] se considera la implementación en hardware de la segmentación por color basado en HSV por delante de HSI al considerarse que se obtiene mejor latencia.
- En [24] los píxeles en RGB también son transformados a HSV para su umbralización.
- En [25] el espacio de color empleado es HSI.
- En [26] se realiza un amplio repaso de numerosas investigaciones sobre detección de señales de tráfico en el que se concluye que HSV y HSL son los espacios de color más utilizados.

5.3.3.3 Segmentación por forma

Además de la segmentación por color, en la detección de señales de tráfico también se suele utilizar la segmentación por forma. La segmentación por forma consiste en identificar objetos de la imagen que tengan una forma geométrica determinada. En el caso de las señales de tráfico, estas suelen ser octogonales, rectangulares, triangulares o circulares.

En la mayoría de los casos, la segmentación por forma suele ir precedida por la segmentación por color. Esto quiere decir que, de una imagen, primero se extraen las regiones que contienen colores determinados, como rojo, azul, etc, y después son clasificadas por su forma. Así, por ejemplo, una señal de 'prohibido' será segmentada por ser roja y después por ser circular.

Pero no tiene porque ser así, existen trabajos en los que solo se utiliza la segmentación por color y trabajos donde solo se utiliza la segmentación por forma.

Existen diversos métodos algorítmicos para encontrar formas geométricas dentro de una imagen. El método empleado dependerá de si la imagen ha sido previamente segmentada por color o no. En el caso de haber sido segmentada por color, la segmentación por forma se procesará, normalmente, sobre una imagen binaria, lo que lo hace mucho más sencillo.

5.3.3.3.1 Segmentación por forma en otros trabajos

Vemos algunos ejemplos donde se aplica la segmentación por forma:

- En [27] se emplea la técnica de *pattern matching* para detectar formas después de la etapa de segmentación por color. Esta técnica consiste en comparar la forma de los objetos previamente detectados con formas geométricas almacenadas en memoria.
- En [23] se utiliza la transformada de Hough para encontrar objetos circulares. Primero, se utiliza la segmentación por color. De la imagen binaria obtenida, se detectan los bordes de los objetos basándose en el gradiente de la imagen. Finalmente, se utilizan estos bordes para encontrar círculos con la transformada de Hough.
- En [24] se utiliza el algoritmo de Ramer-Doublas-Peucker para reducir el número de puntos utilizados en la aproximación de una curva y determinar la forma de un objeto según el número de puntos obtenidos.

5.3.4 Etiquetado de componentes conectados y extracción de las regiones de interés

El etiquetado de componentes conectados (CCL, connected component labeling) es uno de los procesos más importantes y utilizados en los sistemas de visión artificial. CCL supone la interfaz entre el procesamiento de imágenes a bajo nivel y el de alto nivel.

CCL consiste en asignar un identificador único a cada conjunto conexo de píxeles de una imagen [28]. Este conjunto conexo suele recibir el nombre de objeto. La entrada del CCL es una imagen binaria que ha sido producto del procesamiento de nivel bajo del sistema. Como salida, el CCL devuelve un identificador único para cada objeto encontrado en la imagen binaria y sus coordenadas para ser localizado dentro de la imagen. Estas coordenadas son después necesarias para que en el procesamiento de nivel alto se puedan localizar los objetos y tratarlos.

Para determinar si dos píxeles de una imagen están conectados se utiliza el siguiente criterio: en una imagen binaria B, dos píxeles p y q están conectados cuando estos pertenecen al primer plano (Foreground) o al fondo (Background) y existe un camino de píxeles del mismo tipo entre ellos, es decir, que se cumple (11), donde S es un subconjunto de píxeles de la imagen binaria B [13].

$$p\ conectado\ a\ q\ \leftrightarrow \exists\ \{s_i \in S \mid s_1 = p, s_{n+1} = q, s_{i+1} \in N(s_i), i = 1, \dots, n\}, \quad (11)$$

siendo $N(s_i)$ el conjunto de vecinos del píxel s_i.

Los algoritmos de CCL van recorriendo la imagen píxel a píxel analizando cada uno de ellos y a sus vecinos. Para ello, se establece una ventana alrededor del píxel que recoge, según el algoritmo, a cuatro o a ocho de sus vecinos más cercanos. Esta ventana puede tener alguna de las formas que se muestran a continuación:

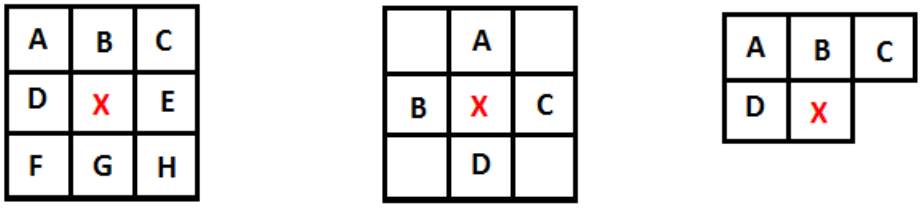


Figura 5-3: Ejemplos de ventanas para un algoritmo de etiquetado

Los algoritmos de CCL suelen estar clasificados según el número de pasadas que se realizan a la imagen, es decir, el número de veces que se recorre la imagen píxel a píxel [13]. Estos pueden ser:

- Algoritmos de múltiples pasadas (multi-scan). En estos algoritmos se realizan varios barridos de la imagen. Estos barridos suelen ser alternos (de arriba a abajo y de izquierda a derecha, y después en el sentido contrario) y son realizados hasta que no haya cambios en el valor de las etiquetas. El acceso a memoria es de forma regular. El problema es que tantos accesos a memoria no resulta eficiente para un sistema de tiempo real y menos para una implementación en hardware.
- Algoritmos de dos pasadas (doble-scan). En los algoritmos de dos pasadas se realizan dos barridos de la imagen. En la primera se asignan etiquetas temporales y en la segunda se determinan las etiquetas definitivas. Se utilizan tablas asociativas para las etiquetas, las cuales se van completando en la primera pasada y son utilizadas en la segunda para calcular el valor de las etiquetas definitivas. Un ejemplo de un algoritmo de dos pasadas para sistemas de tiempo real e implementado sobre hardware lo podemos ver en [13].
- Algoritmos de una pasada (one-scan). Los algoritmos de una pasada están pensados para ser implementados en hardware. Cuando un algoritmo es implementado en hardware, la imagen debe llegar como un streaming de píxeles. Estos píxeles van siendo analizados junto con sus vecinos y, como no pueden ser almacenados en memoria debido a que no habría suficiente, se emplean estructuras de memoria para almacenar solamente los datos que sean necesarios. En [15] se emplea este tipo de algoritmo.

5.4 Etapa de reconocimiento

La etapa de reconocimiento es la que viene detrás de la etapa de detección. La etapa de reconocimiento analiza las regiones que han pasado con éxito la etapa de detección y, por lo tanto, aquellas que son altamente probables de contener una señal de tráfico. Sobre estas regiones, la etapa de reconocimiento decidirá si existe alguna señal y, en tal caso, decidir cuál es.

Existe una gran cantidad de métodos de reconocimiento de objetos. Para la realización de este trabajo se han estudiado algunas de las posibilidades que se presentan en este apartado.

5.4.1 Clasificación por características locales

Una característica local describe las propiedades de un píxel de la imagen con respecto a sus píxeles vecinos. Las características locales proporcionan información sobre estructuras relevantes de la imagen como pueden ser: esquinas, bordes, contornos, texturas, etc [29]. Las características locales ayudan, por lo tanto, a describir una imagen y proporcionar información para diferenciarla de las demás.

Los algoritmos de extracción de características locales se dividen en dos fases, detección y descripción. En la detección de características locales se aplican diferentes algoritmos matemáticos sobre la imagen con el fin de encontrar las características que mejor describan la imagen. Estas características se suelen llamar puntos de interés, ya que han sido encontradas en puntos concretos de la imagen. Los algoritmos matemáticos aplicados dependerán del método implementado. El resultado de esta fase es un conjunto de

puntos de interés que resultan relevantes para definir la imagen.

En la descripción, el objetivo es describir los puntos de interés encontrados en la fase anterior. Esta descripción consiste en asignar un vector a cada punto de interés, los cuales ayudarán a poder clasificar la imagen con algún método de clasificación.

Existen numerosos métodos de descripción y detección de características locales. A continuación, se tratará brevemente algunos de ellos.

5.4.1.1 Descriptor SIFT

SIFT es uno de los detectores y descriptores de características locales más utilizados actualmente. SIFT basa su método de detección de puntos de interés en aplicar una secuencia de filtros Gaussianos a diferentes escalas y resoluciones. Veamos un poco cómo funciona el método SIFT en la detección de putos de interés.

5.4.1.1.1 Diferencia de Gaussianas (DoG)

Aplicar un filtro Gaussiano sobre una imagen consiste en ponderar cada uno de los píxeles con respecto a sus píxeles vecinos empleando la función Gaussiana. Al aplicar este filtro la imagen queda suavizada. El parámetro más importante en el filtro Gaussiano es el parámetro de la desviación estándar, σ. Cuanto mayor sea σ, más se tendrá en cuenta la influencia de los píxeles vecinos y, consecuentemente, mayor será el suavizado.

Figura 5-4: Efecto de aplicar un filtro Gaussiano sobre una imagen con distintos valores de σ

La diferencia de Gaussianas consiste en calcular la diferencia entre dos imágenes suavizadas con filtros Gaussianos con diferente σ. Tras este cálculo, se podrá detectar aquellas zonas de la imagen en las que existan grandes cambios de intensidad. En el método SIFT se van calculando estas diferencias Gaussianas variando σ de manera creciente, lo que permite analizar la imagen a distintas escalas o resoluciones. Cambiar la escala permite encontrar cambios bruscos de intensidad en regiones más localizadas (o pequeñas) y en regiones más grandes. Aquellos puntos donde la DoG tengan valores extremos serán guardados como puntos de interés de la imagen. Para calcular los valores extremos se sigue la norma de que el valor de la diferencia en ese punto tiene que ser extremo con respecto al valor de sus vecinos más cercanos y, a la vez, extremo con respecto a sus vecinos en las escalas inmediatamente mayor y menor [29].

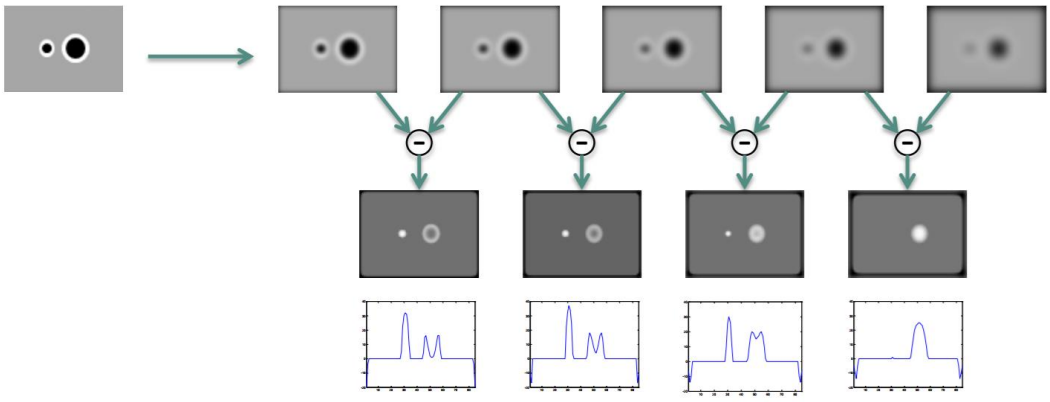


Figura 5-5: Respuesta a la DoG con diferentes escalas [29]

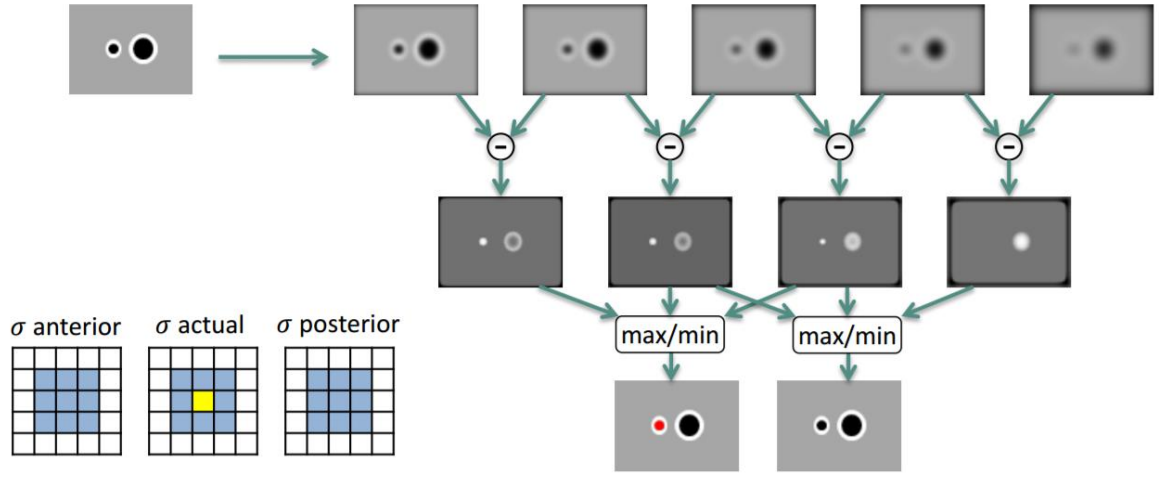


Figura 5-6: Búsqueda de extremos locales [29]

En la figura 5-5 podemos ver cómo varía la respuesta de la DoG cuando σ va cambiando. Para una σ más pequeña la respuesta será mayor cuando la diferencia de intensidad se produce en zonas más localizadas. En cambio, a medida que σ es más grande, notaremos una mayor respuesta en áreas de cambios de intensidad más grandes. En la figura 5-6 vemos el ejemplo de cómo se buscan los extremos locales.

5.4.1.1.2 Pirámide de imágenes

Esta es una técnica utilizada en procesamiento de imágenes que consiste en generar otra imagen más pequeña después de haber aplicado sobre la primera un suavizado (normalmente aplicando previamente un filtro gaussiano). Esta operación se repite varias veces de forma que se obtiene una pirámide de imágenes. En el siguiente apartado veremos para qué se utiliza, en un detector SIFT, esta pirámide de imágenes.

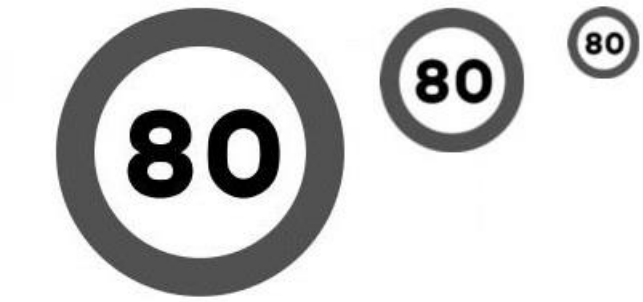


Figura 5-7: Pirámide de imágenes

5.4.1.1.3 Detección de los puntos de interés

SIFT utiliza la DoG para la detección de puntos de interés de una imagen. Tras aplicar DoG a distintas escalas de suavisado, se repite la misma técnica con distintos tamaños de la imagen aplicando la técnica de la pirámide. El hecho de utilizar una pirámide de imágenes en la detección con diferentes niveles de resolución y distintas escalas de suavizado permite detectar siempre los mismos puntos de interés de un objeto, independientemente del tamaño de este [29].

5.4.1.1.4 Asignación de orientaciones

La asignación de orientaciones se utiliza para garantizar la invariancia respecto a la rotación de las imágenes.

Para calcular la orientación, primero se calcula el histograma de direcciones de gradientes locales de los

píxeles vecinos a una escala dada. El valor máximo en el histograma indica la orientación del punto de interés [31]. Vemos a continuación cómo se calcula el gradiente en un píxel y, en el apartado 5.4.2, cómo se calcula el histograma de gradientes.

Figura 5-8: Ejemplo de puntos SIFT detectados con su orientación

5.4.1.1.5 Descripción de las características locales con SIFT

Una vez obtenidos los puntos de interés de una imagen debemos conseguir un descriptor de cada uno de estos puntos. Los descriptores de características locales suelen ser, normalmente, vectores numéricos que representan localmente a cada píxel con respecto a sus píxeles vecinos. Los descriptores de los puntos de interés servirán después para poder clasificar la imagen.

Para calcular los vectores de descripción (en el caso de un descriptor SIFT) se utiliza el concepto de gradiente. El gradiente es un cálculo que permite detectar cambios de intensidad en una imagen. Se compara la intensidad de un píxel con la intensidad de sus píxeles vecinos y se calcula la diferencia con el objetivo de calcular los cambios de intensidad. Esta comparación se hace en dos direcciones: horizontal y verticalmente. De este modo se obtienen dos valores por cada punto de interés: un valor para el eje horizontal y otro para el eje vertical. Estos dos valores son las coordenadas de un vector, llamado vector gradiente. El módulo de este vector es la magnitud del gradiente y el ángulo que forma con el eje horizontal, la orientación del gradiente.

Para calcular la descripción de los puntos de interés de una imagen se calcula primero el vector gradiente en cada uno de estos puntos detectados previamente y de los píxeles vecinos a cada uno de ellos. Después, una vez calculados todos los vectores gradiente, se construye un histograma por cada punto y sus vecinos en el que cada píxel quedará clasificado según el valor de su orientación. Con el conjunto de todos estos histogramas se formará un único vector que describe a la imagen completa.

5.4.2 Descriptor basado en Histograma de Gradientes Orientados

El histograma de gradientes orientados, del inglés histogram of oriented gradients (HOG), es un descriptor que se basa en el gradiente de cada uno de los píxeles como información básica para obtener la descripción de una imagen. Esta es la misma técnica que se utiliza para obtener la descripción de los puntos de interés de una imagen en el método SIFT. La diferencia entre ellos es, básicamente, que en SIFT esta técnica es empleada únicamente sobre los puntos de interés detectados previamente y en un descriptor HOG el gradiente es calculado sobre todos los píxeles.

Para calcular el descriptor HOG de una imagen tenemos que calcular previamente el gradiente de cada uno de sus píxeles. Una vez calculados estos gradientes, el siguiente paso es calcular el histograma de estos gradientes. No se hace un solo histograma de la imagen completa, sino que se calcula un histograma por cada pequeña fracción de la imagen, como veremos a continuación. En conjunto con estos histogramas, se determina un descriptor que describe la imagen. Veamos cómo se realizan cada uno de estos 3 pasos.

5.4.2.1 Cálculo del gradiente

Como ya hemos visto, el gradiente se define como un cambio de intensidad de la imagen en una cierta dirección, aquella en la que el cambio de intensidad es máximo. El gradiente se calcula sobre cada uno de los píxeles de una imagen y queda definido, para cada píxel, con los valores de dirección, hacia donde el cambio de intensidad es máximo, y la magnitud del cambio en esa dirección.

El cálculo del gradiente es muy utilizado en el campo de la visión artificial ya que permite adquirir mucha información sobre una imagen. Por ejemplo, el gradiente se utiliza para detectar los bordes de un objeto contenido en una imagen. Podemos ver en la siguiente figura el resultado de calcular el gradiente sobre una imagen completa, tanto en dirección horizontal como en vertical.

Figura 5-9: Ejemplo de aplicación del gradiente para buscar los bordes de una imagen

Para entender cómo se obtiene el gradiente de una imagen vamos a hacer uso del siguiente ejemplo. Supongamos que la siguiente tabla representa la matriz de píxeles de una imagen y que los valores de cada punto de la matriz representan el valor de intensidad de cada píxel.

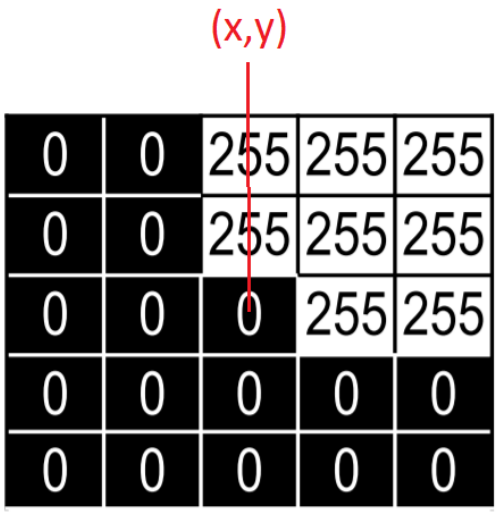


Figura 5-10: Ejemplo para el cálculo del gradiente

Para calcular el gradiente del píxel central, con coordenadas (x, y), se calcula la diferencia de intensidad de sus píxeles vecinos, horizontal y verticalmente. La formulación queda expresada de la siguiente manera:

$$dx = I(x+1, y) - I(x-1, y) = 255 \qquad (12)$$

$$dy = I(x, y+1) - I(x.y-1) = 255 \quad (13)$$

Una vez obtenida la diferencia de intensidad tanto en dirección horizontal como en vertical, se puede calcular la magnitud del gradiente y su orientación de la siguiente manera:

$$\theta(x,y) = arctan\frac{dy}{dx} \quad (14)$$

$$g(x,y) = \sqrt{dx^2 + dy^2} \quad (15)$$

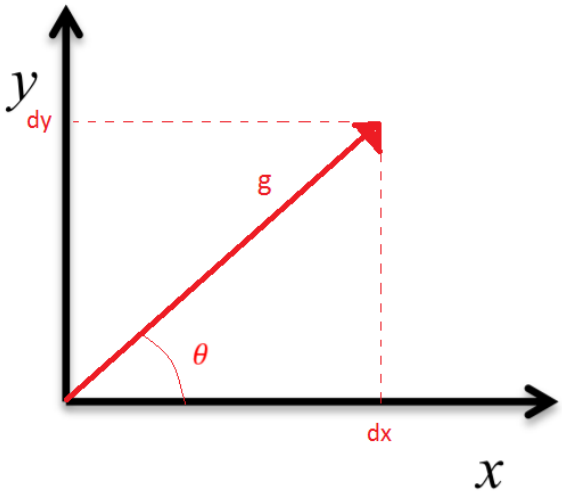


Figura 5-11: Representación de la magnitud y orientación del gradiente

5.4.2.2 Cálculo de los histogramas

Para poder obtener la información contenida en los gradientes calculados y calcular, posteriormente, el descriptor de la imagen, el método HOG almacena los gradientes en forma de histograma. Pero no solo se almacena un único histograma por cada imagen, sino que cada una se divide en celdas de tamaño fijo y se calcula un histograma por celda con los píxeles que quedan contenidos dentro. El tamaño de estas celdas suele ser de 6x6 píxeles. En la siguiente figura se puede ver esta representación.

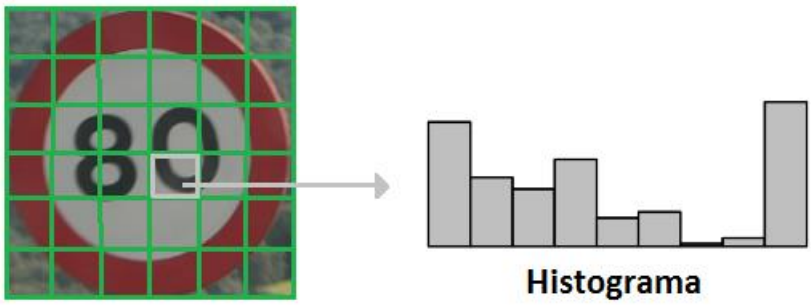


Figura 5-12: Ejemplo de histograma de una celda en una imagen

El campo que se tiene en cuenta para formar el histograma es el de dirección. El campo de dirección de un gradiente es un ángulo contenido entre los valores 0º y 360º, ó entre 0º y 180º si además se tiene en cuenta el sentido de la dirección. El primer paso para calcular el histograma es dividir este rango de valores en intervalos, de modo que cada gradiente se pueda clasificar dentro de alguno de ellos. Una forma típica de

dividir el rango de valores es con intervalos de 20º de manera que el rango 0º-180º queda dividido en 9 intervalos. Una vez fijados, el histograma se formará sumando los gradientes que se clasifican en cada uno de los intervalos.

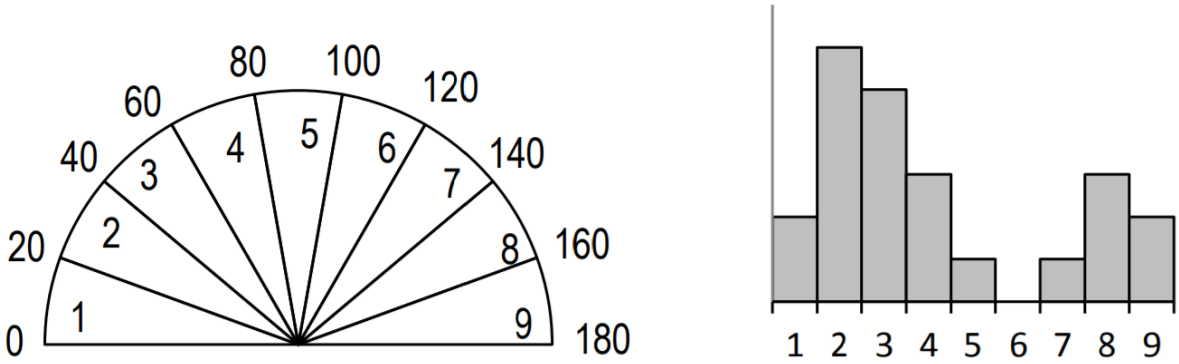


Figura 5-13: Ejemplo de cómo dividir los intervalos para un histograma

Matemáticamente se puede expresar lo anterior de la siguiente manera:

$$k = \sum_{(x,y)\in C} \omega_k(x,y) g(x,y) \quad (16)$$

$$\omega_k(x,y) = \begin{cases} 1 & si \quad (k-1)\delta\theta \le \theta(x,y) < k\delta\theta \\ 0 & en\ caso\ contrario \end{cases} \quad (17)$$

donde k es cada intervalo del histograma y $\omega_k(x,y)$ es la asociación de cada píxel con su intervalo.

5.4.2.3 Cálculo del descriptor HOG

Una vez obtenido el histograma de cada una de las celdas que divide la imagen, el siguiente paso es calcular el descriptor de la imagen. Este es un vector que representa a dicha imagen y su dimensión vendrá determinada por ciertos parámetros configurables.

El descriptor HOG se obtiene de la concatenación de los histogramas calculados. Se dice, por lo tanto, que una imagen queda representada por el conjunto de todos sus histogramas. Antes de realizar la concatenación, estos tienen que ser normalizados para disminuir efectos indeseados debido a agentes externos, como cambios en la iluminación de la imagen.

5.4.3 Clasificador de Máquinas de Vectores Soporte

Según como se define en [29], las máquinas de vectores soporte (o support vector machine, SVM) son un tipo de clasificador binario, cuya solución se basa en encontrar el margen máximo entre dos clases a partir de unos vectores determinados conocidos como vectores soporte.

SVM es un clasificador lineal, es decir, su solución es un hiperplano que permite dividir el espacio de características en dos regiones completamente disjuntas. Este hiperplano se calcula a partir de las muestras más cercanas de cada clase para, de esta forma, conseguir que el espacio libre de muestras a cada lado del hiperplano sea lo mayor posible. Para definir este hiperplano solo se consideran las muestras de entrenamiento de cada clase que quedan más cerca del hiperplano de separación. Estas muestras son las conocidas como vectores soporte.

5.4.3.1 SVM para clasificación binaria de muestras separables linealmente

El escenario más simple para un SVM es el caso en el que se tienen dos únicas clases separables linealmente y sin muestras ruidosas. En la siguiente figura vemos un ejemplo gráfico. Se ha supuesto que todos los vectores de dos clases diferentes pueden representarse en un espacio bidimensional en el que el hiperplano equivalente que separaría las dos clases de forma lineal sería una recta.

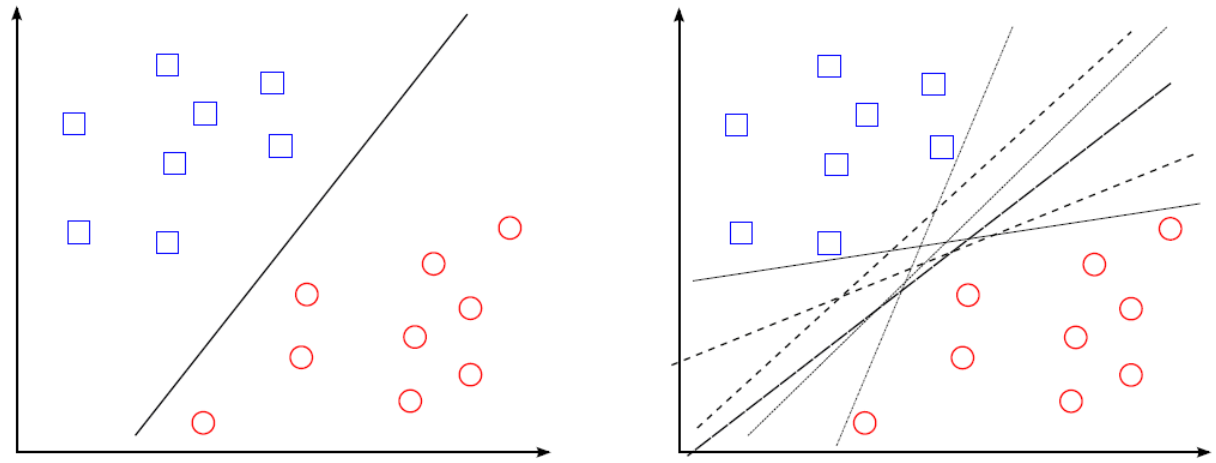


Figura 5-14: Hiperplanos de separación en un espacio bidimensional de un conjunto de muestras separables en dos clases: (izquierda) ejemplo de hiperplano de separación y (derecha) otros ejemplos de hiperplanos de separación, entre los infinitos posibles.

Dado un conjunto de muestras $S = \{(x_1 y_1), \dots, (x_d y_d)\}$, donde $x_i \in \mathbb{R}^d$ e $y_i \in \{+1, -1\}$, se puede definir un hiperplano de separación como una función lineal que es capaz de separar dicho conjunto sin error [32]:

$$D(x) = (\omega_1 x_1 + \cdots + \omega_d x_d) + b = < \omega, x > + b \quad (18)$$

donde $\omega \in \mathbb{R}^d$, $b \in \mathbb{R}$ y el operador $< x_1, x_2 >$ representa el producto escalar de los vectores x_1 y x_2. Para todo el conjunto de muestras x_i, el hiperplano cumplirá las siguientes restricciones:

$$\begin{aligned} < \omega, x_i > + b \geq 0 \quad si \quad y_i = +1 \\ < \omega, x_i > + b \leq 0 \quad si \quad y_i = -1, \\ i = 1, \dots, n \end{aligned} \quad (19)$$

o también:

$$y_i(< \omega, x_i > + b) - 1 \geq 0 \quad i = 1, \dots, n \quad (20)$$

o de forma más compacta:

$$y_i D(x_i) \geq 0, \quad i = 1, \dots, n \quad (21)$$

Como podemos imaginar, no existe un único hiperplano de separación para las muestras, sino que el conjunto de hiperplanos que cumple con las restricciones impuestas anteriormente es infinito. El objetivo de SVM es encontrar aquel hiperplano en el que el margen de separación entre muestras es máximo. Si definimos el margen, denotado por τ como la mínima distancia entre el hiperplano de separación y la

muestra más cercana, la solución óptima de SVM es aquel hiperplano cuyo margen será máximo. Podemos ver en la siguiente figura la solución óptima a unas muestras dadas que hay representadas.

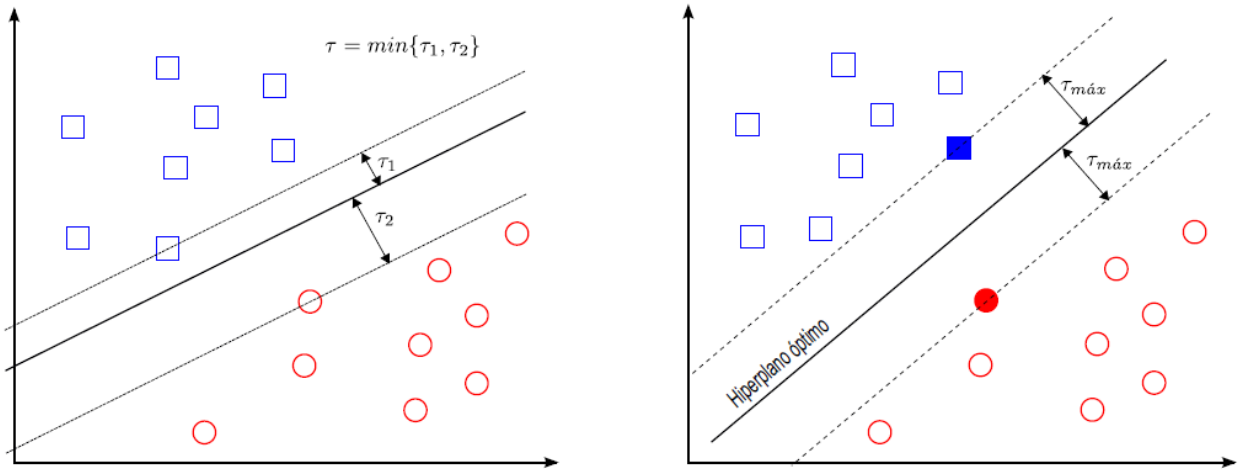


Figura 5-15: Representación en un ejemplo de la distancia entre el hiperplano hasta la muestra más cercana de cada clase (izquierda) y del hiperplano óptimo y de las distancias máximas a los vectores de soporte (derecha)

Como se indica en [32], la distancia entre un hiperplano de separación y una muestra x' viene dada por:

$$distancia\ (D(x), x') = \frac{|D(x')|}{||\omega||}, \tag{22}$$

siendo $|.|$ el comando operador absoluto, $||.||$ el operador norma de un vector y ω el vector que, junto con el parámetro b, define el hiperplano $D(x)$ y que, además, tiene la propiedad de ser perpendicular al hiperplano considerado. Por la definición de margen máximo, y por (21) y (22), todos los ejemplos de entrenamiento cumplirán que la distancia de cada uno de ellos al hiperplano de separación óptimo es mayor o igual que dicho margen, es decir:

$$\frac{y_i D(x_i)}{||\omega||} \geq \tau, \quad i = 1, \dots, n \tag{23}$$

Para el caso de los vectores soporte, que son las muestras situadas justo en la frontera que delimita el margen a cada lado del hiperplano de separación, se cumple por definición:

$$\frac{y_i D(x_i)}{||\omega||} = \tau, \quad \forall i \in V_s \tag{24}$$

donde V_s denota el conjunto de todos los vectores soporte.

Como se indica definitivamente en [32], la búsqueda del hiperplano óptimo para la clasificación binaria de ejemplos linealmente separables, se puede formalizar como el problema de encontrar los valores ω y b que minimizan el funcional $f(\omega) = ||\omega||$ sujeto a las restricciones dadas por (20) o, de forma equivalente[1]:

[1] Obsérvese que es equivalente minimizar $f(\omega) = ||\omega||$ o el funcional $1/2||\omega||^2$ propuesto en (20). El proceso de minimización de este nuevo funcional equivalente, en lugar del original, permitirá simplificar la notación posterior, obteniendo expresiones más compactas.

$$\begin{aligned} \min \quad & \frac{1}{2}\left|\left|\omega\right|\right|^2 \equiv \frac{1}{2} < \omega, \omega > \\ s.a. \quad & y_i(< \omega, x_i > +b) - 1 \geq 0, \quad i = 1, \dots, n \end{aligned} \tag{25}$$

Este problema de optimización con restricciones, como se indica en el documento anteriormente citado, corresponde a un problema de programación cuadrático y es abordable mediante la teoría de la optimización. Según la teoría de optimización, que se puede consultar en la referencia anterior y en [33], se establece que un problema de optimización, denominado primal, tiene una forma dual si la función a optimizar y las restricciones son funciones estrictamente convexas. En estas circunstancias, resolver el problema del dual permite obtener la solución del problema primal.

Se puede demostrar que el problema de optimización (25) es convexo y, por lo tanto, tiene un dual. A continuación, se resumen los pasos establecidos en [32] para transformar el problema primal en su dual.

En primer lugar, se construye la función lagrangiana [33]:

$$L(\omega, b, \alpha) = \frac{1}{2} < \omega, \omega > - \sum_{i=1}^{n} \alpha_i [y_i(< \omega, x_i > +b) - 1] \tag{26}$$

donde α_i son los denominados multiplicadores de Lagrange.

A continuación, se aplican las condiciones de Karush-Kuhn-Tucker(KKT):

$$\frac{\delta L(\omega, b, \alpha)}{\delta \omega} \equiv \omega - \sum_{i=1}^{n} \alpha_i y_i x_i = 0 \tag{27}$$

$$\frac{\delta L(\omega, b, \alpha)}{\delta b} \equiv \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{28}$$

$$\alpha_i [1 - y_i(< \omega, x_i > +b)] = 0, \qquad i = 1, \dots, n \tag{29}$$

Las restricciones (27) y (28) corresponden al resultado de aplicar la primera condición KKT y las expresadas en (29) al resultado de aplicar la denominada condición complementaria (segunda condición KKT). Concretamente, la restricción dada por (27) permite expresar ω en terminos de α_i:

$$\omega = \sum_{i=1}^{n} \alpha_i y_i x_i \tag{30}$$

La restricción (30) establece una restricción adicional para los coeficientes α_i:

$$\sum_{i=1}^{n} \alpha_i y_i = 0 \tag{31}$$

Y, finalmente, las dadas por (29), tal y como se verá más adelante, permitirán obtener el valor de b.

Para construir el problema dual, se hace uso de (30) para expresar la función lagrangiana únicamente mediante los α_i. Antes de ello, se puede reescribir (26) como:

$$L(\omega, b, \alpha) = \frac{1}{2} < \omega, \omega > - \sum_{i=1}^{n} \alpha_i y_i < \omega, x_i > - b \sum_{i=1}^{n} \alpha_i y_i + \sum_{i=1}^{n} \alpha_i \quad (32)$$

Teniendo en cuenta (31), el tercer sumando de la parte derecha de la función anterior es nulo, por lo que la sustitución de (30) en dicha función resulta ser:

$$L(\alpha) = \frac{1}{2}\left(\sum_{i=1}^{n} \alpha_i y_i x_i\right)\left(\sum_{j=1}^{n} \alpha_j y_j x_j\right) - \left(\sum_{i=1}^{n} \alpha_i y_i x_i\right)\left(\sum_{j=1}^{n} \alpha_j y_j x_j\right) + \sum_{i=1}^{n} \alpha_i$$

$$L(\alpha) = -\frac{1}{2}\left(\sum_{i=1}^{n} \alpha_i y_i x_i\right)\left(\sum_{j=1}^{n} \alpha_j y_j x_j\right) + \sum_{i=1}^{n} \alpha_i \quad (33)$$

$$L(\alpha) = + \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \alpha_i \alpha_j y_i y_j < x_i, x_j >$$

De esta forma se obtiene el problema dual, consistente en encontrar un α^* que maximice la función (33) sujeta a las restricciones dadas por (32) y aquellas otras asociadas a los multiplicadores de Lagrange ($\alpha_i \geq 0$). Esto es:

$$\begin{aligned} \max \quad & L(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j < x_i, x_j > \\ & s.a. \quad \sum_{i=1}^{n} \alpha_i y_i = 0 \\ & \alpha_i \geq 0, i = 1, \ldots, n \end{aligned} \quad (34)$$

Al igual que el problema primal, el problema dual es abordable mediante técnicas estándar de programación cuadrática, pero siendo más fácil de solucionar.

La solución del problema dual, α^*, permitirá obtener la solución del problema primal. Para ello, bastará sustituir dicha solución en la expresión (30) y, finalmente, sustituir el resultado en (18), es decir:

$$D(x) = \sum_{i=1}^{n} \alpha_i^* y_i < x, x_i > + b^* \quad (35)$$

Ahora necesitamos calcular b^* para obtener la definición completa del hiperplano. Analizando la formulación del problema dual, fijándonos en (35), podemos ver que ω viene definido por la combinación lineal de las muestras de entrenamiento, pero solo de aquellas en las cuales su multiplicador de Lagrange asociado cumple $\alpha_i > 0$. Son, por lo tanto, estas muestras las únicas denominadas vectores soporte. De esta afirmación se puede confirmar que el hiperplano de separación (35) se construye con una combinación lineal de los vectores soporte, ya que el resto de muestras del conjunto de entrenamiento tendrán asociado un $\alpha_j = 0$.

Para calcular b^* empleamos la siguiente ecuación:

$$b^* = y_{V_s} - < \omega^*, x_{V_s} > \quad (36)$$

donde V_s representa al conjunto de vectores soporte.

Se observa que tanto la definición del problema dual (34) como el hiperplano de separación óptimo (35) dependen del producto escalar de los vectores muestra. Esta propiedad se utiliza para calcular hiperplanos de separación óptimos en espacios transformados de alta dimensionalidad.

5.4.3.2 SVM para clasificación binaria de muestras que no son separables linealmente

En la mayoría de los casos prácticos las muestras de entrenamiento no podrán ser separadas linealmente como se ha visto hasta ahora, es decir, que no existirá un hiperplano definido como función lineal que separe las muestras de distintas clases.

Cuando tenemos un espacio muestral en el que las muestras de distintas clases no pueden ser separadas linealmente existe la posibilidad de hacer una transformación no lineal de este espacio y construir un clasificador lineal en este nuevo espacio transformado. Dicho de otra forma, una frontera lineal en el espacio transformado puede representar una frontera no lineal en el espacio original. Esto permitiría resolver el problema mediante las técnicas de resolución vistas hasta ahora. Para hacer esta transformación se utiliza la técnica del kernel, que permite proyectar un espacio de características en otro de mayor dimensión en el que las muestras sí pueden ser separadas de manera lineal.

Por lo tanto, la solución de un problema SVM en el que las muestras no son separables linealmente pasa por transformar, mediante una función de transformación, el espacio de muestras original a otro espacio de mayor dimensión con la esperanza de que en éste sus muestras sí sean linealmente separables. Esta función de transformación se suele expresar como $\phi(x)$ donde x es cada vector del espacio de muestras original.

En el nuevo contexto del espacio transformado, se demuestra [32] que la función de decisión del problema dual se obtiene transformando la expresión de la frontera de decisión en:

$$D(x) = \sum_{i=1}^{n} \alpha_i^* y_i K(x, x_i) \tag{37}$$

donde $K(x, x')$ se denomina *función Kernel*.

Una función Kernel se define como una función $k: \mathbb{x}\, x\, \mathbb{x} \rightarrow \mathbb{R}$ que asigna a cada par de elementos del espacio de entrada, $\mathbb{x}$, un valor real correspondiente al producto escalar de las imágenes de dichos elementos en un nuevo espacio $\mathcal{F}$ (espacio de características) [32], es decir,

$$K(x, x') = < \phi(x), \phi(x') > \tag{38}$$

donde $\phi: \mathbb{x} \rightarrow \mathcal{F}$ es la función de transformación del espacio original $\mathbb{x}$ al espacio transformado $\mathcal{F}$.

De esta manera, el problema a resolver pasa a ser ahora la búsqueda del valor de los parámetros $\alpha_i^*; i = 1, \dots, n$ que optimiza el problema dual expresado como [32]:

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j K(x_i, x_j) \\ s.a. \quad & \sum_{i=1}^{n} \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C, i = 1, \dots, n \end{aligned} \tag{39}$$

donde C es un parámetro de regularización del cual no existe una forma teórica de encontrar, pero se le suele asignar un valor grande. La metodología de solución a un problema SVM de clasificación de muestras no separables linealmente pasar por calcular los los parámetros α_i^* conocidas las muestras de entrenamiento, el kernel K y el parámetro de regularización C.

5.4.3.3 SVM para la clasificación multiclase

Cuando se aplica el problema SVM en un caso multiclase se tienen en cuenta dos cosas básicas [34]:

- Cada categoría debe ser dividida en otras y todas ellas combinadas entre si.
- Se construyen $k(k-1)/2$ modelos, donde k es el número de clases.

Luego, cuando tenemos un problema SVM de clasificación multiclase, pasamos de tener un problema binario a varios problemas binarios. Para clasificar una muestra, esta tendrá que ser evaluada en cada modelo de clasificación tratado y será clasificada entre dos clases en cada uno.

5.4.4 Etapa de reconocimiento en otros trabajos de detección de señales de tráfico

Existen multitud de estudios sobre la implementación de detectores de señales de tráfico y casi todos tienen su propio sistema diferente. En cuanto a la etapa de reconocimiento, existen muchas posibilidades, pero casi todas están basadas en los estudios de las características locales de la imagen. A continuación, se presentan algunos métodos empleados y trabajos:

- El descriptor HOG es uno de los métodos más empleados para la detección de objetos desde que, en el año 2005, fuera presentado un detector de peatones basado en este método. En [35] se tiene un ejemplo donde se emplea este descriptor junto con SVM como clasificador.
- El descriptor SIFT es también un método bastante empleado en aplicaciones de reconocimiento de objetos, podemos verlo aplicado en [36].
- Las redes neuronales son otro ejemplo muy empleado en la etapa de reconocimiento. Podemos ver cómo se aplican en [24], [27], y [37].
- Otro método también empleado es el de *template matching*. *Template matching* consiste en comparar las imágenes que se quieren clasificar con unas plantillas almacenadas en memoria. Finalmente, se obtiene la clase de la plantilla a la que más se asemeja la imagen. Podemos ver los ejemplos de [14], [23] y [38].

5.5 Aceleración hardware en el diseño de detectores de señales de tráfico

Los sistemas de detección de señales de tráfico son sistemas complejos con alto coste computacional que tardan mucho tiempo en ejecutarse. Sin embargo, estos sistemas deben de cumplir (en la mayoría de los casos) con un requisito fundamental, que es ser un sistema de tiempo real. Normalmente, la etapa con mayor coste computacional suele ser la de reconocimiento. Aplicar una etapa de detección previa a la etapa de reconocimiento hace que esta última requiera de mucho menos trabajo, pero los algoritmos de preprocesamiento de la imagen y de detección también tienen un alto coste computacional. La ventaja de estos algoritmos es que son buenos candidatos para ser implementados en hardware ya que aplican procesos muy repetitivos que pueden ser ejecutados en paralelo. Es por esto que, para mejorar el rendimiento de estos sistemas, suelan ser implementados en arquitecturas heterogéneas con lógica programable, en dispositivos DSPs o GPUs. A continuación, se analiza el rendimiento (en el sentido de la velocidad de procesamiento del sistema) de algunas de las publicaciones estudiadas:

- En [14] un sistema de detección de señales de tráfico es implementado y sintetizado completamente en una FPGA Virtex-5 de la compañía Xilinx. El rendimiento conseguido para el sistema completo es de una frecuencia de 14.7 frames por segundo y un acierto del 91% de las señales. El diseño completo es desarrollado con lenguaje HDL. En la publicación se incluye una tabla comparativa de rendimiento de otros trabajos implementados en diferentes arquitecturas hardware.
- En [16] se ponen a prueba dos versiones de un sistema de detección de señales de tráfico en los que la diferencia entre las dos está en la etapa de reconocimiento. En la primera versión, el reconocimiento de las señales se lleva a cabo mediante una técnica de template matching basada en el cálculo de la distancia entre la señal candidata y cada una de las plantillas mediante el método de Hausdorff. En la segunda versión, se emplea un detector de características tipo SURF y un

clasificador basado en el método del vecino más cercano. La primera versión es implementada en dos dispositivos diferentes: una FPGA Virtex-5 de Xilinx y un SoC Zynq-7020 de una Zedboard. En ellos se obtiene un rendimiento de 1.3 frames/s en la FPGA y 10.3 frames/s en el SoC. La segunda versión es implementada solo en el SoC de la Zedboard, en el que se obtiene un rendimiento de 1 frame/s. La segunda versión toma la ventaja de ser más robusta que con el método de template matching, pero no consigue un rendimiento suficiente para ser un sistema de tiempo real. El diseño también es desarrollado con lenguaje HDL.

- En [39] se propone un complejo algoritmo basado en template matching para el reconocimiento de señales circulares en entornos de noche o de condiciones meteorológicas adversas. El sistema es implementado sobre un SoC Zynq-7020 y el rendimiento estimado es de 83 frames/s (es estimado porque solo está desarrollada la etapa de detección). Esta publicación también incluye una tabla comparativa de rendimiento de otros trabajos con otras arquitecturas hardware.

Según la mayoría de estudios analizados, para poder lograr un alto rendimiento en velocidad resulta necesario la aceleración de funciones en hardware.

6 ESTRUCTURA DEL DETECTOR DE SEÑALES DE TRÁFICO DESARROLLADO Y RESULTADOS EXPERIMENTALES

6.3 Esquema general del detector de señales de tráfico

Como ya se ha visto en el capítulo 5, un sistema de detección de señales de tráfico está compuesto por dos etapas fundamentales: la etapa de detección y la de reconocimiento. A continuación, se explicará en detalle qué se ha realizado en cada etapa del detector desarrollado en este proyecto:

- Etapa de detección: la etapa de detección determina qué regiones de la imagen tienen alta probabilidad de contener una señal de tráfico. El objetivo es reducir la inmensa cantidad de información que suelen contener las imágenes de entornos normales de una señal. A aquellas regiones que puedan contener una señal de tráfico y que han sido detectadas por esta etapa son conocidas como regiones de interés, del inglés regions of interest (RoIs). Para el cometido de esta etapa se han empleado dos técnicas vistas en el capítulo 5: segmentación por color y etiquetado de componentes conectados (CCL).

  Esta etapa tiene un alto coste computacional, pero es muy sintetizable en dispositivos de computación en paralelo. Es por ello que esta etapa queda implementada sobre la lógica programable del sistema.

- Etapa de reconocimiento: La etapa de reconocimiento se encarga de identificar las señales de tráfico contenidas en las RoIs detectadas en la etapa de detección. Para ello, intenta clasificar cada RoI como una señal de tráfico utilizando las características locales de la imagen. El sistema completo de reconocimiento queda compuesto por la técnica HOG+SVM. HOG como técnica para extraer las características locales de la imagen y SVM como clasificador, ambos explicados en el capítulo anterior.

El sistema completo se ha desarrollado sobre una placa Zedboard como la que se muestra en el capítulo 2, la cual contiene un chip de la familia Zynq-7000. Para la elaboración del sistema, se han empleado las herramientas de desarrollo de Xilinx SDSoC y Vivado HLS, presentadas en el capítulo 3. La etapa de detección queda implementada sobre la lógica programable del dispositivo, mientras que la etapa de reconocimiento es ejecutada por el microprocesador. La parte sintetizable sobre la PL ha sido desarrollada mediante lenguaje de alto nivel y aplicando las técnicas de optimización explicadas en el capítulo 4.

Para comprobar el funcionamiento del sistema, se han empleado una colección extensa de imágenes con contextos de carreteras y urbanizaciones en las que hay contenidas señales de tráfico. Las imágenes son leídas de la memoria de una manera secuencial, igual que si se estuvieran leyendo imágenes de una cámara.

6.3.1 Etapa de detección

La etapa de detección recibe las imágenes de la memoria y devuelve, para cada una de ellas, una tabla con las coordenadas donde están contenidas las RoIs para que puedan ser analizadas en la etapa de reconocimiento. La etapa de segmentación queda subdividida en dos: etapa de segmentación por color y etapa de etiquetado de componentes conectados (CCL).

6.3.1.1 Segmentación por color

La etapa de segmentación por color distingue a todos aquellos píxeles de la imagen que contengan el color de una señal de tráfico. Para el detector desarrollado, se distinguen los colores azul y rojo. El algoritmo de segmentación por color recorre los píxeles de la imagen uno a uno cambiándolos de valor según su color. Los píxeles que contengan el color rojo o el azul pasan a tener el valor 1, y los que contengan cualquier otro color pasan a valer 0. De esta manera, el resultado de la segmentación por color es una imagen binaria.

Figura 6-1: Ejemplo de salida de la etapa de segmentación

El espacio de color utilizado en las imágenes es el RGB y la fórmula para determinar si el píxel es rojo o

azul es el que se muestra en el capítulo 5 de esta memoria. El motivo de utilizar este espacio de color y no otro es que éste es el que resulta más restrictivo. Con otro espacio de color, como por ejemplo HSI, habría demasiadas regiones de la imagen que superen el umbral de segmentación, lo que satura de demasiado trabajo a la etapa de reconocimiento y aumenta los casos de falsos positivos.

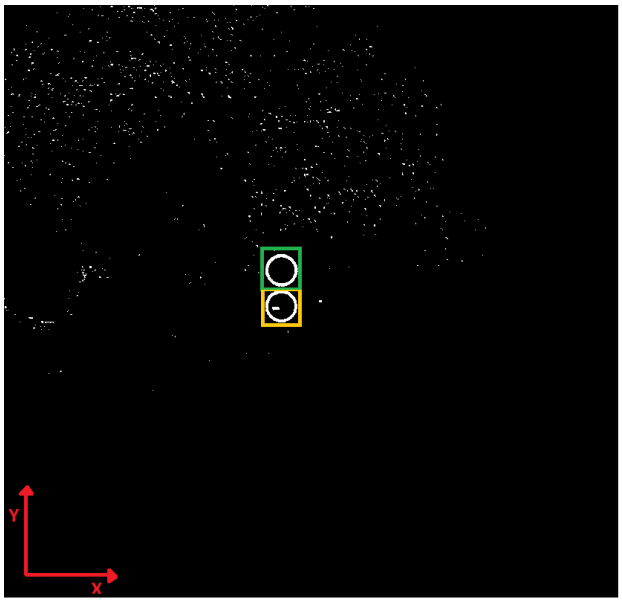
Esta es la primera parte implementada sobre la FPGA del dispositivo Zynq. Para su elaboración, se ha hecho uso de la librería HLS_VIDEO de Xilinx.

6.3.1.2 Etiquetado de componentes conectados

Una vez que se obtiene la imagen binaria de la etapa de segmentación por color, el siguiente paso por la etapa de detección es obtener las coordenadas de aquellos objetos que haya contenidos en la imagen. El algoritmo de CCL recorre toda la imagen buscando conjuntos de píxeles que sean conexos para asignarles una etiqueta a cada uno. El algoritmo que se ha desarrollado en este proyecto es el que se estudia y se elabora en [15], donde se pueden consultar todos los detalles del algoritmo.

Existe una diferencia importante en la forma en que se ha desarrollado este algoritmo en este trabajo con respecto a la referencia [15]. En ambos casos, el algoritmo es implementado sobre una FPGA. La diferencia está en que en el caso de la referencia el algoritmo es desarrollado con lenguaje VHDL, mientras que en este trabajo es desarrollado con lenguaje C++. El resultado de esta diferencia es, como ya se supondrá, que el diseño codificado con lenguaje VHDL obtiene mejor rendimiento que con C++, a cambio de que con VHDL el desarrollo es más costoso. Cabe indicar que, debido a la complejidad del algoritmo de CCL, esta fue la parte más difícil de codificar.

En la siguiente figura vemos un ejemplo de entrada de esta etapa y su salida. La información proveniente de la etapa anterior es una imagen binaria donde los píxeles que sean de color rojo o azul toman el valor 1 y los demás 0. A la salida, se obtiene una tabla con las coordenadas de aquellos objetos iluminados de la imagen que cumplen los siguientes requisitos: ser mayores de 10x10 píxeles y que el ancho no sea mayor que dos veces el alto o viceversa. Esto permite obviar aquellos objetos que son demasiado pequeños o que no tienen una proporcion lógica para ser una señal.



ROIs	Xi	Yi	Xf	Yf
ROI 1	880	330	931	384
ROI 2	881	386	932	432

Figura 6-2: Ejemplo de salida de la etapa de CCL

6.3.1.3 Aceleración de la etapa de detección sobre la FPGA del dispositivo

Como ya se ha dicho, la etapa de detección ha sido implementada sobre la FPGA del dispositivo. La razón es que, por la naturaleza de los algoritmos de la etapa de detección, se puede mejorar sensiblemente el rendimiento del sistema aprovechando el paralelismo que ofrece la PL.

Como ya se explicó en el capítulo 3, SDSoC permite desarrollar el sistema completo HW/SW de un co-diseño. Todo el sistema completo es elaborado en C++, por lo que SDSoC es el encargado de llamar a la herramienta Vivado HLS para realizar la síntesis de aquellas funciones que deban ser implementadas sobre la PL.

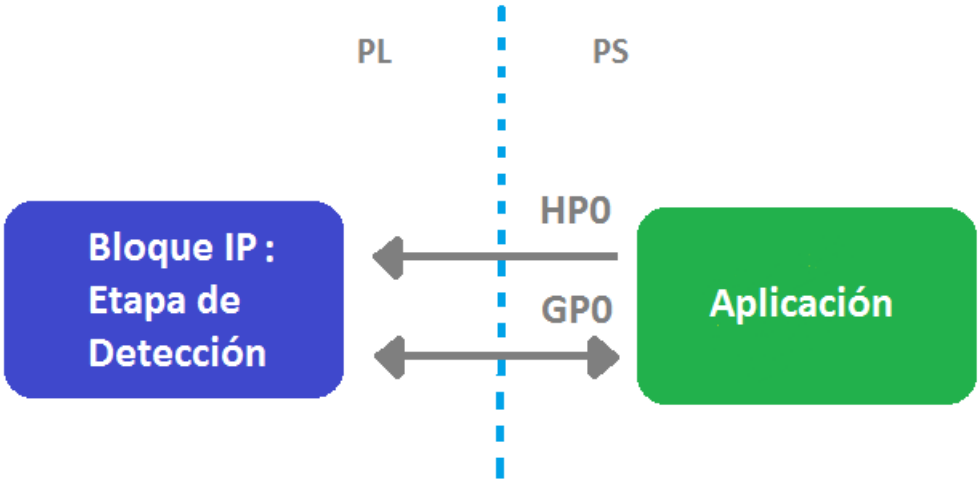


Figura 6-3: Interfaces de comunicación PS-PL

Como parte del sistema es implementado sobre PS y parte sobre PL, es necesario una conexión entre ambas partes para la transferencia de datos. SDSoC es el encargado de elegir de manera automática cuáles serán las interfaces adecuadas para las transacciones PS-PL, pero hay que indicarle al compilador las propiedades de cada conexión mediante directivas (como ya se vio en el capítulo 4). Las interfaces de transferencia que se han generado para este proyecto son:

- Un **puerto de alto rendimiento (HP)** para la transferencia de imágenes desde la aplicación hacia la etapa de detección. Este puerto está preparado para la transferencia de grandes cantidades de datos (puede verse en el capítulo 2 de este documento).

  Para la transferencia de imágenes a través de este puerto, SDSoC genera de manera automática un *data mover* del tipo **axi_dma_simple,** que es el más eficiente para la transferencia de arrays. Este *data mover* permite que la transferencia sea a través de una interfaz DMA, es decir, que el bloque IP puede acceder a memoria sin la intervención del microprocesador. Para dar al compilador las características de esta conexión se tienen que realizar los dos siguientes pasos:

  - Primero, cada imagen que se transfiere desde la aplicación hacia la etapa de detección debe ser almacenada en memoria continua mediante la función *sds_alloc* de la librería *sds_lib.* Además, es necesario indicar al compilador que las imágenes han sido almacenadas de esta manera, para ello se emplea la directiva:

    #pragma SDS data mem_attribute (A: PHYSICAL_CONTIGUOUS).

  - Después, debemos indicar al compilador que la transferencia será secuencial a través de la directiva:

    #pragma SDS data access_pattern (A: SEQUENTIAL).

- Una interfaz **de propósito general (GP)** que se utiliza para dos transferencias distintas. Por un lado, transferir el valor de la variable *color* desde la aplicación hasta la etapa de detección. Esta variable sirve para indicar a la etapa de detección si debe detectar los píxeles que son rojos o los que son azules. Por otro lado, transferir la tabla de coordenadas con las RoIs detectadas desde la lógica programable hacia la aplicación software. Para la variable *color,* se ha generado un *data mover* tipo **axi_lite**, apropiado para la transferencia de una única variable. Para el caso de la tabla de RoIs, un *data mover* del tipo **axi_fifo**.

  La tabla de coordenadas no se puede considerar como transmitida de forma secuencial al igual que con el streaming de imágenes, sino que queda determinada como transmitida mediante un patrón

de tipo aleatorio. Esto se debe a que la tabla de coordenadas no tiene un tamaño fijo, sino que depende de cuantas RoIs contiene cada imagen. En una transmisión secuencial tipo streaming, el tamaño de los datos a transmitir debe ser fijo en cada aplicación. La otra opción sería una transferencia de tipo aleatorio. Para indicar al compilador que la transferencia es de tipo aleatorio se emplea la directiva:

#pragma SDS data access_pattern (A: RANDOM).

En la siguiente tabla se presentan, de manera resumida, las interfaces que se han empleado en este proyecto:

IP puerto	Dirección	Data mover	Conexión
video_in	IN	axi_dma_simple	HP0
RoIs	OUT	axi_lite	GP0
color	IN	axi_lite	GP0

Tabla 3: Resumen de las interfaces empleadas

6.3.2 Etapa de reconocimiento

La etapa de reconocimiento obtiene como datos de entrada las tablas de coordenadas de la imagen donde están contenidas las RoIs, es decir, las zonas con probabilidad de contener una señal de tráfico. El objetivo de esta etapa es analizar estas RoIs para comprobar si existen señales en ellas e identificarlas. Una vez analizadas todas las RoIs de una imagen, la etapa de reconocimiento devuelve las señales de tráfico identificadas.

La técnica utilizada para analizar las imágenes es la del análisis de sus características locales. Como descriptor de la imagen se ha empleado el método HOG. El método HOG se basa en la información del gradiente para calcular un descriptor. Una vez obtenido el descriptor de cada una, será clasificado mediante un clasificador del tipo SVM. El clasificador utilizado en este caso es un clasificador multiclase lineal. La elección de un SVM lineal no tiene por qué ser la mejor opción. No será competencia de este trabajo encontrar las características del SVM óptimo.

Para la implementación del método HOG y del clasificador SVM se ha hecho uso de las funciones de librería OpenCV.

6.3.2.1 Entrenamiento del SVM

Para poder clasificar imágenes con el SVM, previamente es necesario entrenarlo con una base de datos. La base de datos que se ha empleado para el entrenamiento es la que se puede descargar desde [40]. Esta base de datos contiene más de 40 tipos de señales de tráfico y un total de más de 50000 imágenes para entrenar a cualquier clasificador. Todas estas imágenes, que corresponden a imágenes de señales de tráfico recortadas, están perfectamente clasificadas, separadas cada clase en una carpeta distinta. Junto con todas estas imágenes, viene incluido un documento con el nombre de todas las imágenes contenidas en la base de datos y su identificador, lo que la hace mucho más fácil de utilizar.

De los 40 tipos que se proporcionan, solo se entrenará al clasificador con 18, de las cuales 14 son rojas y 4 azules. En la siguiente tabla se tienen las señales de tráfico que podrá detectar el sistema:

Tabla 4: Señales de tráfico que pueden ser detectadas

Se han calculado dos modelos de SVM entrenados distintos, uno para las imágenes rojas y otro para las

imágenes azules. Esto mejora la eficiencia, ya que disminuye la cantidad de errores de clasificación y aumenta el tiempo de clasificación de cada imagen.

6.3.3 Esquema del sistema completo

Podemos ver un esquema del sistema completo del detector de señales en la figura de abajo.

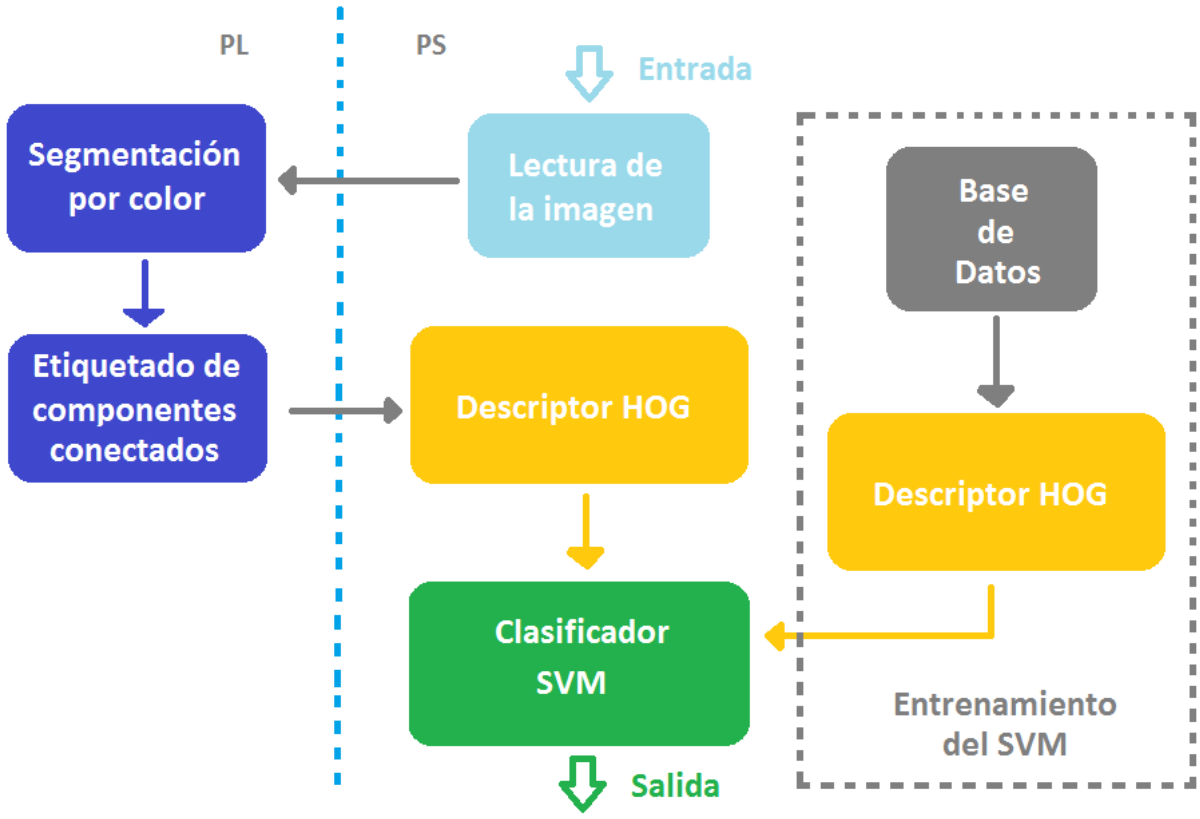


Figura 6-4: Esquema completo del detector de señales de tráfico

Como se puede observar, tanto la etapa de segmentación por color como la de CCL, ambas partes de la etapa de detección, quedan implementadas sobre la lógica programable del sistema.

6.4 Resultados

Para estudiar el rendimiento del sistema se han empleado un total de 900 imágenes de carreteras y de ciudad, de las cuales solo algunas contienen señales de tráfico. Las imágenes son leídas una a una desde la memoria y son procesadas por el sistema para poder detectar las señales de tráfico. El objetivo del experimento es determinar qué porcentaje de las señales contenidas en las imágenes de test pueden ser reconocidas y qué rendimiento es obtenido (teniendo en cuenta que parte del sistema es implementado sobre la lógica programable).

6.4.1 Eficiencia del sistema

Tras el experimento, los resultados que se han obtenido en cuanto a efectividad de detección de señales de tráfico son los que se muestran en la tabla siguiente.

Señal	Total	Detectadas	Reconocidas	% reconocidas	% ef detección	% ef reconocimiento
	79	55	52	65,82	69,62	94,55
	41	29	24	58,53	70,31	82,76
	41	26	24	58,53	63,41	92,31
	38	28	26	68,42	73,68	92,86
	83	40	40	48,19	48,19	100
	32	24	15	46,88	75	62,5
	8	6	5	62,5	75	83,33
	38	19	17	44,73	50	89,47
	13	7	6	46,15	53,84	85,71
	20	10	10	50	50	100
	31	19	17	62,5	61,39	89,71
	18	9	8	44,44	50	88,89
	14	7	5	35,71	50	71,43
	5	3	3	60	60	100
	16	16	14	87,5	100	87,5
	20	16	14	70	80	87,5
	88	69	69	78,41	78,41	100
	6	5	4	66,67	83,33	80

Tabla 5: Resultado para todas las señales tras el experimento con las imágenes de test

En esta tabla se representan los resultados obtenidos para cada señal de tráfico que el sistema puede detectar. Teniendo en cuenta que el número total de señales de tráfico contenidas en las 900 imágenes de test es conocido, se ha procedido a calcular qué tanto por ciento de las imágenes han sido reconocidas. En la columna de *% ef detección* podemos ver la efectividad de la etapa de detección acelerada sobre la FPGA y en la columna *% ef reconocimiento* tenemos la efectividad de la etapa de reconocimiento, es decir, el tanto por ciento de imágenes correctamente reconocidas de las que previamente han sido detectadas.

En la tabla 6 se tiene la media calculada para todas las señales en conjunto.

	Número de señales	Detectadas	Reconocidas	% reconocidas	% ef detección	% ef reconocimiento
Total	591	388	353	59,73	65,65	90,98

Tabla 6: Resultado total tras el experimento con las imágenes de test.

Observándose la tabla de resultados, se podría pensar rápidamente que la efectividad del sistema en el reconocimiento de señales de tráfico no es buena. Si bien es cierto que el porcentaje obtenido no es alto, hay que tener varias cosas en cuenta antes de tomar un juicio.

- En primer lugar, cabe decirse que del número total de señales de tráfico que hay contenidas en las imagenes, muchas de ellas aparecen claramente perjudicadas por consecuencias del entorno, como pueden ser una mala luminosidad u oclusiones por otros objetos. Podemos ver algún ejemplo:

Figura 6-5: Ejemplo de señales con poca luminosidad, indetectables por la etapa de detección

- En segundo lugar, hay que tener en cuenta que cada imagen del conjunto para test está tomada de un paisaje distinto. Esto quiere decir que cada imagen solo tiene una oportunidad de ser reconocida. En el caso de utilizarse este sistema con una cámara de video, se obtendrían muchas imágenes del mismo paisaje, por lo que, si una imagen no ha obtenido la calidad suficiente para poder detectar la señal, es probable que sí lo haga en la siguiente. Se tendrían varias oportunidades de detectar la misma señal y, por lo tanto, multiplica la probabilidad de éxito.

Hay que indicarse también que, para este trabajo, no se han tenido en cuenta los falsos positivos. Los falsos positivos son aquellas RoIs que han pasado la etapa de detección sin contener una señal de tráfico pero que han sido clasificadas como tal. El motivo es que el clasificador creado para este trabajo debería contener una clase excluyente, es decir, una clase a la que deberían ser asignadas todas aquellas señales que no contienen una señal de tráfico.

6.4.2 Aceleración del sistema

Otro de los aspectos importantes que hay que medir es la mejora en cuanto a velocidad de ejecución de la etapa de detección cuando ésta es acelerada. Gracias a las funciones de reloj definidas en la librería *sds_lib* podemos determinar que la etapa de detección, cuando esta es acelerada, tan solo necesita 0,04 segundos en extraer las RoIs de una imagen. Por otro lado, la misma etapa, pero ejecutada por el microprocesador, necesita un total de 1,42s para la misma acción. Todo esto medido durante su ejecución sobre la Zedboard. Podemos notar una mejoría notable en cuanto a velocidad, hasta 71 veces más rápido cuando se aprovecha el paralelismo de la FPGA. En total, teniendo en cuenta también la etapa de reconocimiento, obtendríamos una tasa de 16,7 imágenes procesadas por segundo. Podemos ver de forma resumida estos resultados en la siguiente tabla.

	Etapa de detección	Etapa de reconocimiento	Tiempo/frame	Frames/segundo[2]
Etapa de detección no acelerada	1,42s	0,02s	1,44s	0,69 f/s
Etapa de detección acelerada	0,04s	0,02s	0,06s	16,7 f/s

Tabla 7: Rendimiento obtenido cuando se acelera la etapa de detección

Un dato importante a tener en cuenta es que las imágenes de entrada a la etapa de detección tienen un tamaño de 1280x720 píxeles. Resulta evidente que cuanto mayor sea el tamaño de cada frame, más tiempo se requerirá para procesarlo. Este rendimiento podría ser mayor si el mismo algoritmo hubiera sido desarrollado con lenguaje VHDL.

[2] Este resultado se obtiene teniendo en cuenta la latencia de la etapa de detección más la de reconocimiento. No se tiene en cuenta el tiempo requerido en leer cada imagen de la memoria mediante las funciones de OpenCV y almacenar dicha imagen en memoria continua. Estas dos tareas no serían necesarias en el caso de utilizar una cámara.

7 CONCLUSIONES Y TRABAJOS FUTUROS

7.3 Conclusiones

- SDSoC fue creado para facilitar el desarrollo de diseños HW/SW sin la necesidad de tener grandes conocimientos sobre diseño HW. A pesar de esto, he comprobado que sigue siendo muy necesario tener unos conocimientos básicos de la arquitectura hardware del dispositivo.
- A la hora de desarrollar un detector de señales de tráfico, el principal obstáculo para obtener un buen rendimiento suele estar en la etapa de detección, ya que suele contener algoritmos de alta carga computacional. Para mejorar este, resulta clave acelerar esta etapa implementándola sobre dispositivos de computación en paralelo. La mejora en cuanto a latencia de la etapa de detección cuando es implementada sobre la FPGA ha sido notable en este proyecto, hasta 71 veces más rápido en la ejecución del algoritmo. Se hubiera obtenido mejor rendimiento si el mismo algoritmo se hubiera desarrollado con algún lenguaje de descripción de hardware.
- Como consecuencia directa de acelerar algoritmos en dispositivos heterogéneos, se tiene que también es necesario implementar una comunicación eficiente entre las distintas partes del dispositivo para que el rendimiento siga siendo alto. Se ha aprendido que tener un buen manejo de las interfaces de interconexión y de los accesos a memoria externa son claves para que un sistema funcione de manera óptima.
- Los sistemas heterogéneos, como es el dispositivo Zynq-7000 utilizado en este proyecto, demuestran tener una gran capacidad de versatilidad y de flexibilidad pudiendo hacer uso de las mejores características de los sistemas de procesamiento y de la lógica programable.
- Se ha demostrado en el desarrollo de este trabajo que es posible obtener muy buena eficiencia en los clasificadores de características locales como el SVM que se ha aplicado. Por otro lado, resulta más compleja la etapa de detección, la cual es muy sensible a agentes externos.

7.4 Trabajos futuros

Entre las posibles mejoras y ampliaciones para este trabajo se podría considerar:

- Mejorar la eficiencia obtenida por la etapa de detección aplicando técnicas más complejas. Se ha demostrado en otros trabajos que el uso de otros espacios de color como el HSI puede mejorar el rendimiento cuando las condiciones de luz son adversas. Además, es posible complementar la segmentación por color con la segmentación por la forma aplicando técnicas complejas como *pattern matching* o empleando clasificadores previamente entrenados para encontrar formas

geométricas.

- Un trabajo futuro y necesario sería entrenar al clasificador SVM con una clase excluyente, es decir, una clase a la que quedarán asignadas todas las RoIs que han pasado con éxito la etapa de detección pero que en realidad no contienen ninguna señal de tráfico. Para ello, sería necesario recopilar una gran base de datos de imágenes que contengan fondo, esto es, cualquier cosa que no sea en este caso una señal de tráfico y entrenar al clasificador con esta nueva clase.
- Se desearía encontrar el clasificador SVM óptimo para el caso de este proyecto. Para ello, la estrategia a seguir sería la de tratar el problema de encontrar el hiperplano de separación como un problema no lineal y encontrar el kernel con los parámetros más eficientes.

REFERENCIAS

[1] S. Pillath, «Automated vehicles in the EU,» Enero 2016. [En línea]. Available: http://www.europarl.europa.eu/RegData/etudes/BRIE/2016/573902/EPRS_BRI(2016)573902_EN.pdf. [Último acceso: 30 Mayo 2017].

[2] Xilinx, «Zynq-7000 All Programmable SoC: Technical Reference Manual (UG 585,» 27 Septiembre 2016. [En línea]. Available: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf. [Último acceso: 25 Junio 2017].

[3] L. H. Crockett, «The Zynq Book: Embedded Processing with th ARM Cortex - A9 on the Xilinx Zynq-7000 All Programmable SoC,» Strathclyde Academic Media, 2014, pp. 202-211.

[4] Avnet, «Zedboard,» [En línea]. Available: Zedboard.org. [Último acceso: 03 Junio 2017].

[5] G. Sutter, «Síntesis de Alto Nivel para FPGAs con Vivado-HLS: Como describir HW desde C/C++,» 25 Mayo 2016. [En línea]. Available: https://www.youtube.com/watch?v=j8ijuzNC02g&t=1404s. [Último acceso: 25 Junio 2017].

[6] Xilinx, «Vivado Design Suite User Guide: High-Level Synthesis (UG 902),» 2016.

[7] L. H. Crockett, R. A.Elliot, M. A. Enderwitz y R. W.Stewart, de *Embedded Processing with the ARM Cortex - A9 on the Xilinx Zynq-7000 All Programmable SoC*, 2014, pp. 281-331.

[8] Xilinx, «SDSoC Environment User Guide,» 2016.

[9] OpenCV, «OpenCV,» [En línea]. Available: http://opencv.org/.

[10] S. Neuendorffer y a. D. W. Thomas Li, «Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries,» 2016.

[11] Xilinx, «SDSoC Environment Optimization Guide,» 2016.

[12] F. Vallina Martinez, «Implementing Memory Structures for Video Processing in the Vivado HLS Tool,» 2012.

[13] E. Calvo, P. Brox y Sanchez-Solano, «Un algoritmo en tiempo real para etiquetado de componentes conectados en imágenes,» Marzo 2012. [En línea]. Available: https://idus.us.es/xmlui/handle/11441/56394. [Último acceso: 20 Junio 2017].

[14] R. Hmida, «Hardware implementation and validation of a traffic road sign detection,» 2016. [En línea]. Available: https://www.researchgate.net/publication/299432433_Hardware_implementation_and_validation_of_a_traffic_road_sign_detection_and_identification_system. [Último acceso: 5 Junio 2017].

[15] N. A. Dobernack, «Implementación de un sistema de detección de señales de tráfico mediante

visión artificial basado en FPGA,» Abril, pp. 151-173.

[16] K. V. E. V. S. P. a. E. O. Yan Han, «Hardware/Software Co-Design of a Traffic Sign Recognition System Using Zynq FPGAs,» Marzo 2016. [En línea]. Available: http://www.mdpi.com/2079-9292/4/4/1062/pdf. [Último acceso: 05 Junio 2017].

[17] D. G. Bailey, Design for Embedded Image Processing on FPGAs, Wiley-Blackwell, 2011.

[18] A. V. K.N. Plataniotis, «Color Image Processing and Applications,» 18 February 2000. [En línea]. Available: http://www.comm.toronto.edu/~kostas/Publications2008/pub/bookchapters/2000-SpringerMonograph.pdf. [Último acceso: 07 Junio 2017].

[19] Wikipedia, «Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Espacio_de_color. [Último acceso: 2 Junio 2017].

[20] Wikipedia, «Wikipedia,» 17 Junio 2017. [En línea]. Available: https://es.wikipedia.org/wiki/YCbCr.

[21] R. Timofte, K. Zimmermann y L. V. Gool, «Multi-view traffic sign detection, recognition, and 3D localisation,» 2014.

[22] V. A. Prisacariu, R. Timofte, K. Zimmermann, I. Reid y L. V. Gool, «Integrating Object Detection with 3D Tracking Towards a Better Driver Assistance System,» 2010.

[23] S. F. Matthew Russell, «OpenCV based road sign recognition on Zynq,» Julio 2013. [En línea]. Available: https://www.researchgate.net/publication/261147904_OpenCV_based_road_sign_recognition_on_Zynq.

[24] A. Salhi, B. Minaoui y M. fakir, «Robust Automatic Traffic Signs Recognition Using Fast Polygonal Approximation of Digital Curves and Neural Network,» 2014.

[25] B. Saturnino Maldonado, A. Sergio Lafuente, J. Pedro Gil, M. Hilario Gomez y F. Francisco Lopez, «Road-Sign Detection and Recognition Based on Support Vector Machines,» 2007.

[26] A. Mogelmose, M. M. Trivedi y T. B. Moeslund, «Vision-Based Traffic Sign Detection and Analysis for Intelligent Driver Assistance Systems: Perspectives and Survey,» 2012.

[27] b. Alberto, P. Cerri, P. Medici y P. P. Porta, «Real Time Road Signs Recognition,» 2007.

[28] E. Calvo y S. S.-S. Piedad Brox, «Un algoritmo en tiempo real para etiquetado de componentes conectados en imágenes,» 2012.

[29] E. Valveny, J. G. Sabaté y R. B. Caselles, «Coursera: Clasificación de imágenes: ¿cómo reconocer el contenido de una imagen?,» [En línea]. Available: https://www.coursera.org/learn/clasificacion-imagenes/.

[30] OpenCV, «Docs OpenCV: Introduction to SIFT (Scale-Invariant Feature Transform),» [En línea]. Available: http://docs.opencv.org/trunk/da/df5/tutorial_py_sift_intro.html.

[31] Wikipedia, «Wikipedia: Scale-invariant feature transform,» 31 Mayo 2017. [En línea]. Available:

https://es.wikipedia.org/wiki/Scale-invariant_feature_transform. [Último acceso: 20 Junio 2017].

[32] E. J. Suárez Carmona, «Tutorial sobre Máquinas de Vectores de Soporte (SVM),» 2016.

[33] V. Blanco, «Universidad de Granada: Departamento de Métodos Cuantitativos para la economía y la empresa. Teoría Lagrangiana».

[34] Wikipedia, «Máquinas de vectores soporte,» [En línea]. Available: https://es.wikipedia.org/wiki/M%C3%A1quinas_de_vectores_de_soporte.

[35] C. Yao, F. Wu y H.-j. Chen, «Traffic sign recognition using HOG-SVM and grid search».

[36] N. Cai, W. Z. Liang, S. Q. Xu y F. Z. Li, «Traffic Sign Recognition Based on SIFT Features».

[37] A. Lorsakul y J. Suthakorn, «Traffic Sign Recognition Using Neural Network on OpenCV: Toward Intelligent Vehicle/Driver Assistance System».

[38] J. Miura y Y. Shirai, «An active vision system for on-line traffic sign recognition».

[39] M. Y. T. K. Anh-Tuan Hoang, «High Accuracy and Simple Real-Time Circle Detection on Low-Cost FPGA for Traffic-Sign Recognition on Advanced Driver Assistance System,» 2015. [En línea]. Available: http://sasimi.jp/new/sasimi2015/files/archive/pdf/p397_R4-13.pdf. [Último acceso: 05 Junio 2017].

[40] INSTITU FÜR NEUROINFORMATIK, «The German Traffic Sign Recognition Benchmark,» [En línea]. Available: http://benchmark.ini.rub.de/?section=gtsrb&subsection=news.

[41] A. L. Peña, E. Valveny y M. Vanrell, «Coursera: Detección de objetos,» [En línea]. Available: https://www.coursera.org/learn/deteccion-objetos.

[42] Xilinx, «Vivado Design Suite User Guide: High-Level Synthesis,» 2016.