



UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

Técnicas y metodologías para la implementación de sistemas de visión en All Programmable SoCs utilizando síntesis de alto nivel

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Miguel Ángel García
<miguel.garcia@gmail.com>

Directora: Dra. Patricia Borensztein
Buenos Aires, 2016

Técnicas y metodologías para la implementación de sistemas de visión en All Programmable SoCs utilizando síntesis de alto nivel

Este trabajo estudia y propone técnicas que permiten optimizar el rendimiento de soluciones de visión artificial haciendo uso de co-diseño hardware/software sobre All Programmable SoCs. En particular, el foco esta puesto sobre la construcción de aceleradores hardware utilizando síntesis de alto nivel.

Las optimizaciones aplicables a aceleradores hardware son introducidas de forma genérica y luego aplicadas en un caso de estudio, Odometría Visual Estéreo, para estudiar su impacto. En este contexto la capacidad de procesar vídeo en tiempo real sin perder cuadros es crítica para obtener buenos resultados.

El trabajo, además, describe y estudia, en el contexto del caso de estudio, una metodología de desarrollo específica para co-diseño hardware/software. Esta metodología propone partir de una solución implementada en una computadora de propósito general, luego portarla al AP SoC, optimizar el software y, finalmente, optimizarla construyendo hardware para acelerar las tareas más demandantes.

Durante el desarrollo del trabajo se utiliza la placa de desarrollo Zybo y la suite de herramientas Vivado, de Xilinx.

Palabras claves: Co-diseño hardware/software, Field Programmable Gate Array, All Programmable SoC, Síntesis de alto nivel, Metodología de desarrollo, Visión artificial, Odometría visual, Optimización por hardware.

Techniques and methodologies for implementing vision system on All Programmable SoCs using high level synthesis

This work studies and proposes techniques to optimize artificial vision solutions using hardware/software co-design on All Programmable SoCs. It focus on the building of hardware accelerators using high level synthesis..

The optimizations are introduced in a generic way and then applied in a case study, Stereo Visual Odometry, to analize their impact. On this context the ability to process video on real time, without losing frames, is critical to achieve good results.

This work, also, describes and studies, in the case study context, a hardware/software co-design specific development methodology. The methodology proposes to start implementing the solution on a general purpose computer, then port it to the AP SoC, apply software optimizations and, finally, optimize it building hardware to accelerate the most demanding tasks.

Along this work, a Zybo development board and the Vivado suite, from Xilinx, are used.

Keywords: hardware/software co-design, Field Programmable Gate Array, All Programmable SoC, High Level Synthesis, Development Methodology, Artificial vision, Visual Odometry, Optimization using hardware.

Agradecimientos

A mis papás, mi hermano y toda mi familia, por el afecto y apoyo incondicional desde que tengo memoria.

A mis amigos nuevos, viejos, de aquí y de allá que me preguntaban a cada rato “cómo va la tesis?”.

A Patricia, que me introdujo en el mundo de la lógica programable, del que versa este trabajo, y acepto dirigir mi tesis, con total dedicación y disponibilidad.

A mis compañeros de cursadas y trabajos prácticos, junto a los que he aprendido, trabajado y pasado muchas horas, algunas de estrés y otras de alegría.

A Hernán Melgratti y la “Maratón de tesis”, una buena iniciativa que nos ayudo a mantener el ritmo de trabajo.

A mis profesores durante la carrera, por la dedicación puesta en enseñar y la calidad de los conocimientos transmitidos.

A la universidad pública.

A mi familia.

Índice general

1. Introducción	1
1.1. Definición del problema y alcance	1
1.2. Objetivos	2
1.3. Organización del documento	3
2. Conceptos, Herramientas y Tecnologías	7
2.1. All Programmable System On a Chip	7
2.1.1. Lógica Programable	8
2.2. Herramientas	9
2.2.1. Vivado HLS	9
2.2.2. Vivado IP Integrator	10
2.2.3. Vivado SDK	10
3. Presentación del Caso de Estudio	11
3.1. Dataset KITTI para Odometría Visual	11
3.2. LIBVISO2	11
3.2.1. Funcionamiento	12
3.2.2. Adaptación inicial	13
3.2.3. Análisis del impacto del rendimiento en el resultado	14
3.3. Metodología de trabajo	16
4. Análisis e Implementación de la solución en una computadora de propósito general	17
4.1. Diseño: Relevamiento de LIBVISO2	17
4.1.1. Adquisición de imágenes	18
4.1.2. Detección de features	18
4.1.3. Extracción de features	19
4.1.4. Matching de features entre imágenes	19
4.1.5. Estimación de pose	21
4.2. Pruebas en procesador de propósito general	22
5. Primera implementación de la solución en el AP SoC	25
5.1. Relevamiento de ZYBO	25
5.2. Construcción de la plataforma de hardware	26
5.3. Construcción de la plataforma de software	27
5.4. Migración del software al APSoC	30
5.4.1. Migración de libviso_gp al APSoC	30

5.4.2. Acceso a datasets desde ZYBO	31
5.4.3. Pruebas de la solución en ZYBO	32
5.5. Medición de rendimiento	35
6. Optimización por software	37
6.1. Análisis del impacto de los parámetros	37
6.1.1. Presentación de los parámetros	37
6.1.2. Calidad del resultado	38
6.1.3. Tolerancia a la pérdida de cuadros	44
6.1.4. Impacto en el rendimiento	48
6.2. Análisis detallado del rendimiento de viso	49
6.3. Optimizaciones	50
6.3.1. Optimización de Detección y Extracción de features	51
6.3.2. Optimización de Matching de features entre imágenes	52
6.3.3. Optimización de Estimador de pose	55
6.4. Pruebas de la solución optimizada en ZYBO	56
6.5. Medición de rendimiento	58
7. Optimización mediante aceleración por hardware	59
7.1. Análisis detallado del rendimiento de viso_s	60
7.2. Aplicación de HLS a la construcción de IP Cores para procesamiento de imágenes	60
7.2.1. Vivado HLS	60
7.2.2. Patrón de diseño para procesamiento secuencial de píxeles	61
7.2.3. Optimizaciones y rendimiento	64
7.3. Arquitectura de Hardware	71
7.3.1. Partición Hardware/Software	71
7.3.2. Comunicación entre procesadores, IP Cores y memoria externa	72
7.3.3. Pipeline de procesamiento de imágenes	73
7.4. IP Cores	75
7.4.1. IP Core Sobel	75
7.4.2. IP Core Half Resolution	78
7.4.3. IP Core Blob + Checkerboard	82
7.4.4. IP Core Features	84
7.4.5. IP Core Controller	88
7.5. Integración del hardware	91
7.5.1. Configuración de PS	91
7.5.2. Integración del pipeline de procesamiento de imágenes	93
7.5.3. Integración y configuración de controladores DMA	94
7.5.4. Generación del archivo Bitstream	96
7.5.5. Prueba de la plataforma	97
7.6. Modificaciones a la plataforma software	97
7.6.1. Actualización de los archivos de descripción del hardware	98
7.6.2. Módulo del kernel controlador del pipeline de procesamiento de imágenes	99
7.6.3. Construcción y prueba	100
7.7. Aplicación viso_h	100
7.8. Pruebas de la solución optimizada en Zybo	101

7.9. Medición de rendimiento	103
8. Pruebas y Resultados	105
8.1. Experimento 1: procesamiento en tiempo real	105
8.2. Experimento 2: procesamiento a 5 cuadros por segundo	109
8.3. Experimento 3: procesamiento a 10 cuadros por segundo	113
8.4. Experimento 4: viso_h configuración adhoc vs procesamiento a 10 cuadros por segundo	117
9. Conclusiones	121
9.1. Trabajos futuros	122

Capítulo 1

Introducción

En este capítulo la sección 1.1 presenta el alcance de nuestro trabajo, junto con los principales conceptos involucrados. La sección 1.2 presenta los objetivos y, finalmente, la sección 1.3 detalla la organización de este documento.

1.1. Definición del problema y alcance

Los sistemas embebidos móviles se encuentran por lo general sujetos a restricciones de tamaño y consumo eléctrico. Debido a estas limitaciones, cuando es necesario procesar grandes volúmenes de datos para responder a eventos en tiempo real, se produce un conflicto entre los requerimientos, pues el uso de procesadores de alta velocidad requeridos para satisfacer los tiempos de ejecución necesarios, se ve impedido por las limitaciones de consumo eléctrico.

Una forma de hacer frente a este problema es mediante el uso de soluciones tecnológicas que combinan en un mismo circuito integrado (IC), procesadores, memoria y lógica programable. Esto posibilita la construcción de sistemas que hacen uso de aceleración por hardware para ciertas funciones, delegando parte del trabajo del procesador a **Intellectual Property Cores (IP Cores)**, componentes hardware reutilizables diseñados para tareas específicas.

Al utilizar lógica programable los costos de diseño y construcción de hardware son menores que si se utilizaran **Application Specific Integrated Circuit (ASIC)**, cuando los volúmenes de producción son pequeños.

Las **Field Programmable Gate Arrays (FPGAs)** son circuitos integrados que contienen bloques de lógica programable junto con interconexiones configurables entre estos bloques. Luego de la fabricación del circuito el diseñador puede configurarlo para que realice cualquier tarea que pueda implementarse en lógica digital. En este sentido una **FPGA** tiene el mismo poder de cómputo que una maquina de Turing finita.

Si bien los fabricantes de **FPGAs** integraron en sus productos procesadores físicos hace varias generaciones, en la actualidad este enfoque evolucionó, dando lugar a los **All Programmable Systems on a Chip (AP SoCs)**.

Los **AP SoCs** integran en un mismo IC un sistema completo de procesamiento (**Processing System o PS**), formado por procesadores, cache, controladores de memoria y otros dispositivos y lógica programable (**Programmable Logic o PL**), formada por compuertas lógicas, bloques DSP y bloques de memoria RAM. Los **AP SoCs** contienen además buses dedicados para comunicar **PS** y **PL**.

El foco de nuestro trabajo está puesto en el análisis de la construcción de soluciones utilizando **AP SoCs**, combinando software y síntesis de alto nivel para crear sistemas utilizando *co-diseño hardware/software*, donde ciertas tareas son delegadas por la CPU a aceleradores hardware para funciones específicas.

En este trabajo utilizaremos una placa de desarrollo *ZYBO*¹, provista de un *AP SoC Zynq-7000* y 512MB de memoria central, y la suite de herramientas *Vivado*², de *Xilinx*.

El diseño de soluciones sobre **AP SoCs** puede dividirse en tres grandes tareas, cada una soportado por una o más herramientas.

- Diseño de **IP Cores**: se debe especificar hardware utilizando un lenguaje de descripción de hardware (**HDL** por sus siglas en inglés) o a través de una herramientas de **síntesis de alto nivel (HLS)**, como *Vivado HLS* que permite especificar hardware utilizando lenguajes de programación como C y C++, minimizando los tiempos de desarrollo y pruebas.
- Integración de **IP Cores**: consiste en la configuración e integración de **IP Cores** propios y de terceros para formar el sistema completo. *Vivado IP Integrator* facilita esta tarea automatizando parte del trabajo, fundamentalmente la interconexión de **IP Cores** a través de buses estándar.
- Programación del software que corre en los procesadores: se debe desarrollar el software que correrá en el o los procesadores del **AP SoC**. *Xilinx SDK* es el entorno de desarrollo por defecto de *Vivado*.

Decidimos centrar nuestro estudio en la aplicación de **AP SoCs** a problemas de visión artificial, ya que dan un contexto práctico donde la calidad del resultado depende en gran parte de la capacidad de procesar una gran cantidad de datos en tiempo real. Además, muchas de las funciones comúnmente aplicadas en procesamiento de imágenes, por ejemplo la aplicación de convoluciones, son buenas candidatas a ser aceleradas por hardware, dado que se basan en la aplicación de pocas y sencillas operaciones a un gran volumen de datos que puede ser leído en forma secuencial.

En particular usaremos la implementación de un **sistema de Odometría Visual Estéreo en un AP SoC** como caso de estudio. Partiendo de una implementación software, *LIBVISO2* (Library for Visual Odometry 2)³, analizaremos cómo las **metodologías de desarrollo y herramientas de síntesis de alto nivel** permiten mejorar el rendimiento, a través de la aceleración por hardware.

Utilizaremos la metodología propuesta en “Una nueva metodología para co-diseño de sistemas embebidos centrados en procesador usando FPGAs”⁴ para guiar el trabajo, obviando el paso de diseño inicial, pues partimos de una solución ya implementada en **PC**.

1.2. Objetivos

Es importante resaltar que nuestro objetivo marco es analizar el ciclo completo de desarrollo para implementar soluciones que aprovechen eficientemente los recursos de los AP SoCs, evitando la pérdida de cuadros.

¹Diligent 2014.

²Xilinx 2014a.

³Geiger, Ziegler y Stiller 2011.

⁴Pedre, Krajník, Todorovich y Borensztein 2012.

Dividimos este objetivo global en los siguientes puntos:

- Analizar aceleración por hardware utilizando **HLS**: Investigar técnicas de programación y arquitecturas de hardware que permitan construir aceleradores hardware para tareas de visión artificial. En este apartado el principal desafío radica en que, al especificar hardware, el diseñador debe crear la jerarquía de memoria necesaria y seleccionar las técnicas de optimización que permitan explotar el paralelismo inherente al hardware para resolver el problema eficientemente.
- **Analizar el diseño de todas las capas del sistema**: al estar creando soluciones sobre AP SoCs el, diseñador está a cargo no solo del software de aplicación que resuelve el problema en cuestión, sino además, del diseño de parte del hardware del sistema base y la selección del sistema operativo.

Para diseñar el hardware es necesario primero crear la partición hardware/software de la solución para alcanzar el rendimiento requerido, restringido a los recursos disponibles. Esto incluye decidir que componentes serán implementados en hardware y de que forma los mismos se integraran al resto del sistema. El diseñador debe seleccionar o crear los protocolos de comunicación entre componentes que colaboren entre si y entre los componentes y los procesadores.

La selección del sistema operativo (SO) posibilita o limita explotar características del hardware, por ejemplo utilizar varios procesadores como un sistema de multi-procesamiento simétrico o múltiples sistemas independientes. Si bien al seleccionar un SO los factores puramente técnicos son importantes, también deben tenerse en cuenta otros, como las herramientas, documentación disponible y la facilidad para crear controladores.

- Analizar la potencia de esta tecnología comparando el rendimiento de soluciones implementadas sobre **PS+PL** vs **PS**: nos interesa evaluar el rendimiento relativo de las soluciones y su impacto en la calidad de los resultados del caso de estudio al procesar imágenes en tiempo real.
- Analizar las herramientas disponibles: finalmente nos interesa evaluar la plataforma Vivado para la construcción de soluciones y su análisis de velocidad y consumo de recursos. Haremos foco en dos herramientas *Vivado HLS*, para la creación de aceleradores hardware y *Vivado IP Integrator*, para la integración de **IP Cores** al sistema.

Además de evaluar el estado actual de herramientas, metodologías y técnicas para el desarrollo de soluciones sobre **AP SoCs**, esperamos que el estudio de estos puntos con un enfoque práctico sea aplicable a trabajos futuros. Creemos que esto es importante ya que distintas plataformas, como por ejemplo la *EDU-CIAA*, están evaluando la incorporación de **AP SoCs** en reemplazo de CPUs tradicionales.

Finalmente esperamos que los resultados de este trabajo puedan ser utilizados en la elaboración de productos donde se haga uso de **AP SoCs** para acelerar el procesamiento de imágenes.

1.3. Organización del documento

Este documento está organizado en los siguientes 9 capítulos:

- **Capítulo 1 “Introducción”:** presenta alcance y objetivos de este trabajo.
- **Capítulo 2 “Conceptos, Herramientas y Tecnologías”:** introduce los conceptos, herramientas y tecnologías utilizados en este trabajo, para facilitar su comprensión.
- **Capítulo 3 “Presentación del Caso de Estudio”:** presenta el caso de estudio que abordamos, la construcción de una Solución de Odometría Visual Estéreo en un **AP SoC**. Describe la implementación *LIBVISO2* que utilizamos como punto de partida y un estudio sobre el efecto de la pérdida de cuadros en la calidad del resultado. Este capítulo introduce la metodología que empleamos para guiar el proceso de desarrollo.
- **Capítulo 4 “Análisis e Implementación de la solución en una computadora de propósito general”:** describe el diseño, implementación y prueba de la solución en una computadora de propósito general.
- **Capítulo 5 “Primera implementación de la solución en el AP SoC”:** describe el proceso de implementación y prueba de la solución en *Zybo*, incluyendo la construcción de las plataformas de hardware y software.
- **Capítulo 6 “Optimización por software”:** Comienza presentando un análisis detallado del rendimiento de la solución en el **AP SoC** y un estudio acerca del impacto de los parámetros de *LIBVISO2* sobre el rendimiento y calidad del resultado. A continuación, describe las optimizaciones aplicadas para obtener un mejor rendimiento y las pruebas realizadas sobre la solución optimizada.
- **Capítulo 7 “Optimización mediante aceleración por hardware”:** En este capítulo radican los principales aportes de este trabajo.

El mismo describe la optimización de la solución construida en el capítulo anterior delegando parte del trabajo a aceleradores hardware en **PL**. Introduce la nueva arquitectura de hardware, la cual incluye un ***pipeline*** de procesamiento de imágenes que desarrollamos específicamente para este trabajo.

Nuestro principal aporte se encuentra en la sección [7.2](#). Allí caracterizamos un patrón de diseño muy frecuente en el desarrollo de IP Cores para procesamiento de imágenes utilizando síntesis de alto nivel y, posteriormente, propusimos dos optimizaciones **procesar K píxeles por ciclo** y **fusión de IP Cores**.

El desarrollo de la implementación de los **IP Cores** introduce otra optimización, menos general, **ventanas con su información “condensada”**. La misma permite ahorrar recursos.

Luego se presentan las modificaciones a la plataforma software y al software de aplicación, necesarias para sacar provecho al ***pipeline*** de procesamiento de imágenes en hardware.

Finalmente se describe el proceso de prueba y medición de rendimiento de la solución construida.

- **Capítulo 8 “Pruebas y resultados”:** describe diferentes experimentos y compara sus resultados para las soluciones construidas a lo largo de este trabajo.

- **Capítulo 9 “Conclusiones:** presenta nuestras conclusiones sobre la metodología utilizada, el impacto de las optimizaciones propuestas y otros puntos de este trabajo. Finalmente se describe de que forma creemos que este trabajo puede influir en trabajos futuros.

Capítulo 2

Conceptos, Herramientas y Tecnologías

Este capítulo introduce los principales conceptos utilizados en este trabajo, así como las herramientas y tecnologías utilizadas en el mismo.

2.1. All Programmable System On a Chip

Los **All Programmable Systems On a Chip** (AP SoCs) surgen como una evolución de las **Field Programmable Gate Arrays** (FPGAs), circuitos integrados que contienen lógica que puede ser programada luego de su fabricación.

Al utilizar lógica programable los costos del diseño, prueba e implementación de hardware son varias veces menores que en la construcción de **Application Specific Integrated Circuits** (ASIC).

Desde su invención, en 1985, hasta la fecha la cantidad de lógica programable dentro de las **FPGAs** se ha incrementado, haciendo posible implementar circuitos más grandes y complejos. Además, han incorporado otros recursos, como bloques de memoria RAM y procesadores.

Los **AP SoCs** integran en un mismo IC un sistema completo de procesamiento (**PS**) y lógica programable (**PL**), además de **buses dedicados para comunicar PS y PL**.

En este trabajo utilizamos una placa de desarrollo *ZYBO*¹ fabricada por *Digilent*, la misma integra un *AP SoC Zynq-7000*² diseñado por *Xilinx* y 512MB de memoria RAM externa al **AP SoC**.

¹Digilent 2014.

²Xilinx 2016.

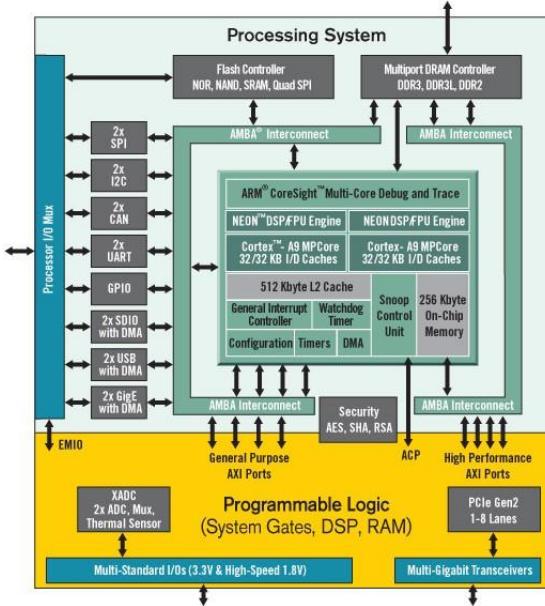


Figura 2.1: Arquitectura Zynq-7000.

La sección **PS** provee la funcionalidad equivalente a la de otros **SoCs** utilizados en muchas **Single Board Computers**, como *Raspberry Pi* y *BeagleBoard*, implementando un sistema de cómputo completo, con procesadores, interfaces de red, USB y otros periféricos.

La sección **PL** permite mejorar el rendimiento de las soluciones construidas en software sobre el **AP SoC**, haciendo posible delegar parte del procesamiento a hardware construido específicamente para acelerar ciertas tareas.

2.1.1. Lógica Programable

La sección **PL** del **AP SoC** funciona de la misma manera que las **FPGAs** modernas. Está formada por distintos tipos de bloques y una red de interconexión entre ellos, definiendo sus configuraciones e interconexiones es posible implementar cualquier circuito de lógica digital.

En particular *Zynq* utiliza los siguientes tipos de bloques³:

- **Configurable Logic Block (CLB)**: Son los principales componentes para implementar circuitos, tanto secuenciales como combinatorios. Cada **CLB** está conectado a una matriz de interconexión para poder acceder a la matriz general de ruteo, que le permite conectarse con otros bloques en **PL**.

Un **CLB** contiene dos **slices**, cada uno compuesto por 4 **tablas de verdad** o “**Look Up Tables**” (LUTs) de 6 entradas, multiplexores y 8 **flip-flops** (FF).

Estos elementos permiten implementar funciones lógicas, aritméticas y de ROM. Adicionalmente, algunos **slices** pueden utilizarse para almacenar datos, es decir como memoria RAM distribuida, o como **registros de desplazamiento** de 32 bits.

³Xilinx 2016.

- **Entrada y salida (IO)**: Estos bloques están conectados a pines físicos del circuito integrado. De esta forma es posible conectar **PL** a señales externas, como por ejemplo distintos tipos de sensores.
- **Clock**: Proporciona señales de reloj a los demás bloques. Su configuración permite definir regiones y árboles de reloj para evitar fenómenos no deseados, como **skew** (cuando una misma señal de **clock** llega a diferentes elementos con retardos distintos)
- **Digital Signal Processing Slice (DSP)**: Permite implementar multiplicadores y acumuladores de forma más eficiente que si se utilizaran CLBs.
- **Block RAM (BRAM)**: Funcionan como una memoria RAM sincrónica de doble puerto con una capacidad de 36 Kbits y palabras de 32 bits. Cada **BRAM** puede utilizarse como dos memorias independientes de 18Kb cada una.

El funcionamiento de la red de **interconexión** entre las entradas y salidas de los bloques, los multiplexores y el contenido de las **LUTs**, **FF** y **Block RAMs** están implementados como RAM. Esto permite reconfigurar el chip luego de su fabricación tantas veces como sea necesario para implementar diferentes circuitos.

2.2. Herramientas

Implementar soluciones utilizando **co-diseño hardware/software** requiere definir no solo el software que correrá, sino también diseñar parte del hardware. Por lo tanto es necesario llevar a cabo al menos tres tareas:

- Construir los **IP Cores** que implementarán funciones en **PL**.
- Instanciar los **IP Cores** y, de ser necesario, conectarlos a **PS**.
- Programar el software que correrá en los procesadores en **PS**.

En **2012** *Xilinx* publicó tres herramientas para facilitar cada una de las tareas, estas son: *Vivado HLS*, *Vivado IP Integrator* y *Vivado SDK*, respectivamente.

2.2.1. Vivado HLS

Esta herramienta permite especificar circuitos digitales utilizando **Síntesis de Alto Nivel**. Es decir, a través de lenguajes de alto nivel, en particular C y C++. Los circuitos construidos son luego exportados como **IP Cores** para ser integrados en la solución.

A diferencia del diseño de hardware con lenguajes de descripción de hardware (**HDL**), como *VHDL* o *Verilog*, usar lenguajes de alto nivel permite al diseñador abstraerse de varios detalles, como el manejo de señales de reloj, paralelización de tareas, codificación de algoritmos como máquinas de estados y codificación de interfaces. Esto reduce la cantidad de errores, permitiendo al diseñador enfocarse en los algoritmos a implementar en hardware.

De todas maneras, dado que el programa en C o C++ no está destinado a correr en una computadora preexistente, sino que es utilizado por la herramienta para **sintetizar** hardware, el diseñador debe tener en cuenta los mecanismos de inferencia para obtener buenos resultados. Esto se debe a que la forma en que la herramienta extrae un circuito

digital a partir de un código de alto nivel afecta a la calidad (velocidad y consumo de recursos) del circuito extraído.

El diseñador debe considerar varios factores, como interfaces y optimizaciones a aplicar, e introducir sus decisiones en la herramienta, modificando el código de alto nivel y anotándolo con **directivas de HLS⁴**. Hay varios tipos de directivas, los principales son:

- Alocación de recursos: define en que tipo de memoria se almacenara una variable o arreglo, por ejemplo **LUTs** o **BRAM**.
- Interfaz: define las señales de entrada y salida del **IP Core**, por ejemplo a que tipo de **bus** se conectará.
- Optimización: algunas de las optimizaciones aplicables son **LOOP UNROLLING** y **PIPELINE**. Su aplicación suele tener un impacto positivo en la velocidad y negativo en el consumo de recursos, por lo tanto, es trabajo del diseñador decidir cuándo aplicarlas.

2.2.2. Vivado IP Integrator

Esta herramienta permite definir la plataforma hardware de la solución. Esta tarea consiste en instanciar y configurar **IP Cores**, definir las conexiones entre ellos y con **PS**.

Luego de definida la plataforma, esta herramienta permite sintetizarla y generar el archivo de configuración (**bitstream**), necesario para cargar la plataforma en el **AP SoC**.

2.2.3. Vivado SDK

Esta herramienta es un **entorno de desarrollo de software** (IDE) que permite desarrollar y realizar debug del software que corre en **PS**.

Vivado *SDK* automatiza la creación de una pequeña capa de software, llamada **Board Support Package**, que abstrae parte de las complejidades del hardware.

En este trabajo el uso de Vivado *SDK* es mínimo, como explica el capítulo 5 decidimos utilizar *PetaLinux*, una distribución de *Linux*, y *PetaLinux SDK* para el desarrollo de la plataforma software de la solución.

⁴Xilinx 2014b.

Capítulo 3

Presentación del Caso de Estudio

El caso de estudio en el que centramos nuestro análisis es el desarrollo de una solución de Odometría Visual Estéreo sobre un **AP SoC Zynq-7000**, de *Xilinx*, utilizando la plataforma de desarrollo *Vivado*.

La odometría visual estéreo consiste en calcular incrementalmente la posición y orientación de un vehículo a partir del análisis de imágenes tomadas por dos cámaras montadas en el mismo.

Para nuestro trabajo utilizaremos como punto de partida la implementación de código abierto *LIBVISO2*. *LIBVISO2* es una solución de odometría visual estéreo para **PC**. La elegimos por dos motivos, en primer lugar está programada en C++ con *libpng* como única dependencia externa, lo que facilita portarla a otras plataformas. En segundo lugar, el código fuente está modularizado y es sencillo de entender, simplificando su análisis.

3.1. Dataset KITTI para Odometría Visual

En este trabajo utilizaremos las secuencias de imágenes en escala de grises del **dataset KITTI para Odometría Visual**¹ para llevar a cabo las pruebas y mediciones.

Cada secuencia de imágenes fue capturada desde un automóvil utilizando dos cámaras de 1.4 Megapíxeles montadas en el mismo mientras se realizaba un recorrido.

Ambas cámaras se encuentran sincronizadas y toman 10 imágenes por segundo. De esta forma para cada segundo se cuenta con 10 cuadros estéreo, capturados por las cámaras 0 (izquierda) y 1 (derecha) de la figura:

Si bien el dataset también proporciona imágenes a color e información de otros sensores, no utilizaremos esos datos en este trabajo.

Para cada secuencia de imágenes se provee en archivos de texto la trayectoria real (**ground truth**) del vehículo y la calibración intrínseca y extrínseca de las cámaras.

3.2. LIBVISO2

Esta sección describe el funcionamiento de **LIBVISO2** junto con las modificaciones iniciales que realizamos para adaptarla a este trabajo. Luego se describe un primer experimento donde analizamos el impacto del rendimiento en la calidad de los resultados, según distintas parametrizaciones.

¹Geiger, Lenz y Urtasun 2012.

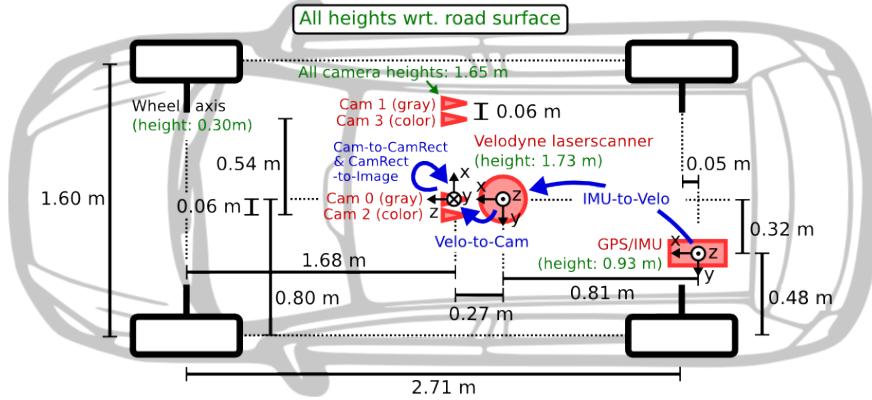


Figura 3.1: Configuración del vehículo para la adquisición de datos.

3.2.1. Funcionamiento

La entrada de *LIBVISO2* para el cálculo de odometría es la calibración intrínseca y extrínseca de las cámaras y una secuencia de cuadros estéreo, cada uno formado por dos imágenes **rectificadas** tomadas simultáneamente.

En primer lugar para cada una de las imágenes de un cuadro estéreo se obtienen sus filtros *Checkerboard* y *Blob* convolucionando la imagen con dos matrices de convolución.

A continuación, se aplica el algoritmo *Non-maximum Supression (NMS)* al resultado de cada filtro para detectar píxeles cuyos valores son mínimos o máximos locales.

Para esto cada imagen filtrada es particionada en una grilla con celdas de $N \times N$ píxeles. Un píxel es máximo si es el mayor dentro de una celda de la grilla, es mayor a *nms_tau* (el valor *nms_tau* es configurable) y, además, dentro de una ventana de $2N \times 2N + 1$ píxeles a su alrededor no existe un píxel con valor mayor. Análogamente se detectan píxeles mínimos.

LIBVISO2 aplica **NMS** dos veces a cada filtro, utilizando por defecto valores de $N=3$ y $N=9$, con el objetivo de detectar puntos sobresalientes “**densos**” y “**dispersos**”, respectivamente. Los puntos detectados son utilizados como características (**features**) de la imagen.

Una vez calculados los **features**, se extraen sus descriptores, utilizando el filtro *Sobel* de cada imagen para calcularlos. *LIBVISO2* utiliza vectores de 32 bytes como descriptores, usando la suma de las diferencias absolutas coordenada a coordenada para compararlos.

Luego se buscan coincidencias entre **features** de la imagen izquierda y derecha, a través de la comparación de descriptores. Asumiendo que si dos **features** coinciden, entonces son la imagen del mismo punto en el espacio, se puede estimar la posición 3D del punto.

Finalmente, el algoritmo calcula el desplazamiento entre un par consecutivo de cuadros estéreo utilizando **RANSAC**. Buscando coincidencias entre sus **features** y la información de los puntos 3D calculados para cada uno, se estima la traslación y rotación que minimiza la diferencia entre las posiciones observadas y esperadas de los puntos. Es decir, se buscan la rotación *R* y la traslación *T* tales que minimizan: $\sum |Rp_{1i} + T - p_{2i}|$, donde *p₁* y *p₂* son las posiciones 3D de los puntos observados en el primer y segundo cuadro, respectivamente.

Acumulando rotaciones y traslaciones es posible estimar la **pose** actual, posición y orientación.

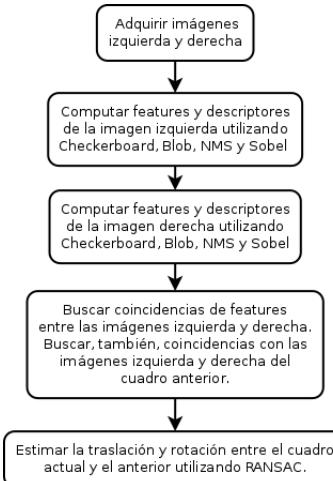


Figura 3.2: Pasos del procesamiento de un cuadro estéreo.

3.2.2. Adaptación inicial

El primer paso de nuestro desarrollo consistió en modificar el programa principal de *LIBVISO2* para que leyera los datos de calibración y configuración en tiempo de ejecución desde archivos de texto². Originalmente estos datos se encontraban en el código fuente.

Los datos de calibración se leen desde archivos de texto que contienen las matrices de proyección de las cámaras izquierda (P_0) y derecha (P_1).

P0:	718.856	0	607.19	0
	0	718.856	185.215	0
	0	0	1	0
P1:	718.856	0	607.19	-386.14
	0	718.856	185.215	0
	0	0	1	0

Figura 3.3: Archivo de calibración de ejemplo.

El archivo de configuración permite redefinir el valor de los siguientes parámetros de *LIBVISO2*:

- **nms_n**: Tamaño de la ventana utilizada en **NMS** para la detección de máximos y mínimos densos. Para los dispersos se utiliza una ventana de tamaño $\max(3 * nms_n, 10)$.
- **nms_tau**: Cota usada en **NMS** para la detección de máximos y mínimos. Variarla permite detectar más o menos puntos.
- **match_bsize**: El algoritmo de búsqueda de coincidencias entre **features** los agrupa en “baldes” según su posición. Este parámetro determina el ancho y alto de los “baldes”.

²El resultado de estas modificaciones se encuentra en el archivo `libviso2_adaptado.tar.gz`.

- **match_radius:** Máxima distancia en píxeles para la búsqueda de coincidencias de **features**. Aumentar este valor hace más probable encontrar coincidencias, pero tiene un impacto negativo en el rendimiento.
- **ransac_iters:** Cantidad de iteraciones **RANSAC** para estimar la rotación y traslación entre cuadros.

```
3
50
50
200
200
```

Figura 3.4: Archivo de configuración con los valores por defecto.

En este paso también eliminamos el código específico para visión monocular, pues no lo utilizamos en este trabajo, y simplificamos la construcción creando un Makefile³.

3.2.3. Análisis del impacto del rendimiento en el resultado

Realizamos un experimento sobre una PC para evaluar cómo la pérdida de cuadros de video afecta a la calidad del resultado. Modificamos *LIBVISO2* para poder simular la pérdida de cuadros y procesamos las secuencias de imágenes del dataset KITTI para odometría visual, originalmente capturadas a 10 cuadros por segundo, simulando capacidades de procesamiento de 10, 5 y 3 cuadros por segundo (**fps**).

A continuación, para cada secuencia de imágenes medimos y graficamos las distancias entre las trayectorias estimadas utilizando *LIBVISO2* y las trayectorias reales.

Estos son los principales resultados⁴:

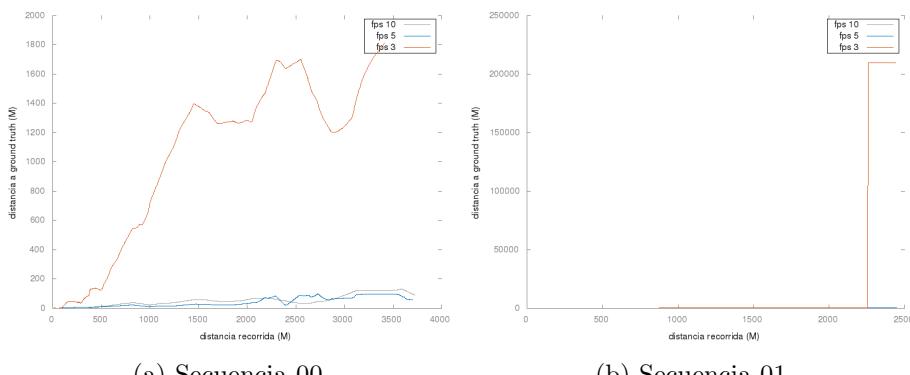
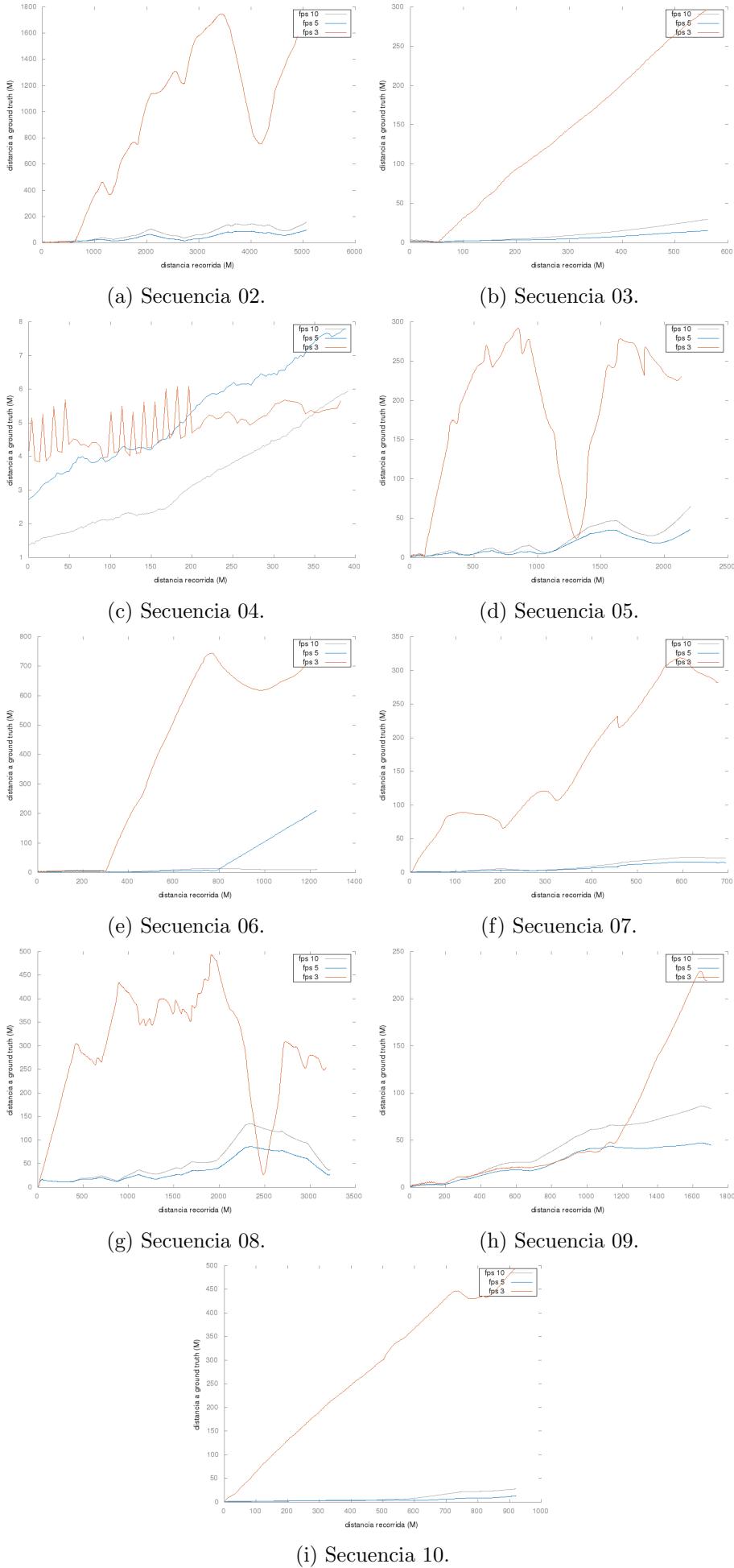


Figura 3.5: Gráficos de distancias entre trayectorias reales y estimadas a 10, 5 y 3 **fps** para las secuencias del dataset **KITTI** para odometría visual

³El programa original utilizaba `cmake`

⁴Los resultados completos pueden encontrarse en el apéndice `estudio-perdida-fps.tar.gz`



Analizando los gráficos se observa que la pérdida de cuadros tiene un gran impacto en la calidad del resultado, especialmente cuando se pierden más de la mitad de los mismos.

Por lo tanto, nuestro trabajo sobre el caso de estudio se centra en analizar cómo la aplicación de metodologías de diseño y técnicas de programación **HLS** permiten lograr una implementación eficiente en el **AP SoC**, capaz de minimizar la pérdida de cuadros.

3.3. Metodología de trabajo

Guaremos el proceso de desarrollo con una metodología basada en la propuesta en “Una nueva metodología para co-diseño de sistemas embebidos centrados en procesador usando FPGAs”⁵.

Esta metodología parte el proceso de desarrollo en 4 etapas:

- A. Diseño
- B. Implementación y testing en procesador de propósito general.
- C. Partición Hardware/Software
- D. Implementación de hardware, testing e integración.

⁵Pedre, Krajnik, Todorovich y Borensztein 2012.

Capítulo 4

Análisis e Implementación de la solución en una computadora de propósito general

Este capítulo describe los pasos de análisis e implementación de la solución de odometría visual estéreo en una computadora de propósito general. Los mismos corresponden a los pasos “A. Diseño” y “B. Implementación y testing en procesador de propósito general” de la metodología que seguimos.

4.1. Diseño: Relevamiento de LIBVISO2

Si bien este trabajo parte de una solución ya implementada en un procesador de propósito general, decidimos documentar los principales componentes del sistema y agruparlos según sus responsabilidades.

La descomposición en componentes sirvió como guía en las etapas de implementación inicial en el **AP SoC** y optimización. Además, creemos que brinda una referencia que facilita el entendimiento del trabajo.

Durante el proceso de documentación pudimos interiorizarnos en el funcionamiento de *LIBVISO2*.

18 CAPÍTULO 4. ANÁLISIS E IMPLEMENTACIÓN DE LA SOLUCIÓN EN UNA COMPUTADORA DE

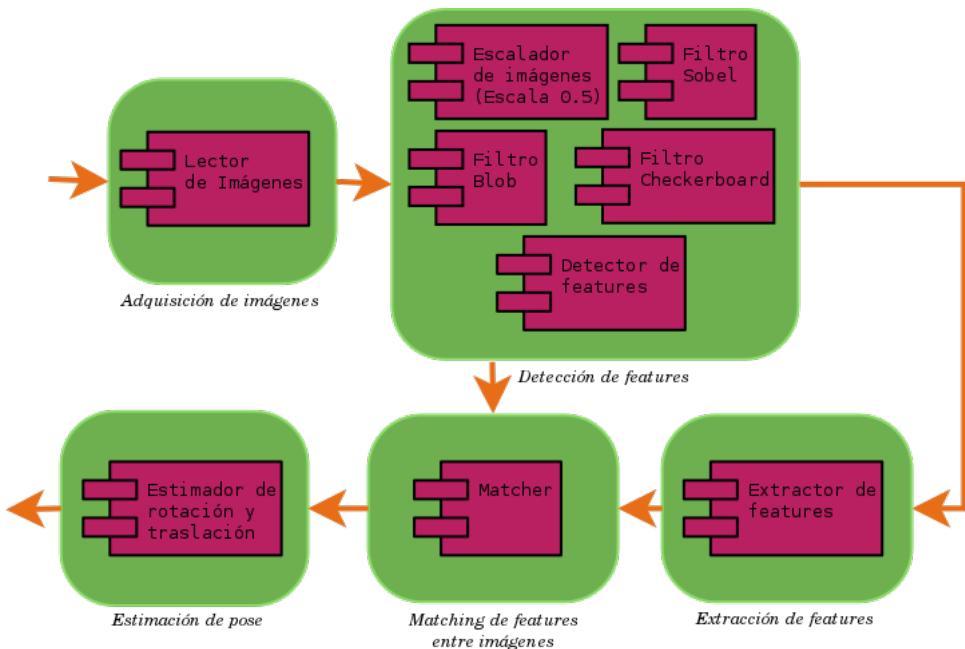


Figura 4.1: Componentes agrupados por tipo de tarea.
Las flechas naranjas representan flujo de datos.

4.1.1. Adquisición de imágenes

El componente de este grupo se encarga de cargar las imágenes izquierda y derecha en **buffers** de memoria, para luego ser procesadas.

La aplicación *demo* de LIBVISO2 utiliza archivos **PNG** (Portable Network Graphics)¹ como entrada.

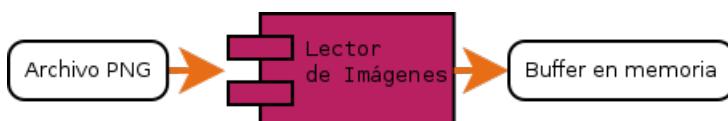


Figura 4.2: Entradas y salidas del “lector de imágenes”.

4.1.2. Detección de features

Los componentes en este grupo colaboran para detectar los **features** de una imagen en memoria.

Para cada imagen se producen dos listas de **features**, “densos” y “dispersos”, aplicando el algoritmo *Non-maximum Supression* a los filtros *Blob* y *Checkerboard*, tal como fue explicado anteriormente en 3.2.1. Al hacerlo de esta forma es posible acelerar la posterior etapa de matching.

LIBVISO2 realiza la detección y extracción de **features** sobre una copia escalada a la mitad del tamaño de la imagen, para hacerlo más rápidamente.

Este grupo de componentes calcula además los filtros *Sobel* de la imagen escalada y sin escalar. Los mismos son luego utilizados en las etapas de extracción de **features** y

¹RFC 2048.

matching, respectivamente.

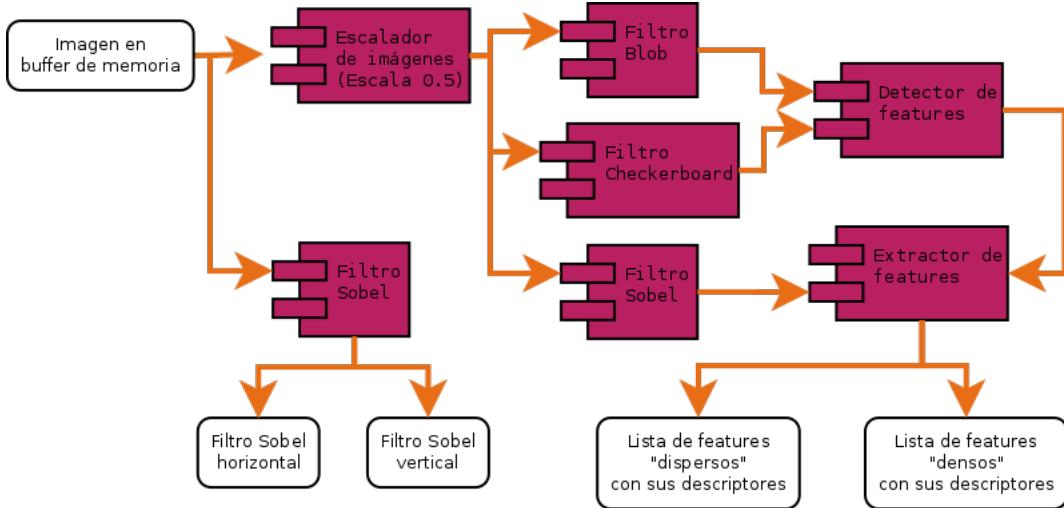


Figura 4.3: Entradas y salidas del grupo de componentes de detección de **features**.

Los componentes de este grupo son los únicos que realizan un procesamiento intensivo de la imagen. Es decir que cada píxel es accedido al menos una vez.

4.1.3. Extracción de features

El componente de este grupo calcula el descriptor de cada **feature** detectado.

Utiliza la información obtenida en la etapa de detección y construye los descriptores, formando para cada **feature** un vector con los valores de los píxeles vecinos en los filtros Sobel horizontal y vertical de la imagen escalada.

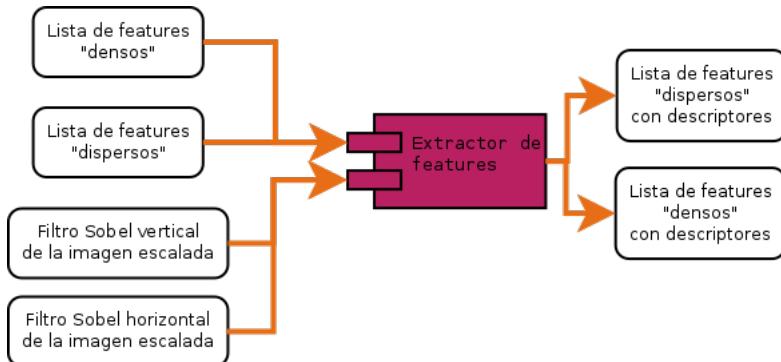


Figura 4.4: Entradas y salidas del componentes de extracción de **features**.

4.1.4. Matching de features entre imágenes

El componente “Matcher” busca coincidencias entre **features** de las imágenes izquierda y derecha de un mismo cuadro y entre el último cuadro provisto y el anterior. Para lograr esto, el componente guarda internamente el resultado del último cuadro estereóo procesado. La información obtenida es utilizada en la etapa de estimación de pose.

20 CAPÍTULO 4. ANÁLISIS E IMPLEMENTACIÓN DE LA SOLUCIÓN EN UNA COMPUTADORA DE VISIONES

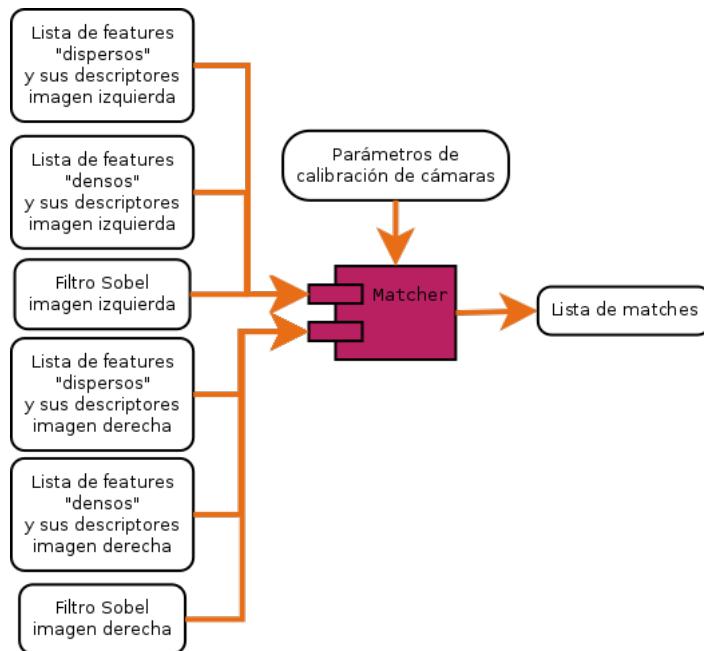


Figura 4.5: Entradas y salidas del componente.

La búsqueda de correspondencias se realiza mediante el algoritmo *quad-matching*, buscando para cada **feature** de la imagen izquierda del último cuadro anteriormente procesado (a) sus correspondencia en las imágenes derecha del mismo cuadro (b) y derecha (c) e izquierda (d) del cuadro actual, intentando cerrar el círculo.

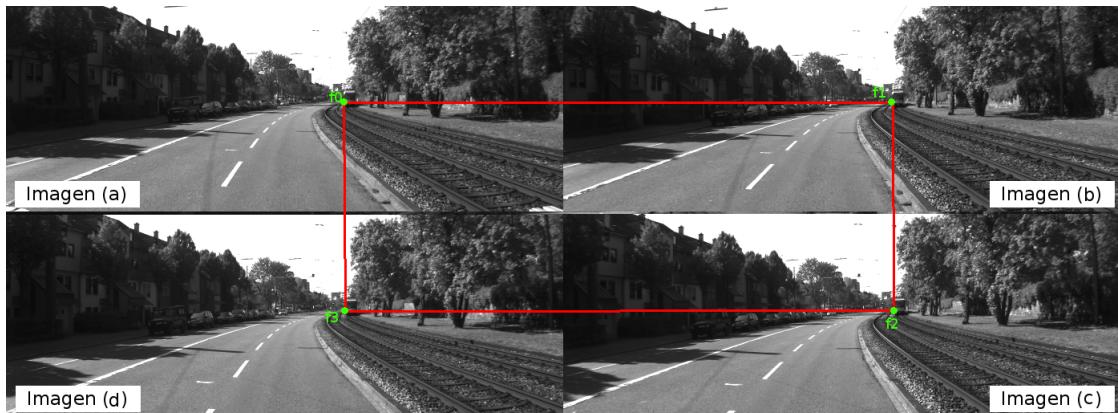


Figura 4.6: Ejemplo de Quad-matching para un **feature**.

Imagen superior: cuadro estéreo anterior.

Imagen inferior: cuadro estéreo actual.

El resultado es una lista que asocia píxeles correspondientes entre las 4 imágenes. Cada elemento de la lista contiene los siguientes datos:

- Coordenadas en la imagen izquierda del cuadro anterior.
- Coordenadas en la imagen derecha del cuadro anterior.

- Coordenadas en la imagen izquierda del cuadro actual.
- Coordenadas en la imagen derecha del cuadro actual.

```

1: matches ← []
2: for all f0 en features imagen izquierda anterior do
3:   f1 ← BUSCARMATCHEN(f0, imgDerechaAnterior)
4:   f2 ← BUSCARMATCHEN(f1, imgDerechaActual)
5:   f3 ← BUSCARMATCHEN(f2, imgIzquierdaActual)
6:   f4 ← BUSCARMATCHEN(f3, imgIzquierdaAnterior)
7:   if f0 = f4 then
8:     matches ← matches + +[(f0, f1, f2, f3)]
9:   end if
10: end for
```

Algoritmo 1: Quad-matching

LIBVISO2 procesa en primer lugar las listas de **features** “dispersos”. Para minimizar los tiempos de ejecución se restringen las áreas de búsqueda para cada **feature** de la siguiente manera:

- Si se buscan coincidencias para el **feature** *f* de una imagen A, en otra imagen B, solo se consideran **features** de B dentro del área [*f.x* − *match_radius*, *f.x* + *match_radius*]x[*f.y* − *match_radius*, *f.y* + *match_radius*].
- Si se buscan coincidencias para el **feature** *f* de una imagen A, en otra imagen B y ambas pertenecen al mismo cuadro estéreo, solo se consideran **features** de B dentro del área [*f.x* − *match_radius*, *f.x* + *match_radius*]x[*f.y* − *match_disp_tolerance*, *f.y* + *match_disp_tolerance*].

Los valores *match_radius* y *match_disp_tolerance* son configurables.

Esta operatoria es acelerada agrupando los **features** de cada imagen en **buckets** según su posición, esto permite minimizar las comparaciones necesarias en la búsqueda de coincidencias.

Finalmente los **outliers** son removidos y las coordenadas de las coincidencias halladas son refinadas, para esto se utilizan los filtros **Sobel** que son entrada del componente.

El procesamiento de los **features** “densos” es similar, pero los rangos de búsqueda se restringen aun más, utilizando la información obtenida del matching de **features** “dispersos”.

Más detalles sobre el funcionamiento de este componente pueden encontrarse en la publicación de Geiger, Ziegler y Stiller 2011.

4.1.5. Estimación de pose

El componente “Estimador de rotación y traslación” se encarga de estimar cuál fue la rotación y traslación de la cámara izquierda entre un par de cuadros estéreo, c0 y c1.

22CAPÍTULO 4. ANÁLISIS E IMPLEMENTACIÓN DE LA SOLUCIÓN EN UNA COMPUTADORA DE

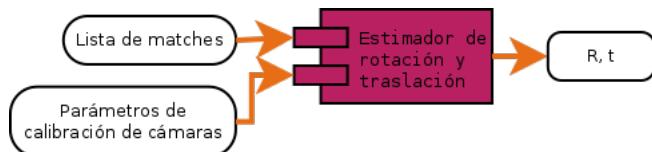


Figura 4.7: Entradas y salidas del componente.

Para esto se utiliza **RANSAC**, construyendo varias estimaciones de R y t a partir de 3 elementos de la lista de coincidencias y seleccionando la que minimiza la cantidad de **outliers**.

Para determinar si un elemento de la lista de coincidencias es **outlier** se triangula su posición en el cuadro c_0 obteniendo un punto 3D p_0 . Luego a p_0 se le aplica la transformación $p_1 = Rp_0 + t$.

Finalmente p_1 se proyecta en las imágenes izquierda y derecha de c_1 .

El elemento es **outlier** si la suma de las distancias entre su posición real en ambas imágenes y las posiciones proyectadas superan cierta cota configurable.

4.2. Pruebas en procesador de propósito general

Siguiendo lo indicado por la metodología, antes de trabajar sobre el **AP SoC**, probamos la solución en un procesador de propósito general. Más específicamente en una computadora de escritorio con arquitectura *x86* y sistema operativo *Debian GNU/Linux*.

Como explicamos en el capítulo anterior en [3.2.2](#) y [3.2.3](#), realizamos las siguientes modificaciones a *LIBVISO2*:

- Lectura de datos de calibración y configuración en tiempo de ejecución.
- Simulación de pérdida de cuadros.
- Reemplazo del sistema de construcción `cmake` por `make`.

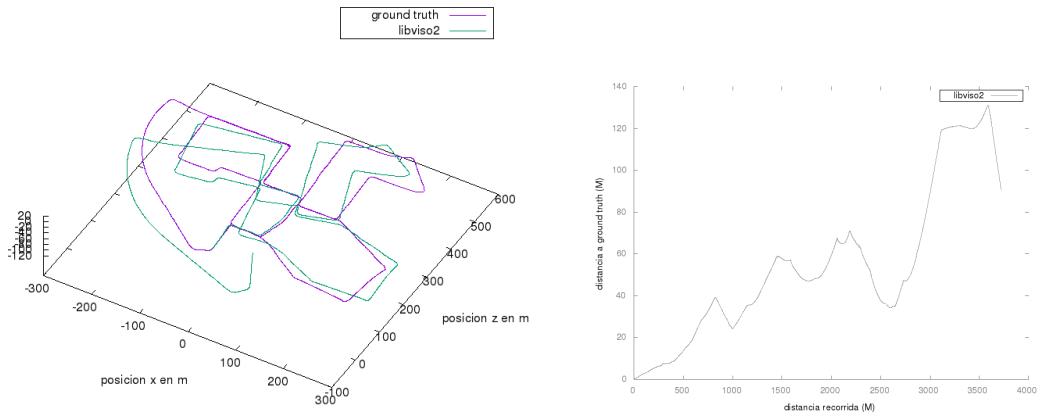
A continuación, probamos la implementación resultante. Las pruebas consistieron en procesar secuencias de imágenes del dataset KITTI para odometría visual y comparar las trayectorias obtenidas con las verdaderas (*“ground truth”*), provistas en el mismo dataset.

De esta forma comprobamos que el funcionamiento de la implementación fuera correcto.

Creamos un programa `run_test_case` para facilitar esta tarea. El mismo recibe como argumento un archivo con la definición de un caso de prueba en formato JSON, lo procesa y crea una gráfica con el resultado.

```
{
  "name": "libviso2_sequence00",
  "sequence": "00",
  "framesN": 999999,
  "ground_truth": true,
  "runs": [
    {
      "name": "libviso2",
      "nms_n": 3,
      "nms_tau": 50,
      "match_binsize": 50,
      "match_radius": 200,
      "ransac_iters": 200,
      "fps": 10
    }
  ]
}
```

(a) Ej: Definición del caso de prueba



(b) Trayectorias real y estimada.

(c) Distancia entre la trayectoria estimada y la real.

Figura 4.8: Caso de prueba para la secuencia 00 del dataset

Los resultados completos de las pruebas realizadas pueden consultarse en el apéndice `apendice-resultados-cpu-gral.tar.gz`.

Llamamos `libviso_gp` a la solución construida en la computadora de propósito general. Su código fuente² sirve como referencia y punto de partida para las siguientes etapas del desarrollo.

²El código fuente se encuentra en el apéndice `libviso2_gp.tar.gz`.

Capítulo 5

Primera implementación de la solución en el AP SoC

Este capítulo presenta el proceso llevado a cabo para migrar la solución de la computadora de propósito general al **AP SoC**. Esta primera implementación fue realizada totalmente en software, haciendo uso solo del **PS** y sin usar lógica programable.

La implementación del software y la plataforma hardware pueden consultarse en el `system.tar.gz`.

5.1. Relevamiento de ZYBO

Antes de comenzar a portar la solución al **AP SoC**, realizamos un relevamiento de las principales características técnicas de la placa de desarrollo *ZYBO*, seleccionada para este trabajo. El objetivo fue detectar limitaciones y obtener información que nos permitiera aprovechar mejor los recursos disponibles durante el resto del trabajo.

Esta placa, fabricada por *Digilent*, integra un **AP SoC** de la familia *Zynq-7000* (*Zynq 7010*) de *Xilinx*, un puerto Ethernet de 1GB, 512MB de memoria RAM DDR3 y UART sobre USB, entre otros periféricos.

El **AP SoC** cuenta con un procesador *ARM Cortex-A9* de doble núcleo, controlador de memoria DDR3 y lógica programable equivalente a una **FPGA Artix-7**.

La comunicación entre **IP Cores** instanciados en **PL** y el **PS** se realiza a través de buses *AMBA*¹ compartidos.

¹Xilinx 2011.

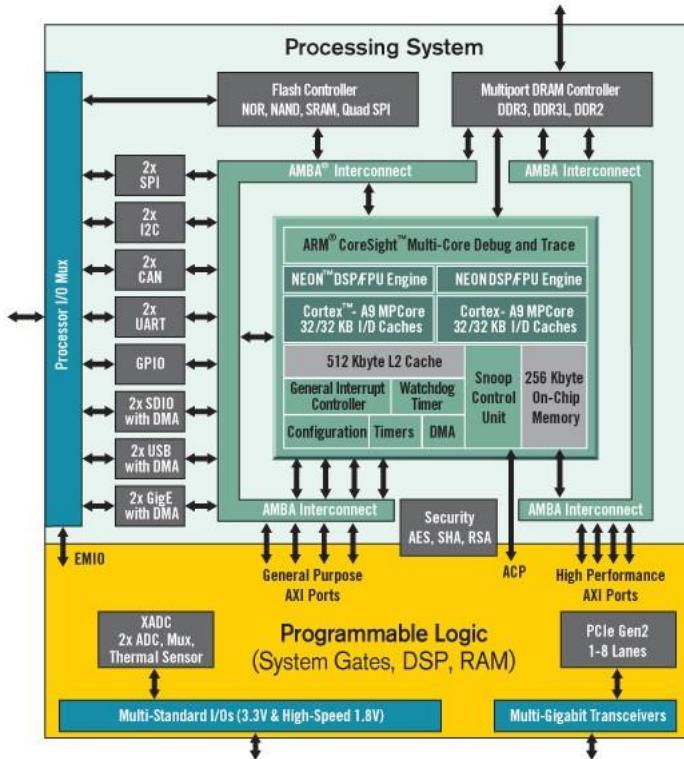


Figura 5.1: Comunicación PS/PL en Zynq 7010.

Como se ve en la figura, existen distintos puertos para la interconexión de **PL** y **PS**:

- High Performance AXI: Permite la transferencia de datos de alta velocidad desde y hacia memoria externa. Provee acceso a un bus exclusivo entre **PL** y el controlador de memoria RAM.
- General Purpose AXI: Permite a los procesadores de **PS** enviar ordenes y pequeños paquetes de datos a **IP Cores** instanciados en **PL**.
- ACP: Provee acceso desde **PL** al mismo universo de direcciones que accede **PS**, compartiendo su cache L2. Al utilizar el mismo cache se garantiza la coherencia.
- EMIO: Lectura y escritura de puertos de E/S multiplexada.

De este relevamiento surge que para aprovechar los recursos de la placa Zybo el software debe distribuir el trabajo entre ambos núcleos del procesador. Además los **IP Cores** dedicados a aceleración de funciones deben conectarse a **PS** a través de los puertos que provean el mejor rendimiento.

5.2. Construcción de la plataforma de hardware

Tal como indica la metodología, configuramos la plataforma de hardware para que sea posible portar la solución implementada en el procesador de propósito general a la placa de desarrollo.

En nuestra implementación de la solución sobre *Zybo* decidimos utilizar ethernet para la transferencia de imágenes a la placa y UART para el control

Para esto creamos un nuevo proyecto en *Vivado IP Integrator*, al que llamamos “**system**”. Dentro del mismo creamos un diseño de bloques al que le agregamos el bloque *ZYNQ7 Processing System (PS)*. Configuramos el **PS** según las especificaciones de *Zybo* y habilitamos solo los periféricos que planeamos utilizar en la solución: ethernet y UART.

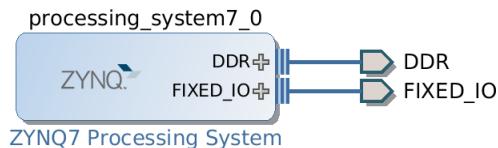


Figura 5.2: Vista del diseño de bloques en Vivado.

Finalmente exportamos el hardware para poder ser utilizado en las herramientas de desarrollo de software. La plataforma construida solo hace uso de **PS** y no de **PL**, por lo tanto las soluciones que corren sobre ella deben implementar la totalidad de su funcionalidad en software.

5.3. Construcción de la plataforma de software

Siguiendo la metodología, a continuación configuramos la plataforma de software.

El punto principal en este apartado fue la selección del sistema operativo (**SO**) a usar. Si bien el uso de un SO no es estrictamente necesario, utilizar uno permite simplificar las tareas de administración de recursos.

Analizamos tres alternativas:

- *standalone*: es una pequeña capa de software de bajo nivel provista por *Xilinx SDK*. Proporciona controladores básicos para los dispositivos y funciones para facilitar el uso de algunas características del procesador, como interrupciones y cache.
Si bien no es un SO completo es útil para realizar pruebas básicas del hardware.
- *FreeRTOS*: Este sistemas operativo de tiempo real proporciona un ambiente multi-tarea y primitivas de sincronización. No incluye controladores para dispositivos, pero se pueden utilizar los generados para *standalone*.
- *PetaLinux*: Es una distribución de *GNU/Linux* desarrollada por *Xilinx* específicamente para sus productos. El kernel de *Linux* proporciona soporte para **SMP** y threads a nivel kernel. A diferencia de las dos alternativas anteriores, utiliza memoria virtual y separa la ejecución en dos dominios, usuario y kernel.

Si bien las alternativas *standalone* y *FreeRTOS* ofrecen un consumo de recursos muy pequeño, decidimos utilizar *PetaLinux* basados en las siguientes razones:

- Soporte nativo para **SMP**, lo que simplifica la utilización de los dos núcleos del procesador.
- Posibilidad de utilizar las mismas herramientas que en la computadora de propósito general.

28 CAPÍTULO 5. PRIMERA IMPLEMENTACIÓN DE LA SOLUCIÓN EN EL AP SOC

- Sistemas de archivos estándar.
- Amplia documentación disponible.

PetaLinux integra un kit de desarrollo llamado *PetaLinux SDK*. El mismo facilita el desarrollo, construcción, prueba y deploy de las soluciones a la placa. Las principales herramientas de este **SDK** son:

- *petalinux-create*: Creación de proyectos, aplicaciones de usuario y módulos del kernel a partir de plantillas.
- *petalinux-config*: Configuración del kernel, activación/desactivación de aplicaciones y módulos.
- *petalinux-build*: Compilación de componentes y construcción de la imagen del kernel y el sistema de archivos raíz.
- *petalinux-boot*: Boot de la solución en el emulador *qemu* o en la placa de desarrollo.

El primer paso fue crear el proyecto usando la herramienta “*petalinux-create*”:

```
~$ petalinux-create -t project --template zynq --name system
```

A continuación, importamos la plataforma de hardware creada en la sección anterior y verificamos que la configuración del sistema fuera correcta:

```
~/system$ petalinux-config --get-hw-description=hardware/system/system.sdk
```

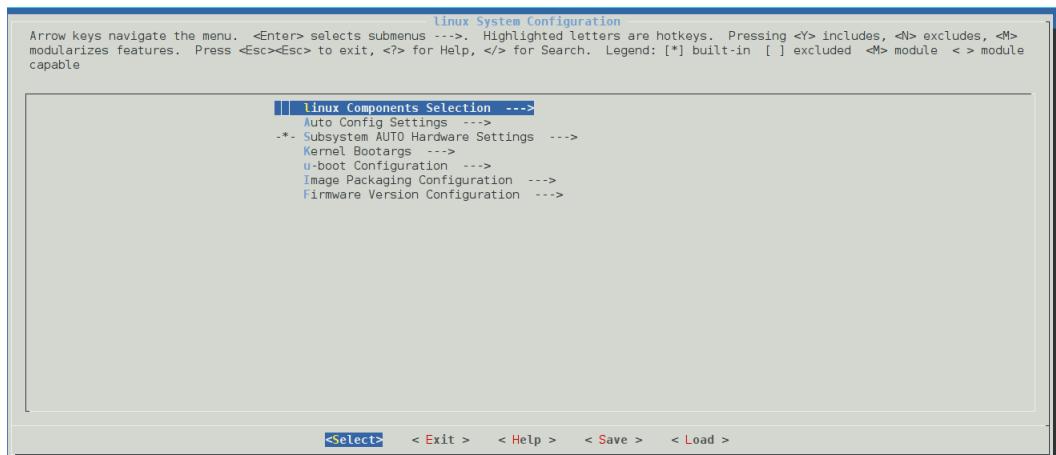


Figura 5.3: Pantalla de configuración de PetaLinux.

Finalmente construimos la plataforma de software:

```
~/system$ petalinux-build
```

A continuación, probamos la plataforma construida en el emulador *qemu*:

```
~/system$ petalinux-boot --qemu --kernel
```

Luego creamos un **shell script** boot.sh para cargar y ejecutar el software en la placa Zybo. El script utiliza *Xilinx Microprocessor Debugger* (xmd) para transferir datos desde una computadora a la memoria RAM de la placa y controlar sus procesadores.

```
#!/bin/sh

xmd <<EOF
connect arm hw

init_user
sleep 1
target 64
mcr p15 0 c1 c0 0 0x08c5187a
dow "images/linux/zynq_fsbl.elf"
con

sleep 1; stop
target 64
mcr p15 0 c1 c0 0 0x08c5187a
dow "images/linux/u-boot.elf"
con

sleep 1; stop
target 64
mcr p15 0 c1 c0 0 0x08c5187a
dow -data "images/linux/system.dtb" 0x15408000
rwr r2 0x15408000
target 64
mcr p15 0 c1 c0 0 0x08c5187a
dow -data "images/linux/zImage" 0x00008000
rwr pc 0x00008000
con

EOF
```

Al correr el script y conectarse mediante un cliente de terminal serie al puerto UART de la placa² se puede observar la siguiente salida:

```
Stopping Bootlog daemon: bootlogd.

Built with PetaLinux v2014.2 (Yocto 1.6) system /dev/ttys0
system login:
```

²En Linux suele ser /dev/ttysUSB1

30 CAPÍTULO 5. PRIMERA IMPLEMENTACIÓN DE LA SOLUCIÓN EN EL AP SOC

La plataforma hasta aquí construida es funcional, es posible iniciar sesión con el usuario root, ejecutar comandos y acceder a la red ethernet.

5.4. Migración del software al APSoC

Está sección describe el trabajo realizado para portar y probar la implementación `libviso_gp` realizada en la sección 4.2 a la placa Zybo. Siguiendo lo propuesto por la metodología, en esta primera instancia implementamos la solución totalmente en software, haciendo uso solo del **PS** del **AP SoC**.

5.4.1. Migración de `libviso_gp` al APSoC

El primer paso fue crear una nueva aplicación dentro del proyecto utilizando las herramientas del **SDK** de *PetaLinux*:

```
~/system$ petalinux-create -t apps -n viso --template c++ --enable
```

A continuación, copiamos el código fuente de `libviso_gp` al directorio recién creado para la aplicación: `system/components/apps/viso`. Luego realizamos las modificaciones necesarias para obtener una implementación compatible con la plataforma anteriormente construida.

`libviso_gp` utiliza instrucciones *Streaming SIMD Extensions (SSE)* a través de la biblioteca `emmintrin.h`. Estas instrucciones no están disponibles en el procesador ARM Cortex-A9 del **PS**. Por lo tanto, reemplazamos el uso de las mismas por instrucciones Neon³ equivalentes.

Para esto utilizamos la biblioteca `SSE2NEON.h` para C++, que mantiene la misma interfaz provista por `emmintrin.h` e internamente usa instrucciones Neon en lugar de **SSE**. De esta forma no fue necesario realizar grandes cambios en el código que hacia uso de funciones pertenecientes a `emmintrin.h`.

El otro cambio que debimos llevar a cabo fue eliminar la dependencia de `libpng` para la carga de imágenes.

Con este objetivo modificamos el código original para que trabaje con archivos de imagen cuyo único contenido son los valores de cada píxel, sin utilizar compresión (llamamos a este formato **raw**). Esta decisión apuntó, también, a que el costo de trabajar con archivos **PNG** no impacte en las mediciones posteriores.

La funcionalidad de carga de imágenes fue implementada en un nuevo archivo `raw_image.c` y utilizada en `demo.cpp`.

Posteriormente modificamos el **Makefile** generado por `petalinux-create` para que incluyera todos los archivos de la solución en la compilación y construyera e instalara dos binarios en el sistema de archivos raíz, `viso` y `viso_profile`. El segundo fue construido activando las opciones de **profiling** del compilador y el **linker**, de esta forma, al ejecutarlo se genera información de **profiling** que puede ser posteriormente analizada.

Finalmente, construimos la aplicación:

```
~/system$ petalinux-build -c rootfs/viso
```

³Neon es un juego de instrucciones SIMD de propósito general para procesadores ARM.
<http://www.arm.com/products/processors/technologies/neon.php>

y empaquetamos el proyecto para actualizar el sistema de archivos raíz:

```
~/system$ petalinux-build -x package
```

5.4.2. Acceso a datasets desde ZYBO

Como explicamos en el apartado anterior, reemplazamos el uso de archivos **PNG** por imágenes **raw**. Para facilitar esta tarea creamos un programa **kitti2raw**⁴ para convertir secuencias de imágenes del dataset KITTI para odometría visual a formato **raw**.

Por ejemplo para convertir una secuencia de imágenes del directorio 00 a **raw** y guardarla en el directorio **00.raw** hay que invocar a **kitti2raw** de la siguiente forma:

```
~/datasets/kitti-odo/sequences$ kitti2raw 00 00.raw
```

Dado que las secuencia de imágenes se encuentran almacenadas en nuestra **PC** de desarrollo, configuramos en la misma un servidor de archivos **NFS** para que los datos pudieran ser accedidos desde la placa **Zybo**, que no cuenta con grandes medios de almacenamiento.

Para esto armamos una red ethernet entre la PC de desarrollo y la placa:

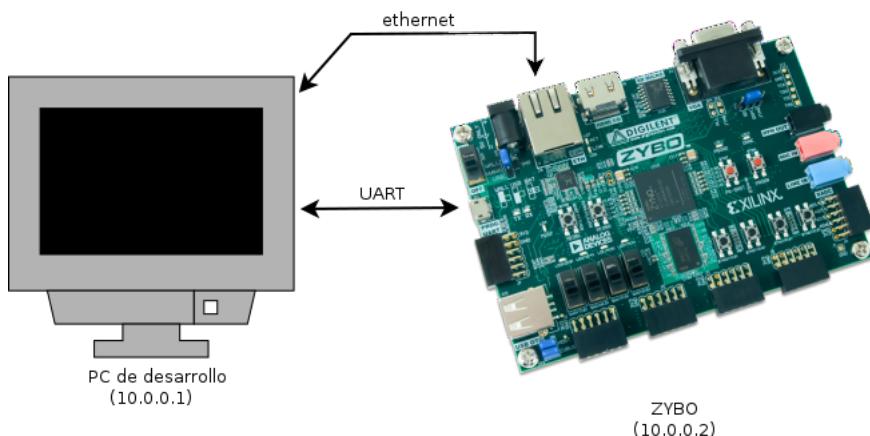


Figura 5.4: Interconexiones ethernet y UART entre PC de desarrollo y placa ZYBO.

En la PC configuramos el servidor **NFS** para que compartiera el directorio **/var/nfs**:

```
/var/nfs      10.0.0.2(rw,sync,no_subtree_check,no_root_squash,insecure)
```

Utilizando el **SDK** de *PetaLinux* creamos una nueva aplicación **myconfig** que instala los archivos necesarios para configurar la red y montar el sistema de archivos **NFS** en la solución.

⁴Este programa y otros programas auxiliares se encuentran en el apéndice **utils.tar.gz**

32CAPÍTULO 5. PRIMERA IMPLEMENTACIÓN DE LA SOLUCIÓN EN EL AP SOC

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
    address 10.0.0.2
    netmask 255.255.255.0
    network 10.0.0.0
    gateway 10.0.0.1
```

Figura 5.5: Configuración de la red en ZYBO: /etc/network/interfaces.

```
10.0.0.1:/var/nfs /media/nfs nfs
```

Figura 5.6: Configuración del punto de montaje en ZYBO: /etc/fstab.

Luego construimos y reempaquetamos la solución:

```
~/system$ petalinux-build -c rootfs/myconfig
~/system$ petalinux-build -x package
```

Cargamos la solución en la placa, con el script `boot.sh` y verificamos que el sistema de archivos **NFS** se hubiera montado correctamente:

```
root@system:~# mount
.
.
.
10.0.0.1:/var/nfs on /media/nfs type nfs (rw,relatime,vers=3,
    rsize=1048576,wsize=1048576,namlen=255,hard,proto=tcp,port=2049,
    timeo=70,retrans=3,sec=sys,local_lock=none,addr=10.0.0.1)
```

5.4.3. Pruebas de la solución en ZYBO

Al igual que en la implementación en la computadora de propósito general, creamos un programa `run_test_case_remote`⁵ para facilitar las pruebas de la solución en la placa Zybo.

El uso de `run_test_case_remote` es similar al de `run_test_case` de la sección 4.2, pero hace uso de **SSH** para ejecutar los casos de prueba de manera remota en la solución desde la PC.

Desde la configuración de *PetaLinux* habilitamos **dropbear**, un servidor **SSH**.

⁵run_test_case_remote se encuentra en el apéndice `system.tar.gz`

```
~/system$ petalinux-config -c rootfs
```

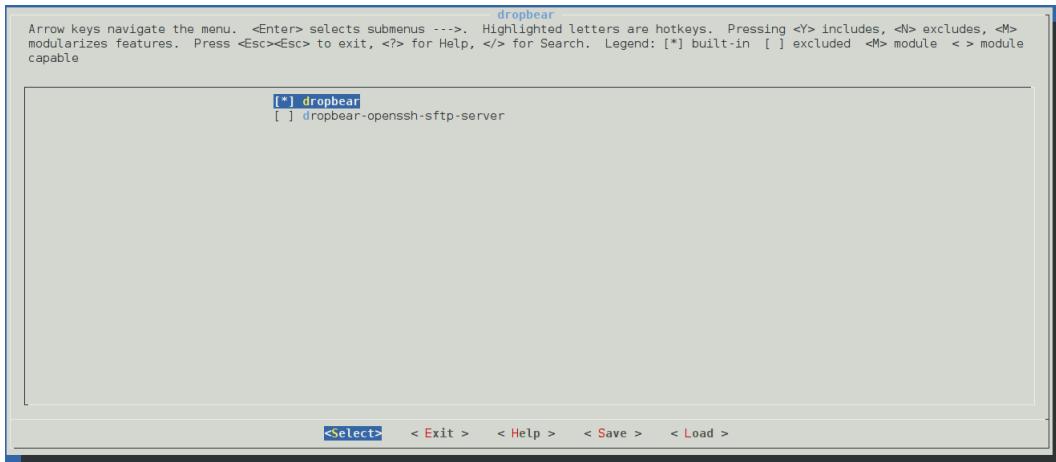


Figura 5.7: Pantalla de configuración de PetaLinux para habilitar dropbear.

A continuación, modificamos la aplicación `myconfig`, creada anteriormente, para instalar la llave pública de la PC en `/home/root/.ssh/authorized_keys` y la llave **RSA** de *Zybo* en `/etc/dropbear/dropbear_rsa_host_key`. De esta forma la ejecución de comandos en *Zybo* desde la PC no requiere ingreso contraseñas, lo que permitió correr los casos de prueba de manera desatendida.

Luego construimos la aplicación nuevamente y reempaquetamos la solución:

```
~/system$ petalinux-build -c rootfs/myconfig
~/system$ petalinux-build -x package
```

Con la plataforma ya lista para realizar las pruebas, el siguiente paso fue convertir todas las secuencias de imágenes del dataset KITTI para odometría visual a formato **raw**, guardando las secuencias convertidas en `/var/nfs`.

Por ejemplo, la secuencia 00 fue convertida de la siguiente forma:

```
~/datasets/kitti-odo/sequences$ kitti2raw 00 /var/nfs/00
```

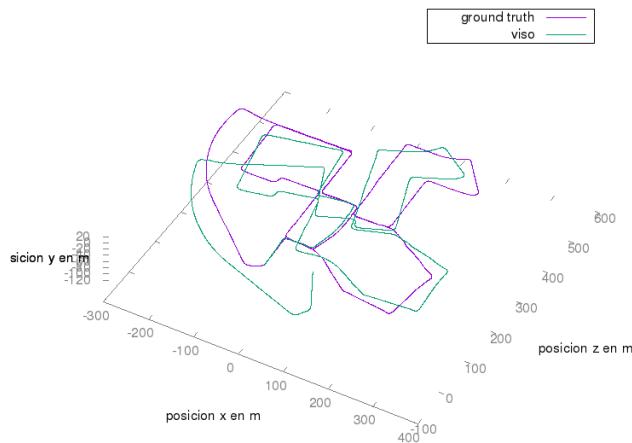
También copiamos los datos de *ground truth* en `/var/nfs/poses`, pues son usados por `run_test_case_remote` para comparar las trayectorias estimadas con las reales.

Luego escribimos los casos de prueba y posteriormente los corrimos.

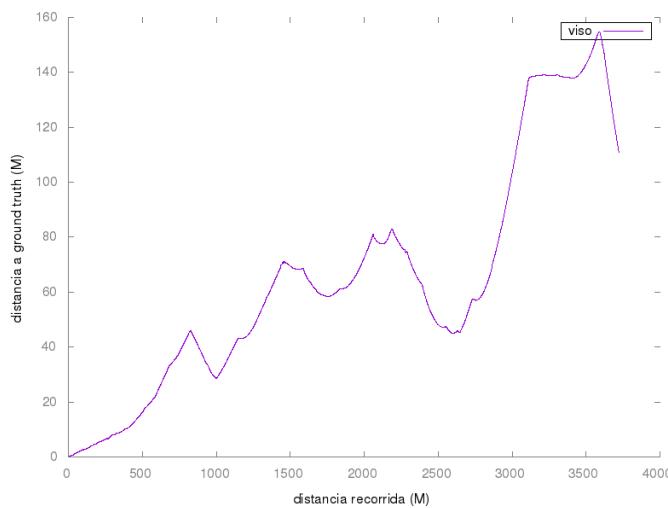
34 CAPÍTULO 5. PRIMERA IMPLEMENTACIÓN DE LA SOLUCIÓN EN EL AP SOC

```
{
    "name": "00",
    "sequence": "00",
    "framesN": 999999,
    "ground_truth": true,
    "runs": [
        {
            "cmd": "viso",
            "name": "viso",
            "nms_n": 5,
            "nms_tau": 60,
            "match_binsize": 20,
            "match_radius": 150,
            "ransac_iters": 100,
            "fps": 10
        }
    ]
}
```

(a) Caso de prueba para la secuencia 00 procesada en Zybo.



(b) Gráfica de trayectoria generada por viso



(c) Distancia entre la trayectoria estimada y la real

Figura 5.8: Caso de prueba de ejemplo ejecutable de forma remota

Los resultados completos de las pruebas realizadas pueden consultarse en el apéndice `apendice-resultados-viso.tar.gz`.

5.5. Medición de rendimiento

En esta sección medimos el rendimiento de la solución construida. Para esto tomamos los tiempos de procesamiento para las distintas secuencias de imágenes del dataset en *Zybo*:

Secuencia	Cuadros	Tiempo (s)	$\sim \frac{\text{cuadros}}{\text{segundo}}$
00	4541	2350	1,93
01	1101	564,5	1,95
02	4661	2402	1,94
03	801	415	1,93
04	271	138	1,95
05	2761	1423	1,94
06	1101	565	1,95
07	1101	567	1,94
08	4071	2109	1,93
09	1591	824	1,93
10	1201	622	1,93

Cuadro 5.1: Mediciones de rendimiento de la solución en Zybo

Se puede observar que la tasa de $\frac{\text{cuadros}}{\text{segundo}}$ es casi totalmente independiente de la secuencia del dataset procesada.

A continuación medimos los tiempos de procesamiento sin considerar los costos del uso de **Ethernet** y **NFS**. Para esto modificamos `run_test_case_remote` agregando la opción de copiar la secuencia de imágenes al disco RAM en *Zybo* antes de procesarla.

Dado que el disco RAM es de 256MB, creamos una nueva secuencia de imágenes más corta, utilizando los cien primeros cuadros de la secuencia 00 del dataset. Llamamos a esta secuencia de imágenes: “`minitest`”.

De esta forma obtuvimos el siguiente resultado:

Secuencia	Cuadros	Tiempo (s)	$\sim \frac{\text{cuadros}}{\text{segundo}}$
minitest	100	39,16	2,55

Cuadro 5.2: Medición de rendimiento de la solución en Zybo con datos en disco RAM

Se observa que, aun ignorando los costos de la transferencia de datos desde la PC a *Zybo*, la tasa de procesamiento es de $\sim 2,55 \frac{\text{cuadros}}{\text{segundo}}$. Este rendimiento según el análisis previamente realizado en [3.2.3](#) no permite estimar trayectorias cercanas a las reales.

En el próximo capítulo aplicaremos técnicas de optimización software en pos de alcanzar un mejor rendimiento.

Capítulo 6

Optimización por software

Este capítulo presenta las optimizaciones por software aplicadas a `viso`, la solución construida en el capítulo 5, para obtener una nueva solución `viso_s` con mejor rendimiento en el **AP SoC**.

En la sección 6.1 realizamos un análisis de los parámetros configurables de *LIBVISO2* y su impacto en la calidad del resultado, tolerancia a la pérdida de cuadros y el rendimiento.

A continuación, en la sección 6.2 realizamos un análisis detallado del rendimiento de la solución en *Zybo*, calculando para cada componente la fracción del tiempo total de procesamiento utilizada.

Luego, utilizando la información obtenida, la sección 6.3 presenta el plan para acelerar `viso` y crear la nueva solución `viso_s`. En esta sección los puntos 6.3.1, 6.3.2 y 6.3.3 detallan las optimizaciones aplicadas a los distintos grupos de componentes.

En la sección 6.4 llevamos a cabo pruebas funcionales sobre `viso_s`.

Finalmente en la sección 6.5 medimos el rendimiento de la solución optimizada y lo comparamos con la anterior.

6.1. Análisis del impacto de los parámetros

En esta sección realizamos un análisis de los parámetros configurables de *LIBVISO2*. Los mismos permite variar el rendimiento, la calidad del resultado y la tolerancia a la pérdida de cuadros.

6.1.1. Presentación de los parámetros

- **nms_n:** Tamaño de la ventana utilizada en **NMS** para la detección de máximos y mínimos densos. Para los dispersos se utiliza una ventana de tamaño $\max(3 * nms_n, 10)$.
- **nms_tau:** Cota usada en **NMS** para la detección de máximos y mínimos, tal como se explicó en 3.2.1. Variarla permiten detectar más o menos puntos.
- **match_binsize:** El algoritmo de búsqueda de coincidencias entre **features** los agrupa en “baldes” según su posición. Este parámetro determina el ancho y alto de los “baldes”.

- **match_radius:** Máxima distancia en píxeles para la búsqueda de coincidencias de **features**. Aumentar este valor hace más probable encontrar coincidencias entre features de distintas imágenes, ya que amplía el rango de búsqueda.
- **ransac_iters:** Cantidad de iteraciones **RANSAC** para estimar la rotación y traslación entre cuadros.

Planteamos los siguientes juegos de parámetros para realizar nuestros análisis:

configuración	nms_n	nms_tau	match_binsize	match_radius	ransac_iters
LIBviso2 Default	3	50	50	200	200
+nms_n	5	50	50	200	200
+tau	3	80	50	200	200
-tau	3	20	50	200	200
+binsize	3	50	70	200	200
-binsize	3	50	20	200	200
+radius	3	50	50	400	200
-radius	3	50	50	100	200
+ransac	3	50	50	200	300
-ransac	3	50	50	200	100
adhoc	5	60	20	150	100

Cuadro 6.1: Juegos de parámetros utilizados en los análisis.

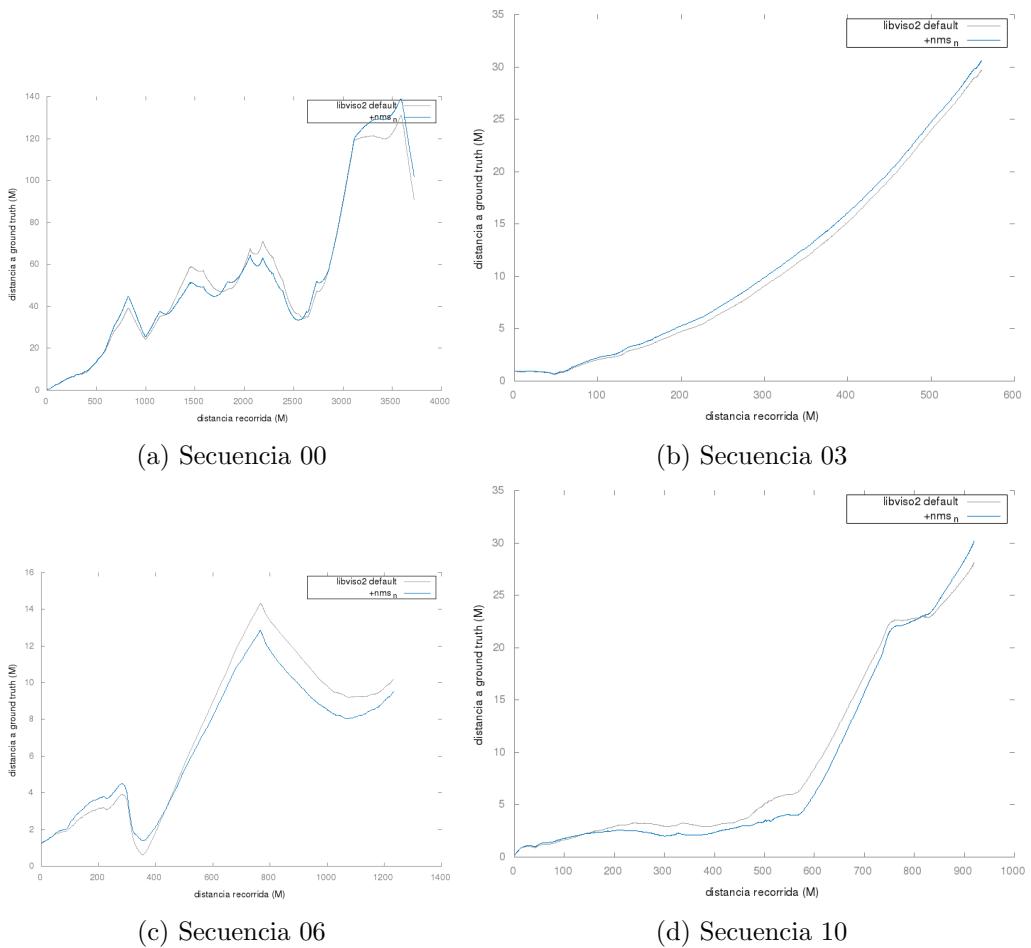
La configuración **ad hoc** fue hallada empíricamente y es utilizada en las etapas posteriores de este trabajo.

6.1.2. Calidad del resultado

A continuación evaluamos el impacto de la variación de cada uno de los parámetros en la calidad del resultado. Para esto estimamos la trayectoria para cada secuencia de imágenes del dataset KITTI utilizando cada una de las configuraciones de la tabla 6.1. Posteriormente medimos la distancia a **ground truth** de las trayectorias estimadas.

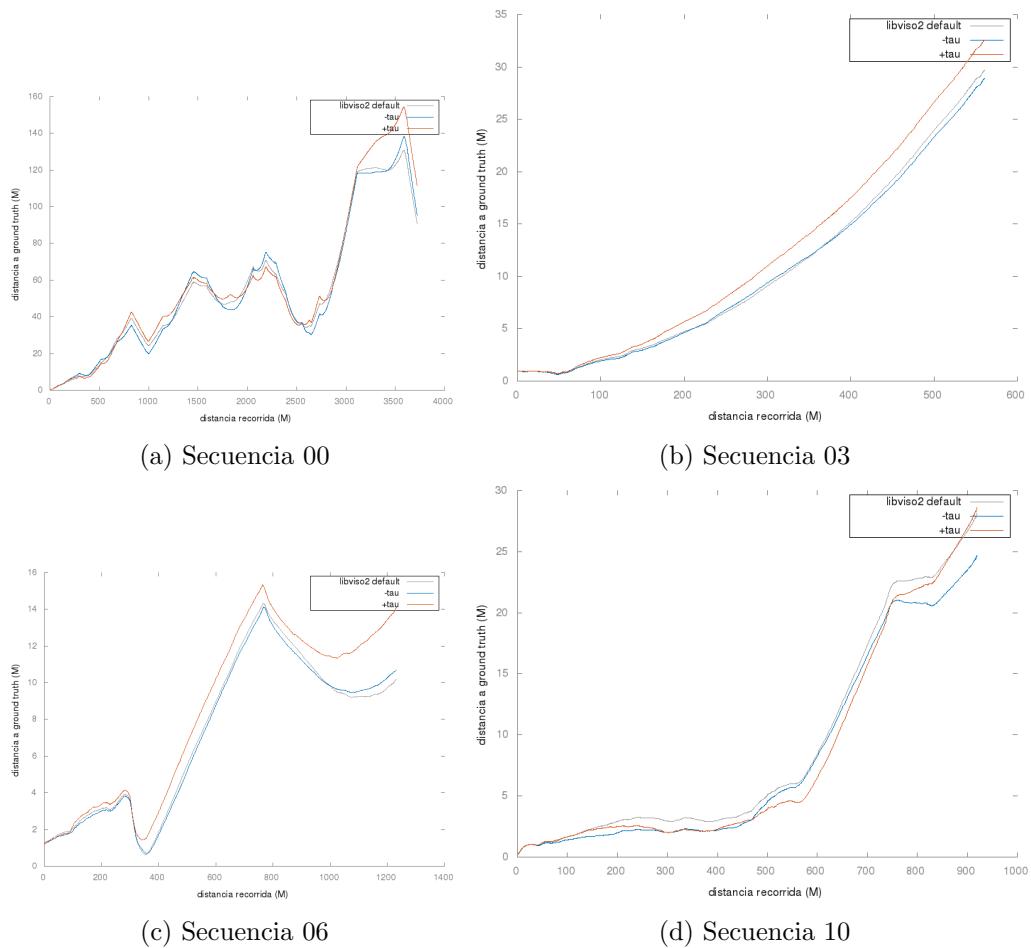
A continuación presentamos los resultados del impacto de variar cada parámetro para las secuencias “00”, “03”, “06” y “10”. Los resultados para todas las secuencias se pueden consultar en el apéndice `estudio-calidad-parametros.tar.gz`.

Análisis del parámetro **nms_n**



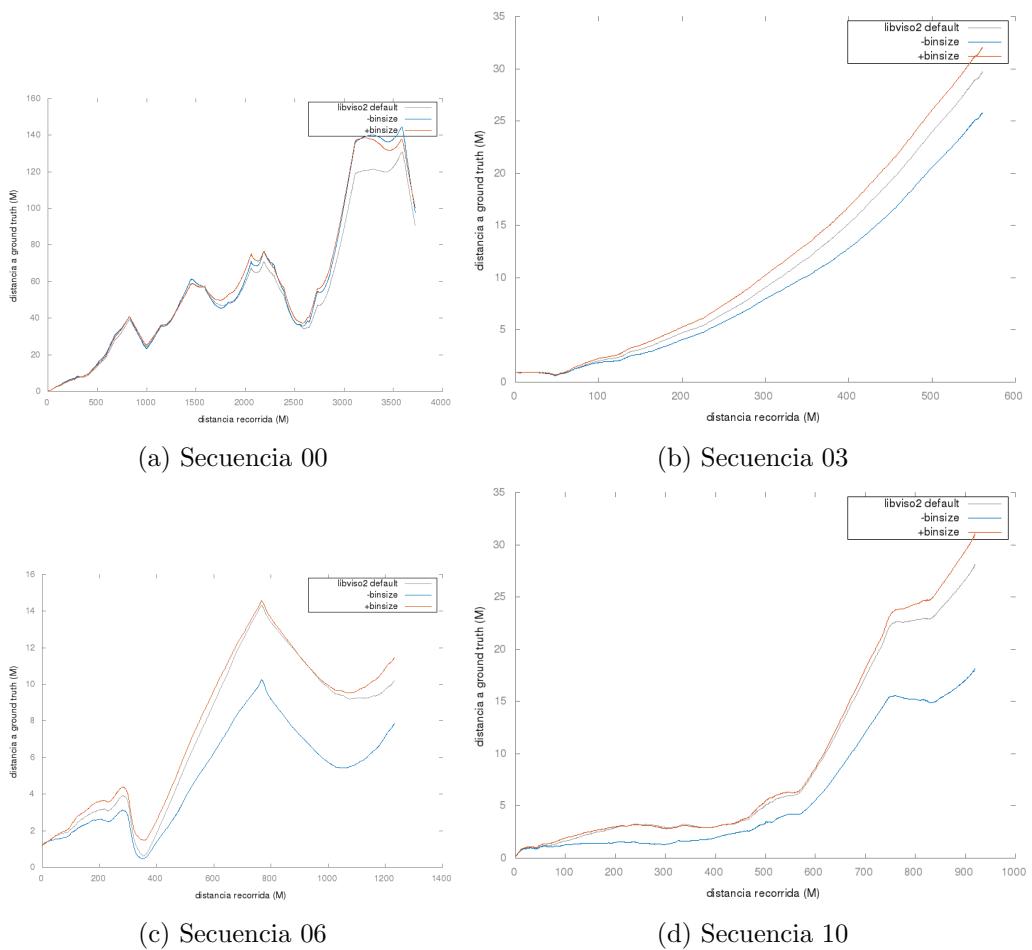
Se observa que en algunos casos incrementar **nms_n**, y por lo tanto detectar en cada imagen menos **features**, pero más relevantes, mejora la precisión en la estimación de trayectorias. Sin embargo, en otros casos la precisión mejora con valores de **nms_n** más pequeños, trabajando de esta forma con más **features**.

Análisis del parámetro nms_tau



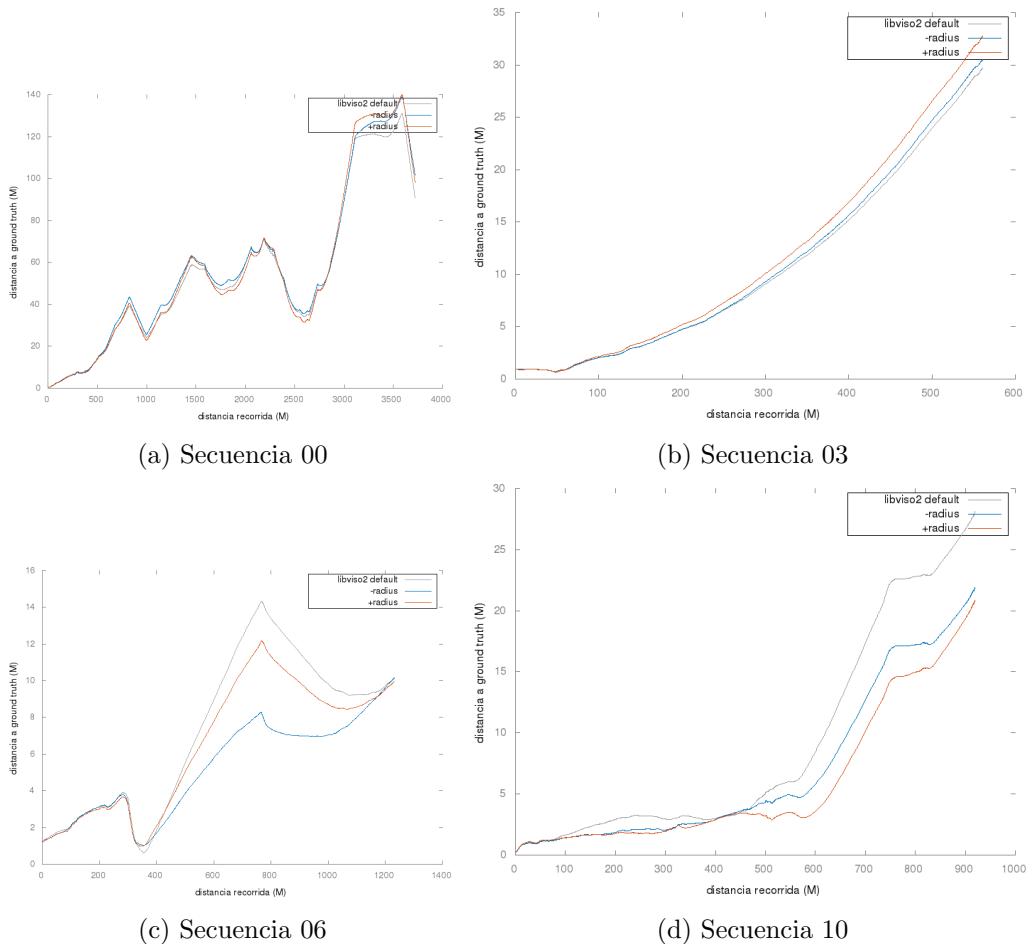
Se observa que en algunos casos reducir **nms_tau**, y por lo tanto detectar más **features** en cada imagen, mejora la precisión en la estimación de trayectorias. Sin embargo, en otros casos la precisión mejora al incrementar **nms_tau**, trabajando de esta forma con los **features** más sobresalientes.

Análisis del parámetro `match_binsize`



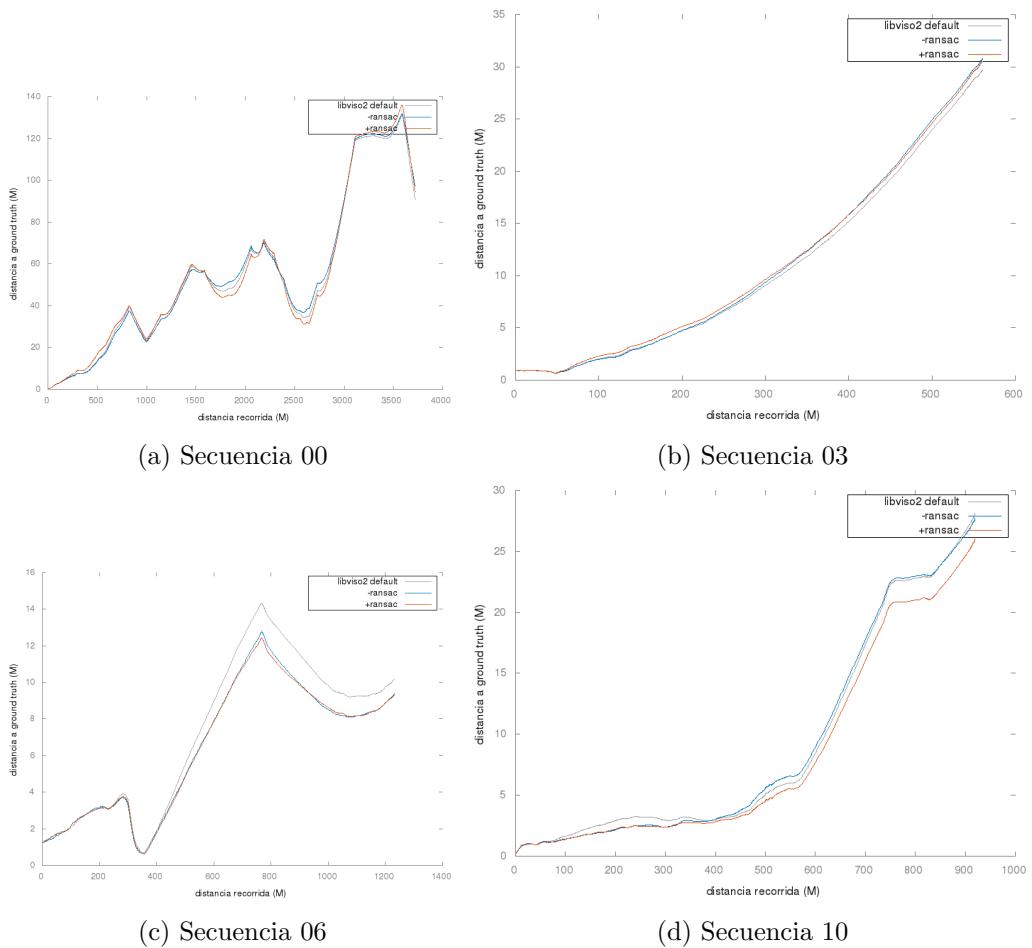
En la mayoría de los casos al reducir `match_binsize` se obtiene una mejora en la precisión de la estimación de trayectorias.

Análisis del parámetro `match_radius`



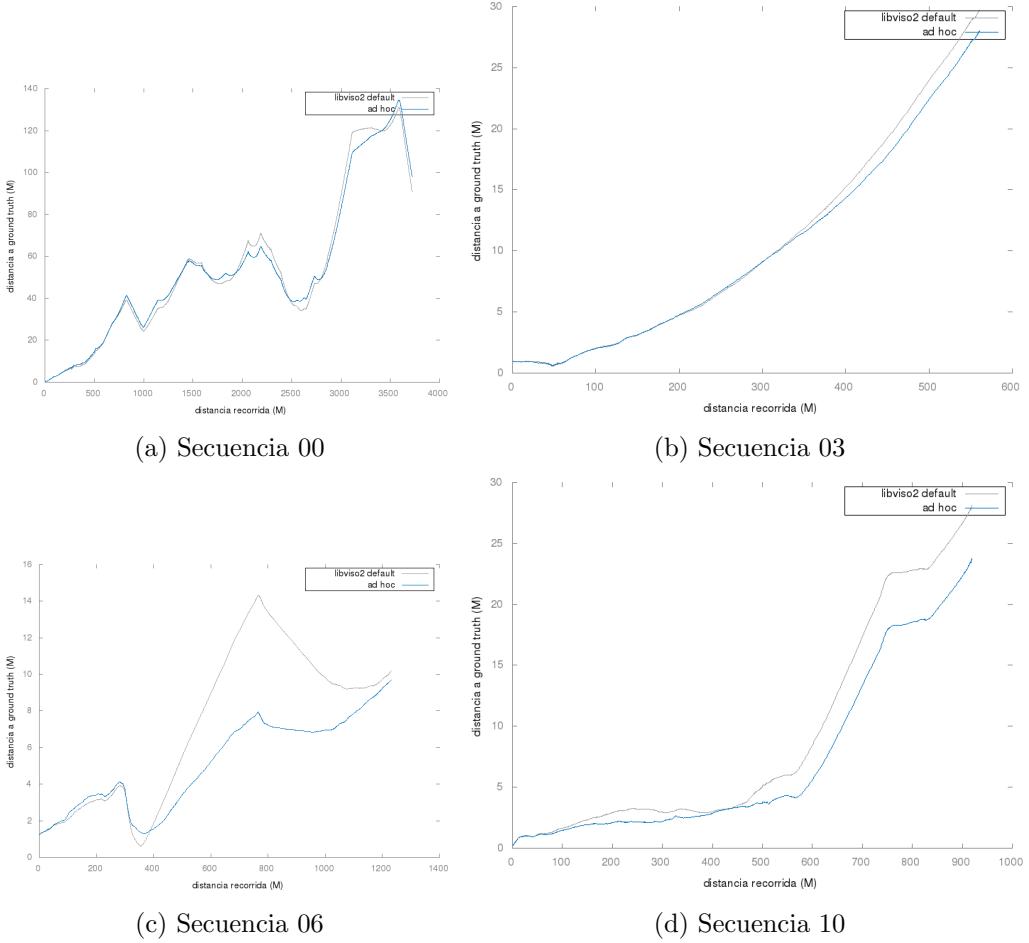
Se observa que en algunos casos incrementar `match_radius` y, por lo tanto, agrandar los rangos de búsqueda de coincidencias para cada `feature` provee una pequeña mejora en la precisión de la estimación de trayectorias. Se puede ver que reducir el valor del parámetro puede afectar negativamente a la precisión, como ocurre al estimar la trayectoria de la secuencia “06”.

Análisis del parámetro `ransac_iters`



Contrariamente a lo que esperábamos, incrementar `ransac_iters` no proporciona siempre una mejora en la estimación de trayectorias.

Comparativa de la configuración ad hoc vs parámetros por defecto de LIBVISO2



Como se observa en los gráficos, en varios casos la configuración **ad hoc** otorga mejores resultados que la configuración por defecto de *LIBVISO2*.

6.1.3. Tolerancia a la pérdida de cuadros

En este apartado estudiamos cómo variando los parámetros es posible mitigar el impacto de la pérdida de cuadros en la calidad del resultado.

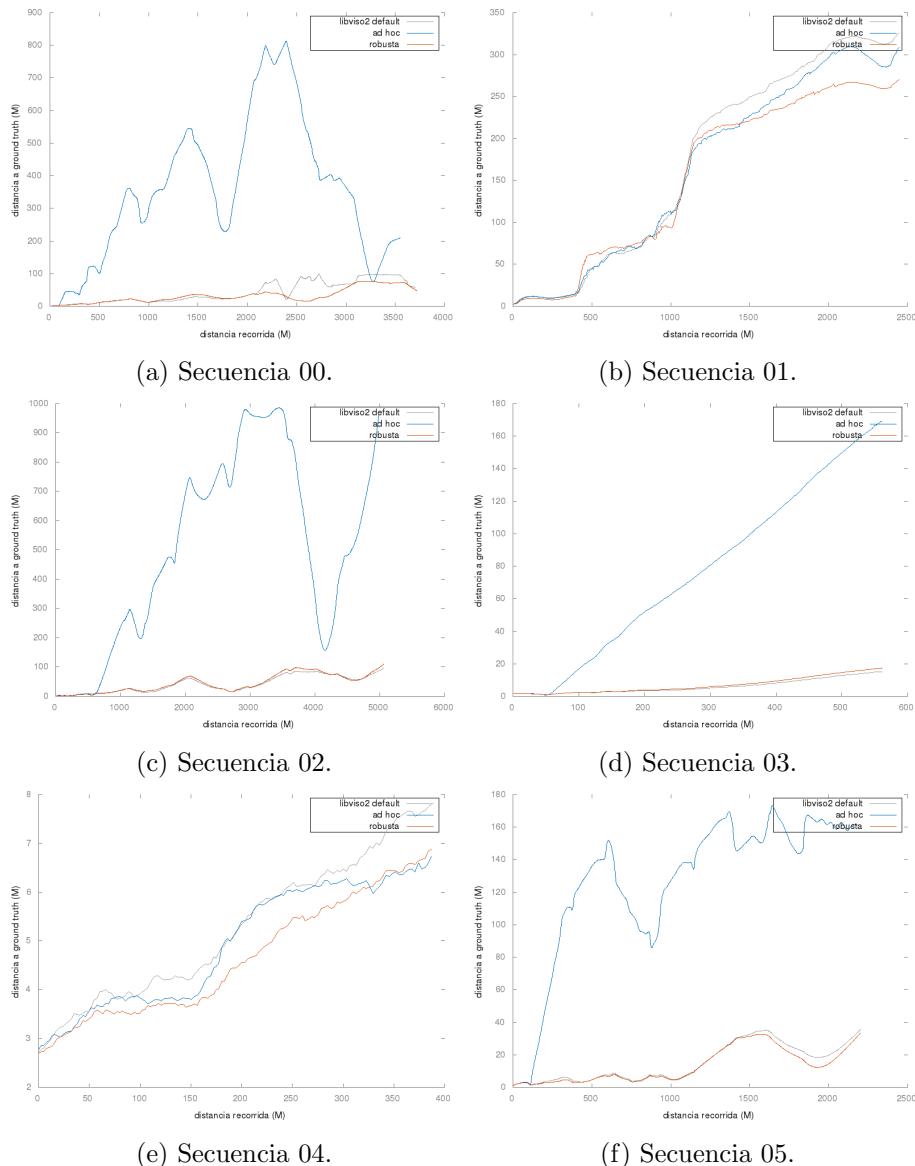
Para esto construimos una nueva configuración a la que llamamos “**robusta**”, teniendo en cuenta los efectos producidos por la pérdida de cuadros. En particular tuvimos en cuenta que cuanto más tiempo haya transcurrido entre dos cuadros, entonces la traslación y rotación de las cámaras sera posiblemente mayor.

Decidimos reducir el impacto de la pérdida de cuadros agrandando el área de búsqueda de coincidencias para cada **feature**. Para esto incrementamos el valor de **match_radius**, de forma tal de detectar coincidencias aun cuando se analizan cuadros no consecutivos, donde las posiciones de los **features** pueden haber sufrido grandes variaciones.

configuración	nms_n	nms_tau	match_binsize	match_radius	ransac_iters
robusta	3	50	20	400	200

Cuadro 6.2: Parámetros de la configuración “robusta”.

A continuación comparamos las configuraciones **ad hoc**, **robusta** y **por defecto de LIBVISO2**. Procesamos las secuencias de imágenes de los datasets simulando capacidades de computo de 3 y 5 cuadros por segundo.

Figura 6.7: Gráficos de distancias entre trayectorias reales y estimadas a 5 **fps** para las secuencias del dataset **KITTI** para odometría visual

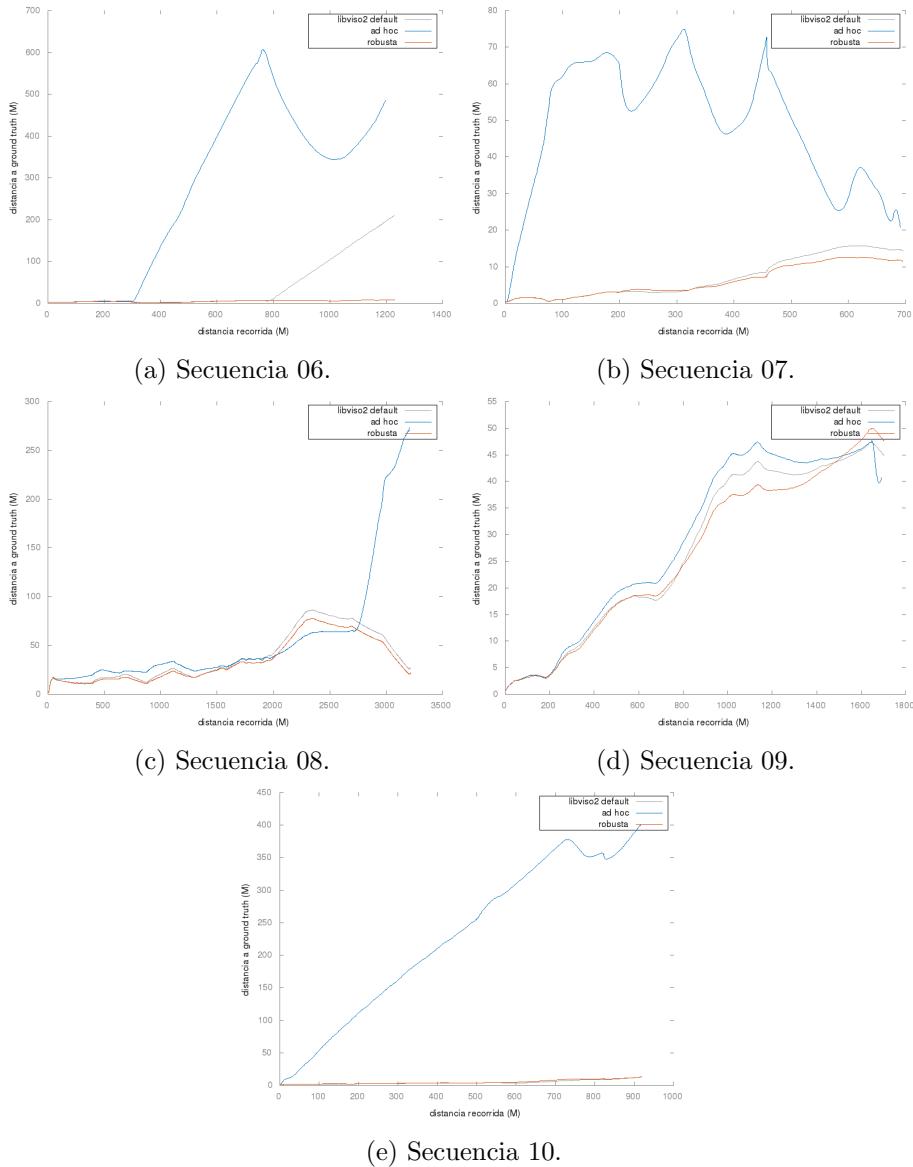


Figura 6.8: Gráficos de distancias entre trayectorias reales y estimadas a 5 **fps** para las secuencias del dataset **KITTI** para odometría visual (cont.)

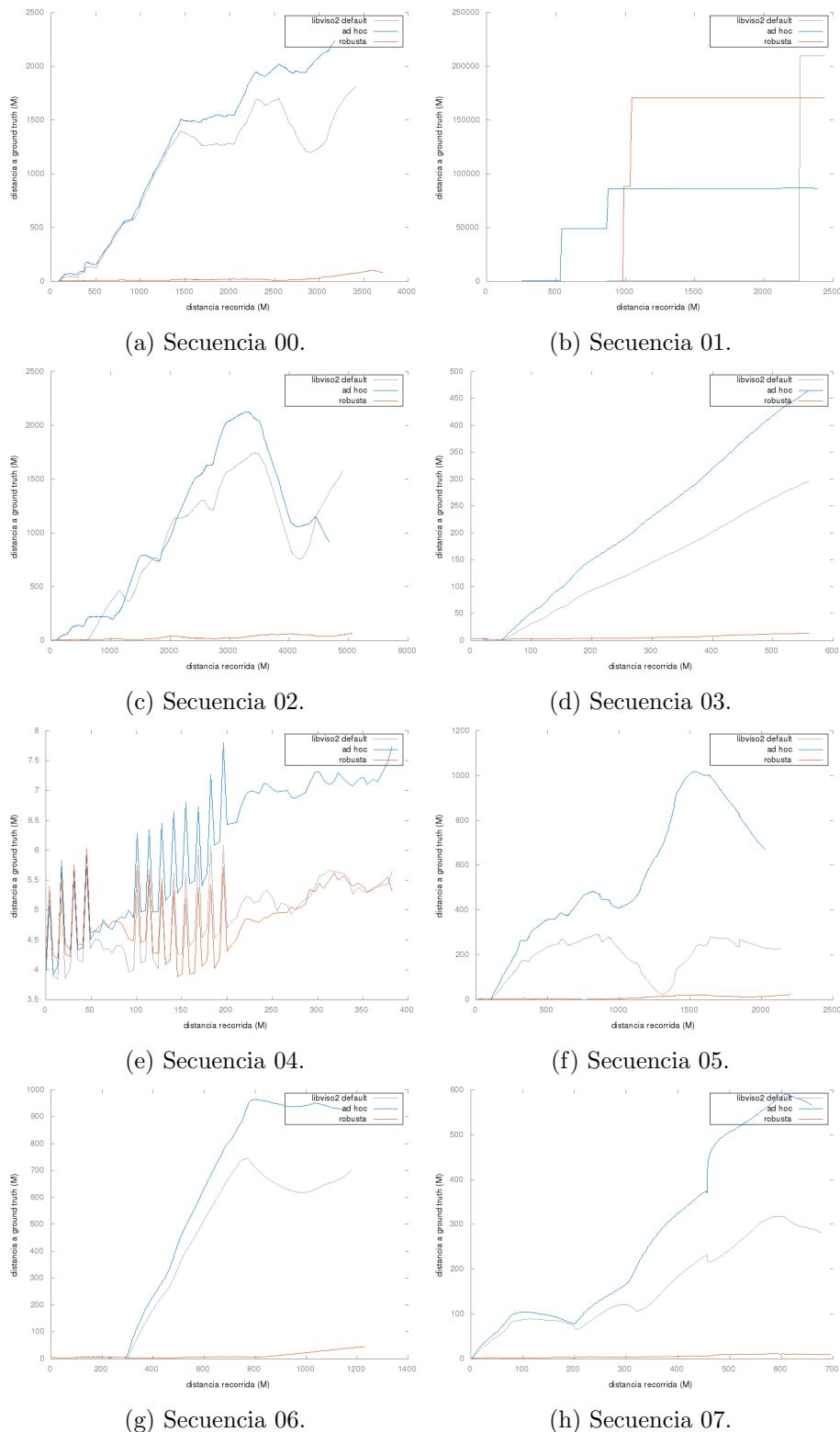


Figura 6.9: Gráficos de distancias entre trayectorias reales y estimadas a 3 **fps** para las secuencias del dataset **KITTI** para odometría visual

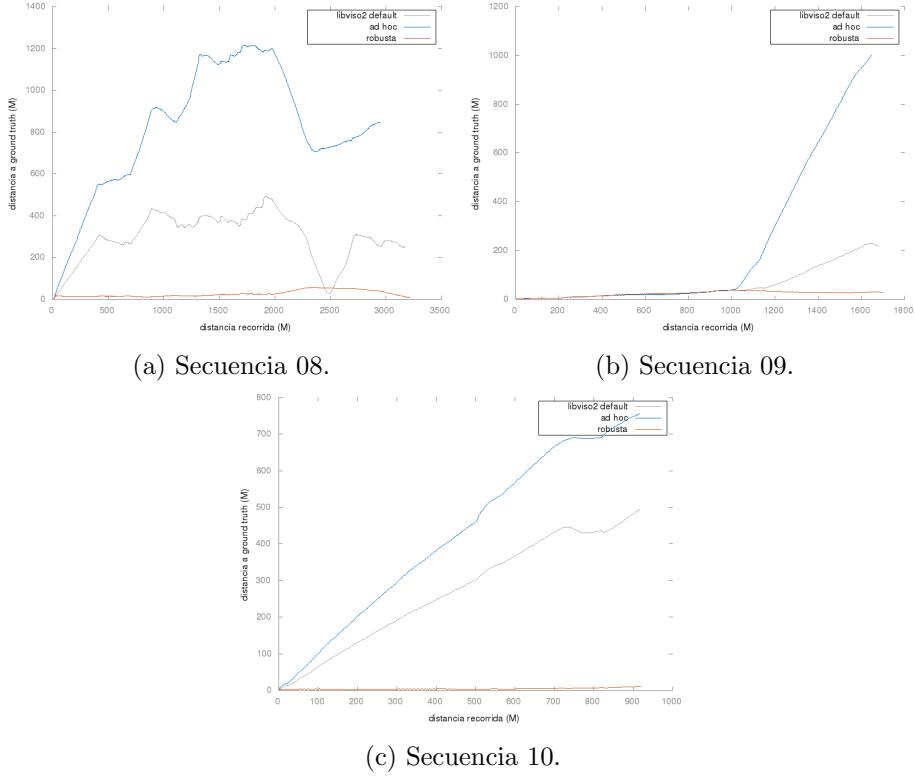


Figura 6.10: Gráficos de distancias entre trayectorias reales y estimadas a 3 *fps* para las secuencias del dataset **KITTI** para odometría visual (cont.)

Como se puede apreciar en los gráficos, modificando los parámetros es posible incrementar la tolerancia a la pérdida de cuadros.

6.1.4. Impacto en el rendimiento

A continuación medimos el impacto de los parámetros en el rendimiento de la solución, calculando para cada configuración la tasa de $\frac{\text{cuadros}}{\text{segundo}}$.

Para tomar estas mediciones procesamos la secuencia `minitest` desde el disco RAM.

configuración	<i>cuadros segundo</i>
LIBVISO2 Default	1,51
+nms_n	2,11
+tau	1,61
-tau	1,34
+binsize	1,55
-binsize	1,16
+radius	1,28
-radius	1,63
+ransac	1,26
-ransac	1,88
adhoc	2,55
robusta	0,72

Cuadro 6.3: Rendimiento de cada una de las configuraciones.

Si bien variando los parámetros de configuración es posible incrementar la tolerancia ante la pérdida de cuadros, esta no es una solución viable cuando la pérdida es originada por falta de poder de cómputo, puesto que el uso de parámetros que ofrecen mayor robustez conlleva un tiempo de procesamiento por cuadro mayor.

Encontramos que la configuración **ad hoc** ofrece un buen balance entre robustez y rendimiento.

Este es el juego de parámetros utilizado en las siguientes etapas de análisis y optimización.

6.2. Análisis detallado del rendimiento de viso

En esta sección realizamos un análisis detallado del rendimiento de la solución utilizando la configuración **ad hoc** introducida en 6.1.1. Medimos el porcentaje del tiempo total de procesamiento utilizado por cada componente. Esta información guiará el proceso de optimización.

Utilizando el ejecutable con información de **profiling** `viso_profile`, construido anteriormente en 5.4.1, procesamos la secuencia `minitest` desde el disco RAM.

Al correr este ejecutable se genera un archivo `gmon.out` con datos de **profiling** particulares de la corrida. Utilizando estos datos es posible tener una estimación del porcentaje de tiempo que pasa el proceso en la ejecución de cada función o método del código fuente.

A continuación copiamos el archivo `gmon.out` a la PC a través del sistema de archivos **NFS** compartido.

Luego creamos un script `process_gmon.sh` para graficar la información del archivo `gmon.out`.

Corriendo el script como se detalla a continuación se generó el gráfico de la figura 6.11¹.

```
~/system$ process_gmon.sh build/linux/rootfs/apps/viso/viso_profile gmon.out
```

¹El gráfico completo se puede consultar en el apéndice `viso_gmon.dot`.

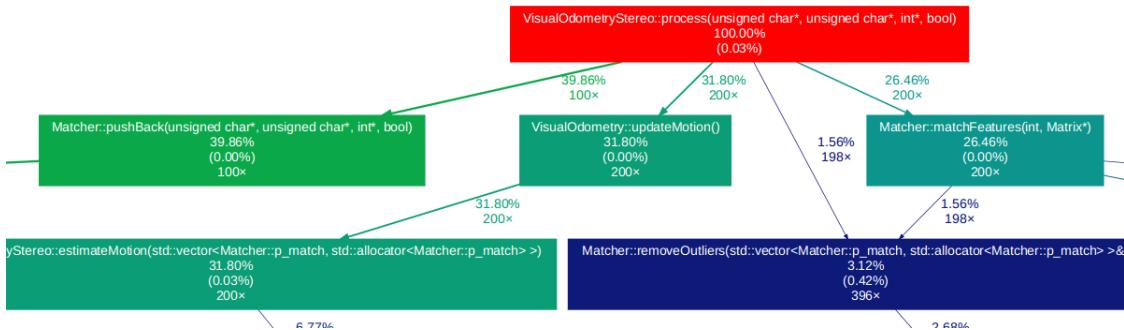


Figura 6.11: Extracto del gráfico generado por process_gmon.sh.

Con la información obtenida construimos un nuevo gráfico agrupando los porcentajes de tiempo de procesamiento por grupo de componentes al que pertenece cada función:

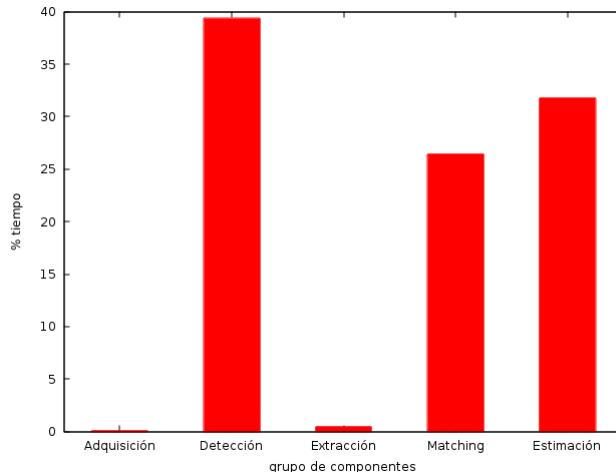


Figura 6.12: Porcentaje tiempos de procesamiento por grupo de componentes.

La información obtenida en este análisis fue utilizada para tomar decisiones sobre las optimizaciones a aplicar.

6.3. Optimizaciones

A partir del análisis realizado en la sección anterior decidimos enfocar el esfuerzo de optimización en los componentes de “Detección de **features**”, “Extracción de **features**”, “Matching de **features** entre imágenes” y “Estimación de pose”.

Creamos una nueva aplicación viso_s la cual explota los dos procesadores ARM del **AP SoC** y el soporte para aplicaciones multithread de *Linux*.

```
~/system$ petalinux-create -t apps -n viso_s --template c++ --enable
```

Figura 6.13: Comando para crear la nueva aplicación viso_s

Copiamos el código de `viso` a la nueva aplicación y modificamos el ***Makefile*** para habilitar la biblioteca ***pthreads***. La misma proporciona funcionalidad para el manejo de threads y su sincronización.

Además modificamos los nombres de los ejecutables a construir por `viso_s` y `viso_s_profile`. Al convivir en el mismo sistema `viso` y `viso_s` se facilita realizar comparaciones entre ambos.

A continuación se detallan los cambios realizados a los componentes de cada grupo.

6.3.1. Optimización de Detección y Extracción de ***features***

Abordamos los componentes de los grupos “Detección de ***features***” y “Extracción de ***features***” en conjunto puesto que su código fuente y flujo de ejecución se encuentran fuertemente ligados.

Tal como se explicó anteriormente en [3.2.1](#), la detección y extracción de ***features*** de una imagen consiste en calcular su filtro *Sobel*, la imagen escalada a la mitad del tamaño y los filtros *Checkerboard*, *Blob* y *Sobel* de esta última. Posteriormente el algoritmo **NMS** es utilizado para detectar ***features*** “dispersos” y “densos”. Finalmente los descriptores son extraídos a partir de los filtros *Sobel* de la imagen escalada.

En `viso` la detección y extracción de ***features*** de cada imagen de un cuadro estéreo se realiza secuencialmente, como se muestra en la figura [6.14](#).

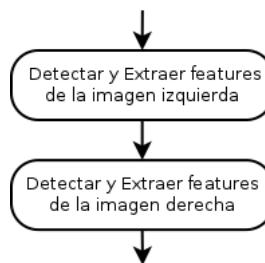


Figura 6.14: Diagrama de actividad del proceso de detección y extracción de ***features*** en `viso`.

Dado que estas tareas son intensivas en cálculos, decidimos acelerarlas aprovechando ambos núcleos del procesador para paralelizarlas.

Realizamos las modificaciones necesarias en `viso_s` para que la detección y extracción de ***features*** se realice simultáneamente para las dos imágenes de cada cuadro.

La figura [6.15](#) muestra el flujo de ejecución.

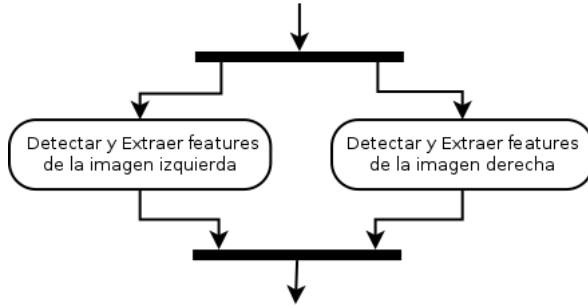


Figura 6.15: Diagrama de actividad del proceso de detección y extracción de **features** en **viso_s**.

6.3.2. Optimización de Matching de features entre imágenes

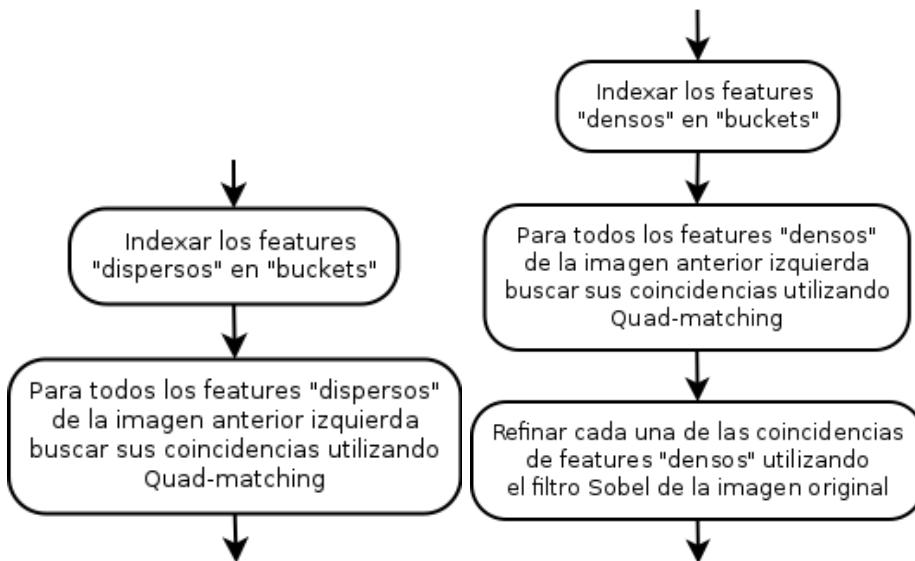
Al igual que en la sección anterior, utilizamos el diagrama de actividad para estudiar el flujo de ejecución de las tareas de este grupo de componentes.

Analizando el código fuente de **viso** construimos el diagrama de actividad de la figura 6.16.



Figura 6.16: Diagrama de actividad del proceso de búsqueda de coincidencias.

Si bien todas las tareas del diagrama son intensivas en cómputo, el nivel de abstracción elegido no permite detectar tareas paralelizables. Por lo tanto miramos con más detalle las tareas de búsqueda y refinamiento de coincidencias.

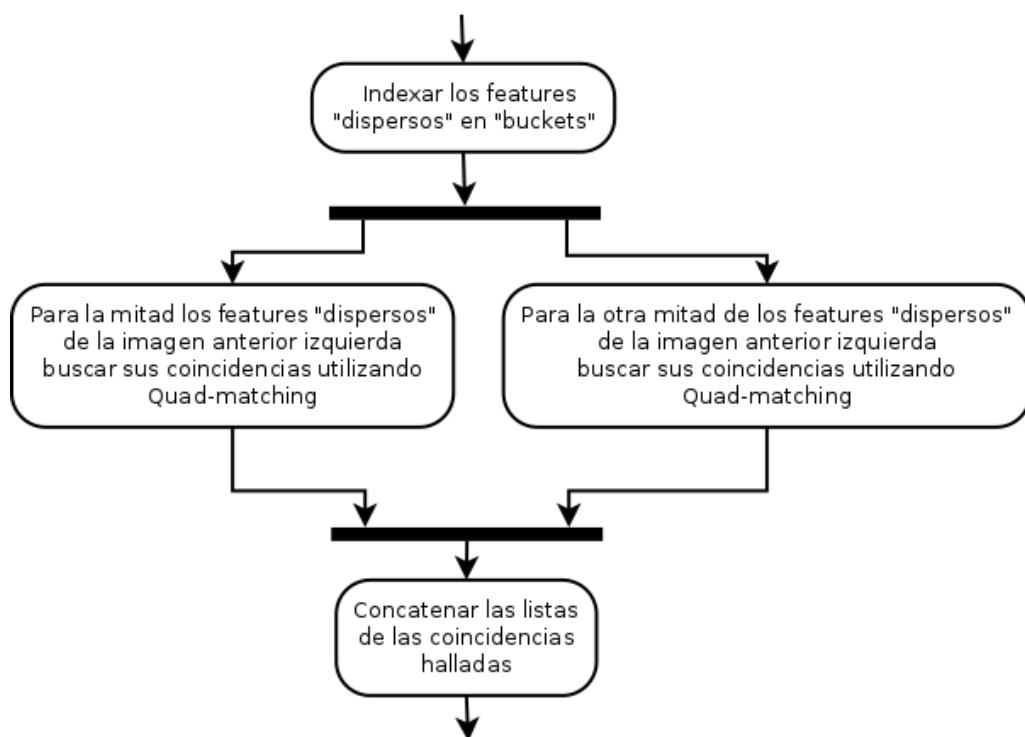


(a) Diagrama de actividad del proceso de búsqueda de coincidencias de **features** “dispersos” y “densos”. (b) Diagrama de actividad del proceso de búsqueda de coincidencias de **features** dispersos.

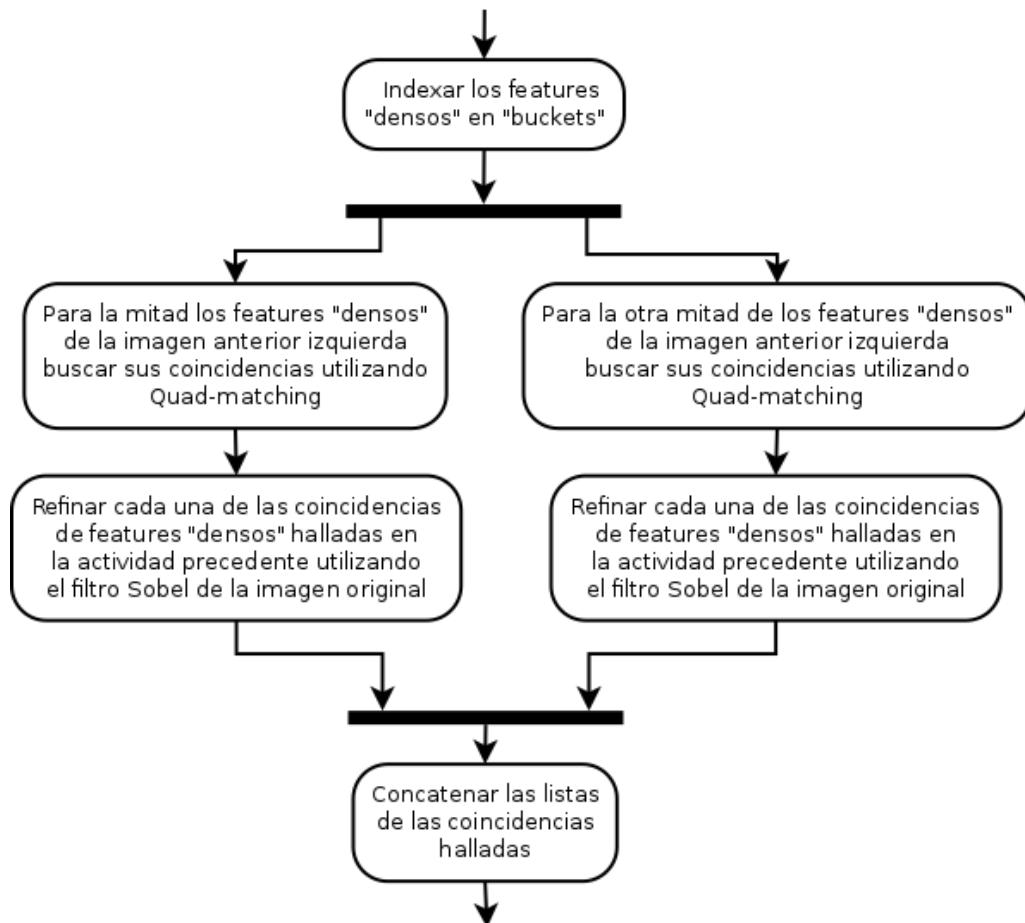
Figura 6.17: Detalle de los procesos de búsqueda de coincidencias de **features** “dispersos” y “densos” en viso.

Las tareas de los diagramas de la figura 6.17 tampoco son paralelizables, pero a este nivel de detalle es posible apreciar que las tareas de búsqueda de coincidencias para cada uno de los **feature** son independientes. Lo mismo ocurre para el refinamiento, el cálculo para cada coincidencia es independiente de las demás.

Por lo tanto partimos las tareas de búsqueda y refinamiento en tareas que trabajan con conjuntos más pequeños de datos que pueden ser tratados de manera independiente. Esto nos permitió paralelizar el trabajo como muestra la figura 6.18.



(a) Diagrama de actividad del proceso de búsqueda de coincidencias de **features** dispersos.



(b) Diagrama de actividad del proceso de búsqueda y refinación de coincidencias de **features** densos.

Figura 6.18: Detalle de los procesos de búsqueda de coincidencias de **features** “dispersos” y “densos” en `viso_s`.

Realizamos las modificaciones necesarias al código fuente de `viso_s`, las cuales incluyeron crear una nueva estructura de datos para agrupar las variables accedidas por cada hilo de ejecución y modificar los límites de varias iteraciones para trabajar solo con la mitad de los **features**.

Aunque la técnica de particionar el trabajo y dividirlo en múltiples hilos de ejecución permite crear más de dos particiones, lo hicimos de esta forma ya que el **AP SoC** utilizado cuenta con dos procesadores.

6.3.3. Optimización de Estimador de pose

De manera análoga al trabajo realizado para los otros grupos de componentes, lo primero que hicimos para el “Estimador de pose” fue un diagrama de actividad [6.19](#).

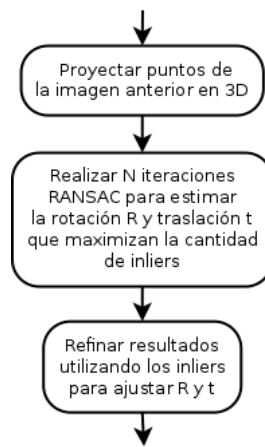


Figura 6.19: Diagrama de actividad del proceso de estimación de pose en `viso`.

Al igual que en la sección anterior, el nivel de abstracción no permite ver tareas paralelizables, pero analizando en detalle el código fuente puede verse que cada iteración del algoritmo **RANSAC** es independiente.

Por lo tanto, en lugar de tener una sola tarea que realiza N iteraciones **RANSAC** dividimos el trabajo en dos tareas, cada una realiza $\frac{N}{2}$ iteraciones. De esta forma obtuvimos dos tareas paralelizables, como muestra la figura [6.20](#).

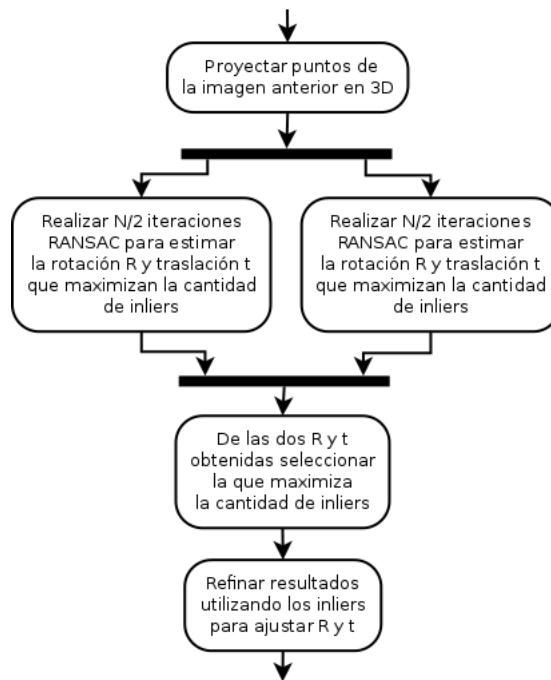


Figura 6.20: Diagrama de actividad del proceso de estimación de pose en `viso_s`.

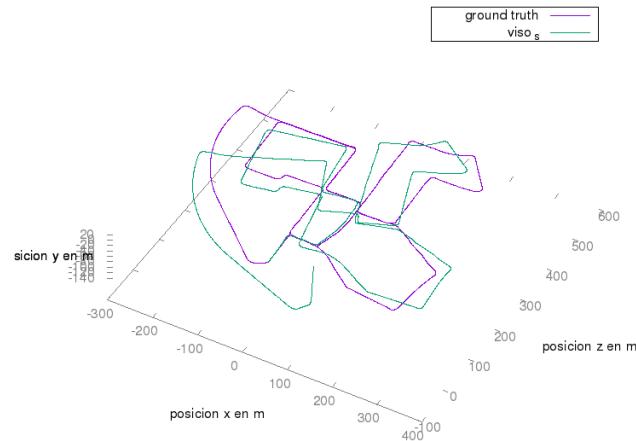
Dado que **RANSAC** utiliza números pseudoaleatorios, fue necesario modificar el código para incorporar un generador de números pseudoaleatorios ***thread-safe***.

6.4. Pruebas de la solución optimizada en ZYBO

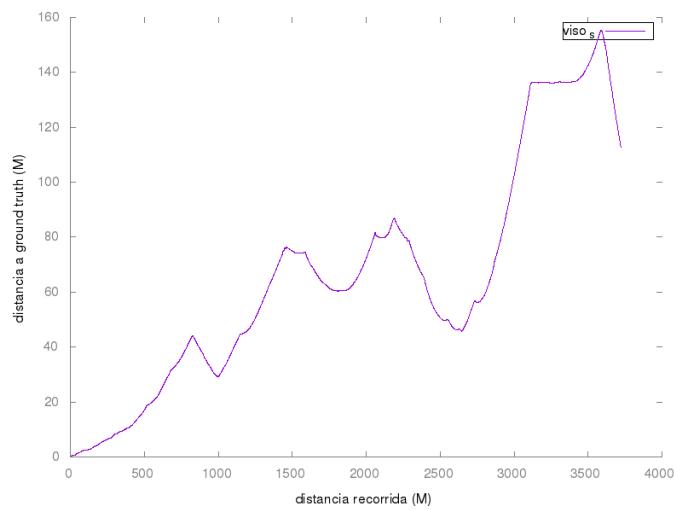
Al igual que en 5.4.3, utilizamos `run_test_case_remote` para probar `viso_s` en **Zybo**.

```
{
  "name": "00",
  "sequence": "00",
  "framesN": 999999,
  "ground_truth": true,
  "runs": [
    {
      "cmd": "viso_s",
      "name": "viso_s",
      "nms_n": 5,
      "nms_tau": 60,
      "match_binsize": 20,
      "match_radius": 150,
      "ransac_iters": 100,
      "fps": 10
    }
  ]
}
```

(a) Caso de test para la secuencia 00 procesada en ZYBO.



(b) Gráfica de trayectoria generada por `viso_s`



(c) Distancia entre la trayectoria estimada y la real

Figura 6.21: Caso de test de ejemplo ejecutable de forma remota para `viso_s`

Los resultados completos de las pruebas realizadas pueden consultarse en el apéndice `apendice-resultados-viso_s.tar.gz`.

6.5. Medición de rendimiento

En esta sección medimos el rendimiento de la solución construida y lo comparamos con la anterior. Al igual que en [5.5](#) tomamos los tiempos de procesamiento de la secuencia de imágenes “`minitest`” desde el disco RAM en `Zybo`.

De esta forma obtuvimos el siguiente resultado:

solución	configuración	<i>cuadros segundo</i>
viso	adhoc	2,55
viso_s	adhoc	4,23

Cuadro 6.4: Comparativa de rendimiento entre `viso` y `viso_s` para la configuración **adhoc**.

Como puede verse, la solución `viso_s` procesa aproximadamente un 66 % más de cuadros por segundo que `viso`.

Capítulo 7

Optimización mediante aceleración por hardware

Este capítulo es el **núcleo de nuestro trabajo**. Aquí se presentan las optimizaciones por hardware aplicadas a `viso_s`, la solución construida en el capítulo 6, para obtener una nueva solución `viso_h` con mejor rendimiento.

El desarrollo de este capítulo abarca los pasos “C - Partición Hardware/Software” y “D - Implementación de hardware, testing e integración” de la metodología de desarrollo presentada en 3.3.

Para esto en la sección 7.1 realizamos un análisis detallado del rendimiento de la solución `viso_s` en *Zybo*. La información obtenida nos permitió estimar el impacto de las optimizaciones en el rendimiento y la distribución del trabajo entre los componentes.

Posteriormente, en la sección 7.2 llevamos a cabo un estudio preliminar sobre la aplicación de *Vivado HLS* a la construcción de **IP Cores** para el procesamiento de imágenes. En este apartado estudiamos las técnicas propuestas por Vallina 2012¹ y extrajimos un **patrón de diseño**.

Nuestro principal aporte es presentado en el apartado 7.2.3. Analizamos la aplicación de optimizaciones que permiten la reducción del consumo de recursos y el aumento de la velocidad en **IP Cores** construidos siguiendo el patrón extraído anteriormente. Durante este trabajo elaboramos un paper² que sintetiza este apartado.

A continuación, la sección 7.3 presenta la arquitectura de hardware. Aquí se detalla la **partición hardware/software** y la comunicación entre procesador, memoria y los **IP Cores**.

La sección 7.4 detalla el diseño, desarrollo y prueba de cada uno de los **IP Cores**. En la sección 7.5 se detalla el proceso de integración de los **IP Cores** a la plataforma hardware.

Posteriormente, la sección 7.6 describe los cambios realizados a la plataforma software. En particular se describe el desarrollo de un módulo del **kernel** encargado de controlar el nuevo hardware, permitiendo que el mismo sea utilizado por aplicaciones de usuario.

Luego, la sección 7.7 detalla las modificaciones realizadas a `viso_s` para delegar el procesamiento de las imágenes al hardware construido.

Finalmente, en las secciones 7.8 y 7.9 llevamos a cabo pruebas funcionales sobre `viso_h`, medimos el rendimiento de la solución optimizada y lo comparamos con la anterior, `viso_s`.

¹Vallina 2012.

²García y Borensztein 2016.

7.1. Análisis detallado del rendimiento de viso_s

Siguiendo los mismos pasos que en la sección 6.2, construimos un gráfico agrupando los porcentajes de tiempo de procesamiento real por grupo de componentes al que pertenece cada función:

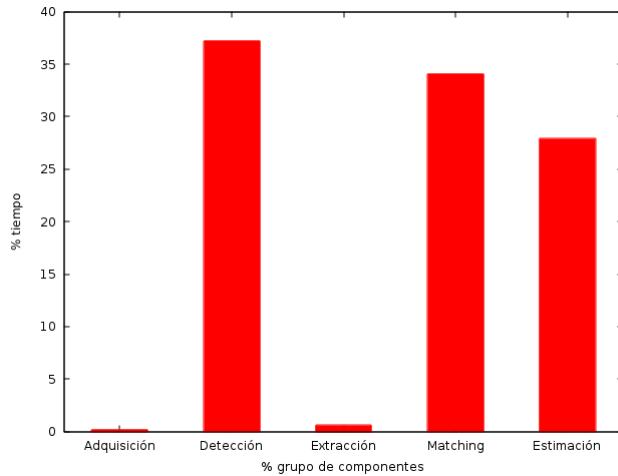


Figura 7.1: Porcentaje tiempos de procesamiento real por grupo de componentes en `viso_s`.

La información obtenida nos permitió estimar el impacto de las optimizaciones en el rendimiento y la distribución del trabajo entre los componentes.

7.2. Aplicación de HLS a la construcción de IP Cores para procesamiento de imágenes

En esta sección estudiamos la aplicación de Vivado HLS³ y las técnicas propuestas en “Implementing Memory Structures for Video Processing in the Vivado HLS Tool”⁴ a la construcción de **IP Cores** para procesamiento de imágenes.

7.2.1. Vivado HLS

Vivado HLS posibilita, a través del uso de **síntesis de alto nivel**, el desarrollo de **IP Cores** utilizando C o C++ para especificar lógica digital. De esta forma el proceso de desarrollo y prueba es simplificado, permitiendo el uso de herramientas estándar y abstrayendo parte de la complejidad comúnmente encontrada al trabajar con lenguajes de descripción de hardware (**HDL**).

De todas maneras, existen dificultades inherentes al desarrollo de hardware que no pueden ser evitadas. Esto se debe a que el código no es ejecutado por una CPU, sino que es utilizado por la herramienta para inferir hardware. Por lo tanto, para lograr buenos resultados deben tenerse en cuenta los mecanismos de inferencia, es decir la forma en que la herramienta extrae a partir de un código de alto nivel un circuito digital equivalente.

³Xilinx 2014a.

⁴Vallina 2012.

7.2. APLICACIÓN DE HLS A LA CONSTRUCCIÓN DE IP CORES PARA PROCESAMIENTO DE IMÁGENES

Esta es una lista de algunos factores que el diseñador debe tener en cuenta e introducir sus decisiones sobre los mismos en la herramienta, realizando modificaciones al código de alto nivel:

- **Alocación de recursos:** Los distintos tipos de memoria disponibles: memoria externa al **AP SoC**, **BRAM**, memoria distribuida en **LUTs** y **flip-flops** forman una jerarquía de memorias con diferentes tiempos de acceso, capacidades y posibilidad de acceso a múltiples elementos en simultáneo. Al usar herramientas **HLS** el diseñador debe mapear las variables del programa a memorias físicas teniendo en cuenta esta jerarquía y las características del algoritmo a implementar.
- **Paralelismo:** Es posible explotar el paralelismo, por ejemplo replicando unidades funcionales para operar sobre varios conjuntos de datos al mismo tiempo. Esto genera múltiples circuitos que realizan el mismo trabajo, la desventaja de esta solución es el mayor consumo de área y energía.
- **Comunicación entre componentes:** En sistemas grandes es más sencillo trabajar con componentes más pequeños interconectados. Al estar creando circuitos las conexiones son señales, por lo tanto el diseñador debe decidir qué señales usar, pudiendo emplear **buses** estándar o definiendo propios.
- **Optimizaciones:** A comparación del mundo del software en donde el compilador realiza muchas optimizaciones de forma automática, las herramientas de **HLS** dejan gran parte de estas decisiones al diseñador. Esto se debe a que no pueden basarse en el comportamiento de una máquina ya existente para tomar decisiones y a que se deben considerar tres factores críticos: velocidad, área ocupada y consumo de energía.

Por ejemplo aplicar **loop unrolling**, si bien podría incrementar la velocidad, acarrea un incremento de área y consumo de energía.

Si bien no hay reglas generales para obtener buenos resultados utilizando **HLS**, en el contexto de **IP Cores** para procesamiento de imágenes se pueden caracterizar algunos patrones y técnicas de diseño que permiten un buen balance entre velocidad y uso de área.

7.2.2. Patrón de diseño para procesamiento secuencial de píxeles

Este apartado describe un patrón de diseño aplicable a la construcción de **IP Cores** que implementan algoritmos donde se procesan todos los píxeles de una imagen en forma **secuencial**. Por ejemplo, la aplicación de convoluciones a imágenes.

En primer lugar el patrón propone caracterizar la entrada y salida del **IP Core** como **streams** de datos. De esta forma es posible independizar al **IP Core** de la fuente de la imagen y destino del resultado, permitiendo que los mismos sean memoria RAM, cámaras, salidas de video, etc.

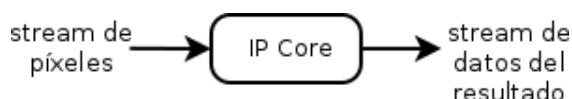


Figura 7.2: IP Core con **streams** como entrada y salida.

62 CAPÍTULO 7. OPTIMIZACIÓN MEDIANTE ACELERACIÓN POR HARDWARE

Dado que para operar sobre un píxel suele ser necesario contar con un contexto de N filas y M columnas de píxeles alrededor del mismo, este patrón utiliza un **buffer** para almacenar las últimas N líneas leídas de la imagen. Este **buffer** es llamado **Line Buffer**.

También utiliza un **buffer**, llamado **Window**, de $N \times M$ píxeles que contiene el contexto necesario para procesar cada píxel. Si bien este último **buffer** no es estrictamente necesario, agrega claridad y facilita la posterior optimización.

El algoritmo 2 ejemplifica la aplicación de este patrón para procesar todos los píxeles de una imagen con una ventana de $N \times M$ píxeles como contexto.

```

1: function PROCESARIMAGEN(input_stream, output_stream)
2:   for i=0; i< alto; i ++ do
3:     for j=0; j< ancho; j ++ do
4:       nuevo_pixel ← READ(input_stream)
5:       for k=0; k< N; k ++ do                                ▷ Shift window loop
6:         for l=0; l< N – 1; l ++ do
7:           Window[k][l] ← Window[k][l + 1]
8:         end for
9:       end for
10:      for k=0; k< N – 1; k ++ do                         ▷ Shift line buffers loop
11:        LineBuffer[k][j] ← LineBuffer[k + 1][j]
12:      end for
13:      LineBuffer[N – 1][j] ← nuevo_pixel
14:      for k=0; k< N; k ++ do                            ▷ Feed window loop
15:        Window[k][M – 1] ← LineBuffer[k][j]
16:      end for
17:      resultado ← PROCESAR(Window)
18:      WRITE(output_stream, resultado)
19:    end for
20:  end for
21: end function
```

Algoritmo 2: Procesamiento de imagen utilizando line buffer

Vallina 2012 guía la decisión del tipo de memoria a usar para **Line Buffers** y **Window** apuntando a minimizar la cantidad de ciclos de reloj necesarios para llevar a cabo las operaciones de los bucles **Shift window loop**, **Shift line buffers loop** y **Feed window loop**.

Para esto considera la cantidad de accesos simultáneos a cada variable y propone utilizar uno o más **Block RAMs** independientes para cada línea del **Line Buffer** y registros independientes (en LUTs) para cada celda de **Window**.

Al hacerlo de esta manera, los bucles pueden ejecutarse de forma completa cada uno en un ciclo de reloj.

Si bien es posible implementar **Line Buffers** y **Window** manualmente como arreglos en C++, su uso es tan frecuente que Vivado HLS provee una biblioteca, `hls_video.h`, con la lógica necesaria para su manejo.

7.2. APLICACIÓN DE HLS A LA CONSTRUCCIÓN DE IP CORES PARA PROCESAMIENTO DE IMÁGENES

Esta biblioteca ya contiene las directivas de **HLS**⁵ necesarias para que los **Line Buffer** se implementen utilizando uno o más **Block RAMs** independientes por línea y las **Window** utilizando un registro independiente para cada celda.

El código a continuación implementa el algoritmo 2 en Vivado HLS utilizando C++ y **hls_video.h**:

```
#include <ap_axi_sdata.h>
#include <hls_video.h>
#define IMAGE_MAX_WIDTH 1024

void procesarImagen(ap_axiu<8,1,1,1> *input_stream,
                    ap_axiu<8,1,1,1> *output_stream,
                    u16 width, u16 height)
{
#pragma HLS INTERFACE ap_ctrl_hs port=return
#pragma HLS INTERFACE axis port=input_stream
#pragma HLS INTERFACE axis port=output_stream
#pragma HLS INTERFACE ap_none port=width
#pragma HLS INTERFACE ap_none port=height

    hls::Window<N, M, u8> window;
    hls::LineBuffer<N, IMAGE_MAX_WIDTH, u8> line_buffers;

    u16 row, col;
    for(row=0; row<height; row++)
    {
        for(col=0; col<width; col++)
        {
            u8 nuevo_pixel = input_stream->data;

            window.shift_left();
            line_buffers.shift_up(col);
            line_buffers.insert_top(nuevo_pixel, col);
            for(int i=0; i<N; i++)
            {
                window.insert(line_buffers.val[i][col], i, M-1);
            }

            output_stream->data = procesar(window.val);
            output_stream->last = input_stream->last;
            output_stream->keep = 1;
            output_stream++;

            input_stream++;
        }
    }
}
```

Figura 7.3: Implementación del algoritmo 2 en Vivado HLS.

Al implementar el algoritmo es necesario limitar el ancho máximo de la imagen para que la herramienta pueda calcular la cantidad de **Block RAMs** a utilizar para cada línea del **Line Buffer**. Además es necesario especificar la interfaz del IP Core empleando directivas **HLS**.

Al sintetizar el código anterior, completando la función **procesar** para que convolu-

⁵Xilinx 2014b.

cione la imagen con la matriz

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

⁶ y fijando $N = 3$ y $M = 3$, se generó el siguiente **IP Core**:

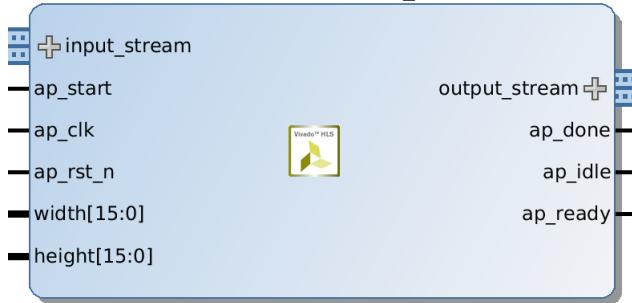


Figura 7.4: IP Core `procesarImagen` generado con **Vivado HLS**.

7.2.3. Optimizaciones y rendimiento

Este apartado presenta optimizaciones aplicables al código 7.3 para obtener mejores resultados al generar **IP Cores** utilizando **Vivado HLS**.

Estimación del rendimiento

La directiva **LOOP_TRIPCOUNT** sirve para informar a la herramienta la cantidad de iteraciones que realizará cada bucle con límites variables. De esta manera, la herramienta puede estimar la cantidad de ciclos de reloj que empleará el **IP Core** para realizar el trabajo.

Si bien esta directiva no es una optimización, resulta útil para medir el impacto de las optimizaciones aplicadas.

El código a continuación muestra la aplicación de esta directiva al código de la figura 7.3 para estimar la cantidad de ciclos y recursos necesarios para procesar una imagen de 1024x768 píxeles:

```
...
for(row=0; row<height; height; row++)
{
#pragma HLS LOOP_TRIPCOUNT min=768 max=768 avg=768
    for(col=0; col<width; col++)
    {
#pragma HLS LOOP_TRIPCOUNT min=1024 max=1024 avg=1024
    ...
}
```

Figura 7.5: Aplicación de la directiva **LOOP_TRIPCOUNT**.

Sintetizando el nuevo código **Vivado HLS** provee información sobre la cantidad de ciclos de reloj necesarios para procesar una imagen y sobre el consumo de recursos de **PL**.

⁶Esta convolución es útil para detectar ejes en imágenes.

7.2. APLICACIÓN DE HLS A LA CONSTRUCCIÓN DE IP CORES PARA PROCESAMIENTO DE IMÁGENES

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	250
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	3	-	0	0
Multiplexer	-	-	-	71
Register	-	-	237	-
Total	3	0	237	321
Available	120	80	35200	17600
Utilization (%)	2	0	~0	1

(a) Velocidad

(b) Consumo de recursos

Figura 7.6: Estimación de velocidad y consumo de recursos generada por **Vivado HLS**.

La tabla (a) de la figura 7.6 muestra la latencia (**Latency**), cantidad de ciclos de reloj que le toma al **IP Core** procesar una imagen, y el intervalo de iniciación (**Interval**), la cantidad de ciclos de reloj desde que el **IP Core** comienza a procesar una imagen hasta que puede comenzar a procesar otra.

El **IP Core** construido presenta un consumo de recursos bajo, pero su **productividad** ($\frac{\text{pixeles}}{\text{ciclo de reloj}}$) es muy pobre. La misma se calcula como $\frac{\text{pixeles a procesar}}{\text{latency}}$, en este caso es $\frac{1024 \times 768 \text{ pixeles}}{11798017 \text{ ciclos}} = \sim 0,067 \frac{\text{pixeles}}{\text{ciclos}}$.

A continuación se describen optimizaciones aplicables al **IP Core** para mejorar su rendimiento.

Optimización pipeline

La directiva **PIPELINE**⁷ de Vivado HLS permite reducir la latencia, es decir incrementar la velocidad, y por lo tanto mejorar la **productividad**.

Al aplicar la directiva a una función o bucle, Vivado HLS intentara que sus operaciones se realicen en paralelo, respetando las dependencias de datos.

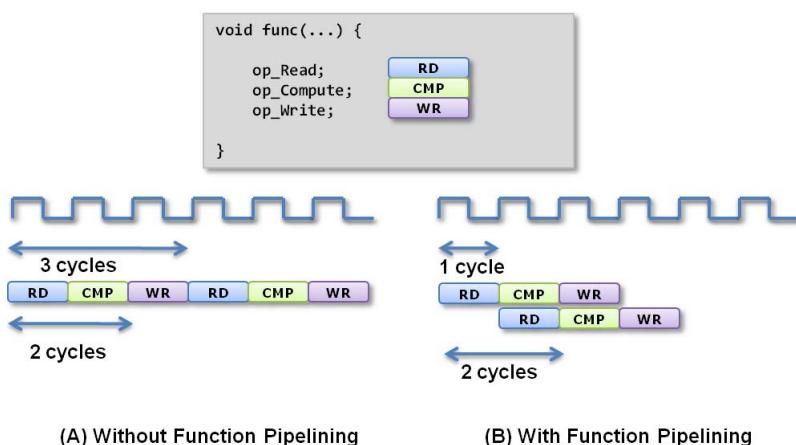


Figura 7.7: Comparativa de comportamiento del código sin y con utilizar la directiva **PIPELINE** extraída del manual de Vivado HLS.

⁷Xilinx 2014b.

El código a continuación muestra la aplicación de esta directiva al segundo bucle del código de la figura 7.5:

```
...
for(row=0; row<height; height; row++)
{
#pragma HLS LOOP_TRIPCOUNT min=768 max=768 avg=768
    for(col=0; col<width; col++)
    {
#pragma HLS LOOP_TRIPCOUNT min=1024 max=1024 avg=1024
#pragma HLS PIPELINE
    ...
}
```

Figura 7.8: Aplicación de la directiva **PIPELINE**.

Sintetizando el nuevo código se puede observar que, al aplicar la directiva al bucle, se reduce el consumo de recursos y se incrementa la velocidad.

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	188
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	2	-	0	0
Multiplexer	-	-	-	43
Register	-	-	202	10
Total	2	0	202	241
Available	120	80	35200	17600
Utilization (%)	1	0	~0	1

(a) Velocidad

Latency		Interval		Type
min	max	min	max	
791041	791041	791042	791042	none

(b) Consumo de recursos

Figura 7.9: Estimación de velocidad y consumo de recursos generada por **Vivado HLS** para la implementación que utiliza la directiva **PIPELINE**.

Aplicando esta optimización la **productividad** paso de $\sim 0,06 \frac{\text{pixeles}}{\text{ciclo de reloj}}$ a $\sim 1 \frac{\text{pixeles}}{\text{ciclo de reloj}}$. Es decir, la **productividad** obtenida al aplicar la directiva **PIPELINE** es más de 16 veces el original.

Experimentalmente encontramos que al optimizar código que sigue el patrón descrito en 7.2.2 la mayor ganancia en **productividad** se obtiene aplicando la directiva **PIPELINE** al segundo bucle.

Al aplicar la directiva a toda la función `procesarImagen` o al bucle exterior se obtuvo en ambos casos una **productividad** de $\sim 0,20 \frac{\text{pixeles}}{\text{ciclo de reloj}}$.

Optimización: K píxeles por ciclo

Analizando el algoritmo 2 y el código de la figura 7.8 notamos que el ancho de los **streams** de entrada y salida es un factor limitante en la velocidad, pues se requiere al menos un ciclo para leer o escribir un dato en ellos.

Por lo tanto probamos una optimización que consiste en procesar en cada ciclo varios píxeles.

Utilizando **streams** más anchos es posible que el **IP Core** lea y escriba varios datos en cada ciclo.

Para esto es necesario aplicar las siguientes modificaciones al algoritmo:

7.2. APLICACIÓN DE HLS A LA CONSTRUCCIÓN DE IP CORES PARA PROCESAMIENTO DE IMÁGENES

- Modificar el ancho de los **streams** de entrada y salida multiplicándolos por K.
- A nivel **Line Buffers** trabajar con paquetes de K píxeles.
- Mantener K ventanas, dado que muchos píxeles de las K ventanas se comparten, alcanza con mantener una ventana (**Window**) de $N \times (M+K-1)$ píxeles que las contenga a todas.
- Modificar los límites del bucle por columnas para realizar $\frac{ancho}{K}$, pues en cada iteración se procesaran K píxeles.
- En cada iteración del segundo bucle procesar K ventanas.

Por simplicidad, al aplicar esta optimización es conveniente restringir el ancho de las imágenes a procesar a múltiplos de K píxeles. Lo mismo ocurre con el ancho de la ventana.

El código a continuación muestra la aplicación de esta optimización al código [7.8](#) del **IP Core procesarImagen** para procesar 4 píxeles por iteración.

68 CAPÍTULO 7. OPTIMIZACIÓN MEDIANTE ACCELERACIÓN POR HARDWARE

```

#define IMAGE_MAX_WIDTH 1024
#define N 3
#define M 3

u8 procesar_window(u8 window[N][8], int offset)
{
#pragma HLS INLINE
    s16 tmp =
        -window[0][offset] - window[0][1+offset] - window[0][2+offset]
        -window[1][offset] + 8 * window[1][1+offset] - window[1][2+offset]
        -window[2][offset] - window[2][1+offset] - window[2][2+offset];
    if(tmp < 0)
        return 0;
    if(tmp > 255)
        return 255;
    return tmp;
}

void procesarImagen(ap_axiu<32,1,1,1> *input_stream,
                    ap_axiu<32,1,1,1> *output_stream,
                    u16 width, u16 height)
{
#pragma HLS INTERFACE ap_ctrl_hs port=return
#pragma HLS INTERFACE axis port=input_stream
#pragma HLS INTERFACE axis port=output_stream
#pragma HLS INTERFACE ap_none port=width
#pragma HLS INTERFACE ap_none port=height

    hls::Window<N, 8, u8> window;
    hls::LineBuffer<N, IMAGE_MAX_WIDTH/4, u32> line_buffers;

    u16 row, col;
    for(row=0; row<height; row++)
    {
#pragma HLS LOOP_TRIPCOUNT min=768 max=768 avg=768
        for(col=0; col<width/4; col++)
        {
#pragma HLS LOOP_TRIPCOUNT min=256 max=256 avg=256
#pragma HLS PIPELINE
            u32 nuevos_píxeles = input_stream->data;

            for(u8 i = 0; i < N; i++) {
                for(u8 j = 0; j < 4; j++) {
                    window.val[i][j] = window.val[i][j+4];
                }
            }

            line_buffers.shift_up(col);
            line_buffers.insert_top(nuevos_píxeles, col);
            for(u8 i=0; i<N; i++)
            {
                for(int j=0; j < 4; j++)
                {
                    window.val[i][4+j] = (u8) (line_buffers.val[i][col] >> (8 * j)) & 0xFF;
                }
            }
            u32 tmp = 0;
            for(int i=0; i < 4; i++)
            {
                tmp = tmp | (((u32) procesar_window(window.val, i)) << (8*i));
            }
            output_stream->data = tmp;
            output_stream->last = input_stream->last;
            output_stream->keep = 0xF;
            output_stream++;
            input_stream++;
        }
    }
}
}

```

Figura 7.10: Aplicación de la optimización para procesar 4 píxeles por iteración.

7.2. APLICACIÓN DE HLS A LA CONSTRUCCIÓN DE IP CORES PARA PROCESAMIENTO DE IMÁGENES

Sintetizando el nuevo código en Vivado HLS, se puede observar que, a comparación de la implementación que procesa un píxel por iteración, la latencia se reduce y el consumo de recursos crece, aunque en menor medida.

Latency		Interval				Name	BRAM_18K	DSP48E	FF	LUT
min	max	min	max	Type						
196616	196616	196617	196617	none		Expression	-	-	0	525
						FIFO	-	-	-	-
						Instance	-	1	0	0
						Memory	2	-	0	0
						Multiplexer	-	-	-	44
						Register	-	-	486	34
						Total	2	1	486	603
						Available	120	80	35200	17600
						Utilization (%)	1	1	1	3

(a) Velocidad

(b) Consumo de recursos

Figura 7.11: Estimación de velocidad y consumo de recursos generada por Vivado HLS para la implementación que procesa 4 píxeles por iteración.

Aplicando esta optimización y la directiva **PIPELINE** de Vivado HLS la **productividad** pasa de $\sim 0,06 \frac{\text{pixeles}}{\text{ciclo de reloj}}$ a $\sim 4 \frac{\text{pixeles}}{\text{ciclo de reloj}}$. Es decir, la **productividad** obtenida es más de 66 veces la original.

Optimización: fusión de IP Cores

Esta optimización es aplicable cuando en un diseño existen dos o más **IP Cores** que procesan una misma imagen siguiendo el patrón antes descrito en [7.2.2](#).

Consiste en combinar la funcionalidad de los **IP Cores** en uno solo, lo que permite compartir **Line Buffers**, **Window** y parte de la lógica. De esta manera, se ahorran recursos sin perder velocidad.

Esto es posible pues, al estar todas las celdas de **Window** implementadas como registros independientes, las mismas son accesibles físicamente en simultáneo.

Para esto es necesario aplicar las siguientes modificaciones al código:

- Agregar un nuevo **stream** de salida.
- En cada iteración calcular el resultado correspondiente a cada uno de los **IP Cores** a combinar, escribiendo cada resultado en un **stream** distinto.

El siguiente código muestra la aplicación de esta optimización a dos **IP Cores**.

70 CAPÍTULO 7. OPTIMIZACIÓN MEDIANTE ACELERACIÓN POR HARDWARE

```

void procesarImagen(ap_axiu<8,1,1,1> *input_stream,
                    ap_axiu<8,1,1,1> *output_stream_1,
                    ap_axiu<8,1,1,1> *output_stream_2,
                    u16 width, u16 height)
{
    #pragma HLS INTERFACE ap_ctrl_hs port=return
    #pragma HLS INTERFACE axis port=input_stream
    #pragma HLS INTERFACE axis port=output_stream_1
    #pragma HLS INTERFACE axis port=output_stream_2
    #pragma HLS INTERFACE ap_none port=width
    #pragma HLS INTERFACE ap_none port=height

    ...

    for(row=0; row<height; row++)
    {
        for(col=0; col<width; col++)
        {

            ...

            output_stream_1->data = procesar_1(window.val);
            output_stream_1->last = input_stream->last;
            output_stream_1->keep = 1;
            output_stream_1++;

            output_stream_2->data = procesar_2(window.val);
            output_stream_2->last = input_stream->last;
            output_stream_2->keep = 1;
            output_stream_2++;

        }
    }
}

```

Figura 7.12: Aplicación de la optimización para combinar dos IP Cores.

Las funciones `procesar_1` y `procesar_2` corresponden a la función `procesar` de cada **IP Core**.

Sintetizando el nuevo código en Vivado *HLS*, se puede observar que la latencia no se alterada. El consumo de **Block RAMs** se reduce, pues existe un solo juego de **Line Buffers** compartido por ambas funcionalidades.

El consumo de **flip-flops** (FF) y **LUTs**, es menor a la suma de los recursos de los dos **IP Cores** independientes. Esto se debe a que existe un solo **Window** y a que la lógica de control es compartida por ambas funcionalidades.

Latency		Interval			Name	BRAM_18K	DSP48E	FF	LUT
min	max	min	max	Type	Expression	-	-	0	259
791041	791041	791042	791042	none	FIFO	-	-	-	-
					Instance	-	-	-	-
					Memory	2	-	0	0
					Multiplexer	-	-	-	43
					Register	-	-	264	10
					Total	2	0	264	312
					Available	120	80	35200	17600
					Utilization (%)	1	0	~0	1

(a) Velocidad

(b) Consumo de recursos

Figura 7.13: Estimación de velocidad y consumo de recursos generada por **Vivado HLS** para la implementación que combina dos IP Cores.

La única desventaja de esta optimización es que la prueba del **IP Core** construido puede resultar más difícil que la de varios **IP Cores** más pequeños por separado.

Esta optimización y la anterior son combinables. Juntas permiten mejorar la **productividad** y **reducir el consumo de recursos**.

7.3. Arquitectura de Hardware

Esta sección presenta la **partición hardware/software** de la funcionalidad, la comunicación entre componentes hardware y el procesador a través de **buses** y el **pipeline** de procesamiento de imágenes que diseñamos.

7.3.1. Partición Hardware/Software

En la nueva solución el procesamiento intensivo de las imágenes, es decir, aquel que requiere procesar todos sus píxeles, es realizado completamente en hardware, liberando al procesador de estas tareas.

Utilizando el patrón de diseño y técnicas de optimización descritas en 7.2, la funcionalidad del grupo de componentes “**Detección de features**” fue implementada en lógica programable.

El siguiente diagrama muestra la **partición hardware/software** de funcionalidad:

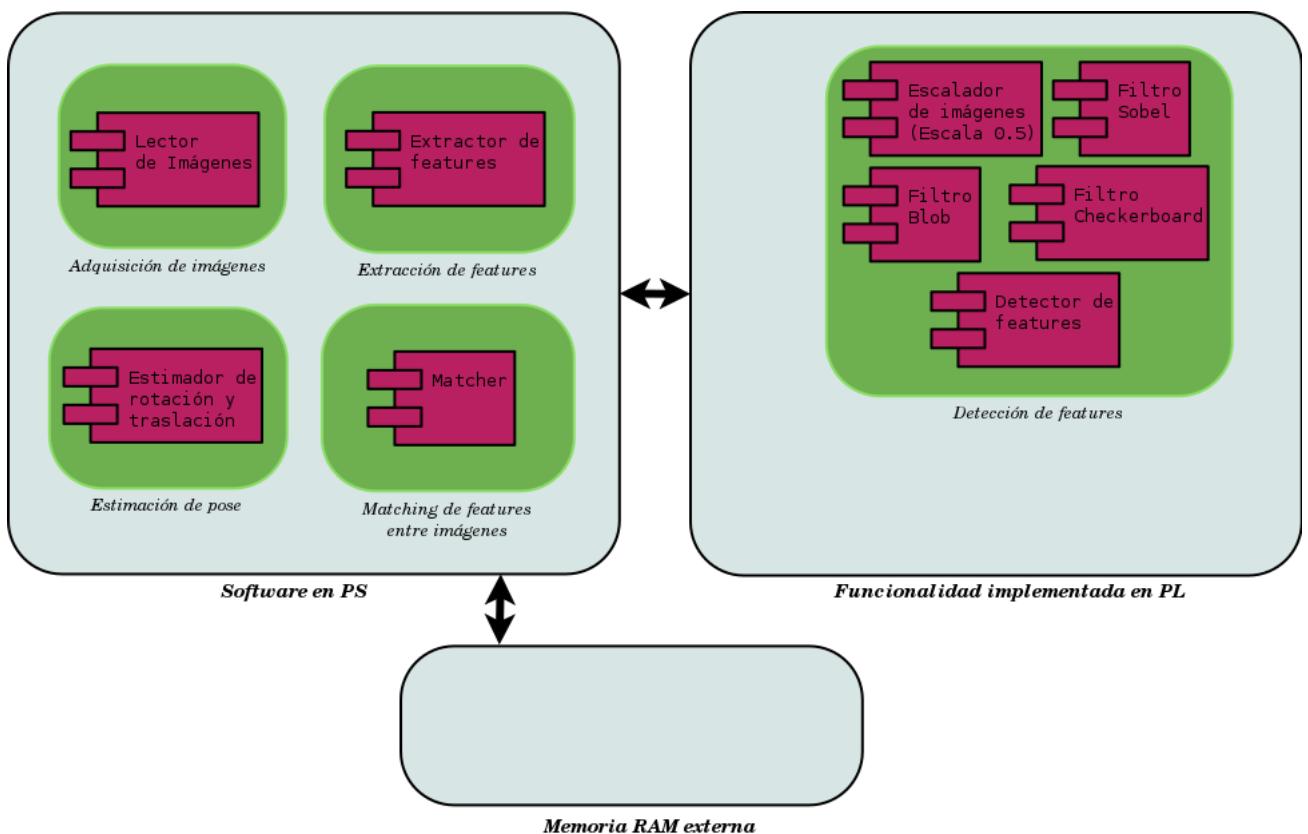


Figura 7.14: Mapeo de funcionalidad a PS y PL.

En este proceso construimos los siguientes **IP Cores**:

- **Sobel**: Calcula los filtros *Sobel* horizontal y vertical de una imagen.
- **Half Resolution**: Escalador de imágenes a la mitad del tamaño.
- **Blob + Checkerboard**: Calcula los filtros *Blob* y *Checkerboard* de una imagen.
- **Features**: Detecta puntos sobresalientes en una imagen utilizando **NMS**.
- **Controller**: Controla de manera sincronizada todos los **IP Cores**.

7.3.2. Comunicación entre procesadores, IP Cores y memoria externa

En el siguiente diagrama de **buses** pueden observarse las señales que permite comunicar procesadores, **IP Cores** y memoria RAM externa.

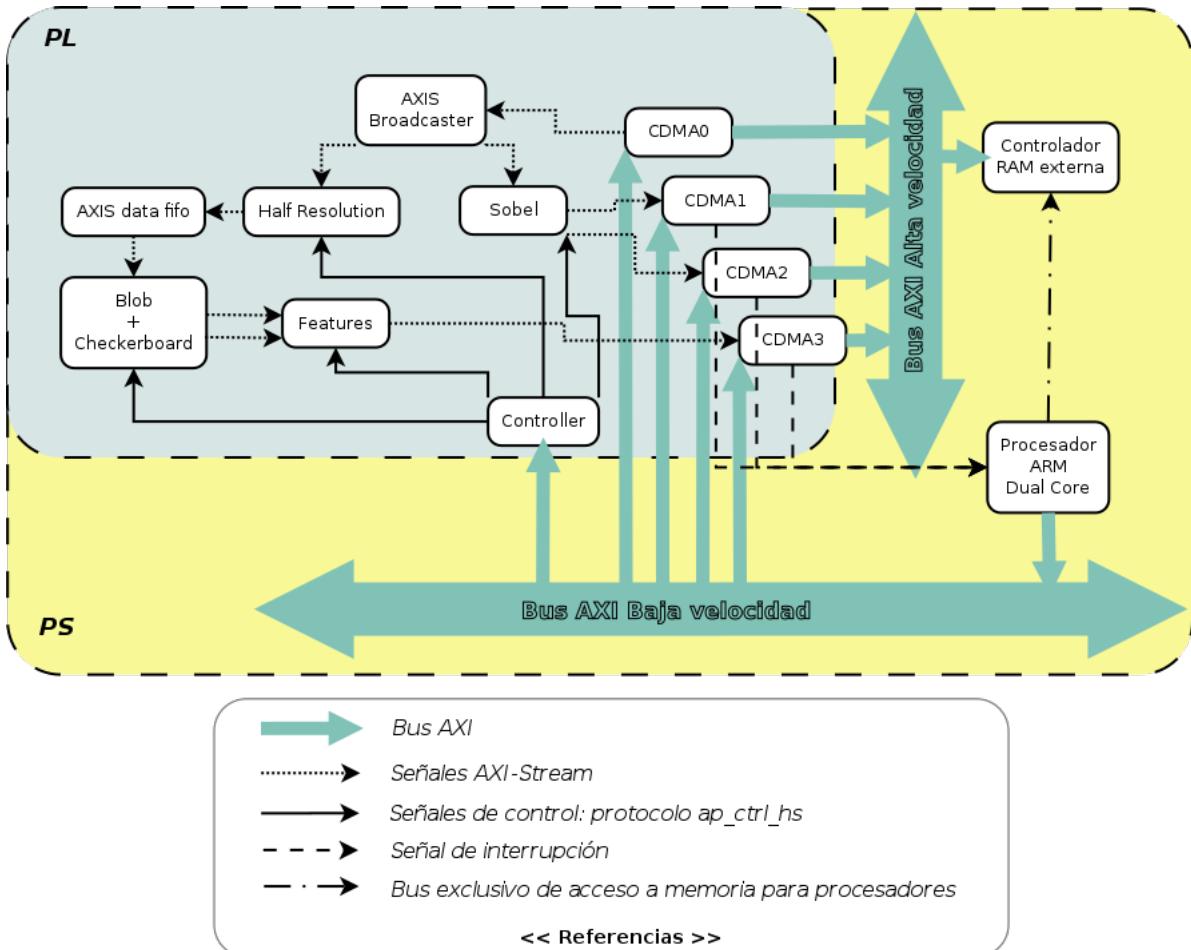


Figura 7.15: Diagrama de buses.

- **AXI** (Advanced eXtensible Interface)⁸ es la tercera generación de la interfaz **AMBA** (Advanced Microcontroller Bus Architecture), el estándar para la interconexión y control de bloques funcionales en diseños **SoC** adoptado por **ARM** y **Xilinx**.
- **AXI-Stream**⁹ es un protocolo usado en aplicaciones enfocadas en el flujo de datos donde el concepto de **dirección de memoria** no es necesario. Cada **stream** AXI se comporta como un canal unidireccional que permite el intercambio de datos entre un productor y un consumidor con señales de **handshake**.
- **ap_ctrl_hs**¹⁰: los **IP Cores** que implementan este protocolo de control cuentan con dos señales: **ap_start** y **ap_done**. Cuando se le asigna el valor “1” a **ap_start**, el **IP Core** comienza a funcionar. Al finalizar este activara la señal **ap_done**.

7.3.3. Pipeline de procesamiento de imágenes

En la implementación que construimos los **IP Cores Sobel, Half Resolution, Blob + Checkerboard** y **Features** forman un **pipeline** de procesamiento, donde la salida de

⁸Xilinx 2011.

⁹Xilinx 2011.

¹⁰Xilinx 2014b.

74 CAPÍTULO 7. OPTIMIZACIÓN MEDIANTE ACCELERACIÓN POR HARDWARE

uno es entrada de otro, como muestra la figura a continuación:

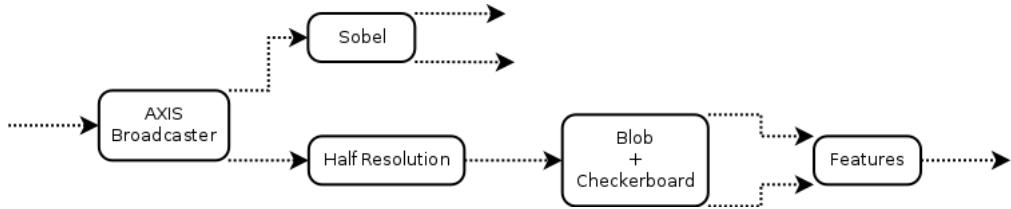


Figura 7.16: Pipeline de procesamiento de imágenes para detección de **features**.

De esta forma un **IP Core** que consume la salida de otro no tiene que esperar a que el primero termine de procesar una imagen completa, sino que va consumiendo datos a medida que el primero los produce. Esto permite minimizar la latencia de procesamiento de una imagen.

Analizamos el **pipeline** para calcular la **productividad** (medida en $\frac{\text{datos producidos}}{\text{ciclo de reloj}}$) de cada uno de los **IP Cores** necesaria para evitar cuellos de botella que bloquen el **pipeline**.

La información obtenida permite evitar la aplicación de **optimizaciones de velocidad**, que tienen un impacto negativo en el consumo de recursos, que no proveen una ganancia en la velocidad de procesamiento de la solución completa. Es decir optimizaciones que llevan a que un **IP Core** tenga una productividad mayor a la velocidad de consumo de datos del **IP Core** que consume su salida.

La siguiente figura ilustra la productividad y la velocidad de consumo de datos de cada **IP Core** necesarias para evitar cuellos de botella y acotar el uso de **buffers** a unas pocas líneas de imagen en función de una velocidad de entrada de datos al **pipeline** de $X \frac{\text{datos}}{\text{ciclo}}$.

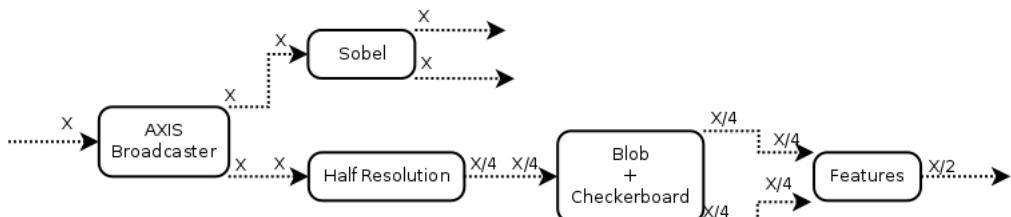


Figura 7.17: Análisis de requerimientos de productividad de los IP Cores del **pipeline** de procesamiento de imágenes.

El análisis realizado sigue las siguientes reglas:

- Si la velocidad de consumo de datos de un **IP Core** desde un **stream** es T y cada n unidades de datos consumidos se producen como máximo m en un **stream** de salida, entonces la productividad debe ser al menos $T * \frac{m}{n}$ para evitar bloquear el **pipeline**.
- Asume **productividades constantes**
- Si dos **IP Cores** consumen un **stream**, su **productividad** es igual a la del más lento.

Luego de analizados los requerimientos relativos de **productividad** entre los **IP Cores**, decidimos comenzar la construcción diseñando en primer lugar el más complejo y difícil de optimizar. A partir del análisis de la información de **profiling** de `viso_s` y el estudio de su código fuente, decidimos comenzar por el **IP Core Features**. La información obtenida nos permitió obtener cotas máximas para la **productividad** y evitar optimizar demás otros componentes.

La sección [7.4.4](#) describe el desarrollo del **IP Core Features** para el cual obtuvimos los siguientes rendimientos:

velocidad de consumo	productividad
$\sim 0,5 \frac{\text{pixel}}{\text{ciclo}}$	$\sim 1 \frac{\text{feature}}{\text{ciclo}}$

Cuadro 7.1: Productividad del IP Core **Features**

Con estos datos y el análisis anterior obtuvimos las siguientes **productividades** mínimas para evitar bloqueos del **pipeline**:

IP Core	velocidad de consumo	productividad
Sobel	$\sim 2 \frac{\text{pixel}}{\text{ciclo}}$	$\sim 2 \frac{\text{pixel}}{\text{ciclo}}$
Half Resolution	$\sim 2 \frac{\text{pixel}}{\text{ciclo}}$	$\sim 0,5 \frac{\text{pixel}}{\text{ciclo}}$
Blob + Checkerboard	$\sim 0,5 \frac{\text{pixel}}{\text{ciclo}}$	$\sim 0,5 \frac{\text{pixel}}{\text{ciclo}}$
Features	$\sim 0,5 \frac{\text{pixel}}{\text{ciclo}}$	$\sim 1 \frac{\text{feature}}{\text{ciclo}}$

Cuadro 7.2: Velocidades de consumo de datos y productividades mínimas de los IP Cores para evitar bloquear el **pipeline**

Utilizando esta información en la siguiente sección construimos y optimizamos los distintos **IP Cores**.

7.4. IP Cores

Esta sección presenta el desarrollo de los IP Cores: **Sobel**, **Half Resolution**, **Blob + Checkerboard**, **Features** y **Controller**. Para esto utilizamos las técnicas y optimizaciones presentadas en [7.2](#).

La implementación de estos **IP Cores** puede encontrarse en el apéndice `ipcores.tar.gz`.

7.4.1. IP Core Sobel

Este **IP Core** calcula para una imagen sus filtros *Sobel* horizontal y vertical. Los mismos se obtienen convolucionando la imagen con las matrices (a) y (b) de la figura a continuación, respectivamente.

$$\begin{bmatrix} -1 & -2 & 0 & 2 & 1 \\ -4 & -8 & 0 & 8 & 4 \\ -6 & -12 & 0 & 12 & 6 \\ -4 & -8 & 0 & 8 & 4 \\ -1 & -2 & 0 & 2 & 1 \end{bmatrix}$$

(a) Matriz de convolución para el filtro Sobel horizontal

$$\begin{bmatrix} -1 & -4 & -6 & -4 & -1 \\ -2 & -8 & -12 & -8 & -2 \\ 0 & 0 & 0 & 0 & 0 \\ 2 & 8 & 12 & 8 & 2 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

(b) Matriz de convolución para el filtro Sobel vertical

Aplicamos el patrón de diseño descrito en [7.2.2](#) junto con la optimización “fusión de IP Cores” ([7.2.3](#)) para implementar ambas convoluciones en un mismo **IP Core**.

Debido a las restricciones de **productividad** obtenidas en la sección [7.3.3](#) decidimos emplear la optimización “K píxeles por ciclo” ([7.2.3](#)) para procesar 2 píxeles en cada ciclo.

velocidad de consumo de datos	productividad
$\sim 2 \frac{\text{pixel}}{\text{ciclo}}$	$\sim 2 \frac{\text{pixel}}{\text{ciclo}}$

Cuadro 7.3: Restricciones de productividad para el IP Core Sobel

Además agregamos lógica para que las imágenes de salida estén alineadas con la de entrada. Sin esta modificación las primeras 5 líneas del resultado contendrían basura pues la ventana (**Window**) no contendría todavía valores válidos en todas sus posiciones.

```
...
if(row > 2 || (row == 2 && col > 0))
{
    ... // Escribir streams de salida
}
...
```

Figura 7.19: Lógica para alinear imágenes de salida con la imagen de entrada

Las imágenes resultantes tienen dos filas menos que la original, pues el **IP Core** trata a los píxeles que se encuentran a una distancia menor a 2 píxeles de los bordes, es decir aquellos para los cuales no se cuenta con todos sus vecinos para calcular la convolución, de la siguiente manera:

- Borde inferior: no figuran en la imagen resultante.
- Bordes superior, izquierda y derecho: completados con basura.

Los píxeles de las imágenes resultantes son de 8 bits, para evitar **overflows** los resultados se saturan (si es menor a 0, se asigna el valor 0. Si es mayor a 255, se asigna el valor 255).

Antes de sintetizar el **IP Core**, realizamos una simulación del código. Para esto creamos un programa que lo invoca para procesar una imagen guardada en un archivo y guarda el resultado en dos archivos de salida.

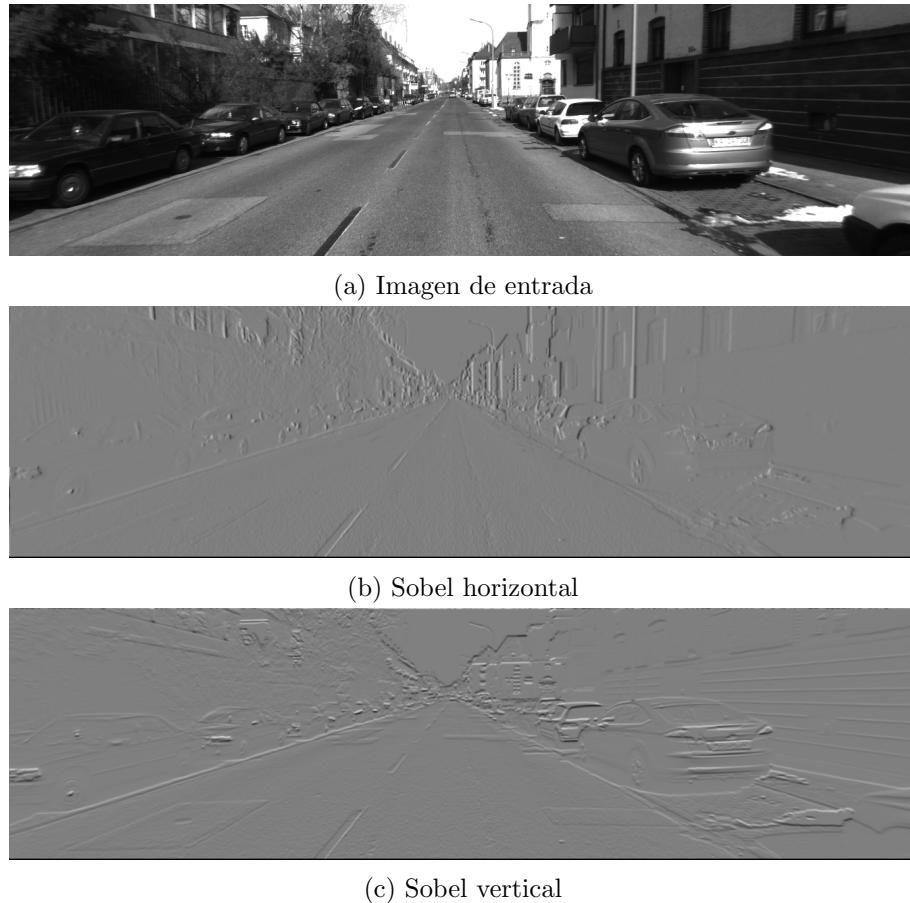


Figura 7.20: Imágenes de entrada y salidas del código a sintetizar

Dado que este **IP Core** procesa 2 píxeles (16 bits) por ciclo, lo llamamos **Sobel16**. Al sintetizar el código con **Vivado HLS** se generó el siguiente IP Core:



Figura 7.21: IP Core **Sobel16**.

La figura a continuación muestra el consumo estimado de recursos:

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	1018
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	8	-	0	0
Multiplexer	-	-	-	29
Register	-	-	813	4
Total	8	0	813	1051
Available	120	80	35200	17600
Utilization (%)	6	0	2	5

Figura 7.22: Consumo de recursos del IP Core **Sobel16**.

Utilizando la directiva **LOOP_TRIPCOUNT** obtuvimos la siguiente información sobre velocidad de procesamiento para una imagen de 1344x372 píxeles:

Latency		Interval			
min	max	min	max	Type	
252217	252217	252218	252218	none	

Figura 7.23: Latencia e intervalo de iniciación del IP Core **Sobel16**.

Dado que en cada iteración se leen 2 píxeles del **stream** de entrada y se escriben 2 píxeles en cada **stream** de salida, la velocidad de consumo de datos y la productividad son iguales. Las mismas son de $\frac{1344 \times 372 \text{ pixels}}{252217 \text{ ciclos}} = \sim 1,98 \frac{\text{pixels}}{\text{ciclo}}$. Muy cercano a los mínimos requeridos.

7.4.2. IP Core Half Resolution

Este **IP Core** escala una imagen a la mitad de su tamaño. Para esto la imagen es dividida en cuadrados de 2x2 píxeles. Luego para cada cuadrado la imagen resultante tiene en su posición un píxel cuyo valor es el promedio de los 4 píxeles del cuadrado.

Debido a las restricciones de **productividad** obtenidas en la sección 7.3.3 este **IP Core** debe leer 2 píxeles en cada ciclo y escribir un píxel cada 2 ciclos.

velocidad de consumo de datos	productividad
$\sim 2 \frac{\text{pixel}}{\text{ciclo}}$	$\sim 0,5 \frac{\text{pixel}}{\text{ciclo}}$

Cuadro 7.4: Restricciones de productividad para el IP Core **Half Resolution**

El **IP Core** trabaja guardando en un **buffer** las filas pares y calculando el resultado para cada cuadrado de 2x2 en las filas impares. Al leer de a 2 píxeles no es necesario el uso de una ventana de registros.

```

1: function HALFRESOLUTION(input_stream, output_stream)
2:   for i=0; i< alto/2; i ++ do
3:     for j=0; j< ancho/2; j ++ do
4:       nuevos_2_pixeles ← READ(input_stream)
5:       LineBuffer[j] ← nuevo_pixel
6:     end for
7:     for j=0; j< ancho/2; j ++ do
8:       abajo_2_pixeles ← READ(input_stream)
9:       arriba_2_pixeles ← LineBuffer[j]
10:      resultado ← (abajo_2_pixeles[0 : 7] + abajo_2_pixeles[8 : 15] +
    arriba_2_pixeles[0 : 7] + arriba_2_pixeles[8 : 15])/4
11:      WRITE(output_stream, resultado)
12:    end for
13:  end for
14: end function

```

Algoritmo 3: Algoritmo para escalar una imagen. El ancho del **stream** de entrada es de 2 píxeles y el de salida 1 píxel

El algoritmo utilizado asume que tanto el ancho como el alto de la imagen son enteros pares. Este supuesto permitió simplificar la implementación.

En este caso no aplicamos el patrón de diseño descrito en 7.2.2, pues el procesamiento de cada píxel depende de su posición en la imagen, algunos solo deben guardarse en el **buffer** y para otros debe calcularse una salida, y el patrón no abarca estos casos.

Sin embargo, el estudio de optimizaciones aplicables al patrón resultó útil para optimizar el código de este **IP Core**. En particular pudimos aplicar las optimizaciones “K píxeles por ciclo” (7.2.3) y “Pipeline” (7.2.3).

Dado que este **IP Core** procesa 2 píxeles (16 bits) por ciclo, lo llamamos **HalfResolution16**.

80 CAPÍTULO 7. OPTIMIZACIÓN MEDIANTE ACCELERACIÓN POR HARDWARE

```

void half_resolution16(ap_axiu<16,1,1,1> *src, ap_axiu<8,1,1,1> *dest, u16 width, u16 height)
{
    #pragma HLS INTERFACE ap_ctrl_hs port=return
    #pragma HLS INTERFACE axis port=src
    #pragma HLS INTERFACE axis port=dest
    #pragma HLS INTERFACE ap_none port=width
    #pragma HLS INTERFACE ap_none port=height

    u16 line_buffer[IMAGE_MAX_WIDTH / 2];
    u16 row, input_col;

    for_row:for(row=0; row<height/2; row++)
    {
        #pragma HLS LOOP_TRIPCOUNT min=186 max=186 avg=186
        for_col_even_row:for(input_col=0; input_col<width/2; input_col++)
        {
            #pragma HLS LOOP_TRIPCOUNT min=672 max=672 avg=672
            line_buffer[input_col] = src->data;
            src++;
        }
        for_col_odd_row:for(input_col=0; input_col<width/2; input_col++)
        {
            #pragma HLS LOOP_TRIPCOUNT min=672 max=672 avg=672
            #pragma HLS PIPELINE rewind
            u16 new_pix = src->data;

            u16 up = line_buffer[input_col];
            dest->data = ((new_pix >> 8)+(new_pix & 0xFF) + (up >> 8) + (up & 0xFF)) /4;
            dest->last = src->last;
            dest->keep = 1;

            src++;
            dest++;
        }
    }
}

```

Figura 7.24: Extracto del código a sintetizar para el IP Core **HalfResolution16**

Antes de sintetizar el **IP Core**, realizamos una prueba del código. Para esto creamos un programa que lo invoca para procesar una imagen guardada en un archivo y guarda el resultado en un archivo de salida.



(a) Imagen de entrada



(b) Imagen escalada

Figura 7.25: Imágenes de entrada y salida del código a sintetizar

Al sintetizar el código con Vivado HLS se generó el siguiente **IP Core**:

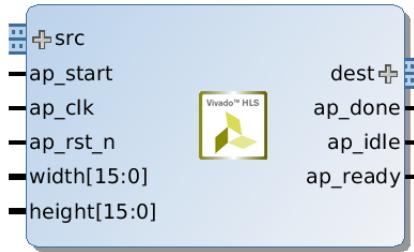


Figura 7.26: IP Core **HalfResolution16**.

La figura a continuación muestra el consumo estimado de recursos:

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	173
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	1	-	0	0
Multiplexer	-	-	-	76
Register	-	-	161	-
Total	1	0	161	249
Available	120	80	35200	17600
Utilization (%)	~0	0	~0	1

Figura 7.27: Consumo de recursos del IP Core **HalfResolution16**.

Utilizando la directiva **LOOP_TRIPCOUNT** obtuvimos la siguiente información sobre velocidad de procesamiento para una imagen de 1344x372 píxeles:

Latency		Interval		
min	max	min	max	Type
1	249571	2	249572	none

Figura 7.28: Latencia e intervalo de iniciación del IP Core **HalfResolution16**.

Considerando la latencia máxima, la velocidad de consumo de datos promedio puede calcularse como $\frac{1344 \times 372 \text{ pixels}}{249571 \text{ ciclos}} = \sim 2 \frac{\text{pixels}}{\text{ciclo}}$. Análogamente la **productividad** es $\frac{672 \times 186 \text{ pixels}}{249571 \text{ ciclos}} = \sim 0,5 \frac{\text{pixels}}{\text{ciclo}}$, pues cada 4 píxeles leídos se produce un píxel.

Si bien los rendimientos obtenidos son muy cercanos a los mínimos requeridos, no se cumple la premisa de **productividad** constante. Esto se debe a que para las filas pares la **productividad** de $0 \frac{\text{pixels}}{\text{ciclo}}$ pues no se realizan escrituras, mientras que para las filas impares la **productividad** es $1 \frac{\text{pixels}}{\text{ciclo}}$.

Si se agrega una cola con capacidad para al menos una fila completa de la imagen escalada a la salida de este **IP Core**, entonces pueden consumirse de la cola $0,5 \frac{\text{pixels}}{\text{ciclo}}$ de manera constante. Es decir que considerando al **IP Core** y la cola como un todo, se puede asumir una **productividad constante** de $0,5 \frac{\text{pixels}}{\text{ciclo}}$.

Al integrar el IP Core a la solución agregamos una cola para satisfacer los supuestos del análisis del **pipeline** de procesamiento realizado en 7.3.3.

7.4.3. IP Core Blob + Checkerboard

Este IP Core calcula para una imagen sus filtros *Blob* y *Checkerboard*. Los mismos se obtienen convolucionando la imagen con las matrices (a) y (b) de la figura a continuación, respectivamente.

$$\begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 & -1 \\ -1 & 1 & 8 & 1 & -1 \\ -1 & 1 & 1 & 1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

(a) Matriz de convolución para el filtro **Blob**

$$\begin{bmatrix} -1 & -1 & 0 & 1 & 1 \\ -1 & -1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & -1 & -1 \\ 1 & 1 & 0 & -1 & -1 \end{bmatrix}$$

(b) Matriz de convolución para el filtro **Checkerboard**

Aplicamos el patrón de diseño descrito en [7.2.2](#) junto con la optimización “fusión de IP Cores” ([7.2.3](#)) para implementar ambas convoluciones en un mismo **IP Core**.

Agregamos lógica para que las imágenes de salida estén alineadas con la de entrada. Sin esta modificación las primeras 5 líneas del resultado contendrían basura pues la ventana (**Window**) no contendría valores válidos en todas sus posiciones.

```
...
if(row > 2 || (row == 2 && col > 0))
{
    ... // Escribir streams de salida
}
...
```

Figura 7.30: Lógica para alinear imágenes de salida con la imagen de entrada

Las imágenes resultantes tienen el mismo tamaño que la original. El **IP Core** trata a los píxeles que se encuentran a una distancia menor a 2 píxeles de los bordes, es decir aquellos para los cuales no se cuenta con todos sus vecinos para calcular la convolución, de la siguiente manera:

- Borde inferior: completado con 0s.
- Bordes superior, izquierda y derecho: completados con basura.

Los píxeles de las imágenes resultantes son de 16 bits para evitar **overflows**.

Antes de sintetizar el **IP Core**, realizamos una prueba del código. Para esto creamos un programa que lo invoca para procesar una imagen guardada en un archivo y guarda el resultado en dos archivos de salida.

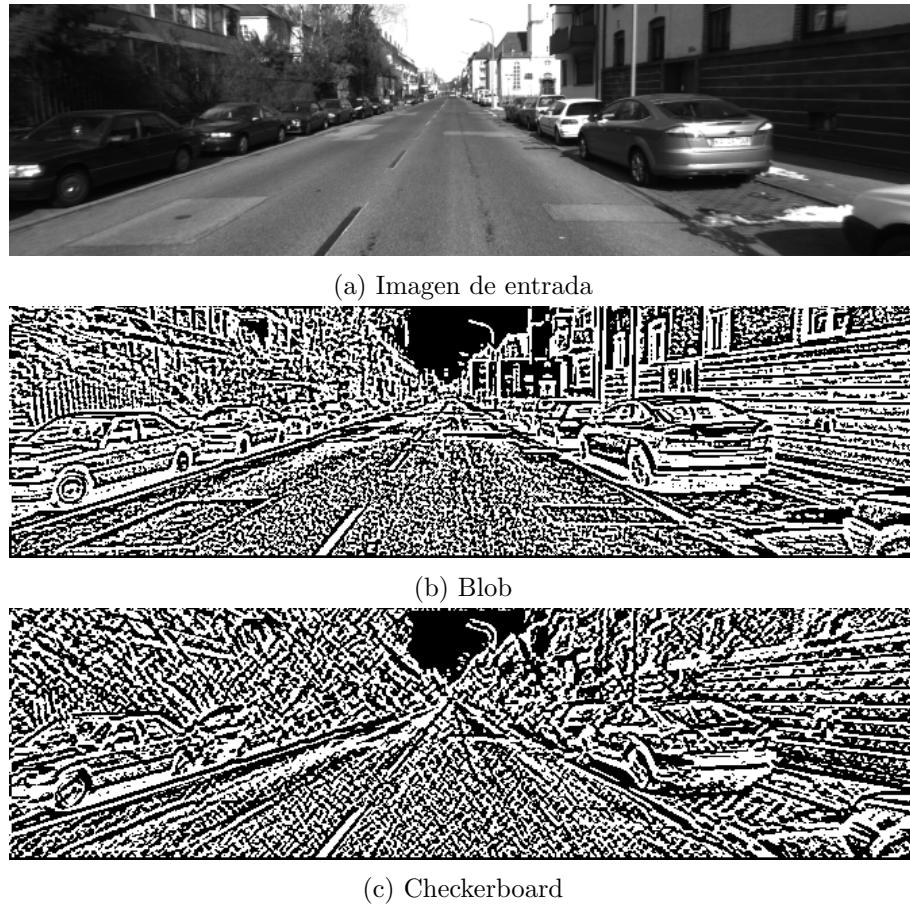


Figura 7.31: Imágenes de entrada y salidas del código a sintetizar

Al sintetizar el código con Vivado HLS se generó el siguiente **IP Core**:



Figura 7.32: IP Core **Blob+Checkerboard**.

La figura a continuación muestra el consumo estimado de recursos:

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	516
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	4	-	0	0
Multiplexer	-	-	-	93
Register	-	-	666	119
Total	4	0	666	728
Available	120	80	35200	17600
Utilization (%)	3	0	1	4

Figura 7.33: Consumo de recursos del IP Core **Blob+Checkerboard**.

Utilizando la directiva **LOOP_TRIPCOUNT** obtuvimos la siguiente información sobre velocidad de procesamiento para una imagen de 672x186 píxeles:

Latency		Interval		
min	max	min	max	Type
127712	127712	127713	127713	none

Figura 7.34: Latencia e intervalo de iniciación del IP Core **Blob+Checkerboard**.

Dado que en cada iteración se lee un píxel del **stream** de entrada y se escribe un píxel en cada **stream** de salida, la **productividad** y la **velocidad de consumo de datos** son iguales. Las mismas son de $\frac{672 \times 186 \text{ pixeles}}{127712 \text{ ciclos}} = \sim 0,98 \frac{\text{pixeles}}{\text{ciclo}}$. Mayores a las mínimas requeridas, aún sin haber aplicado otras optimizaciones.

7.4.4. IP Core Features

Este **IP Core** detecta de manera **simultanea features** dispersos y densos en dos imágenes con píxeles de 16 bits.

Como se explica en [3.2.1](#), *LIBVISO2* aplica dos veces el algoritmo **Non-maximum Supression (NMS)** al resultado de los filtros *Blob* y *Checkerboard*, variando los tamaños de ventana, para detectar píxeles cuyos valores son mínimos o máximos locales.

Para esto se partitiona cada imagen en una grilla con celdas de $N \times N$ píxeles. Un píxel es máximo si es el mayor dentro de una celda de la grilla, es mayor a **nms_tau** (el valor **nms_tau** es configurable) y, además, dentro de una ventana de $2N \times 2N + 1$ píxeles a su alrededor no existe un píxel con valor mayor. Análogamente se detectan píxeles mínimos.

El **IP Core** implementado utiliza una simplificación del algoritmo **NMS**, cambiando la definición de máximo y mínimo. Un píxel es máximo si es mayor a **nms_tau** y dentro de una ventana de $2N \times 2N + 1$ píxeles a su alrededor no existe un píxel con valor mayor. Análogamente para la detección de mínimos.

Decidimos utilizar valores de $N = 10$ para la detección de **features** dispersos y $N = 5$ para **features** densos.

El siguiente algoritmo ejemplifica de que manera el **IP Core** cataloga a los píxeles de una imagen en 5 tipos: “mínimo disperso”, “mínimo denso”, “máximo disperso”, “máximo denso” y “nada”.

```

1: for i=0; i< alto; i ++ do
2:   for j=0; j< ancho; j ++ do
3:     if imagen[i][j] <= min(imagen[i - 10 : i + 10][j - 10 : j + 10]) then
4:       tipo_feature ← MIN_DISPERSO
5:     else if imagen[i][j] <= min(imagen[i - 5 : i + 5][j - 5 : j + 5]) then
6:       tipo_feature ← MIN_Denso
7:     else if imagen[i][j] >= max(imagen[i - 10 : i + 10][j - 10 : j + 10]) then
8:       tipo_feature ← MAX_DISPERSO
9:     else if imagen[i][j] >= max(imagen[i - 5 : i + 5][j - 5 : j + 5]) then
10:      tipo_feature ← MAX_Denso
11:    else
12:      tipo_feature ← NADA
13:    end if
14:  end for
15: end for

```

Algoritmo 4: Algoritmo para detectar **features** densos y dispersos en una imagen

Este algoritmo puede implementarse utilizando el patrón 7.2.2 con píxeles de 16 bits. En un primer análisis puede verse que, dado que la ventana que se utiliza para detectar mínimos y máximos densos es una sub ventana de la que se utiliza para los dispersos, basta con utilizar 21 **Line Buffers** y una sola **Window** de 21x21 celdas, pues utilizamos $N = 10$ y son necesarios $2N + 1$ **Line Buffers** y una ventana de $2N + 1 \times 2N + 1$.

Analizando con mayor detenimiento el uso de la ventana en el algoritmo notamos que es posible **condensar** su información y, de esta manera, reducir el uso de recursos. Esto es posible, pues para cada configuración de la ventana no es necesario conocer el valor de todas las celdas, sino que alcanza con conocer para cada columna el valor y fila del máximo y mínimo.

Por lo tanto reemplazamos la ventana **Windowde** 21x21 celdas por 8 ventanas:

- **WinMinDensos** de 21 celdas. Contiene el valor mínimo de cada columna de **Window** entre las filas 5 y 15.
- **WinMinDispersos** de 21 celdas. Contiene el valor mínimo de cada columna de **Window**.
- **WinFilaMinDensos** de 21 celdas. Contiene el índice de fila del mínimo valor entre las filas 5 y 15 de cada columna de **Window**.
- **WinFilaMinDispersos** de 21 celdas. Contiene el índice de fila del mínimo valor de cada columna de **Window**.
- **WinMaxDensos**, **WinMaxDispersos**, **WinFilaMaxDensos** y **WinFilaMaxDispersos** de 21 celdas cada una, análogas a las anteriores, pero con información sobre máximos.

De esta forma una ventana de 21x21 celdas de 16 bits pudo ser reemplazada por cuatro de 21 celdas de 16 bits y otras cuatro de 21 celdas de 5 bits (pues los índices de fila están entre 0 y 20). Esto permite un **ahorro de recursos del 75 %** al reducir el consumo memoria de $21 * 21 * 16 = 7056$ bits a $21 * 4 * 16 + 21 * 4 * 5 = 1764$ bits.

86 CAPÍTULO 7. OPTIMIZACIÓN MEDIANTE ACCELERACIÓN POR HARDWARE

A continuación se muestra un algoritmo para la detección de mínimos densos en una imagen utilizando **ventanas con su información “condensada”**:

```

1: function DETECTARMINIMOSDENSOS(input_stream, output_stream, tau, ancho, alto)
2:   for i=0; i< alto; i ++
3:     for j=0; j< ancho; j ++
4:       nuevo_pixel ← READ(input_stream)
5:       for l=0; l< 20; l ++
6:         WinMinDensos[l] ← WinMinDensos[l + 1]                                ▷ Shift window loop
7:         WinFilaMinDensos[l] ← WinFilaMinDensos[l + 1]
8:       end for
9:       for k=0; k< 20; k ++
10:        LineBuffer[k][j] ← LineBuffer[k + 1][j]                               ▷ Shift line buffers loop
11:      end for
12:      LineBuffer[20][j] ← nuevo_pixel                                         ▷ Obtener información de mínimo denso en la columna actual
13:      min_denso ← tau
14:      min_denso_row ← 0
15:      for k=5; k<= 15; k ++
16:        if LineBufer[k][j] < min_denso then
17:          min_denso ← LineBufer[k][j]
18:          min_denso_row ← j
19:        end if
20:      end for                                                               ▷ Guardar información para la columna actual en ventanas condensadas
21:      WinMinDensos[20] ← min_denso
22:      WinFilaMinDensos[20] ← min_denso_row
23:      if i >= 20 and j >= 15 then      ▷ Chequear si el valor del centro de la ventana es mínimo denso
24:        es_min_denso ← true
25:        for x = 5; x <= 15; x ++
26:          if WinMinDensos[x] < WinMinDensos[10] then
27:            es_min_denso ← false
28:          end if
29:        end for
30:        if WinFilaMinDensos[10]! = 10 or WinMinDensos[10] > -tau then
31:          es_min_denso ← false
32:        end if
33:      else
34:        es_min_denso ← false
35:      end if                                                               ▷ Computar salida
36:      if es_min_denso then
37:        WRITE(output_stream, Feature(MIN_DENSO, i - 10, j - 10, WinMinDensos[10]))
38:      end if
39:    end for
40:  end for
41: end function

```

Algoritmo 5: Detección de mínimos densos en una imagen utilizando ventanas “condensadas”

A partir del algoritmo anterior, con el objetivo de compartir la lógica de control y el **stream** de salida, implementamos en Vivado HLS la detección de **features**, es decir, mínimos y máximos dispersos y densos, para dos imágenes en simultaneo.

Utilizamos un **stream** de salida de 64 bits, codificando la información de **features** detectados de la siguiente forma:

Contenido	Tipo de Feature	Valor	Columna	Fila
Bits	63..48	47..32	31..16	15..0

Cuadro 7.5: Codificación de **features** en **stream** de salida.

0	mínimo disperso de la imagen 0
1	máximo disperso de la imagen 0
2	mínimo disperso de la imagen 1
3	máximo disperso de la imagen 1
4	mínimo denso de la imagen 0
5	máximo denso de la imagen 0
6	mínimo denso de la imagen 1
7	máximo denso de la imagen 1

Cuadro 7.6: Codificación de tipos de **features**.

La implementación realizada al terminar escribe un **feature** con todos sus valores en 0 para indicar el fin, facilitando el posterior uso de los **features** detectados desde el software.

Antes de sintetizar el **IP Core**, realizamos una simulación del código. Para esto creamos un programa que lo invoca para procesar una dos imágenes guardadas en archivos y muestra por salida estándar información sobre los **features** detectados.



(a) Imagen de entrada 0.



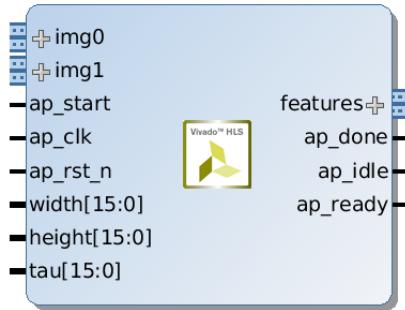
(b) Imagen de entrada 1.

fila	columna	valor	tipo
...			
16	137	1658	1
16	138	-395	2
16	175	-1483	0
16	215	561	7
16	355	182	7
16	358	-98	6
...			

(c) **Features** detectados.

Figura 7.35: Imágenes de entrada y salida del código a sintetizar

Al sintetizar el código con Vivado *HLS* se generó el siguiente **IP Core**:

Figura 7.36: IP Core **Features**.

La figura a continuación muestra el consumo estimado de recursos:

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	9476
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	40	-	0	0
Multiplexer	-	-	-	233
Register	-	-	6763	137
Total	40	0	6763	9846
Available	120	80	35200	17600
Utilization (%)	33	0	19	55

Figura 7.37: Consumo de recursos del IP Core **Features**.

Utilizando la directiva **LOOP_TRIPCOUNT** obtuvimos la siguiente información sobre la velocidad de procesamiento para imágenes de 672x186 píxeles:

Latency		Interval		
min	max	min	max	Type
255380	255380	255381	255381	none

Figura 7.38: Latencia e intervalo de iniciación del IP Core **Features**.

Dado que en cada iteración se lee un píxel de cada **stream** de entrada, la **velocidad de consumo de datos** es $\frac{672 \times 186 \text{ pixels}}{255380 \text{ ciclos}} = \sim 0,49 \frac{\text{pixels}}{\text{ciclo}}$.

La cantidad de **features** detectables está acotada por la cantidad de píxeles en cada imagen. Entonces la productividad esta acotada por $\frac{2 \times 672 \times 186 \text{ features}}{255380 \text{ ciclos}} = \sim 0,98 \frac{\text{features}}{\text{ciclo}}$.

Si bien no es cierto que la productividad sea constante, ya que no todos los píxeles de una imagen son **features**, al ser este el último **IP Core** del **pipeline**, solo su velocidad de lectura tiene un impacto sobre los bloqueos de este.

7.4.5. IP Core Controller

Este **IP Core** permite al procesador controlar de manera coordinada todos los **IP Cores** del **pipeline** de procesamiento de imágenes.

A través del bus **AXI** el procesador le indica a este **IP Core** las dimensiones de la imagen a procesar y el valor de **tau** a utilizar en la detección de **features**. Además el procesador indica al **IP Core** cuando comenzar a procesar una imagen.

Con la información provista por el procesador el **IP Core Controller** genera las siguientes señales, que permiten controlar a los **IP Cores Sobel16**, **Half Resolution16**, **Blob + Checkerboard** y **Features**:

- **slv_width**: ancho de la imagen a procesar.
- **slv_height**: alto de la imagen a procesar.
- **slv_half_width**: ancho de la imagen luego de ser escalada.
- **slv_half_height**: alto de la imagen luego de ser escalada.
- **slv_tau**: valor a **tau** a usar en la detección de **features**.
- **slv_start**: indica a los **IP Cores** que comiencen a procesar.

Este **IP Core** no contiene lógica propia para el procesamiento de imágenes, por lo tanto no tiene sentido intentar aplicar el patrón [7.2.2](#).

El código a continuación muestra de que forma el **IP Core** establece los valores de las señales de salida a partir de los recibidos por el bus **AXI** y cómo activa durante un ciclo la señal **slv_start** para que los demás **IP Cores** comiencen a procesar una imagen.

```
void controller(u16 width, u16 height, s16 tau,
               volatile bool &slv_start, u16 &slv_width, u16 &slv_height,
               u16 &slv_half_width, u16 &slv_half_height, s16 &slv_tau)
{
    #pragma HLS INTERFACE ap_none port=slv_tau
    #pragma HLS INTERFACE ap_none port=slv_half_height
    #pragma HLS INTERFACE ap_none port=slv_half_width
    #pragma HLS INTERFACE ap_none port=slv_height
    #pragma HLS INTERFACE ap_none port=slv_width
    #pragma HLS INTERFACE ap_none port=slv_start
    #pragma HLS INTERFACE s_axilite port=tau
    #pragma HLS INTERFACE s_axilite port=height
    #pragma HLS INTERFACE s_axilite port=width
    #pragma HLS INTERFACE s_axilite port=return

    slv_width = width;
    slv_height = height;
    slv_half_width = width / 2;
    slv_half_height = height / 2;
    slv_tau = tau;
    slv_start = true;
    ap_wait();
    slv_start = false;
}
```

Figura 7.39: Código fuente del IP Core **Controller**.

Sintetizando el código anterior con Vivado *HLS* se generó el siguiente **IP Core**:

Figura 7.40: IP Core **Controller**.

La figura a continuación muestra el consumo estimado de recursos:

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	-	548	700
Memory	-	-	-	-
Multiplexer	-	-	-	78
Register	-	-	80	-
Total	0	0	628	778
Available	120	80	35200	17600
Utilization (%)	0	0	1	4

Figura 7.41: Consumo de recursos del IP Core **Controller**.

Observamos que el consumo de **LUTs** y **Flip-Flops** (FF) es alto si lo comparamos con otros **IP Cores**, como por ejemplo **Half Resolution 16**, que realizan tareas más complejas.

Analizando en detalle el consumo de recursos encontramos que esto se debe al uso de la interfaz **AXI Slave**.

Instance	Module	BRAM_18K	DSP48E	FF	LUT
controller_AXILiteS_s_axi_U	controller_AXILiteS_s_axi	0	0	548	700
Total		1	0	0	548 700

Figura 7.42: Consumo de recursos de la interfaz **AXI Slave** del IP Core **Controller**.

Junto con el objetivo de forzar la coordinación entre los **IP Cores** a nivel hardware, esta fue otra razón que nos llevo a utilizar este esquema de una **jerarquía de control**. Teniendo un solo **IP Core** que se conecta al procesador a través del bus **AXI**, en lugar de conectar todos los **IP Cores** al bus **AXI**, y que el procesador controle a cada uno individualmente, se logra un ahorro de recursos.

Al igual que para los otros **IP Cores**, obtuvimos información sobre la latencia e intervalo de iniciación:

Latency		Interval		
min	max	min	max	Type
1	1	2	2	none

Figura 7.43: Latencia e intervalo de iniciación del IP Core **Controller**.

En la siguiente sección 7.5 se pueden observar en detalle las interconexiones entre este **IP Core** y los **IP Cores** del **pipeline** de procesamiento.

7.5. Integración del hardware

Esta sección detalla como integramos los **IP Cores** sintetizados en la sección anterior a la plataforma de hardware inicial, construida anteriormente en [5.2](#), utilizando *Vivado IP Integrator*. También se detallan las pruebas realizadas sobre la plataforma utilizando *Vivado SDK*.

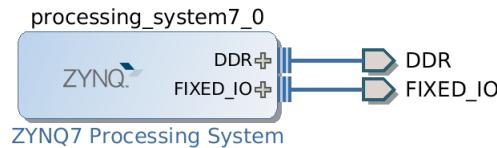


Figura 7.44: Vista del diseño de bloques de la plataforma de hardware inicial.

Dividimos estas tareas en los siguientes puntos: Configuración de PS [7.5.1](#), integración del **pipeline** de procesamiento de imágenes [7.5.2](#), integración y configuración de controladores DMA [7.5.3](#), generación del archivo **bitstream** [7.5.4](#) y prueba de la plataforma [7.5.5](#).

7.5.1. Configuración de PS

En este apartado detallamos la configuración del bloque ZYNQ7-PS para habilitar las señales necesarias para el control y comunicación con **IP Cores** instanciados en **PL**.

En primer lugar habilitamos las señales de **clock** y **reset** a ser usadas por los **IP Cores** instanciados en **PL**.

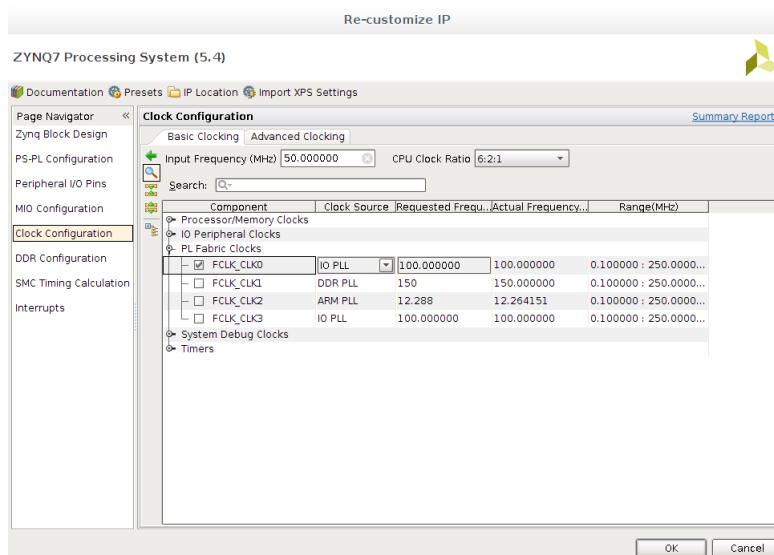


Figura 7.45: Configuración de señal de **clock** para **PL**.

92 CAPÍTULO 7. OPTIMIZACIÓN MEDIANTE ACCELERACIÓN POR HARDWARE

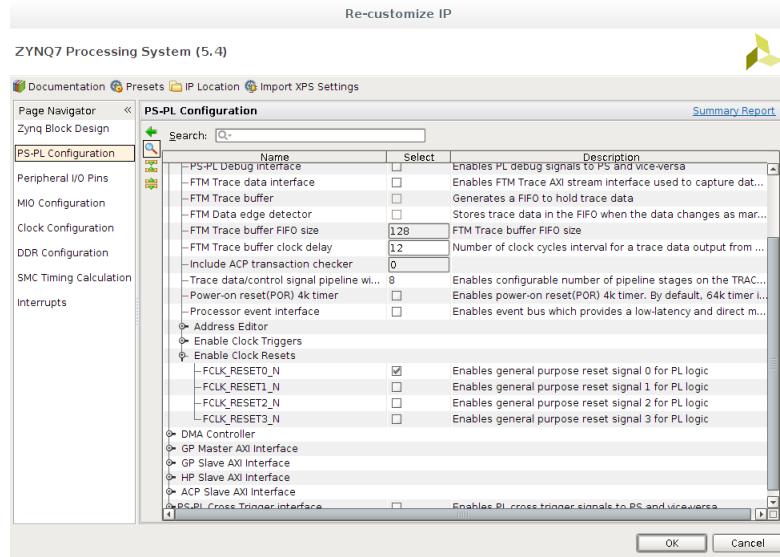


Figura 7.46: Configuración de señal de **reset** para PL.

Luego habilitamos el bus **AXI** de propósito general (**AXI GP**), necesario para que el procesador pueda controlar al **IP Core Controller** y a los controladores **DMA**.

También habilitados el bus **AXI** de alta velocidad (**AXI HP**), el mismo permite a los controladores **DMA** acceder al controlador de RAM externa sin intervención del procesador.

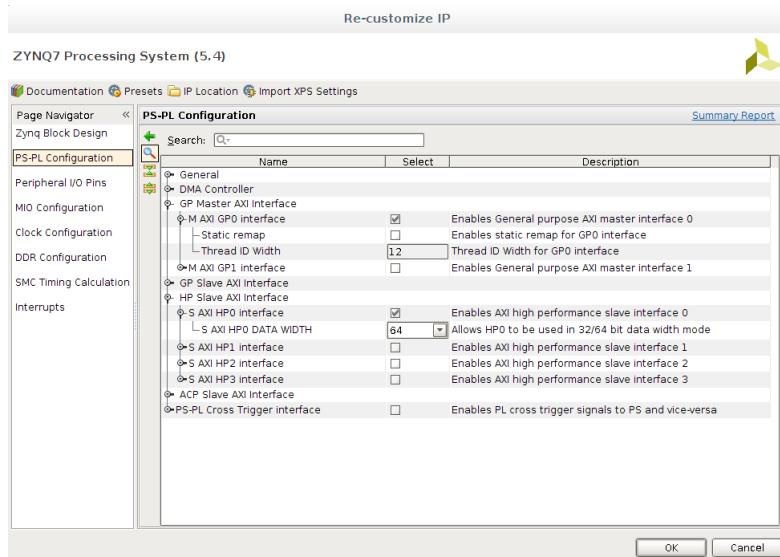


Figura 7.47: Configuración de buses AXI entre PS y PL.

Finalmente habilitamos las interrupciones provenientes de **PL** a **PS**, utilizadas por los controladores **DMA** para notificar el fin de las transacciones.

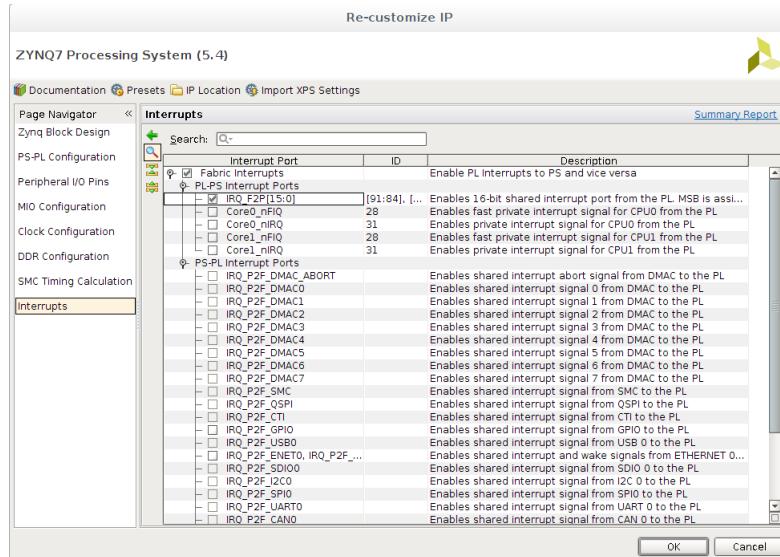
Figura 7.48: Configuración de señales de interrupción desde **PL** a **PS**.

Figura 7.49: Bloque Zynq7-PS luego de ser configurado.

7.5.2. Integración del pipeline de procesamiento de imágenes

En este apartado agregamos al diseño de bloques los **IP Cores** del **pipeline de procesamiento de imágenes**, analizado en [7.3.3](#).

En primer lugar conectamos todos los **IP Cores** a las señales de **clock** y **reset**.

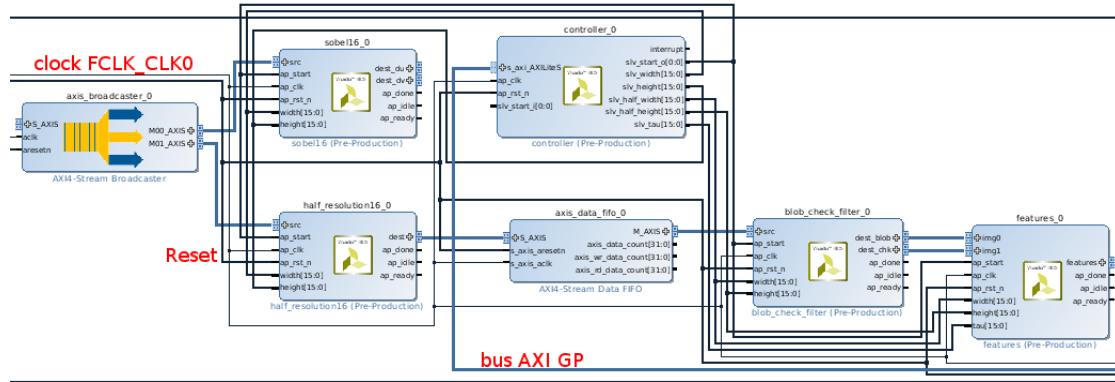
A continuación construimos el camino de datos, interconectando los **streams** de los **IP Cores**.

Como vimos en [7.4.2](#), fue necesario agregar una cola entre la salida de **Half Resolution16** y la entrada de **Blob + Checkerboard**. Vivado proporciona el **IP Core “AXI4-Stream Data FIFO”**¹¹ para tal fin, el mismo implementa una cola utilizando el estándar **AXI Stream** para su entrada y salida.

Configuramos la cola con un tamaño de 672 bytes, media linea de imagen del ancho máximo considerado en este trabajo.

Finalmente, agregamos al diseño de bloques el **IP Core Controller**, conectamos el mismo al bus **AXI** de propósito general y sus señales a cada uno de los **IP Cores** del **pipeline**.

¹¹Xilinx 2015b.

Figura 7.50: Vista de bloques del **pipeline** en Vivado IP Integrator.

7.5.3. Integración y configuración de controladores DMA

Utilizamos controladores DMA para generar el **stream** de entrada al **pipeline de procesamiento** desde una imagen en memoria y para guardar los datos de salida también en memoria.

Empleamos el IP Core “**AXI Direct Memory Access**”, incluido en Vivado. Dado que el mismo tiene un canal de lectura y/o uno de escritura, utilizamos tres instancias, **DMA0**, **DMA1** y **DMA2**.

- **DMA0**: genera un **stream** desde una imagen en RAM y guarda el **stream** de **features** resultante en RAM. El **stream** con los datos de la imagen tiene un ancho de 16 bits, pues empaqueta los píxeles de a dos.
- **DMA1** y **DMA2** guardan en RAM los **streams** resultantes de los filtros **Sobel** horizontal y vertical, respectivamente. Utilizan **streams** de 16 bits de ancho, pues las salidas del IP Core **Sobel16** son de este tamaño.

Conectamos los controladores **DMA** al bus **AXI GP** para que el procesador pueda manejar las transacciones **DMA**, es decir iniciarlas y consultar su estado. Además conectamos las salidas de interrupción de cada controlador al procesador, de esta forma el mismo puede ser notificado sobre cambios de estado en transferencias de datos sin utilizar **polling**.

También conectamos los controladores **DMA** al bus **AXI HP**. Esta conexión de alta velocidad es usada para la lectura y escritura de datos en la memoria RAM externa.

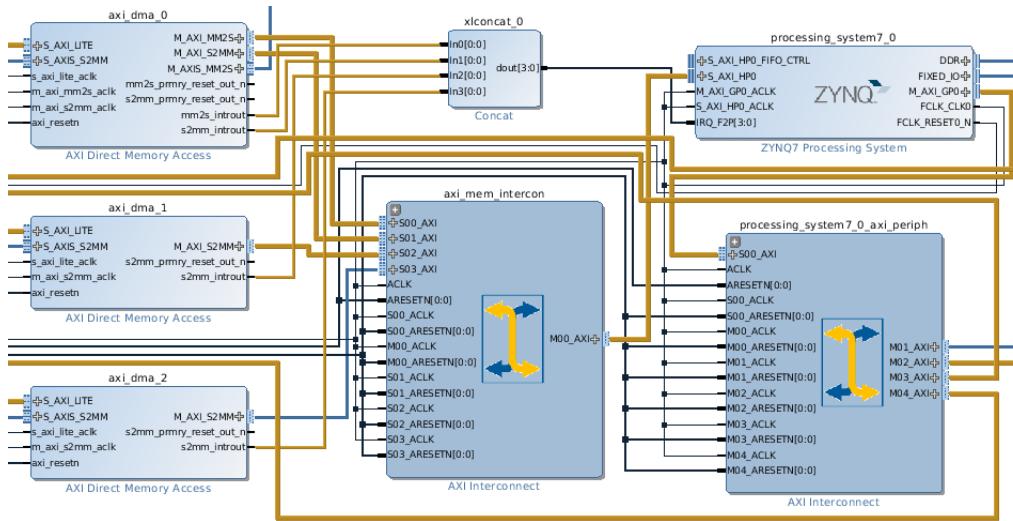


Figura 7.51: Conexiones de los controladores DMA a AXI GP, AXI HP e interrupciones.

Finalmente interconectamos los ***streams*** de los controladores DMA con los del ***pipeline*** de procesamiento de la siguiente forma:

- ***Stream*** de salida del controlador DMA0 a entrada al ***pipeline***.
- ***Stream*** de entrada del controlador DMA0 a salida del **IP Core Features**.
- ***Stream*** de entrada del controlador DMA1 a salida del filtro horizontal de **Sobel16**.
- ***Stream*** de entrada del controlador DMA2 a salida del filtro vertical de **Sobel16**.

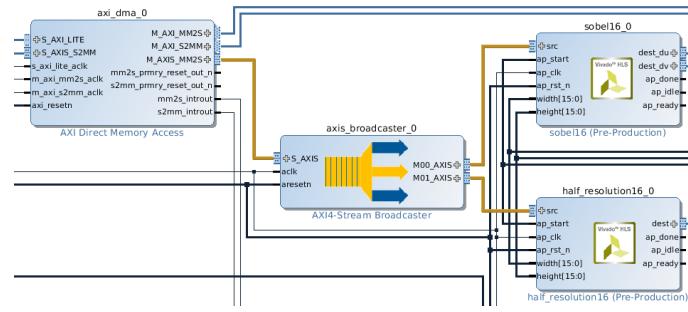
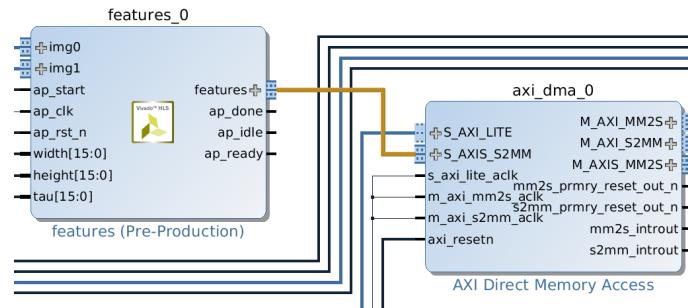
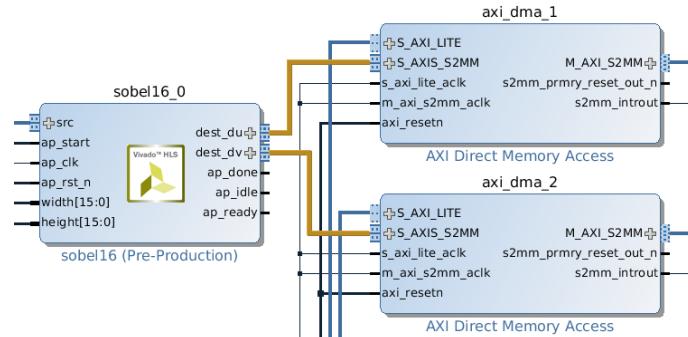
(a) Conexión DMA0 a entrada del **pipeline**.(b) Conexión DMA0 a salida del IP Core **Features**.(c) Conexiones DMA1 y DMA2 a salidas del IP Core **Sobel16**.

Figura 7.52: Conexiones entre controladores DMA y *streams* del **pipeline** de procesamiento.

7.5.4. Generación del archivo Bitstream

Desde Vivado IP Integrator, generamos el archivo **bitstream** de la plataforma construida. Al finalizar el proceso obtuvimos información sobre el consumo de recursos, tal como se muestra en la figura a continuación:

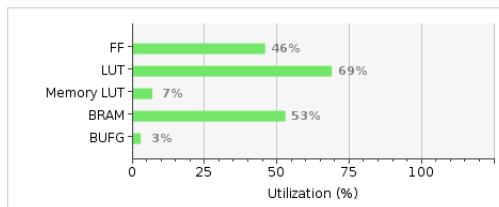


Figura 7.53: Reporte sobre consumo de recursos generado por **Vivado IP Integrator**.

Cabe destacar que si bien en teoría sería posible implementar dos ***pipelines*** de procesamiento de imágenes, los limitados recursos del **AP SoC** utilizado no lo permiten. Esto se debe a que el **IP Core Features** ocupa el 55 % de las **LUTs** del **AP SoC Zynq7010**, el más pequeño de la familia **Zynq**, utilizado en **Zybo**. Por lo tanto no es posible tener más de una instancia de ese **IP Core** en el sistema.

Finalmente exportamos la plataforma hardware creada para que pueda ser utilizada en las herramientas de desarrollo de software, como **Vivado SDK** y **PetaLinux SDK**.

7.5.5. Prueba de la plataforma

Utilizando **Vivado SDK** desarrollamos un programa **standalone**, es decir sin hacer uso de sistemas operativos, para probar el funcionamiento de la plataforma de hardware construida.

El programa configura los controladores **DMA** para que realicen las transferencias de datos desde y hacia RAM. Las direcciones de memoria para las transferencias son asignadas previo a la ejecución del programa utilizando **xmd**.

Luego el programa asigna al **IP Core Controller** el tamaño de la imagen, un valor de **tau** fijo y le indica que comience a procesar.

Al recibir la interrupción de fin de transferencia de los 3 canales DMA de lectura (filtros **Sobel** y **features**), el programa muestra por salida estándar datos de los **features** detectados.

Además, usando **xmd** es posible descargar a la computadora de escritorio los resultados.

Con el objetivo de medir el tiempo de procesamiento real, considerando costos de acceso a memoria, agregamos a la plataforma un cronómetro (**AXI Timer**¹²) que cuenta **ticks** del reloj de 100Mhz.

Antes de comenzar las transferencias **DMA**, el programa reinicia el cronómetro y al finalizar la última transferencia lo detiene.

Para una imagen de 1344x372 píxeles el procesamiento tomo **325901 ticks**. Es decir, una imagen fue procesada en 0,003 segundos, o lo que es lo mismo, el hardware construido es capaz de calcular los filtros **Sobel** y detectar **features** en **más de 300 imágenes por segundo**.

7.6. Modificaciones a la plataforma software

Esta sección presenta las modificaciones a la plataforma de software construida en 5.3 para dar soporte al nuevo hardware. Es decir, permitir a los procesos en espacio de usuario

¹²Xilinx 2015a.

utilizar el ***pipeline*** de procesamiento de imágenes, agregado a la plataforma hardware en la sección anterior, [7.5](#).

Dividimos a este trabajo en tres tareas: actualización de los archivos de descripción del hardware, implementar un módulo del kernel controlador del ***pipeline*** de procesamiento de imágenes y, finalmente, construcción y prueba de la plataforma.

Las siguientes secciones describen en detalle estas tres tareas.

7.6.1. Actualización de los archivos de descripción del hardware

Este primer paso consistió en sincronizar la plataforma de hardware utilizada en el proyecto *PetaLinux* con la plataforma exportada en [7.5.4](#).

Al igual que en [5.3](#), utilizamos la herramienta `petalinux-config`

```
~/system$ petalinux-config --get-hw-description=hardware/system/system.sdk
```

Al sincronizar la plataforma se generó el árbol de dispositivos (***device tree***). El mismo le permite al sistema operativo conocer la configuración de los dispositivos no ***plug & play***, como por ejemplo el **IP Core Controller** y los controladores DMA.

```
...
    axi_dma_2: axi-dma@40420000 {
        compatible = "xlnx,axi-dma";
        interrupt-parent = <&ps7_scugic_0>;
        interrupts = <0 32 4>;
        reg = <0x40420000 0x10000>;
        #dma-cells = <1>;
        #dma-channels = <1>;
        #dma-requests = <1>;
        dma-channel@40420030 {
            compatible = "xlnx,axi-dma-s2mm-channel";
            interrupts = <0 32 4>;
            xlnx,datawidth = <0x40>;
            xlnx,device-id = <0x1>;
            xlnx,include-dre = <0x0>;
        } ;
    } ;
    controller_0: controller@43c00000 {
        compatible = "miguel,controller-1.4";
        interrupt-parent = <&ps7_scugic_0>;
        interrupts = <0 4>;
        reg = <0x43c00000 0x10000>;
        miguel,s-axi-axilites-addr-width = <0x6>;
        miguel,s-axi-axilites-data-width = <0x20>;
    } ;
...
}
```

Figura 7.54: Extracto del archivo `subsystems/linux/configs/device-tree/pl.dtsi`. Contiene información sobre el ***device tree*** de los IP Cores instanciados en **PL**.

Finalmente, agregamos manualmente al ***device tree*** información acerca de los canales DMA utilizados para cada tarea. Esta información es usada en el módulo controlador del **IP Core Controller**:

```

...
    controller_0: controller@43c00000 {
        compatible = "miguel,controller-1.4";
        interrupt-parent = <&ps7_scugic_0>;
        interrupts = <0 4>;
        reg = <0x43c00000 0x10000>;
        miguel,s-axi-axilites-addr-width = <0x6>;
        miguel,s-axi-axilites-data-width = <0x20>;
        dma-image-in = <&axi_dma_0 0>;
        dma-du-out = <&axi_dma_1 0>;
        dma-dv-out = <&axi_dma_2 0>;
        dma-features-out = <&axi_dma_0 1>;
    } ;
...

```

Figura 7.55: Extracto del archivo `subsystems/linux/configs/device-tree/pl.dtsi` modificado con información sobre el uso de cada canal **DMA**.

7.6.2. Módulo del kernel controlador del pipeline de procesamiento de imágenes

Para proveer a los procesos que corren en espacio de usuario acceso a los controladores **DMA** y al **IP Core Controller** construimos un módulo del kernel.

A diferencia de los programas de usuario, los módulos del kernel de Linux son código ejecutable que se carga dinámicamente y se ejecuta en modo kernel, es decir que puede acceder sin restricciones a todo el hardware.

El módulo que diseñamos, `image_proc_controller`, provee dos servicios:

- Acceso a 16 **buffers** 1MB de memoria RAM **físicamente contigua**.
- Procesamiento de imágenes mediante el **pipeline**, utilizando los **buffers** anteriores como fuente de datos y destino de resultados.

Cada **buffer** contiguo tiene un archivo asociado, `/proc/image_proc_buffer_XX`, esto permite a los procesos en espacio de usuario referenciarlo y mapearlo a su espacio de memoria utilizando `mmap`. `mmap` es una llamada al sistema, compatible con *POSIX*, que mapea archivos o espacios de memoria física en el espacio memoria de un proceso.

El módulo también crea un archivo `/dev/image_proc_controller` al cual pueden acceder los procesos de usuario para realizar peticiones de procesamiento de imágenes. Para esto se utiliza *IOCTL* (input/output control), una llamada al sistema para operaciones que no pueden expresarse utilizando las llamadas estándar.

De esta forma, sin agregar una nueva llamada al sistema específica, pudimos implementar una interfaz entre los procesos en espacio de usuario y el módulo `image_proc_controller`. Esta interfaz permite a los primeros procesar imágenes utilizando el **pipeline** implementado en hardware en la sección [7.5](#).

Al invocar al módulo utilizando esta interfaz el proceso en espacio de usuario debe especificar:

- Ancho y alto de la imagen a procesar.
- Valor de **tau**.

- Índices de **buffers** contiguos para: fuente de imagen y destino de filtros *Sobel* y **features** detectados.

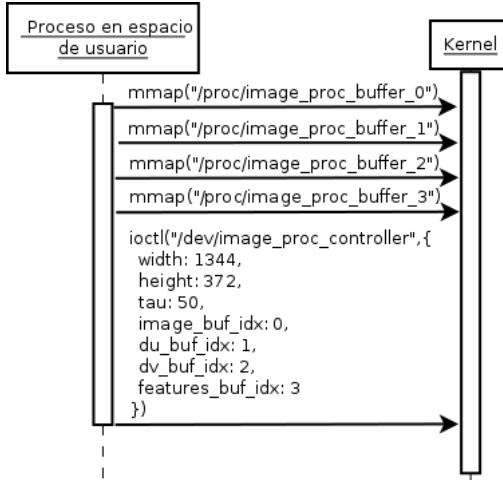


Figura 7.56: Secuencia de llamadas de proceso de usuario a kernel para procesar una imagen.

Luego de invocar al módulo el proceso de usuario, o el hilo particular que realizó la invocación si se trata de un proceso con múltiples hilos, queda bloqueado hasta que el procesamiento de la imagen en hardware finaliza.

7.6.3. Construcción y prueba

Luego de sincronizar la plataforma software con la plataforma hardware y crear el módulo del kernel para el control del **pipeline** de procesamiento, construimos y probamos la plataforma completa.

Desarrollamos una aplicación, `image_proc_controller_test`, que procesa una imagen siguiendo la secuencia de la figura 7.56.

A continuación construimos la plataforma software, incluyendo el módulo y la aplicación de prueba:

```
~/system$ petalinux-build
```

Posteriormente, modificamos el archivo `boot.sh`, creado en 5.3, para programar **PL** con el archivo **bitstream** generado en 7.5.4.

Finalmente, iniciamos la solución en *Zybo*, corriendo el archivo `boot.sh`, y comprobamos su correcto funcionamiento.

7.7. Aplicación viso_h

Esta sección describe las modificaciones realizadas a `viso_s` para obtener una nueva solución, `viso_h`, que hace uso del uso del **pipeline** de procesamiento de imágenes implementado en hardware para lograr un mejor rendimiento en el **AP SoC**.

En primer lugar reemplazamos el mecanismo de alocación de memoria para **buffers** utilizados en `viso_s` para imágenes y resultados de filtros *Sobel* por los **buffers** contiguos provistos por el módulo del kernel implementado en [7.6.2](#).

Para esto implementamos una biblioteca, `image_buffers` que proporciona dos funciones:

- `image_buffers_get`
- `image_buffers_release`

Al tener estas funciones una semántica similar a `malloc` y `free`, utilizadas en `viso_s`, pudimos de manera sencilla realizar las modificaciones al código necesarias para que haga uso de los **buffers** contiguos.

La segunda modificación consistió en aplicar cambios al cálculo de descriptores para que el mismo se lleve a cabo utilizando los filtros *Sobel* de la imagen sin escalar, puesto que el hardware implementado solo calcula los filtros para la imagen completa.

Para esto, antes de calcular el descriptor de un **feature**, se calcula su posición en la imagen original, multiplicando sus coordenadas por 2. Luego se calcula el descriptor del punto en las coordenadas obtenidas.^{[13](#)}

Finalmente modificamos el código de detección de **features**^{[14](#)}, reemplazando el uso del algoritmo **NMS** y los filtros *Sobel* implementados en software por una invocación del módulo del kernel `image_proc_controller`.

Agregamos el código necesario para compatibilizar la interfaz de salida utilizada por el **pipeline** de procesamiento para los **features** con la ya existente en `viso_s`. De esta forma acotamos el impacto de los cambios.

7.8. Pruebas de la solución optimizada en Zybo

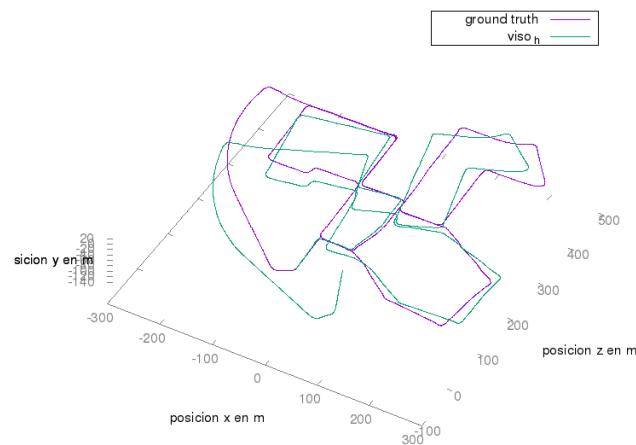
Al igual que en [5.4.3](#), utilizamos `run_test_case_remote` para probar `viso_h` en **Zybo**.

¹³Ver archivo `matcher.cpp`, función `computeDescriptors`

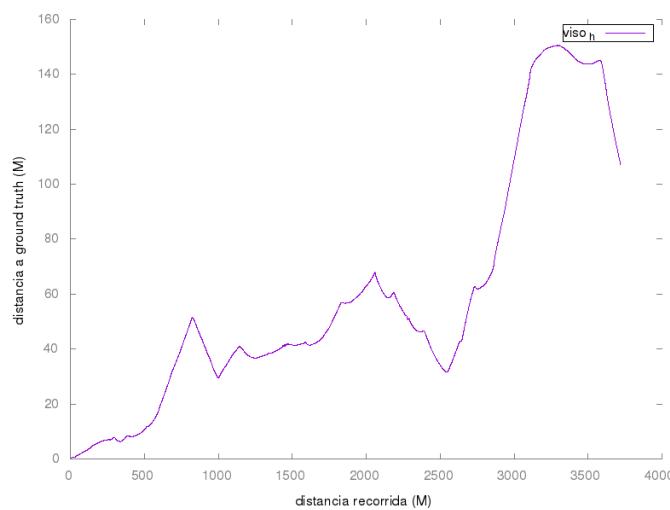
¹⁴Ver archivo `matcher.cpp`, función `computeFeatures`

```
{
  "name": "00",
  "sequence": "00",
  "framesN": 999999,
  "ground_truth": true,
  "runs": [
    {
      "cmd": "viso_h",
      "name": "viso_h",
      "nms_n": 5,
      "nms_tau": 60,
      "match_binsize": 20,
      "match_radius": 150,
      "ransac_iters": 100,
      "fps": 10
    }
  ]
}
```

- (a) Caso de test para la secuencia 00 procesada en ZYBO.



(b) Gráfica de trayectoria generada por viso_h



(c) Distancia entre la trayectoria estimada y la real

Figura 7.57: Caso de test de ejemplo ejecutable de forma remota para viso_h

Los resultados completos de las pruebas realizadas pueden consultarse en el apéndice `apendice-resultados-viso_h.tar.gz`.

7.9. Medición de rendimiento

En esta sección medimos el rendimiento de la solución construida y lo comparamos con la anterior. Al igual que en [5.5](#) tomamos los tiempos de procesamiento de la secuencia de imágenes “`minitest`” desde el disco RAM en *Zybo*.

De esta forma obtuvimos el siguiente resultado:

solución	configuración	<i>cuadros segundo</i>
viso	adhoc	2,55
viso_s	adhoc	4,23
viso_h	adhoc	7,42

Cuadro 7.7: Comparativa de rendimiento entre `viso`, `viso_s` y `viso_h` para la configuración **ad hoc**.

Como se ve, la solución `viso_h` procesa aproximadamente un 75 % más de cuadros por segundo que `viso_s` y un 191 % más de cuadros por segundo que `viso`.

Capítulo 8

Pruebas y Resultados

En este capítulo realizamos pruebas comparativas sobre las tres soluciones construidas, `viso`, `viso_s`y `viso_h`. Estas pruebas apuntan a estudiar de que forma las mejoras en rendimiento, obtenidas a través de optimizaciones software y hardware, impactan en la calidad de los resultados.

A partir de las mediciones de rendimiento obtenidas en los capítulos 5, 6 y 7 llevamos a cabo cuatro experimentos.

El primero consistió en procesar las secuencias de imágenes del dataset, simulando la pérdida de cuadros, según la tabla 8.1 que representa el rendimiento de cada solución para la configuración **adhoc**. En el segundo y el tercero modificamos las configuraciones de `viso_s` y `viso_h` para procesar las secuencias de imágenes a 5 y 10 cuadros por segundo respectivamente.

Para los experimentos 2 y 3 no consideramos la solución `viso`, pues su rendimiento no nos permitió obtener tasas de procesamiento de 5 y 10 cuadros por segundo.

Finalmente, en el experimento 4 comparamos para `viso_h` los resultados obtenidos al procesar las secuencias de imágenes con la configuración **adhoc**, simulando la pérdida de cuadros acorde al rendimiento, y con la configuración hallada en el experimento 3 sin pérdida de cuadros.

solución	configuración	<i>cuadros segundo</i>
viso	adhoc	2,55
viso_s	adhoc	4,23
viso_h	adhoc	7,42

Cuadro 8.1: Comparativa de rendimiento entre `viso`, `viso_s` y `viso_h` para la configuración **ad hoc**.

Los resultados completos y las definiciones de los casos de prueba pueden encontrarse en el `comparativa.tar.gz`.

8.1. Experimento 1: procesamiento en tiempo real

Este experimento consistió en procesar las secuencias de imágenes del dataset **KITTI** para odometría visual con las tres soluciones construidas, simulando en cada una la pérdida de cuadros acorde a su rendimiento.

Al igual que en 5.4.3, utilizamos `run_test_case_remote` para realizar este experimento sobre **Zybo**.

```
{
  "name": "adhoc_realtime_sequence",
  "sequence": "X",
  "framesN": 999999,
  "ground_truth": true,
  "runs": [
    {
      "name": "viso",
      "cmd": "viso",
      "nms_n": 5,
      "nms_tau": 60,
      "match_binsize": 20,
      "match_radius": 150,
      "ransac_iters": 100,
      "fps": 2.55
    },
    {
      "name": "viso_s",
      "cmd": "viso_s",
      "nms_n": 5,
      "nms_tau": 60,
      "match_binsize": 20,
      "match_radius": 150,
      "ransac_iters": 100,
      "fps": 4.23
    },
    {
      "name": "viso_h",
      "cmd": "viso_h",
      "nms_n": 5,
      "nms_tau": 60,
      "match_binsize": 20,
      "match_radius": 150,
      "ransac_iters": 100,
      "fps": 7.42
    }
  ]
}
```

Figura 8.1: Caso de test utilizado en el experimento.

A continuación medimos las distancias entre las trayectorias estimadas y las reales.

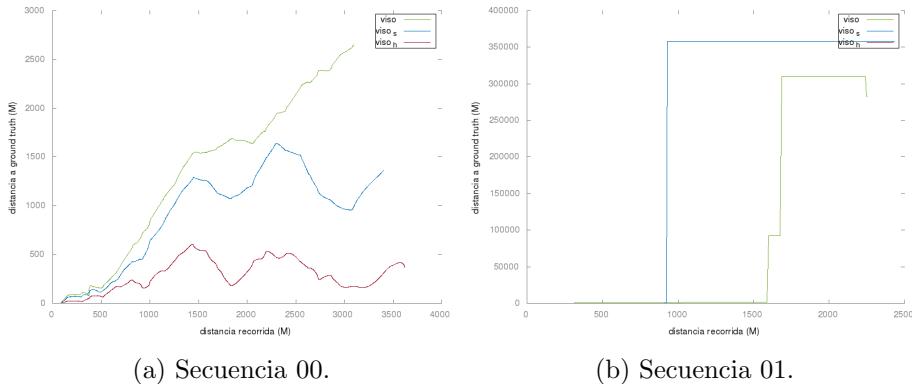
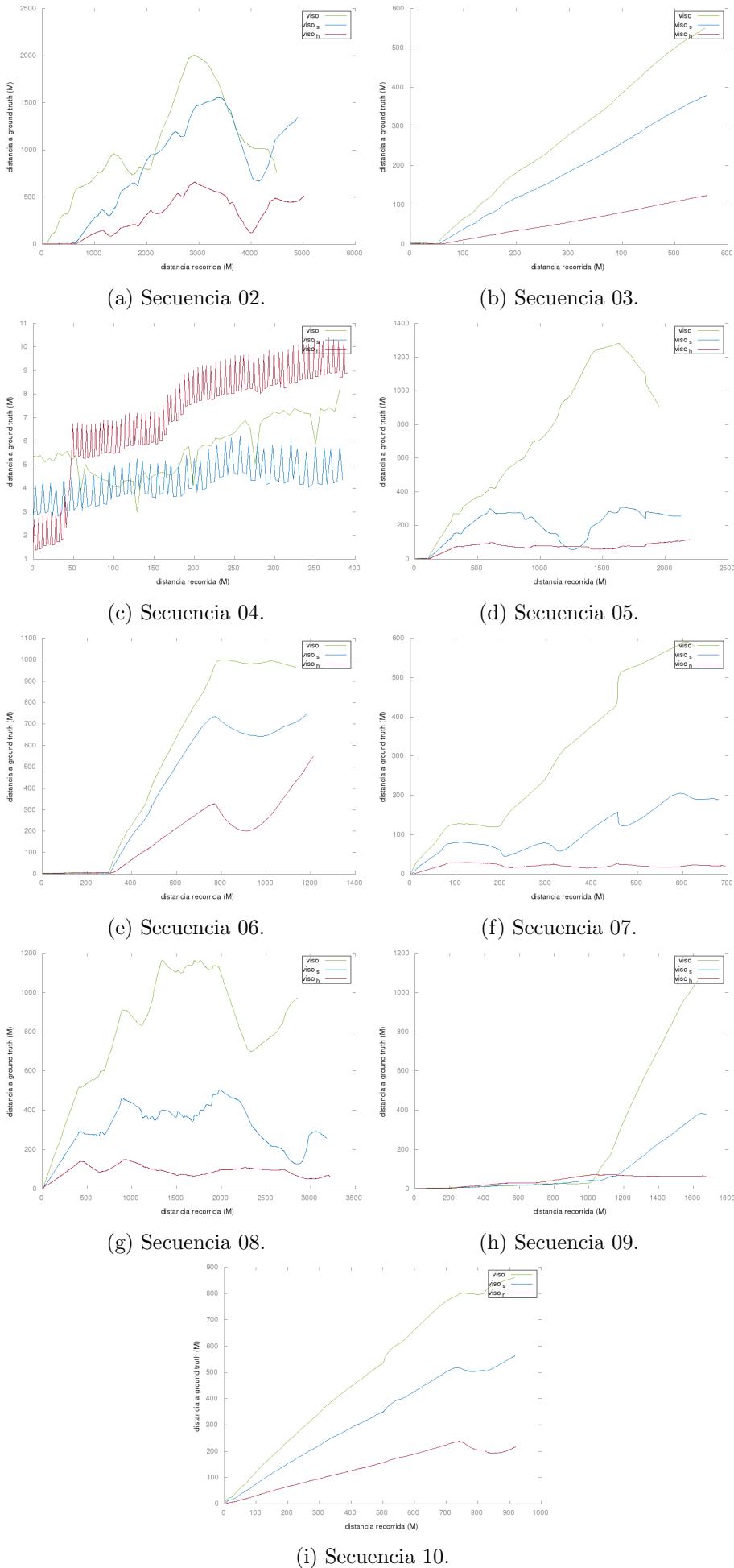


Figura 8.2: Resultados del experimento.



Los extraños gráficos de las secuencias 01 y 04 pueden explicarse revisando [3.2.3](#), allí vimos que en particular para estas secuencias la pérdida de cuadros altera drásticamente al resultado.

A partir de la información obtenida construimos las siguientes tablas, las mismas sintetizan y comparan las distancias máximas y promedio de las distintas soluciones a la trayectoria real de cada secuencia de imágenes.

secuencia	viso (m)	viso_s (% de viso)	viso_h (% de viso)
00	2649,63	61	23
01	310096,47	115	0,078
02	2004,72	77	32
03	549,05	69	22
04	8,22	75	126
05	1281,4	24	9
06	1000,01	74	54
07	590,36	34	5
08	1165,46	43	13
09	1060,84	36	7
10	860,97	65	27

Cuadro 8.2: Comparativa: Distancias máximas de **viso** a **ground truth** en m. y distancias máximas de **viso_s** y **viso_h** a **ground truth** como proporción de la distancia máxima de **viso**.

secuencia	viso (m)	viso_s (% de viso)	viso_h (% de viso)
00	1255,55	69	22
01	92595,34	241	0,1
02	1042,16	76	28
03	255,46	68	22
04	5,58	77	124
05	680,72	28	10
06	491,14	76	35
07	293,23	36	7
08	830,36	38	10
09	178,56	44	23
10	488,3	65	28

Cuadro 8.3: Comparativa: Distancias promedio de **viso** a **ground truth** en m. y distancias promedio de **viso_s** y **viso_h** a **ground truth** como proporción de la distancia promedio de **viso**.

Como se puede observar, la mejora en el rendimiento, es decir el procesamiento de más cuadros por segundo, permite una mejora en la calidad del resultado.

Tomando la media truncada, descartando el mayor y menor valor, observamos que la media de las distancias máximas de **viso_s** a **ground truth** como porcentajes de las distancias máximas de la solución inicial, **viso**, es del 59,33 %.

De la misma manera, la media de la distancias promedio de `viso_s` a ***ground truth*** como porcentajes de las distancias máximas de la solución inicial, `viso`, es del 61 %.

Análogamente para `viso_s` con respecto a `viso` estos valores son 21,33 % y 20,55 %, respectivamente.

Es decir que las optimizaciones aplicadas se tradujeron, efectivamente, en mejoras sustantivas en la calidad de los resultados.

8.2. Experimento 2: procesamiento a 5 cuadros por segundo

Para este experimento modificamos los parámetros de configuración de `viso_s` y `viso_h` de forma tal que ambas soluciones tuvieran tasas de procesamiento de 5 cuadros por segundo.

Dado que el rendimiento de `viso_s` para la configuración **adhoc** es de 4,23 cuadros por segundo, fue necesario modificar su configuración para trabajar con una menor cantidad de **features**, restringir los rangos de búsqueda de coincidencias y realizar una menor cantidad de iteraciones **RANSAC**.

En cambio, para `viso_h` el rendimiento para la configuración **adhoc** es de 7,42 cuadros por segundo, por lo tanto, modificamos la configuración para trabajar con más **features**, agrandar los rangos de búsqueda de coincidencias y la cantidad de iteraciones **RANSAC**.

La siguiente tabla muestra las configuraciones utilizadas:

solución	nms_n	nms_tau	match_binsize	match_radius	ransac_iters
<code>viso_s</code>	5	200	50	150	120
<code>viso_h</code>	5	40	20	200	224

Cuadro 8.4: Juegos de parámetros utilizados en este experimento.

A continuación procesamos las secuencias de imágenes del dataset **KITTI** para odometría visual a 5 cuadros por segundo, es decir, perdiendo la mitad de los cuadros.

Al igual que en el experimento anterior, utilizamos `run_test_case_remote` para realizar este sobre **Zybo**.

```
{
  "name": "comparativa_5fps_sequence",
  "sequence": "X",
  "framesN": 999999,
  "ground_truth": true,
  "runs": [
    {
      "name": "viso_s",
      "cmd": "viso_s",
      "nms_n": 5,
      "nms_tau": 200,
      "match_binsize": 50,
      "match_radius": 150,
      "ransac_iters": 120,
      "fps": 5
    },
    {
      "name": "viso_h",
      "cmd": "viso_h",
      "nms_n": 5,
      "nms_tau": 40,
      "match_binsize": 20,
      "match_radius": 200,
      "ransac_iters": 224,
      "fps": 5
    }
  ]
}
```

Figura 8.4: Caso de test utilizado en el experimento.

A continuación medimos las distancias entre las trayectorias estimadas y las reales.

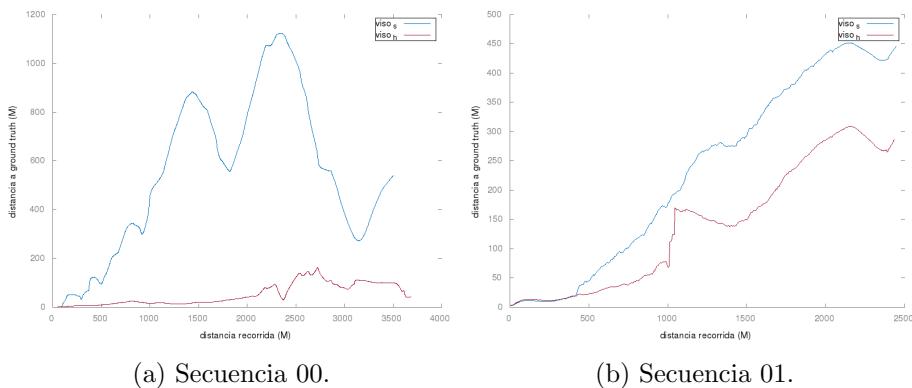
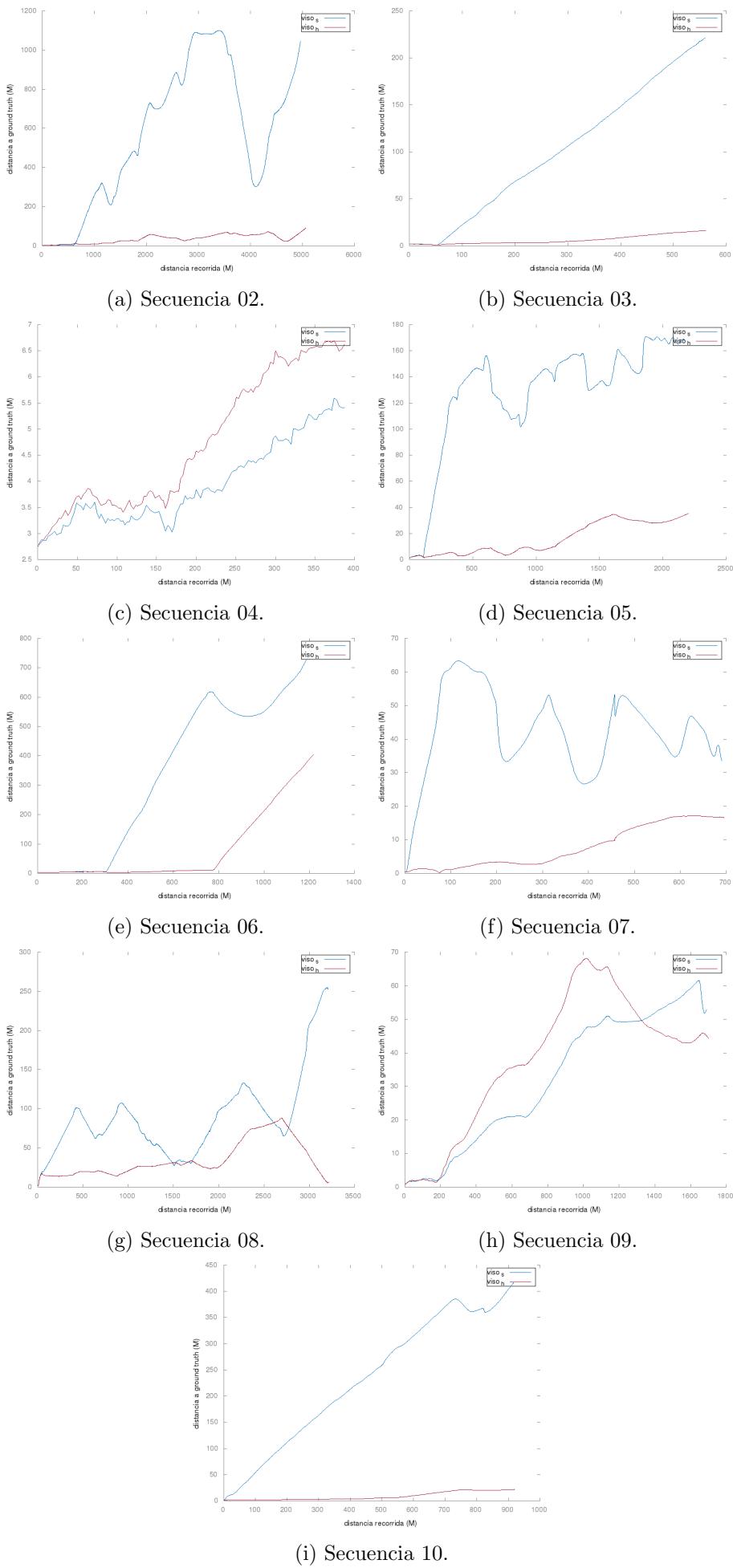


Figura 8.5: Resultados del experimento.



Al igual que en el experimento anterior, a partir de la información obtenida construimos las siguientes tablas, las mismas sintetizan y comparan las distancias máximas y promedio de las distintas soluciones a la trayectoria real de cada secuencia de imágenes.

secuencia	viso_s (m)	viso_h (% de viso_s)
00	1123,02	14,5
01	451,52	68
02	1099,06	8
03	221,28	7
04	5,59	120
05	170,93	20
06	727,69	55
07	63,30	27
08	254,92	34,5
09	61,55	110,6
10	416,67	5

Cuadro 8.5: Comparativa: Distancias máximas de **viso_s** a **ground truth** en m. y distancias máximas de **viso_h** a **ground truth** como proporción de la distancia máxima de **viso_s**.

secuencia	viso_s (m)	viso_h (% de viso_s)
00	528,95	9
01	241,89	61
02	558,81	6
03	102,58	6,5
04	3,87	119
05	125,77	12
06	322,47	24
07	42,29	18
08	86,69	39
09	31,32	121
10	234,79	3,9

Cuadro 8.6: Comparativa: Distancias promedio de **viso_s** a **ground truth** en m. y distancias promedio de **viso_h** a **ground truth** como proporción de la distancia promedio de **viso_s**.

Tomando la media truncada, descartando el mayor y menor valor, observamos que la media de las distancias máximas de **viso_h** a **ground truth** como porcentajes de las distancias máximas de **viso_s**, es del 34,45 %.

De la misma manera, la media de las distancias promedio de **viso_h** a **ground truth** como porcentajes de las distancias máximas de **viso_s**, es del 32,72 %.

Es decir que, la mejora en el rendimiento permite, a iguales tasas de procesamiento, utilizar parámetros de configuración que proporcionan en la mayoría de los casos mejores resultados.

8.3. Experimento 3: procesamiento a 10 cuadros por segundo

Para este experimento modificamos los parámetros de configuración de `viso_s` y `viso_h` de forma tal que ambas soluciones tuvieran tasas de procesamiento de 10 cuadros por segundo.

La siguiente tabla muestra las configuraciones utilizadas:

solución	nms_n	nms_tau	match_binsize	match_radius	ransac_iters
viso_s	7	650	100	55	4
viso_h	5	65	20	95	45

Cuadro 8.7: Juegos de parámetros utilizados en este experimento.

A continuación procesamos las secuencias de imágenes del dataset **KITTI** para odometría visual a 10 cuadros por segundo, es decir, sin pérdida de cuadros.

Al igual que en el experimento anterior, utilizamos `run_test_case_remote` para realizar este sobre **Zybo**.

```
{
  "name": "relajado_10fps_sequence",
  "sequence": "X",
  "framesN": 999999,
  "ground_truth": true,
  "runs": [
    {
      "name": "viso_s",
      "cmd": "viso_s",
      "nms_n": 7,
      "nms_tau": 650,
      "match_binsize": 100,
      "match_radius": 55,
      "ransac_iters": 4,
      "fps": 10
    },
    {
      "name": "viso_h",
      "cmd": "viso_h",
      "nms_n": 5,
      "nms_tau": 65,
      "match_binsize": 20,
      "match_radius": 95,
      "ransac_iters": 45,
      "fps": 10
    }
  ]
}
```

Figura 8.7: Caso de test utilizado en el experimento.

A continuación medimos las distancias entre las trayectorias estimadas y las reales.

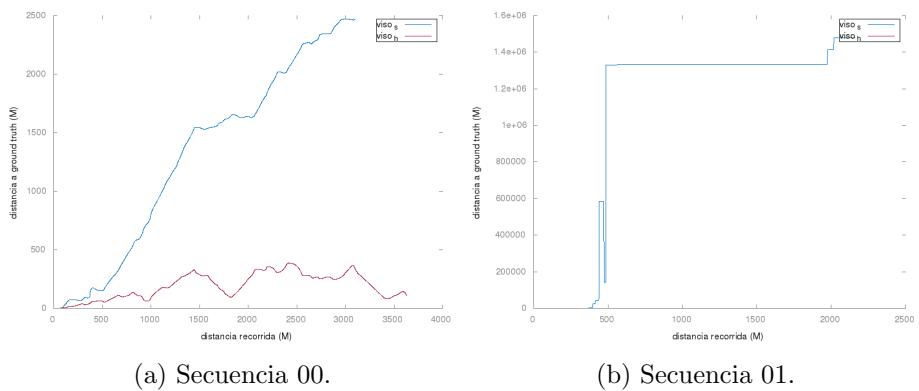
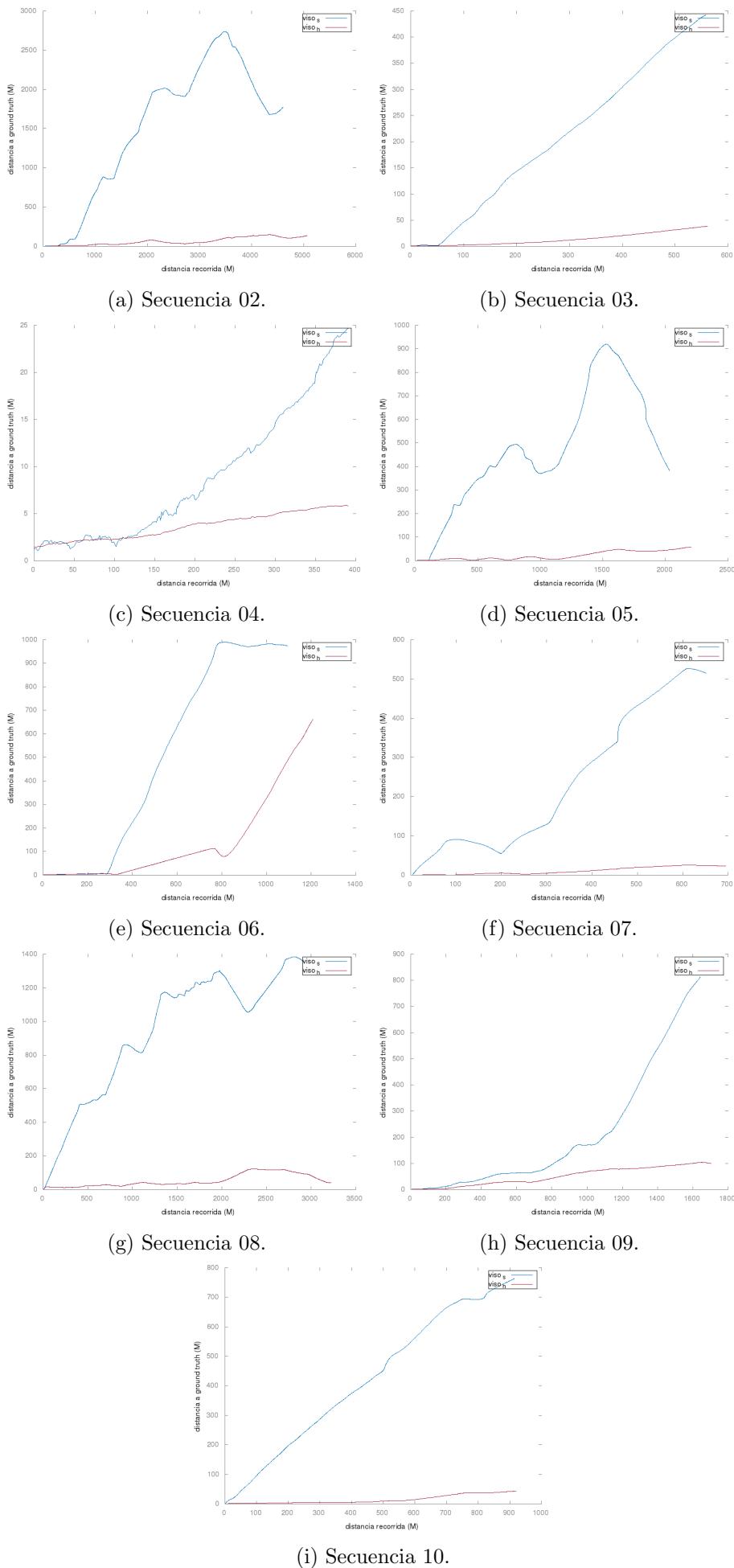


Figura 8.8: Resultados del experimento.



Al igual que en el experimento anterior, a partir de la información obtenida construimos las siguientes tablas, las mismas sintetizan y comparan las distancias máximas y promedio de las distintas soluciones a la trayectoria real de cada secuencia de imágenes.

secuencia	viso_s (m)	viso_h (% de viso_s)
00	2471,68	15,6
01	1478947,93	0,016
02	2739,58	5
03	442,37	8
04	24,66	23
05	920,11	6,4
06	990,33	66
07	526,83	4,8
08	1384,56	8,9
09	812,41	12
10	764,33	5,7

Cuadro 8.8: Comparativa: Distancias máximas de `viso_s` a **ground truth** en m. y distancias máximas de `viso_h` a **ground truth** como proporción de la distancia máxima de `viso_s`.

secuencia	viso_s (m)	viso_h (% de viso_s)
00	1247,62	15
01	241,89	61
02	1003652,09	0,012
03	204,52	7
04	8,35	42
05	457,22	4,5
06	474,06	29
07	229,94	5
08	949,15	5,4
09	186,96	26,5
10	417,64	3,9

Cuadro 8.9: Comparativa: Distancias promedio de `viso_s` a **ground truth** en m. y distancias promedio de `viso_h` a **ground truth** como proporción de la distancia promedio de `viso_s`.

En este experimento observamos que `viso_s` utilizando los parámetros hallados para alcanzar tasas de procesamiento de 10 cuadros arroja peores resultados que al utilizar la configuración del experimento anterior y perder la mitad de los cuadros.

Por lo tanto, decidimos realizar el experimento 4, para evaluar si en el caso de `viso_h` provee mejores resultados no perder cuadros o utilizar la configuración **adhoc** a una tasa de procesamiento acorde a su rendimiento.

8.4. EXPERIMENTO 4: VISO_H CONFIGURACIÓN ADHOC VS PROCESAMIENTO A 10 CUADROS POR SEGUNDO

8.4. Experimento 4: viso_h configuración adhoc vs procesamiento a 10 cuadros por segundo

En este experimento evaluamos que alternativa provee mejores resultados: no perder cuadros o utilizar la configuración **adhoc** a una tasa de procesamiento acorde a su rendimiento. Para esto utilizamos la configuración **adhoc** y la hallada en el experimento 3.

La siguiente tabla muestra las configuraciones utilizadas:

configuración	nms_n	nms_tau	match_binsize	match_radius	ransac_iters
adhoc	5	60	20	150	100
10fps	5	65	20	95	45

Cuadro 8.10: Juegos de parámetros utilizados en este experimento.

Al igual que en el experimento anterior, utilizamos `run_test_case_remote` para realizar este sobre **Zybo**.

```
{
  "name": "adhoc_vs_10fps_sequence",
  "sequence": "X",
  "framesN": 999999,
  "ground_truth": true,
  "runs": [
    {
      "name": "adoc",
      "cmd": "viso_h",
      "nms_n": 5,
      "nms_tau": 60,
      "match_binsize": 20,
      "match_radius": 150,
      "ransac_iters": 100,
      "fps": 7.42
    },
    {
      "name": "10 fps",
      "cmd": "viso_h",
      "nms_n": 5,
      "nms_tau": 65,
      "match_binsize": 20,
      "match_radius": 95,
      "ransac_iters": 45,
      "fps": 10
    }
  ]
}
```

Figura 8.10: Caso de test utilizado en el experimento.

A continuación medimos las distancias entre las trayectorias estimadas y las reales.

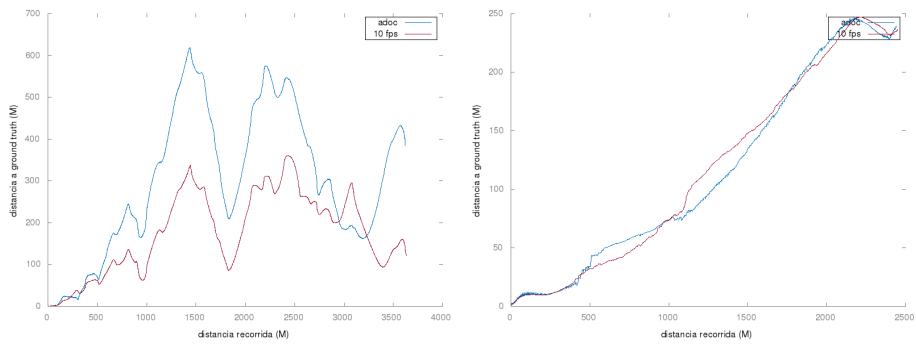
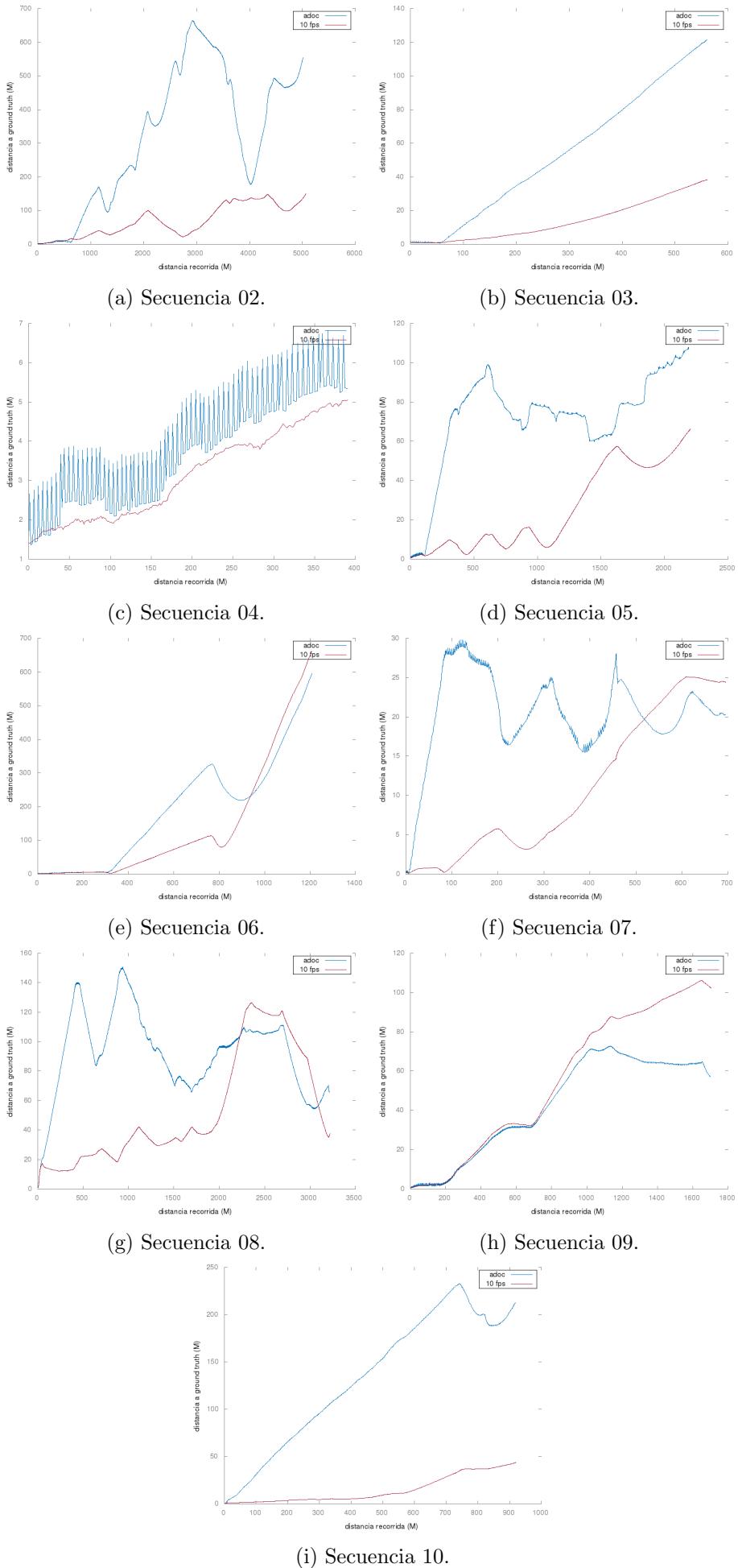


Figura 8.11: Resultados del experimento.

8.4. EXPERIMENTO 4: VISO_H CONFIGURACIÓN ADHOC VS PROCESAMIENTO A 10 CUADROS P



Al igual que en el experimento anterior, a partir de la información obtenida construimos las siguientes tablas, las mismas sintetizan las distancias máximas y promedio a la trayectoria real de cada secuencia de imágenes utilizando las configuraciones planteadas.

secuencia	adhoc (m)	10fps (m)
00	618,63	360,11
01	246,10	247,22
02	663,98	149,7
03	121,61	38,38
04	6,82	5,05
05	107,82	66,28
06	594,78	661,51
07	29,80	25,1
08	150,55	126,32
09	72,81	106,02
10	232,88	43,74

Cuadro 8.11: Comparativa: Distancias máximas a **ground truth** en m. de `viso_h` utilizando las configuraciones **adhoc** y **10fps**.

secuencia	adhoc (m)	10fps (m)
00	295,34	173,67
01	121,14	123,20
02	315,13	68,85
03	55,54	14,85
04	3,88	3,05
05	71,09	24,02
06	184,43	138,86
07	20,76	11,43
08	91,26	51,24
09	42,26	53,89
10	134,23	16,49

Cuadro 8.12: Comparativa: Distancias promedio a **ground truth** en m. de `viso_h` utilizando las configuraciones **adhoc** y **10fps**.

En este experimento observamos que, utilizando los parámetros hallados para alcanzar tasas de procesamiento de 10 cuadros por segundo `viso_h` proporciona en la mayoría de los casos mejores resultados que al utilizar la configuración **adhoc** con una tasa de procesamiento acorde a su rendimiento (**7.42 fps**).

Es decir que, en el caso de `viso_h` el rendimiento de la solución hace que no sea necesario utilizar parámetros muy poco exigentes para alcanzar tasas de procesamiento de 10 cuadros por segundo.

Capítulo 9

Conclusiones

Este capítulo presenta nuestras conclusiones sobre los distintos aspectos del desarrollo de soluciones que implementan visión artificial sobre **AP SoCs** y nuestros aportes a los mismos.

En primer lugar, nos parece importante resaltar que la **metodología** elegida permitió llevar a cabo este trabajo con éxito. Considerando que durante el desarrollo de la solución construimos todas las capas que la conforman, desde el diseño de alto nivel hasta la implementación del hardware que le da soporte, contar con una metodología que integra todas las etapas del desarrollo resultó clave para ordenar y guiar el trabajo.

La propuesta de la metodología de ir avanzando en pequeños pasos, partiendo de una implementación en una computadora de propósito general hasta llegar a la solución optimizada en el AP SoC, resultó útil pues permitió contar desde una etapa temprana con una solución funcionando que sirvió como referencia en las etapas posteriores.

El estudio temprano de los parámetros de configuración de **LIBVIS02** y el efecto de la pérdida de cuadros resultó útil para determinar la configuración a usar como punto de partida y para guiar la etapa de experimentación.

El enfoque utilizado en la etapa de diseño, donde dividimos la solución en **grupos de componentes** con funcionalidades bien definidas, fue de gran ayuda no solo para el desarrollo, sino también para el estudio del rendimiento y la aplicación de optimizaciones.

En la etapa de optimización por software, el uso de diagramas de actividad facilitó la detección de tareas paralelizables y tareas divisibles en subtareas paralelizables. Sin embargo, no fue trivial hallar el nivel de detalle necesario para que los diagramas de actividad proveyeran información útil.

Haber caracterizado el funcionamiento de los **IP Cores** para procesamiento de imágenes utilizando un patrón de diseño nos permitió explorar optimizaciones sobre el mismo de forma genérica.

Las optimizaciones que propusimos en el capítulo 7, “**K píxeles por ciclo**” y “**fusión de IP Cores**” resultaron determinantes en la construcción de **IP Cores** para satisfacer los niveles de velocidad requeridos por el **pipeline** de procesamiento de imágenes.

El capítulo 8 demuestra que la mejora en rendimiento fruto de delegar tareas al hardware tiene un impacto muy positivo en la calidad del resultado. Además, el mayor rendimiento hace posible explorar más configuraciones según el ambiente de uso particular.

En resumen, al utilizar la metodología elegida, técnicas de optimización de software y síntesis de alto nivel, aplicando las optimizaciones estudiadas y propuestas por nosotros, para la construcción de IP Cores, fue posible el desarrollo de una solución de odometría

visual estéreo con buenos rendimientos.

9.1. Trabajos futuros

Creemos que hay varios elementos de este trabajo que pueden ser utilizados en desarrollos futuros.

La plataforma de hardware construida puede ser utilizada de forma directa en otras aplicaciones que saquen provecho de la detección de **features** por hardware. Además, la misma y el conocimiento adquirido durante su desarrollo proveen un buen punto de partida para soluciones sobre **AP SoCs** que realicen procesamiento de señales en hardware.

La solución construida en el caso de estudio puede servir como base para aplicaciones en robótica, donde las restricciones de tamaño y consumo energético hagan prohibitivo el uso de otras plataformas de cómputo.

Este trabajo presenta varios aportes que consideramos valiosos para el desarrollo de futuros trabajos utilizando **AP SoCs**:

- El diseño inicial de la solución centrado en componentes y grupos de componentes.
- Estudio del pipeline de procesamiento de señales (en nuestro caso imágenes).
- Estudio del patrón de diseño para el desarrollo de **IP Cores** para procesamiento de imágenes utilizando **HLS**.
- Optimizaciones aplicables a **IP Cores** para procesamiento de imágenes.

Índice de figuras

2.1.	Arquitectura Zynq-7000	8
3.1.	Configuración del vehículo para la adquisición de datos.	12
3.2.	Pasos del procesamiento de un cuadro estéreo.	13
3.3.	Archivo de calibración de ejemplo.	13
3.4.	Archivo de configuración con los valores por defecto.	14
3.5.	Gráficos de distancias entre trayectorias reales y estimadas a 10, 5 y 3 <i>fps</i> para las secuencias del dataset KITTI para odometría visual	14
3.6.	Gráficos de distancias entre trayectorias reales y estimadas a 10, 5 y 3 <i>fps</i> para las secuencias del dataset KITTI para odometría visual (cont.)	15
4.1.	Componentes agrupados por tipo de tarea. Las flechas naranjas representan flujo de datos.	18
4.2.	Entradas y salidas del “lector de imágenes”.	18
4.3.	Entradas y salidas del grupo de componentes de detección de features	19
4.4.	Entradas y salidas del componentes de extracción de features	19
4.5.	Entradas y salidas del componente.	20
4.6.	Ejemplo de Quad-matching para un feature . Imagen superior: cuadro estéreo anterior. Imagen inferior: cuadro estéreo actual.	20
4.7.	Entradas y salidas del componente.	22
4.8.	Caso de prueba para la secuencia 00 del dataset	23
5.1.	Comunicación PS/PL en Zynq 7010.	26
5.2.	Vista del diseño de bloques en Vivado.	27
5.3.	Pantalla de configuración de PetaLinux.	28
5.4.	Interconexiones ethernet y UART entre PC de desarrollo y placa ZYBO.	31
5.5.	Configuración de la red en ZYBO: <code>/etc/network/interfaces</code>	32
5.6.	Configuración del punto de montaje en ZYBO: <code>/etc/fstab</code>	32
5.7.	Pantalla de configuración de PetaLinux para habilitar dropbear.	33
5.8.	Caso de prueba de ejemplo ejecutable de forma remota	34
6.7.	Gráficos de distancias entre trayectorias reales y estimadas a 5 <i>fps</i> para las secuencias del dataset KITTI para odometría visual	45
6.8.	Gráficos de distancias entre trayectorias reales y estimadas a 5 <i>fps</i> para las secuencias del dataset KITTI para odometría visual (cont.)	46
6.9.	Gráficos de distancias entre trayectorias reales y estimadas a 3 <i>fps</i> para las secuencias del dataset KITTI para odometría visual	47

6.10. Gráficos de distancias entre trayectorias reales y estimadas a 3 <i>fps</i> para las secuencias del dataset KITTI para odometría visual (cont.)	48
6.11. Extracto del gráfico generado por <code>process_gmon.sh</code>	50
6.12. Porcentaje tiempos de procesamiento por grupo de componentes.	50
6.13. Comando para crear la nueva aplicación <code>viso_s</code>	50
6.14. Diagrama de actividad del proceso de detección y extracción de features en <code>viso</code>	51
6.15. Diagrama de actividad del proceso de detección y extracción de features en <code>viso_s</code>	52
6.16. Diagrama de actividad del proceso de búsqueda de coincidencias.	52
6.17. Detalle de los procesos de búsqueda de coincidencias de features “dispersos” y “densos” en <code>viso</code>	53
6.18. Detalle de los procesos de búsqueda de coincidencias de features “dispersos” y “densos” en <code>viso_s</code>	54
6.19. Diagrama de actividad del proceso de estimación de pose en <code>viso</code>	55
6.20. Diagrama de actividad del proceso de estimación de pose en <code>viso_s</code>	56
6.21. Caso de test de ejemplo ejecutable de forma remota para <code>viso_s</code>	57
 7.1. Porcentaje tiempos de procesamiento real por grupo de componentes en <code>viso_s</code>	60
7.2. IP Core con streams como entrada y salida.	61
7.3. Implementación del algoritmo 2 en Vivado HLS	63
7.4. IP Core <code>procesarImagen</code> generado con Vivado HLS	64
7.5. Aplicación de la directiva LOOP_TRIPCOUNT	64
7.6. Estimación de velocidad y consumo de recursos generada por Vivado HLS	65
7.7. Comparativa de comportamiento del código sin y con utilizar la directiva PIPELINE extraída del manual de Vivado HLS.	65
7.8. Aplicación de la directiva PIPELINE	66
7.9. Estimación de velocidad y consumo de recursos generada por Vivado HLS para la implementación que utiliza la directiva PIPELINE	66
7.10. Aplicación de la optimización para procesar 4 píxeles por iteración.	68
7.11. Estimación de velocidad y consumo de recursos generada por Vivado HLS para la implementación que procesa 4 píxeles por iteración.	69
7.12. Aplicación de la optimización para combinar dos IP Cores.	70
7.13. Estimación de velocidad y consumo de recursos generada por Vivado HLS para la implementación que combina dos IP Cores.	71
7.14. Mapeo de funcionalidad a PS y PL.	72
7.15. Diagrama de buses.	73
7.16. Pipeline de procesamiento de imágenes para detección de features	74
7.17. Análisis de requerimientos de productividad de los IP Cores del pipeline de procesamiento de imágenes.	74
7.19. Lógica para alinear imágenes de salida con la imagen de entrada	76
7.20. Imágenes de entrada y salidas del código a sintetizar	77
7.21. IP Core Sobel16	77
7.22. Consumo de recursos del IP Core Sobel16	78
7.23. Latencia e intervalo de iniciación del IP Core Sobel16	78
7.24. Extracto del código a sintetizar para el IP Core HalfResolution16	80

7.25. Imágenes de entrada y salida del código a sintetizar	80
7.26. IP Core HalfResolution16	81
7.27. Consumo de recursos del IP Core HalfResolution16	81
7.28. Latencia e intervalo de iniciación del IP Core HalfResolution16	81
7.30. Lógica para alinear imágenes de salida con la imagen de entrada	82
7.31. Imágenes de entrada y salidas del código a sintetizar	83
7.32. IP Core Blob+Checkerboard	83
7.33. Consumo de recursos del IP Core Blob+Checkerboard	84
7.34. Latencia e intervalo de iniciación del IP Core Blob+Checkerboard	84
7.35. Imágenes de entrada y salida del código a sintetizar	87
7.36. IP Core Features	88
7.37. Consumo de recursos del IP Core Features	88
7.38. Latencia e intervalo de iniciación del IP Core Features	88
7.39. Código fuente del IP Core Controller	89
7.40. IP Core Controller	90
7.41. Consumo de recursos del IP Core Controller	90
7.42. Consumo de recursos de la interfaz AXI Slave del IP Core Controller	90
7.43. Latencia e intervalo de iniciación del IP Core Controller	90
7.44. Vista del diseño de bloques de la plataforma de hardware inicial	91
7.45. Configuración de señal de clock para PL	91
7.46. Configuración de señal de reset para PL	92
7.47. Configuración de buses AXI entre PS y PL	92
7.48. Configuración de señales de interrupción desde PL a PS	93
7.49. Bloque Zynq7-PS luego de ser configurado	93
7.50. Vista de bloques del pipeline en Vivado IP Integrator	94
7.51. Conexiones de los controladores DMA a AXI GP , AXI HP e interrupciones	95
7.52. Conexiones entre controladores DMA y streams del pipeline de procesamiento	96
7.53. Reporte sobre consumo de recursos generado por Vivado IP Integrator	97
7.54. Extracto del archivo subsystems/linux/configs/device-tree/pl.dtsi . Contiene información sobre el device tree de los IP Cores instanciados en PL	98
7.55. Extracto del archivo subsystems/linux/configs/device-tree/pl.dtsi modificado con información sobre el uso de cada canal DMA	99
7.56. Secuencia de llamadas de proceso de usuario a kernel para procesar una imagen	100
7.57. Caso de test de ejemplo ejecutable de forma remota para viso.h	102
8.1. Caso de test utilizado en el experimento	106
8.2. Resultados del experimento	106
8.3. Resultados del experimento (cont.)	107
8.4. Caso de test utilizado en el experimento	110
8.5. Resultados del experimento	110
8.6. Resultados del experimento (cont.)	111
8.7. Caso de test utilizado en el experimento	113
8.8. Resultados del experimento	114

8.9. Resultados del experimento (cont.)	115
8.10. Caso de test utilizado en el experimento.	117
8.11. Resultados del experimento.	118
8.12. Resultados del experimento (cont.)	119

Índice de cuadros

5.1. Mediciones de rendimiento de la solución en Zybo	35
5.2. Medición de rendimiento de la solución en Zybo con datos en disco RAM	35
6.1. Juegos de parámetros utilizados en los análisis.	38
6.2. Parámetros de la configuración “robusta”	45
6.3. Rendimiento de cada una de las configuraciones.	49
6.4. Comparativa de rendimiento entre <i>viso</i> y <i>viso_s</i> para la configuración ad hoc	58
7.1. Productividad del IP Core Features	75
7.2. Velocidades de consumo de datos y productividades mínimas de los IP Cores para evitar bloquear el pipeline	75
7.3. Restricciones de productividad para el IP Core Sobel	76
7.4. Restricciones de productividad para el IP Core Half Resolution	78
7.5. Codificación de features en stream de salida.	86
7.6. Codificación de tipos de features	87
7.7. Comparativa de rendimiento entre <i>viso</i> , <i>viso_s</i> y <i>viso_h</i> para la configuración ad hoc	103
8.1. Comparativa de rendimiento entre <i>viso</i> , <i>viso_s</i> y <i>viso_h</i> para la configuración ad hoc	105
8.2. Comparativa: Distancias máximas de <i>viso</i> a ground truth en m. y distancias máximas de <i>viso_s</i> y <i>viso_h</i> a ground truth como proporción de la distancia máxima de <i>viso</i>	108
8.3. Comparativa: Distancias promedio de <i>viso</i> a ground truth en m. y distancias promedio de <i>viso_s</i> y <i>viso_h</i> a ground truth como proporción de la distancia promedio de <i>viso</i>	108
8.4. Juegos de parámetros utilizados en este experimento.	109
8.5. Comparativa: Distancias máximas de <i>viso_s</i> a ground truth en m. y distancias máximas de <i>viso_h</i> a ground truth como proporción de la distancia máxima de <i>viso_s</i>	112
8.6. Comparativa: Distancias promedio de <i>viso_s</i> a ground truth en m. y distancias promedio de <i>viso_h</i> a ground truth como proporción de la distancia promedio de <i>viso_s</i>	112
8.7. Juegos de parámetros utilizados en este experimento.	113

8.8. Comparativa: Distancias máximas de <code>viso_s</code> a <i>ground truth</i> en m. y distancias máximas de <code>viso_h</code> a <i>ground truth</i> como proporción de la distancia máxima de <code>viso_s</code>	116
8.9. Comparativa: Distancias promedio de <code>viso_s</code> a <i>ground truth</i> en m. y distancias promedio de <code>viso_h</code> a <i>ground truth</i> como proporción de la distancia promedio de <code>viso_s</code>	116
8.10. Juegos de parámetros utilizados en este experimento.	117
8.11. Comparativa: Distancias máximas a <i>ground truth</i> en m. de <code>viso_h</code> utilizando las configuraciones adhoc y 10fps	120
8.12. Comparativa: Distancias promedio a <i>ground truth</i> en m. de <code>viso_h</code> utilizando las configuraciones adhoc y 10fps	120

Bibliografía

- Diligent, Inc. (2014). *ZYBO Reference Manual*. URL: http://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPZYBO/documentation/ZYBO_RM_B_V6.pdf.
- García, Miguel y Patricia Borensztein (2016). “200 times speedup on a case of Image filters optimization using High Level Synthesis”. En:
- Geiger, Andreas, Philip Lenz y Raquel Urtasun (2012). “Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite”. En: *Conference on Computer Vision and Pattern Recognition (CVPR)*.
- Geiger, Andreas, Julius Ziegler y Christoph Stiller (2011). “StereoScan: Dense 3D Reconstruction in Real-time”. En: *Intelligent Vehicles Symposium (IV)*.
- Pedre, Sol, Tomás Krajník, Elías Todorovich y Patricia Borensztein (2012). “A co-design methodology for processor-centric embedded systems with hardware acceleration using FPGA”. En: *Programmable Logic (SPL), 2012 VIII Southern Conference on*.
- Vallina, Fernando Martínez (2012). “Implementing Memory Structures for Video Processing in the Vivado HLS Tool”. En: *XAPP793*.
- Xilinx (2011). *AXI Reference Guide*. Ed. por Xilinx. URL: http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.
- (2014a). *Vivado Design Suite User Guide*. URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_3/ug893-vivado-ide.pdf.
- (2014b). “Vivado Design Suite User Guide: High-Level Synthesis”. En: *Vivado Design Suite User Guide*.
- (2015a). *AXI Timer v2.0*. Ed. por Xilinx. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_timer/v2_0/pg079-axi-timer.pdf.
- (2015b). *AXI4-Stream Infrastructure IP Suite*. Ed. por Xilinx. URL: http://www.xilinx.com/support/documentation/ip_documentation/axis_infrastructure_ip_suite/v1_1/pg085-axi4stream-infrastructure.pdf.
- (2016). *Zynq-7000 All Programmable SoC Overview*. Ed. por Xilinx. URL: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.