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

Miguel Ángel García

Directora: Patricia Miriam Borensztejn

Departamento de Computación, FCEN, UBA - 2016

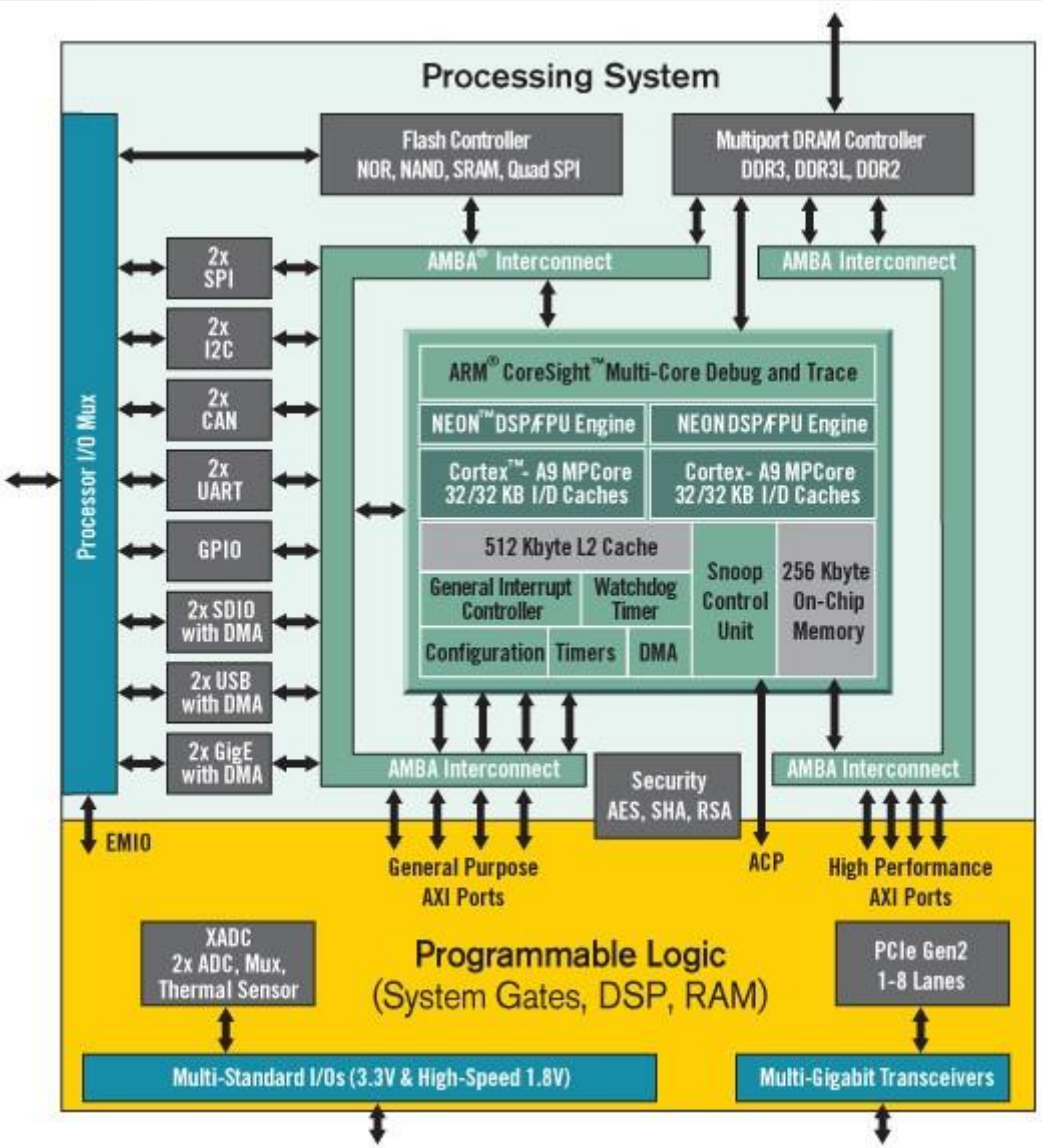
Por qué usar AP SoCs ?

Sistemas embebidos → restricciones:

- Espacio
- Peso
- Energía
- Rendimiento

Los All Programmable SoCs (**AP SoC**) las atacan delegando parte del trabajo a hardware específico, que puede diseñarse utilizando **HLS**.

Qué son los **AP SoC**s



Xilinx Zynq-7000

Capitulo 2

Objetivos de este trabajo

Analizar la construcción de soluciones de visión artificial sobre AP SoCs, considerando:

- Metodología para el ciclo completo de desarrollo.
- Aceleración por hardware utilizando Síntesis de Alto Nivel (**HLS**).
- Diseño de todas las capas del sistema.
- Potencia de Processing System **(PS)** vs Processing System + Lógica Programable **(PL)**.
- Herramientas disponibles.

Placa de desarrollo Zybo

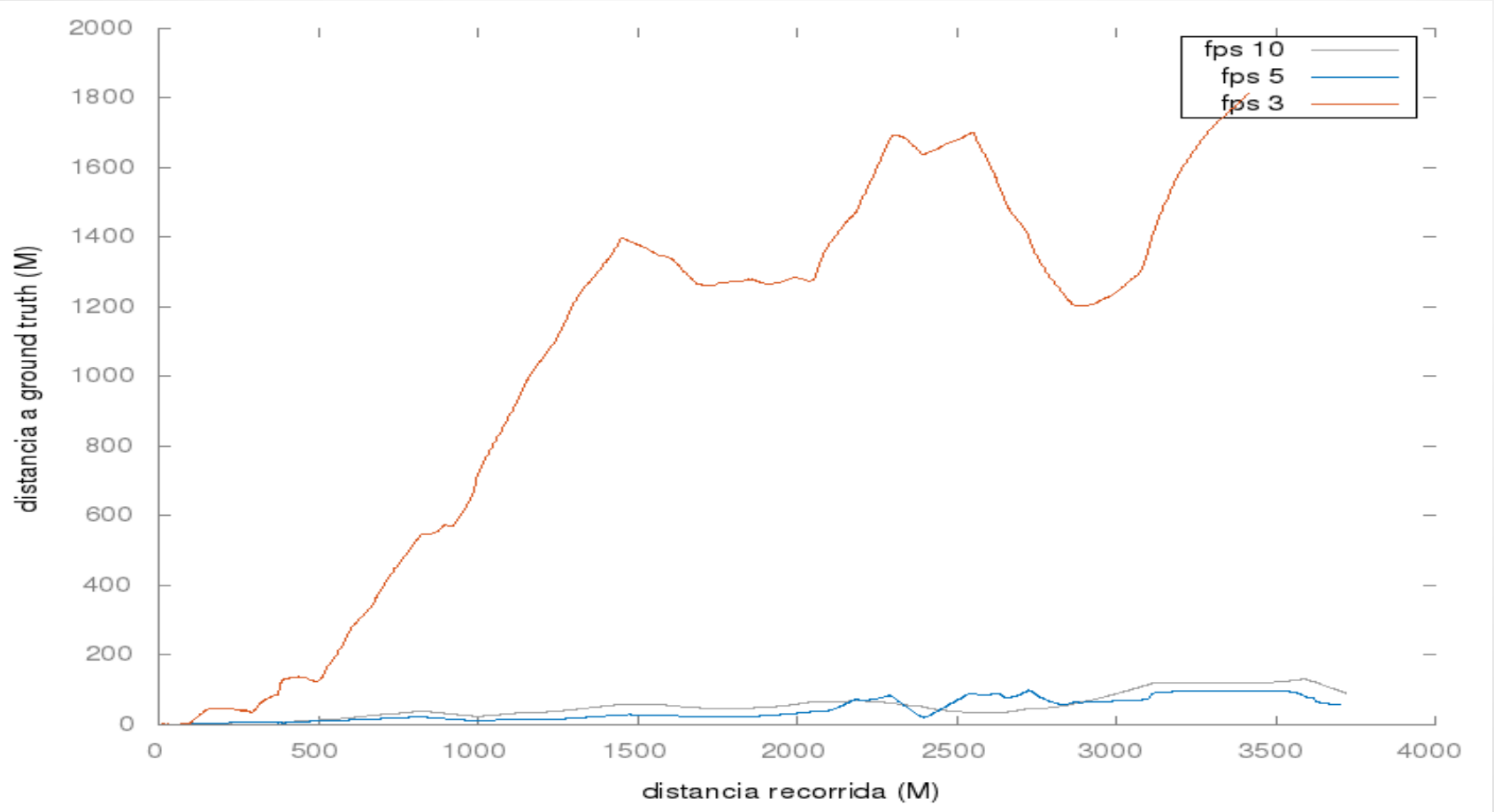
- AP SoC Zynq-7010
  - ARM Cortex-A9 **Dual Core** a 650Mhz
  - La versión más chica de **Zynq** en lo referido a PL (La parte de PS es siempre igual)
- RAM externa: 512 MB DDR3
- Ethernet: 10Mb/100Mb/1Gb
- Precio u$d189

Caso de Estudio

- Implementación de una solución de **Odometría Visual Estéreo** sobre Zybo.
  - **LIBVISO2** como punto de partida.
  - Data set **KITTI**: capturado a **10 fps**.

Impacta el rendimiento en el resultado ?

- Experimento en **PC** para probar cómo afecta al resultado la pérdida de cuadros:

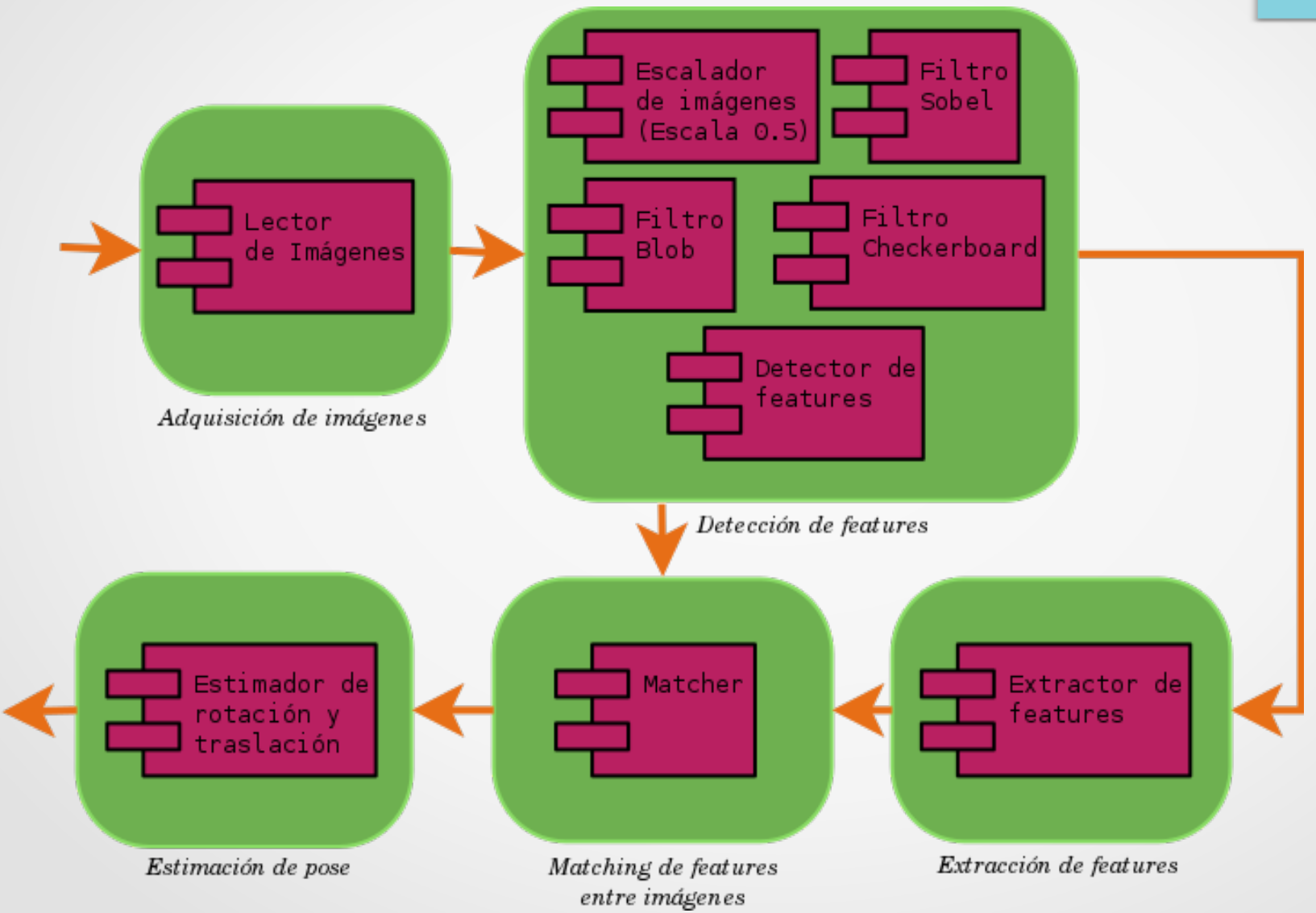


Metodología de Trabajo

Metodología **integral** para el desarrollo de sistemas utilizando **co-diseño hardware/software**

- A) Diseño
- B) Implementación y testing en CPU de propósito general
- C) Partición hardware/software
  - Migrar software al AP SoC y aplicar **optimizaciones software**
  - Medir rendimiento → **Partición preliminar**
- D) Implementación de hardware (**IP Cores**), testing e integración

A) Diseño


Lector de Imágenes
Adquisición de imágenes
Escalador de imágenes (Escala 0.5)
Filtro Sobel
Filtro Blob
Filtro Checkerboard
Detector de features
Detección de features
Extractor de features
Extracción de features
Matcher
Matching de features entre imágenes
Estimador de rotación y traslación
Estimación de pose

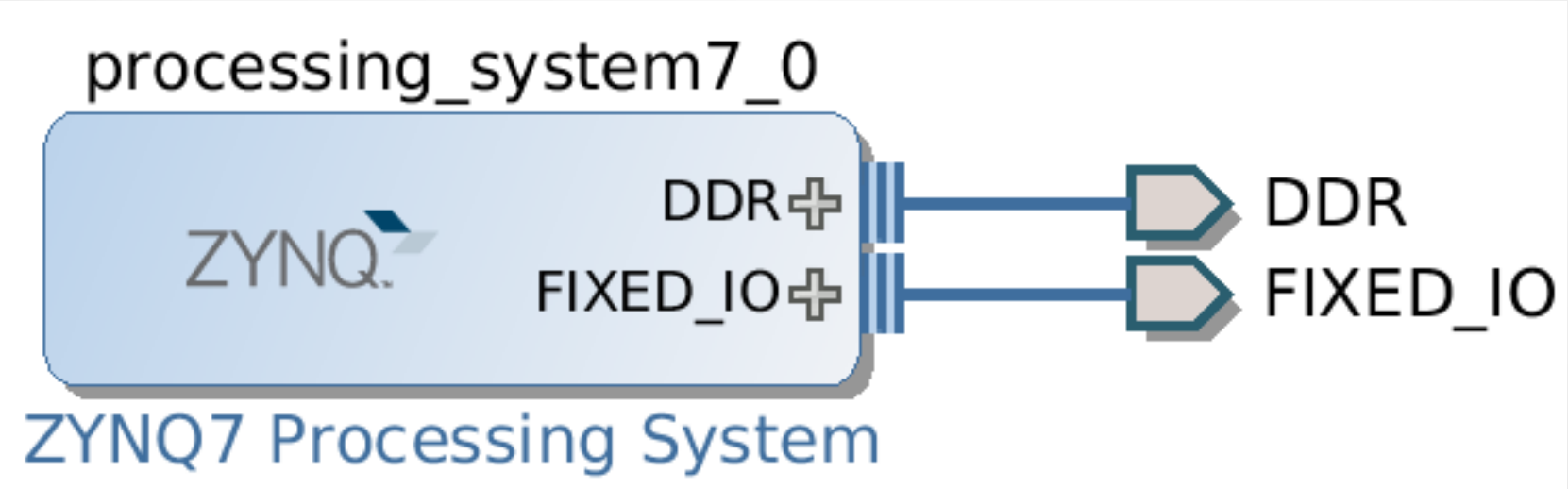
B) Implementación en **CPU** de propósito general

- Objetivo: construir un **“golden model”**
- Modificaciones a **LIBVISO2**:
  - Calibración y parametrización desde archivos.
  - Simulación pérdida de cuadros.

- Automatización de pruebas:
  - **run_test_case**: json + gnuplot

C) Migración a Zybo

- Configuración plataforma hardware
  - Solo utilizamos PS y algunos periféricos



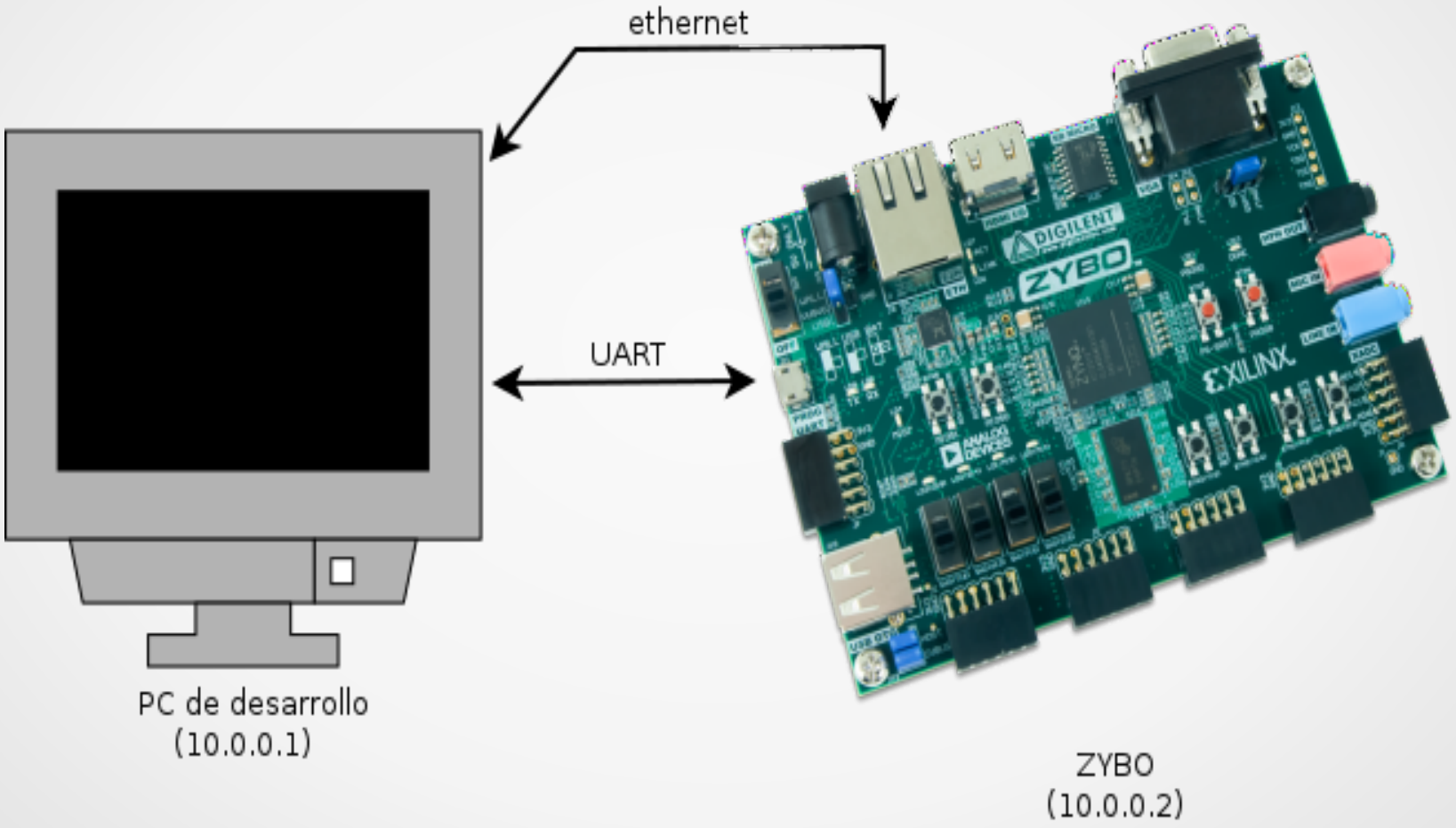
Plataforma de Software

- **Permite explotar la plataforma de hardware.**
- **PetaLinux:** distribución Linux de Xilinx.
  - SDK para co-diseño
  - Niveles de privilegio: Usuario y Kernel
  - Soporta SMP y VM
  - Documentación
  - Boot en la placa usando **xmd**

Migración de la aplicación

- Nueva aplicación “**viso**”
  - **SSE → Neon**
  - **PNG → raw**
    - kitti2raw
- Aplicación **viso_profile**: igual a viso pero compilada con opciones de profiling.

Acceso a datasets desde Zybo: **NFS**



Pruebas de la solución en Zybo

- Automatización de las pruebas:
  - **run_test_case_remote**: json + ssh + nfs + gnuplot

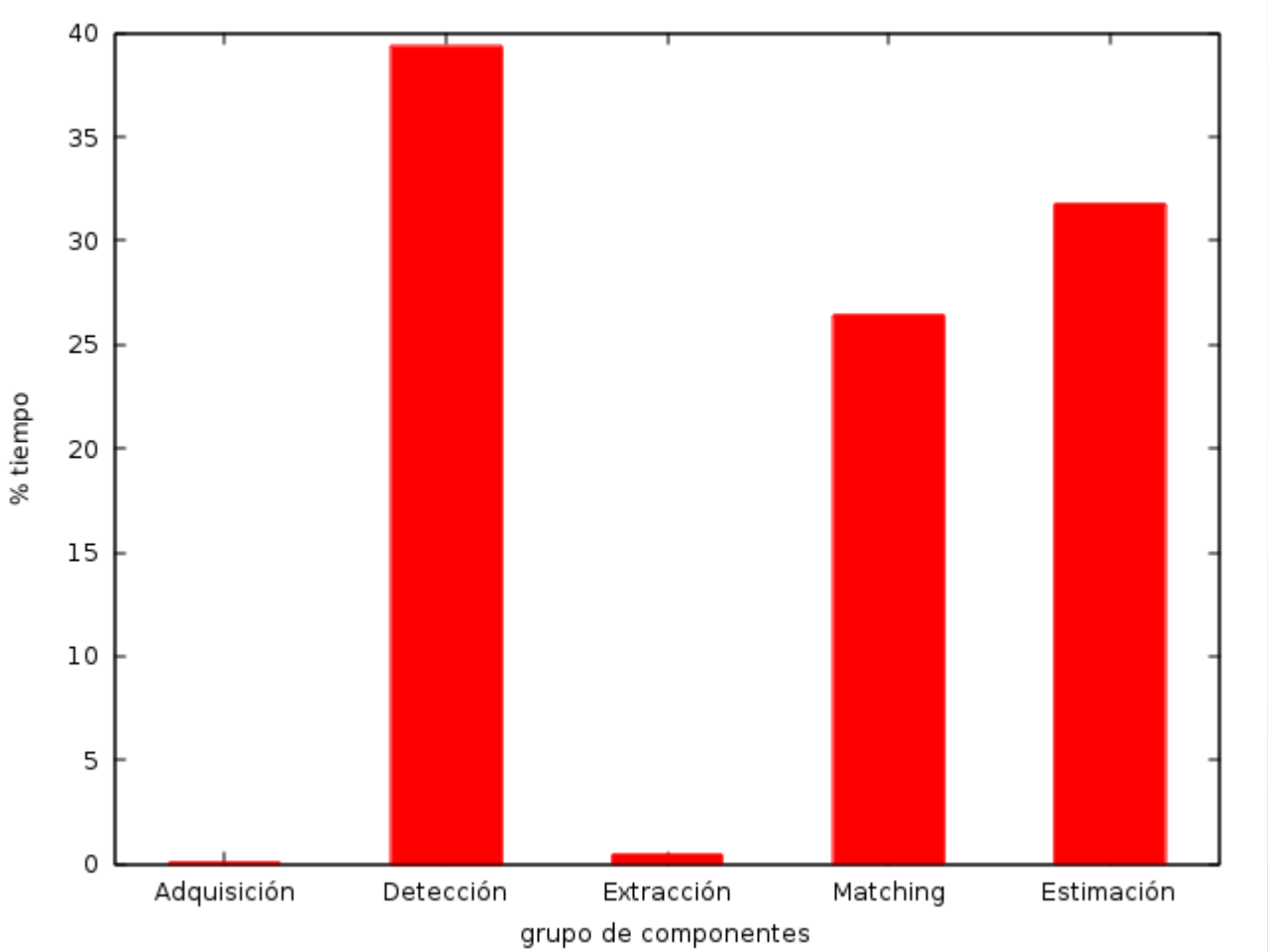
- Rendimiento, dependiendo de la fuente de datos:
  - NFS: 1,94 FPS
  - Disco RAM: **2,55 FPS**

Optimización Software

- Estudio del impacto de los parámetros
  - Calidad del resultado
  - Rendimiento
  - Tolerancia a la pérdida de cuadros
    - → Configuración **“ad hoc”**
- Análisis de **viso**
- Aplicación de optimizaciones
- Medición

Análisis del rendimiento de viso

- Resultados para **viso_profile** utilizando la configuración **“ad hoc”**:

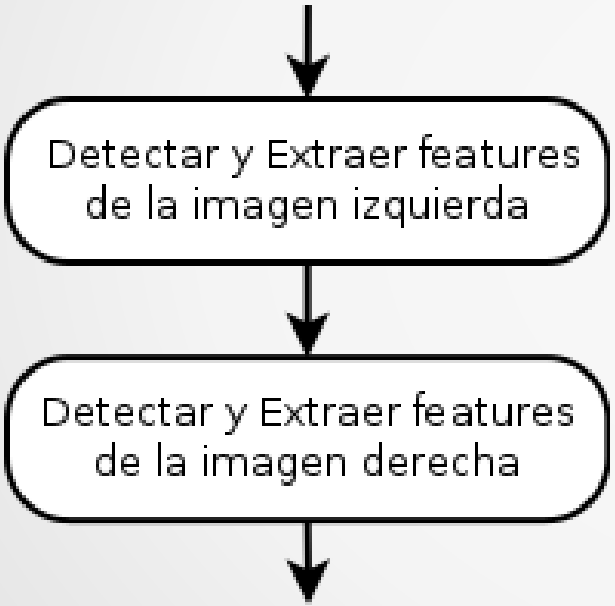


Aplicación viso_s

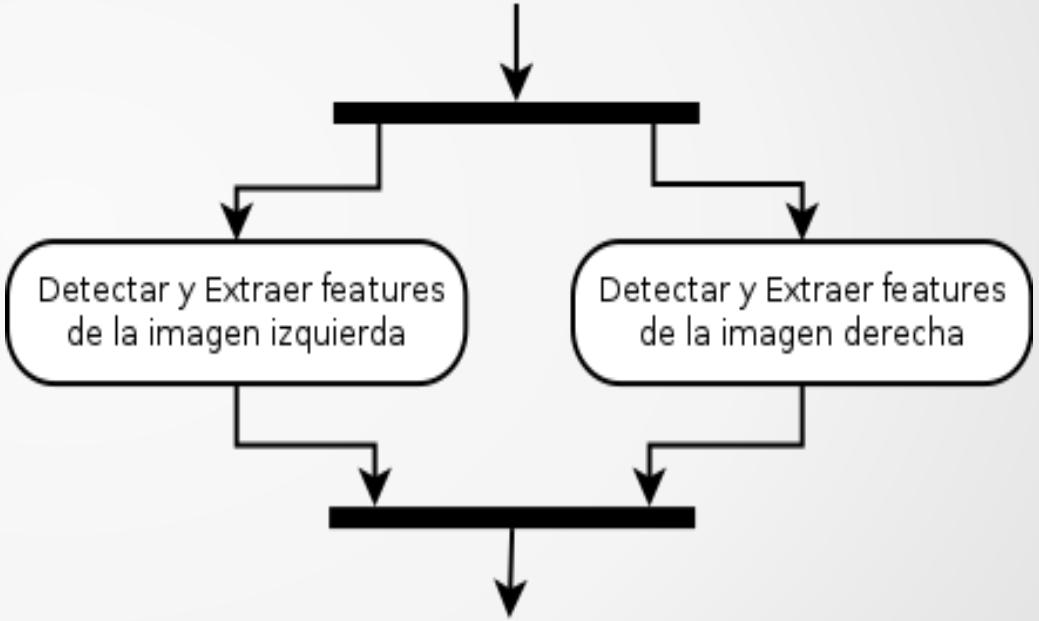
- Nueva aplicación **viso_s**, objetivo: mejorar el rendimiento de **viso** explotando los dos procesadores.
- Paralelizamos el procesamiento:
  - Threads, mediante la biblioteca pthreads.
  - Requirió detectar secciones paralelizables y modificar el código.

Detección y Extracción de features

VISO:



VISO_S:



Resultados de viso_s

- Rendimiento (Disco RAM): 4,23 **FPS**

  65% más rápido que viso

D) Optimización Hardware

- Consiste en delegar parte de la funcionalidad al hardware:
  - Particionar funcionalidad
  - Diseñar IP Cores mediante **HLS**
  - Integrar: hardware y software

Dificultades del diseño de hardware

- **Vivado HLS** permite utilizar lenguajes de alto nivel, pero el código no es ejecutado por una CPU.

- Deben introducirse decisiones:
  - Interfaces
  - Optimizaciones y Paralelismo
  - Jerarquía de memoria

- Codificar algoritmos aprovechando las características del hardware.

Índice Optimización Hardware

- **Patrón de diseño para procesamiento de imágenes.**
  - **Optimizaciones aplicables**
- Arquitectura de hardware
  - IP Cores
  - Integración a la plataforma hardware
- Plataforma software
- Modificaciones a la aplicación

Patrón de diseño IP Cores para imágenes

- Basado en documento técnico de **Xilinx** (Vallina).
- Aplicable a IP Cores que procesan **secuencialmente** los píxeles de una imagen:
  - Caracteriza entradas y salidas como flujos unidireccionales (**streams)**.

Patrón: Jerarquía de memoria

- Para procesar cada píxel suele ser necesario un contexto de NxM alrededor del mismo.

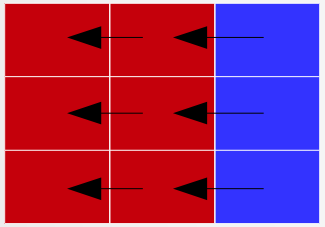
  → Guardamos una porción de la imagen en dos tipos de **buffers**:

  - **Line Buffers (LB):** acceso a un elemento para escritura y otro para lectura por ciclo: Block RAM
  - **Window:** acceso simultáneo a todos sus elementos: LUTs RAM

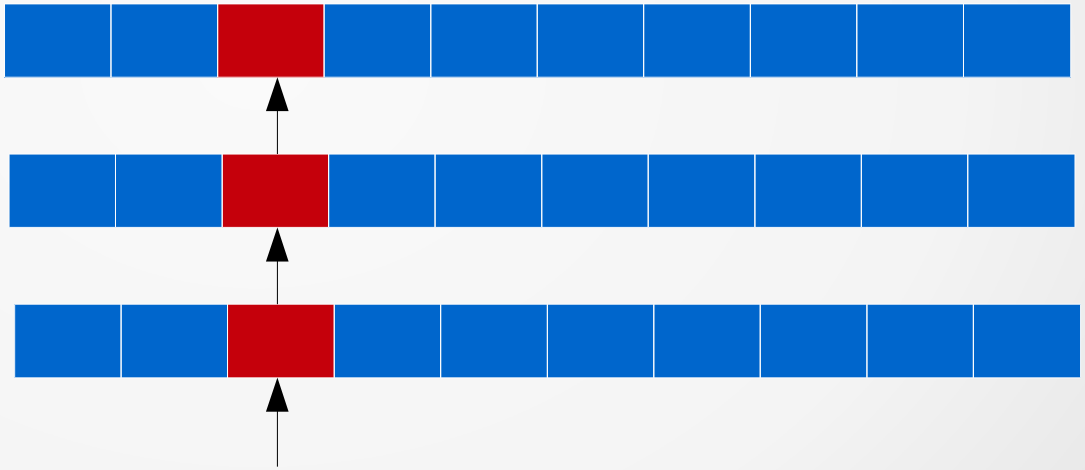
Patrón: algoritmo general, paso 1

Leer un píxel del stream + **desplazar** Window y Line Buffers (ej 3er píxel de una línea):

Window:

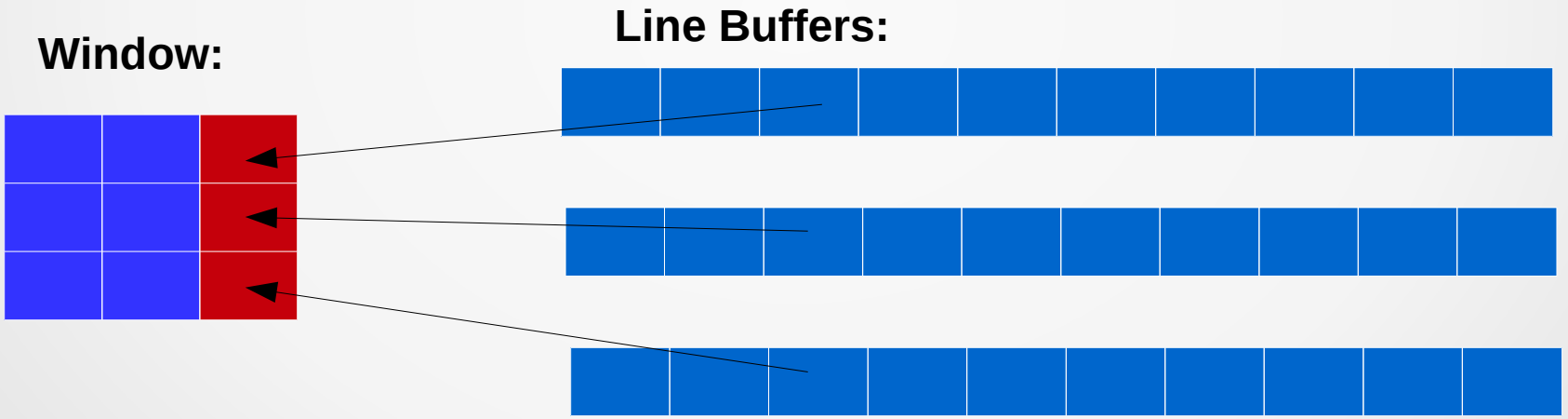


Line Buffers:



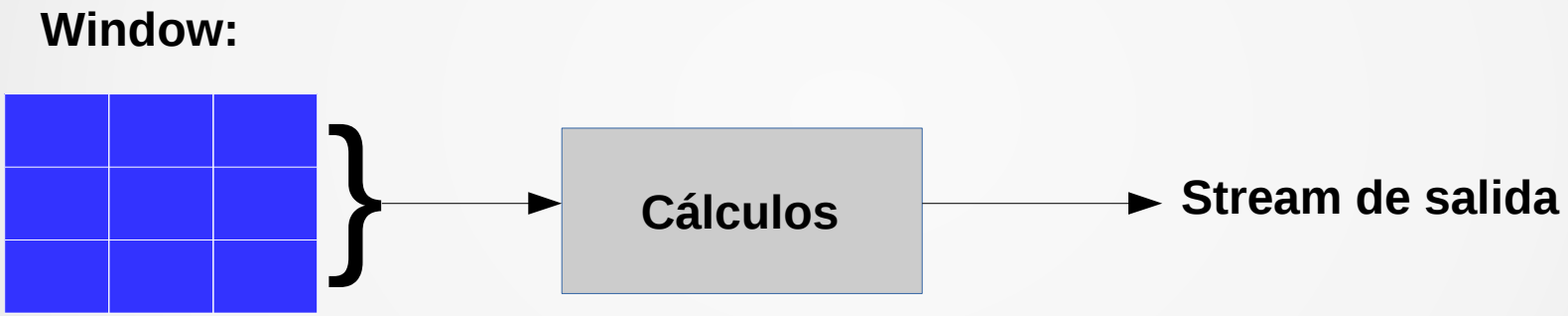
Patrón: algoritmo general, paso 2

Alimentar Window desde Line Buffers:



Patrón: algoritmo general, paso 3

Procesar Window:



Patrón: Análisis del rendimiento

- Uso de LOOP_TRIPCOUNT para estimar los ciclos que toma procesar una imagen de 1024x768 píxeles:

Latency		Interval		
min	max	min	max	Type
11798017	11798017	11798018	11798018	none

→ **~0,066 píxeles/ciclo**

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

Patrón: Optimización directiva PIPELINE

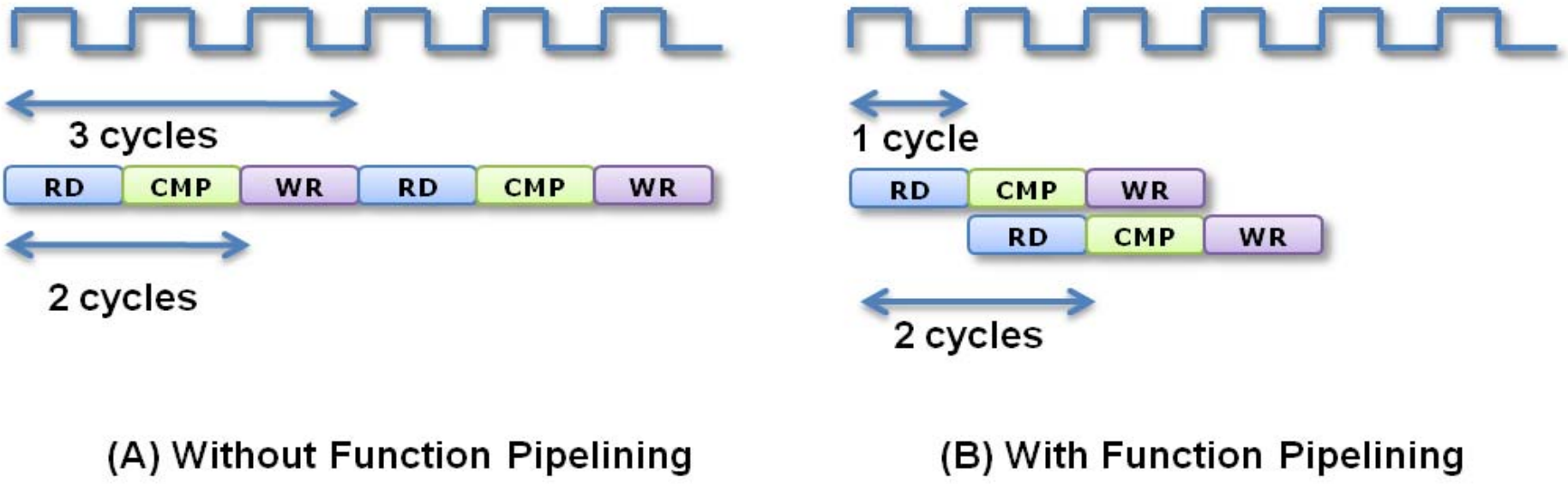
- Aplicable a funciones y bucles

```
void func(...) {

    op_Read;
    op_Compute;
    op_Write;

}
```

RD
CMP
WR



(A) Without Function Pipelining

(B) With Function Pipelining

Patrón: Optimización directiva PIPELINE

- Resultados:

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

→ **~0,99 píxeles/ciclo**

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

Patrón: Optimización K píxeles por ciclo

- **Problema:** de un **stream** solo se pueden leer o escribir un dato por ciclo.
- **Idea propuesta:** usar **streams** más “anchos”. Ahora cada dato es un **grupo de píxeles**.
- Implica modificar:
  - Streams
  - Line Buffers
  - Window
  - Algoritmo

Patrón: Ejemplo **4** píxeles por ciclo

- Resultados

Latency		Interval		
min	max	min	max	Type
196616	196616	196617	196617	none

~3,99 píxeles/ciclo

Name	BRAM_18K	DSP48E	FF	LUT
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

< 250% de la implementación de ~0,99 píxeles/ciclo

Patrón: fusión de IP Cores

- Útil cuando se realizan múltiples operaciones sobre una misma imagen.
- Reduce recursos al **compartir**:
  - Streams
  - Line Buffers
  - Window
  - Parte de la lógica de control

Patrón: fusión de IP Cores

- Resultados

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

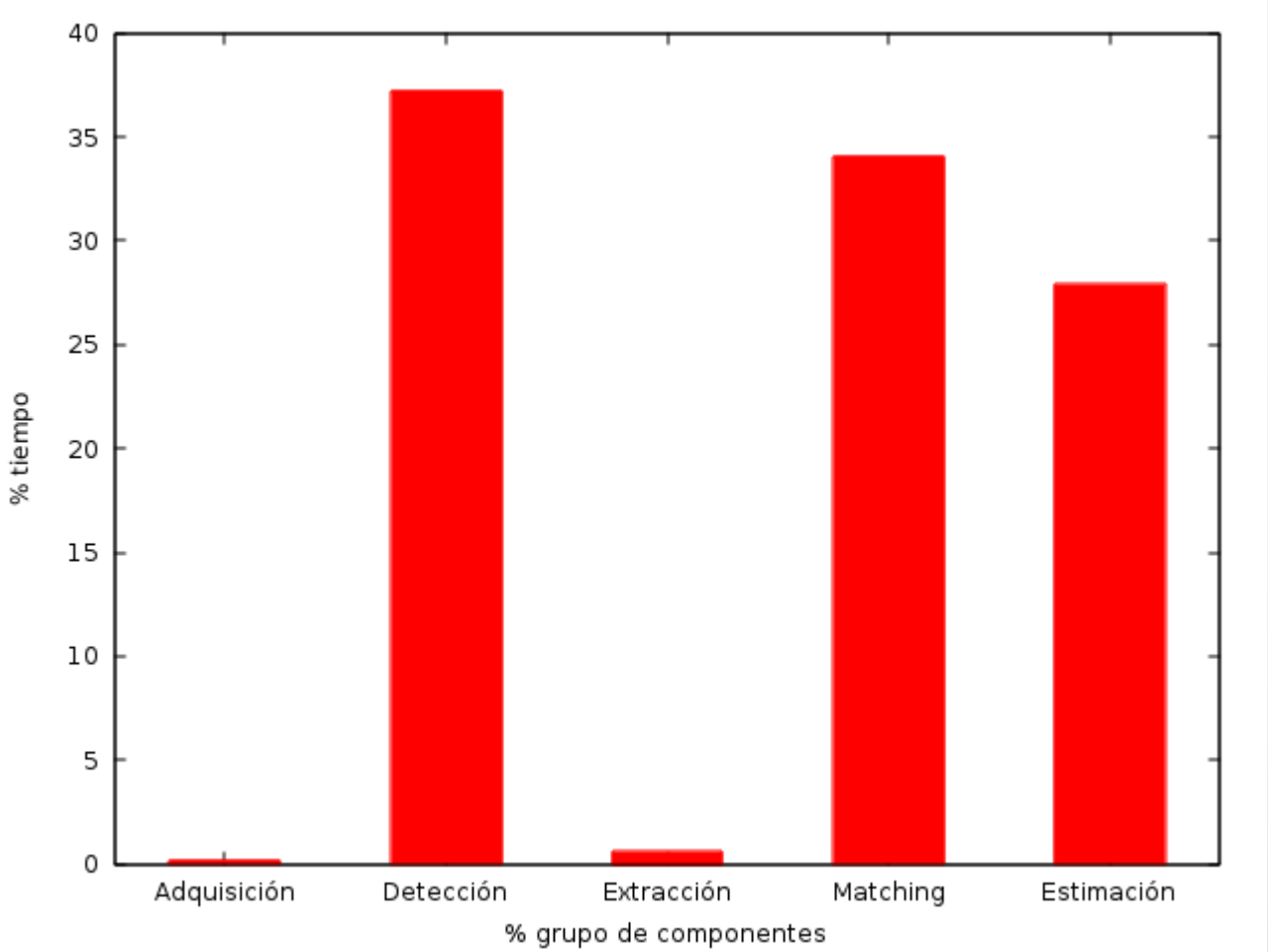
No se altera

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	259
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

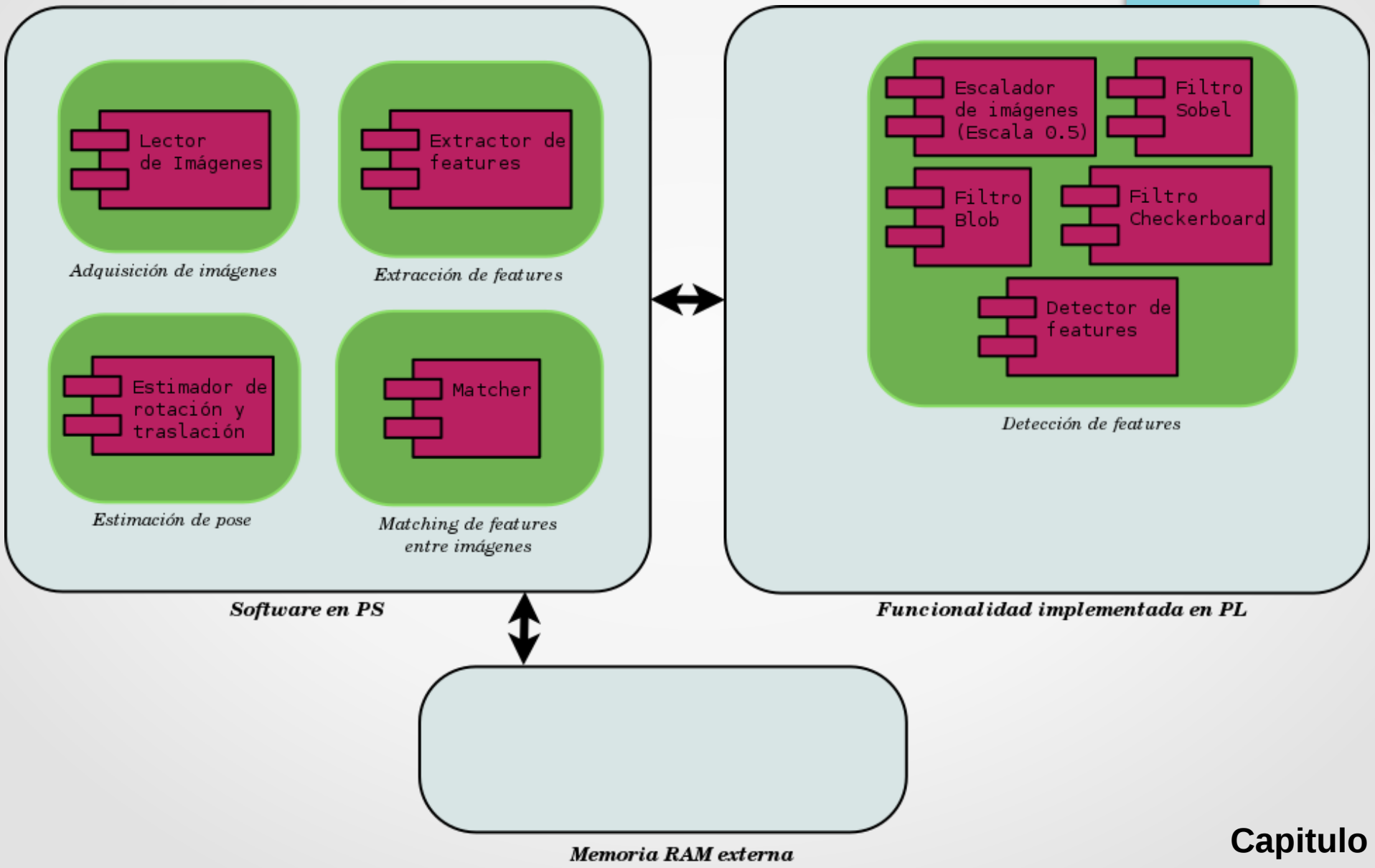
Consumo de recursos: ~60% que dos IP Cores independientes

Análisis del rendimiento de viso_s

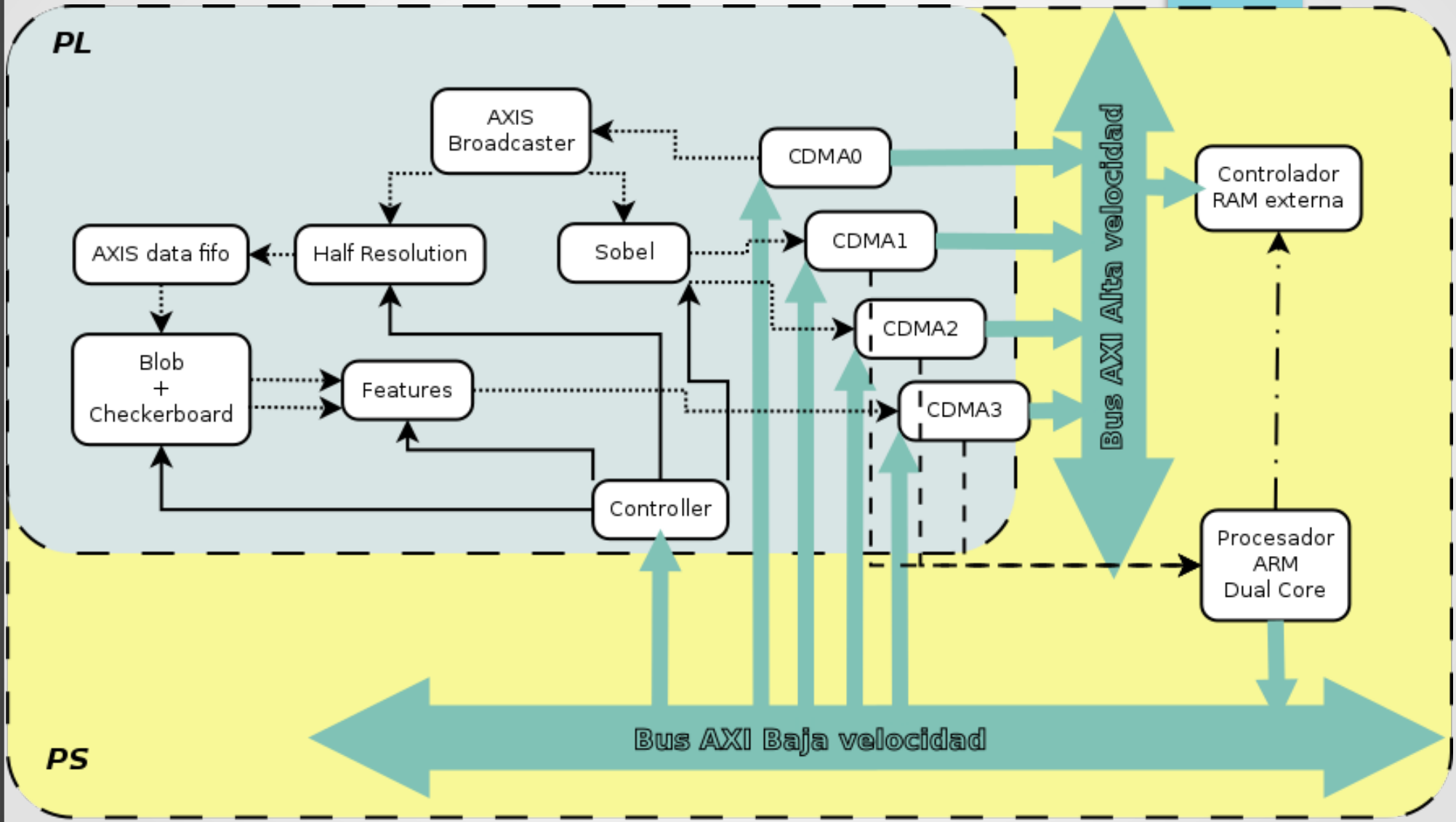
- Resultados para **viso_s_profile** utilizando la configuración **“ad hoc”**



Partición hardware/software

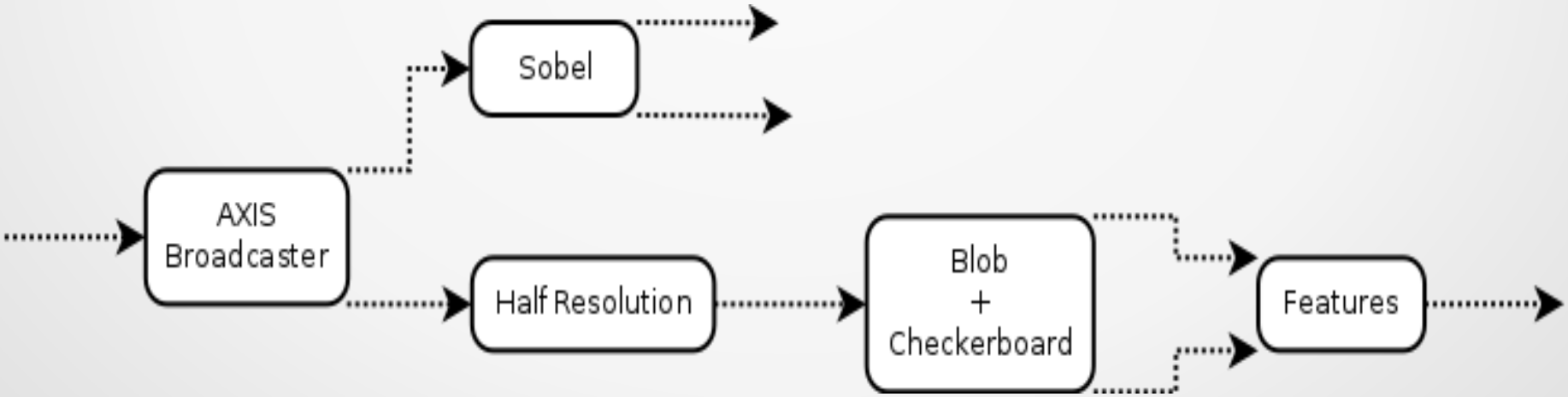


Arquitectura de hardware



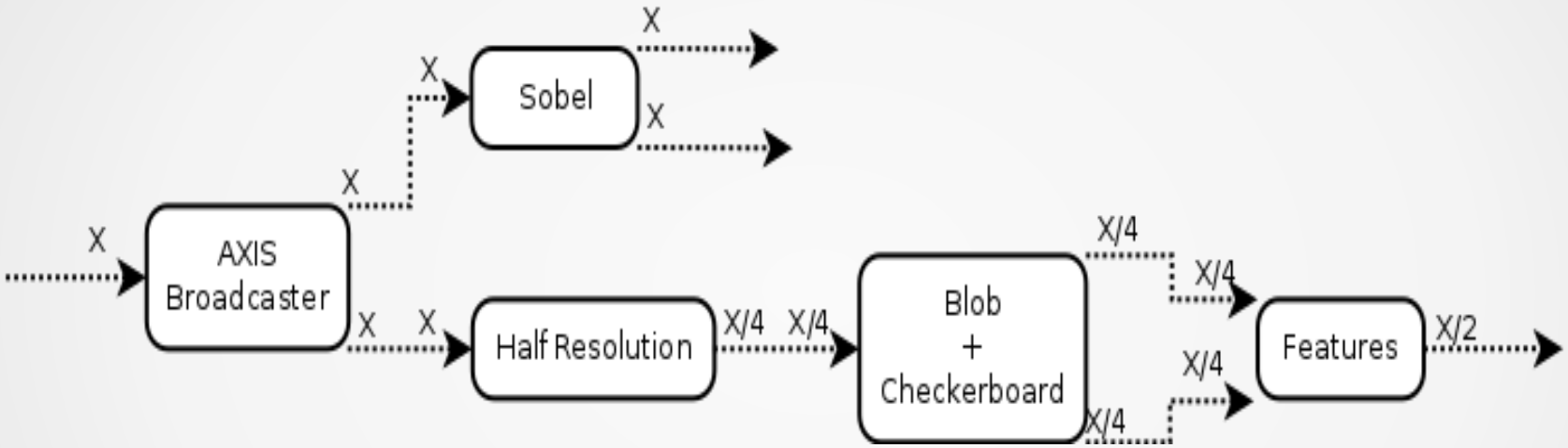
Pipeline de procesamiento de imágenes

- Los **IP Cores** forman un **pipeline**: consumen lo producido por otros.
- No es necesario esperar a que el anterior termine.
- Queremos evitar **cuellos de botella**.



Pipeline de procesamiento de imágenes

- Estudiamos el rendimiento requerido para cada IP Core.



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

Índice de IP Cores implementados

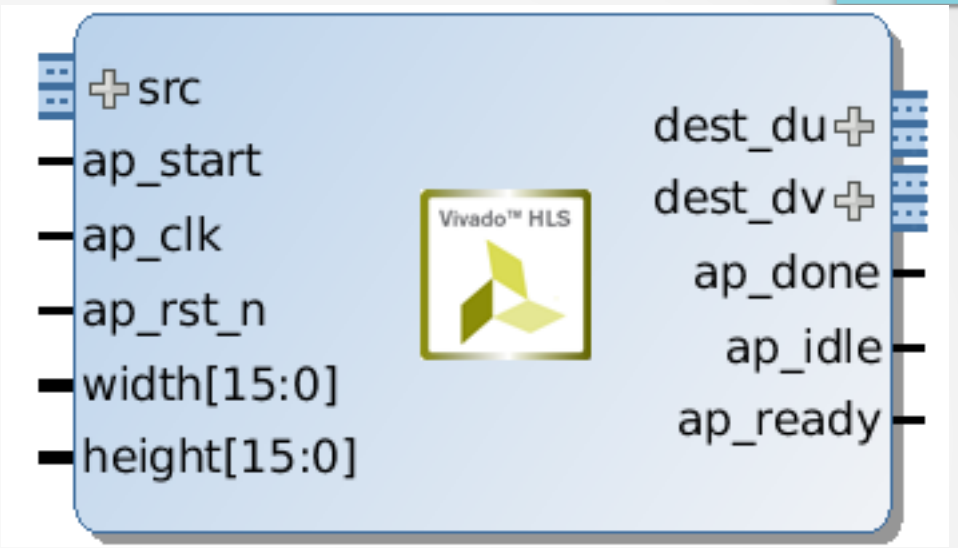
1) Sobel
2) Half Resolution
3) Blob + Checkerboard
4) Features
5) Controller

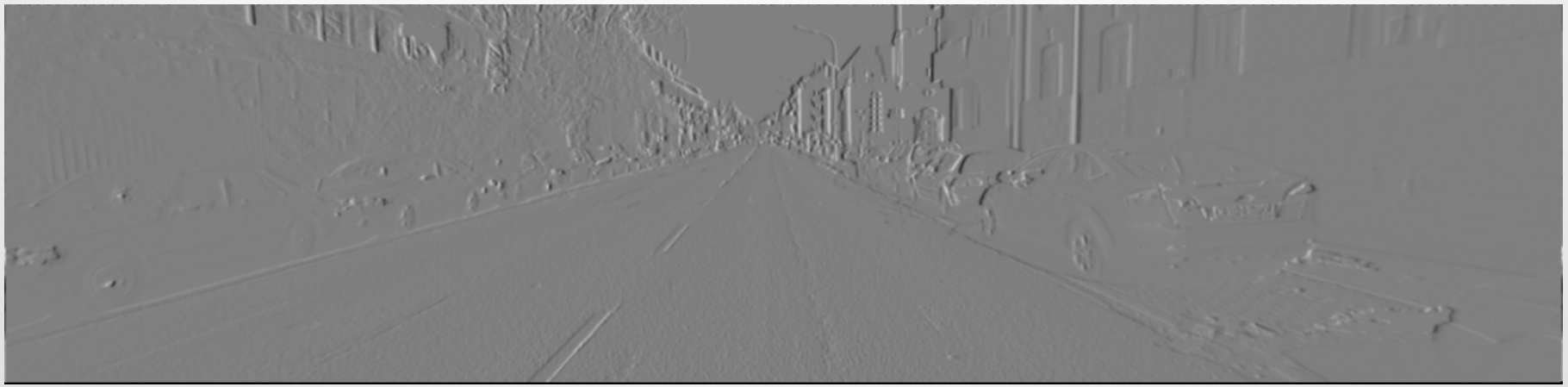
1) IP Core Sobel

- Los filtros Sobel horizontal y vertical resultan de aplicar dos convoluciones de 5x5 a la imagen.
- Productividad requerida: 2 píxeles/ciclo
- Usamos:
  - Patrón para procesamiento de imágenes
    - Window de 5x5 y 5 LB
  - Optimizaciones:
    - K píxeles por ciclo
    - Fusión de IP Cores
    - Directiva PIPELINE

1) IP Core Sobel

- IP Core **Sobel16**:





Resultado de la **simulación**: Sobel horizontal

1) IP Core Sobel

- Rendimiento y consumo de recursos de **Sobel16** (1344x372 píxeles):

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

→ **~1,98 píxeles/ciclo**

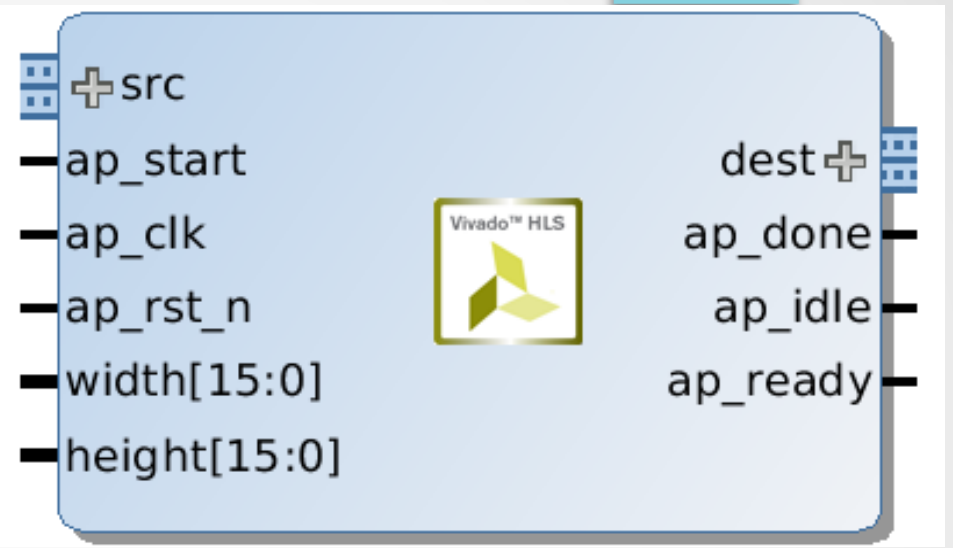
Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	1018
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	8	-	0	0
Multiplexer	-	-	-	29
Register	-	-	813	4
Total	8	0	813	1051
Available	120	80	35200	17600
Utilization (%)	6	0	2	5

2) IP Core Half Resolution

- Para cada cuadrado de 2x2 píxeles se genera un solo píxel con su promedio.
- Productividad requerida: 0,5 píxeles/ciclo
- **No** empleamos el patrón de diseño
- Aunque nos sirvieron las ideas:
  - E/S mediante streams
  - Procesar K píxeles.
  - Aplicar la directiva PIPELINE.

2) IP Core Half Resolution

- IP Core **HalfResolution16**:



Resultado de la **simulación**

2) IP Core Half Resolution

- Rendimiento y consumo de recursos de **HalfResolution16 (672x186 píxeles)**:

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

Productividad: ~0,5 píxeles/ciclo

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

3) IP Core Blob + Checkerboard

- Los filtros Blob y Checkerboard resultan de aplicar una convolución de 5x5, cada uno, a la imagen.
- Productividad requerida: 0,5 píxeles/ciclo
- Usamos:
  - Patrón para procesamiento de imágenes
  - Optimizaciones:
    - Fusión de IP Cores
    - Directiva PIPELINE

3) IP Core Blob + Checkerboard

- IP Core **Blob + Checkerboard**:

Resultado de la **simulación**: Blob

3) IP Core Blob + Checkerboard

- Rendimiento y consumo de recursos de Blob + Checkerboard (672x186 píxeles):

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

→ **~0,97 píxeles/ciclo**

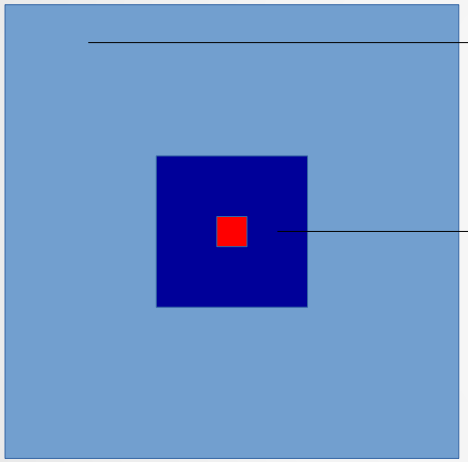
Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	516
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	4	-	0	0
Multiplexer	-	-	-	93
Register	-	-	666	119
Total	4	0	666	728
Available	120	80	35200	17600
Utilization (%)	3	0	1	4

4) IP Core Features

- Realiza la detección de mínimos y máximos locales, en dos imágenes de manera simultanea.
- Velocidad de lectura requerida: 0,5 píxeles/ciclo
- Usamos:
  - Patrón para procesamiento de imágenes
  - Optimizaciones:
    - Fusión de IP Cores
    - Directiva PIPELINE

4) IP Core Features

En cada iteración se analiza el píxel del centro de la ventana:



Window para cada imagen

Dispersos: mínimos y máximos en la ventana de 21x21

Densos: mínimos y máximos en la **sub ventana** de 11x11

4) IP Core Features

- Para saber si el píxel del centro es extremo alcanza con conocer el valor y fila de los extremo para cada columna, completa y restringida entre las filas 5 y 15
- Reemplazamos una ventana de 21x21 celdas de 16 bits por 4 de 21x1 de 16 bits y 4 de 21x1 de 5 bits →

Condensando la información de las ventanas ahorramos **75%** de recursos.

4) IP Core Features

- IP Core **Features**:

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
...			

Resultado de la **simulación**

4) IP Core Features

- Rendimiento y consumo de recursos de **Features** (672x186 píxeles):

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

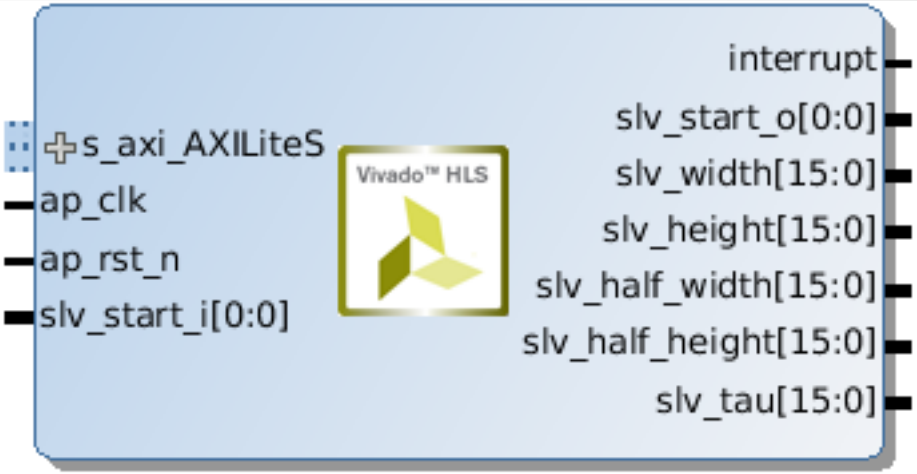
~0,49 píxeles/ciclo

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

En Zynq7010 no puede utilizarse más de una instancia

5) IP Core Controller

- Es el **único** que es accedido desde los procesadores, mediante el bus AXI:
  - Ahorro de recursos al minimizar interfaces AXI Slave
  - Coherencia
  - Simplifica software



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

Recursos necesarios para implementar la interfaz AXI Slave

Integración

- **Plataforma de hardware**
  - Incorporar los IP Cores
  - Configurar DMA y PS
- **Plataforma de software**
  - Actualizar árbol de dispositivos
  - Implementar controlador

Integración del hardware

- Configuración de PS, habilitando los buses para la comunicación con PL

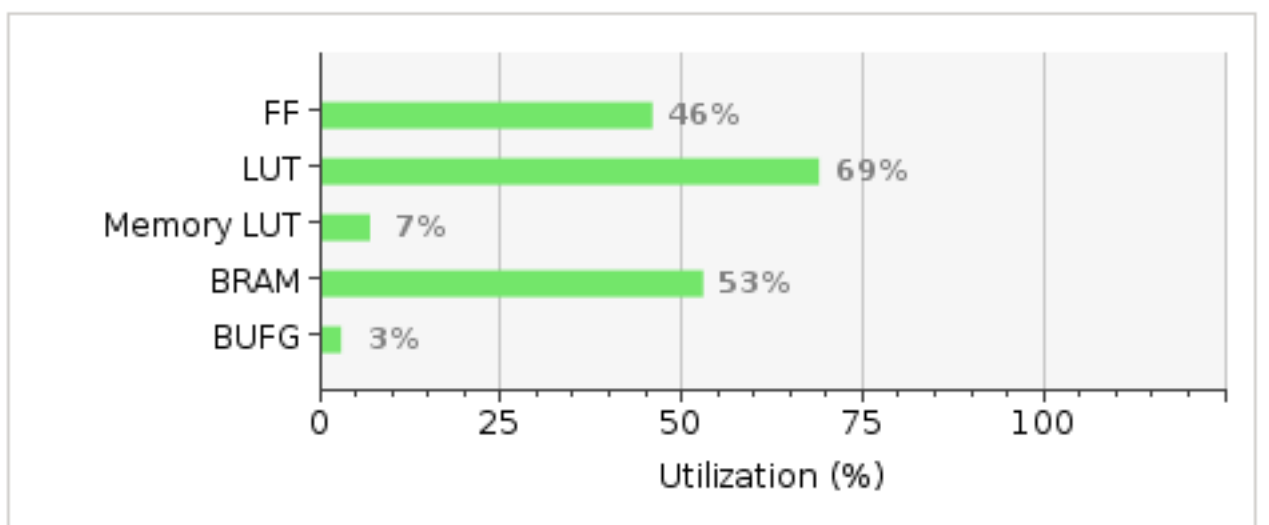
Integración del hardware

- Consistió en:
  - Instanciar IP Cores
  - Configurar canales DMA
  - Realizar las conexiones necesarias entre los IP Cores y con PS.
  - Generar archivo Bitstream

  Veamoslo en Vivado IP Integrator...

Prueba del hardware

- Realizamos una prueba **standalone** (sin SO).
- Detectar Features y calcular filtros Sobel de una imagen de 1344x372 píxeles tomó 0,003 segundos → la plataforma es capaz de procesar **300 imágenes/segundo**.



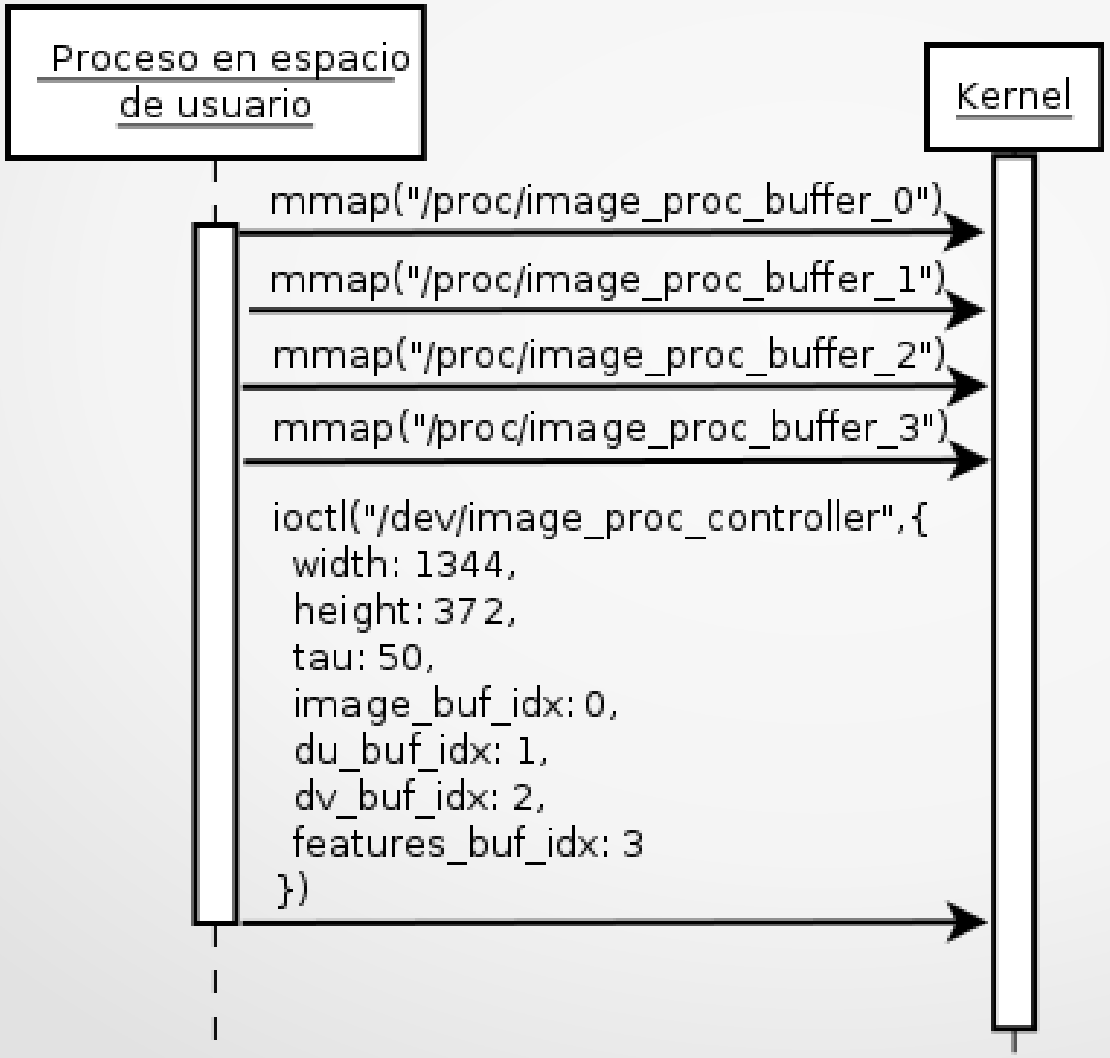
Consumo de recursos.

Plataforma software

- Árbol de dispositivos
- Módulo controlador (driver): módulo del kernel
  - Transferencias DMA
  - Manejo del IP Core Controller
  - Alocación de buffers contiguos en RAM
  - API
    - mmap
    - IOCTL

Prueba del módulo controlador

- Construimos una aplicación de prueba para el módulo.



Nueva aplicación viso_h

- Nuevas aplicaciones **viso_h** y **viso_h_profile** a partir de **viso_s**
  - Reemplazo de malloc y free por buffers contiguos.
  - Uso del filtro Sobel de la imagen completa para el cálculo de descriptores.
- Delegar tareas al módulo controlador
  - Cálculo de filtros Sobel.
  - Detección de features.

Pruebas de viso_h en Zybo

- Rendimiento (Disco RAM): 7,42 **FPS**

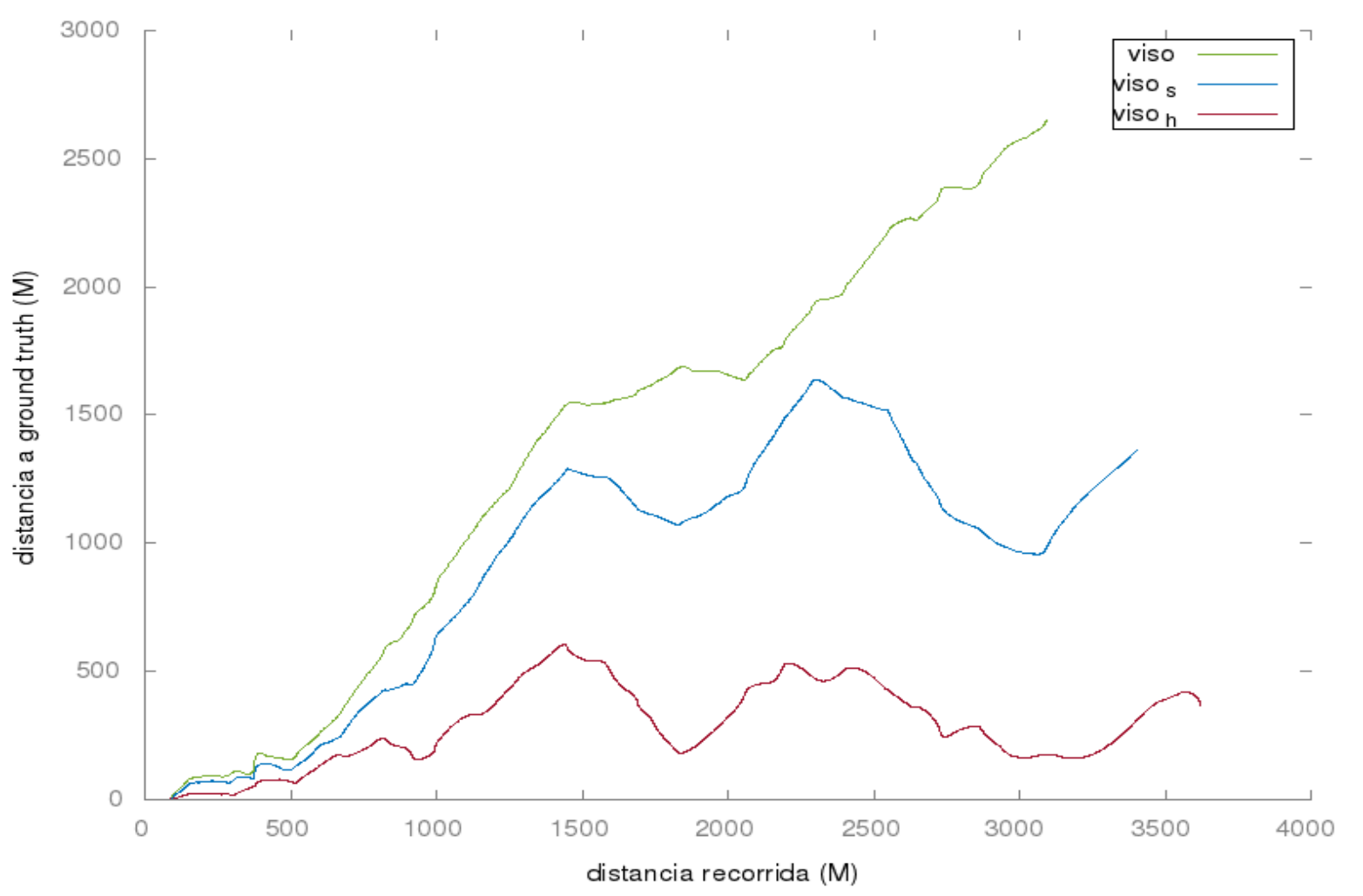
  75% más rápido que viso_s

  190% más rápido que viso

  Se pierden menos de la mitad de los cuadros !

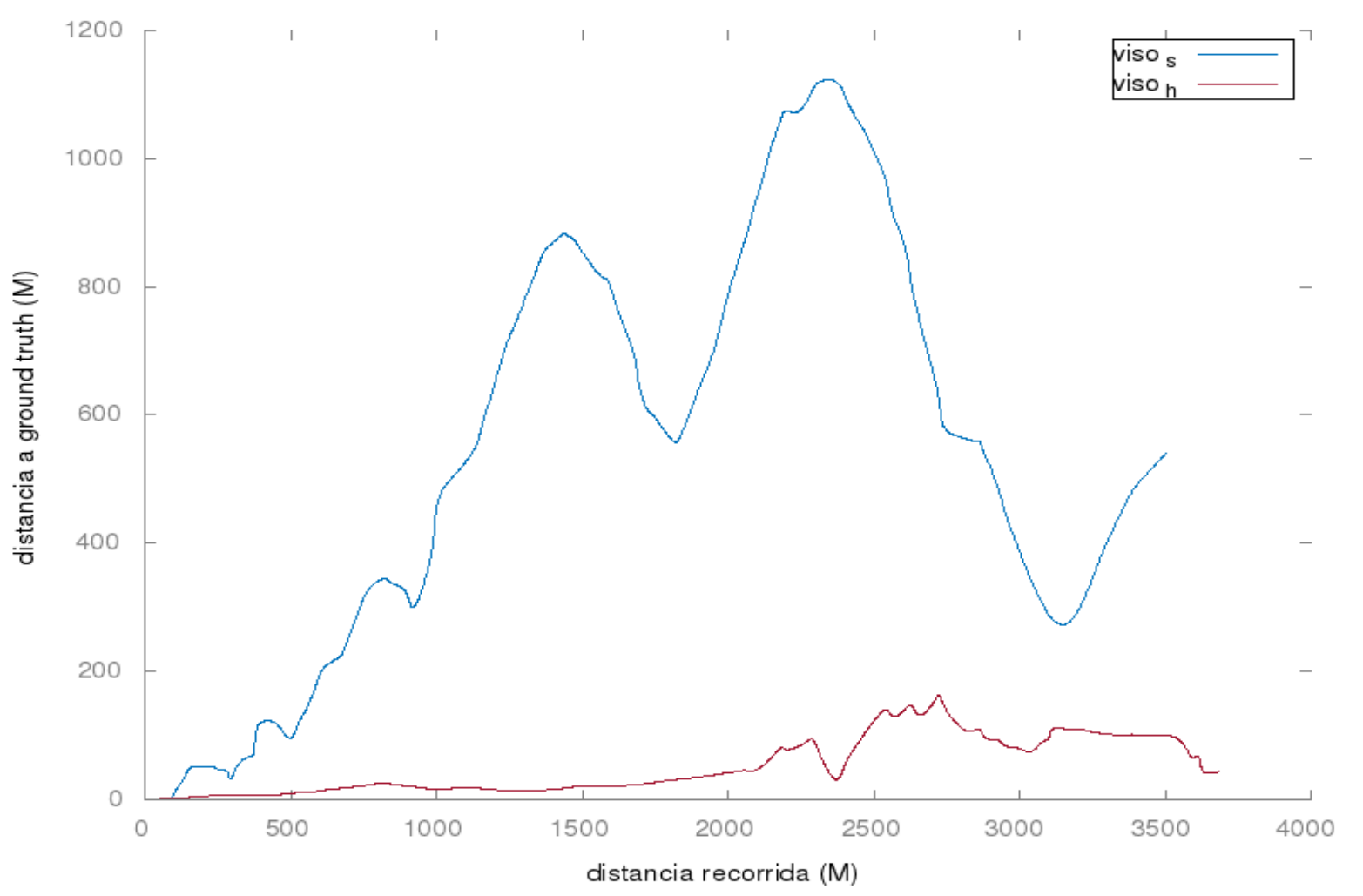
Resultados: Experimento 1

- Simulación de pérdida de cuadros acorde al rendimiento:



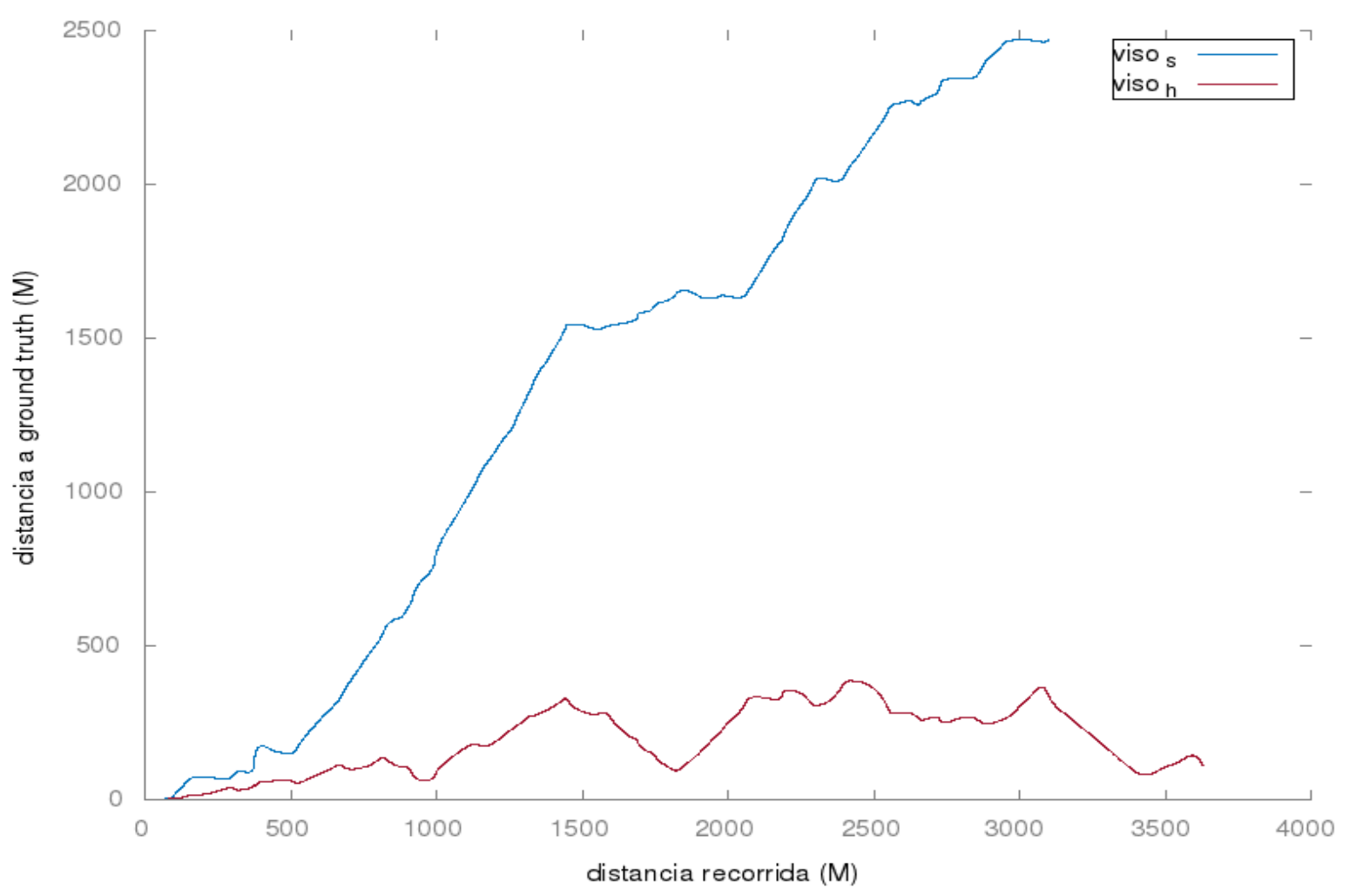
Resultados: Experimento 2

- Modificación de los parámetros para un rendimiento de 5 fps:



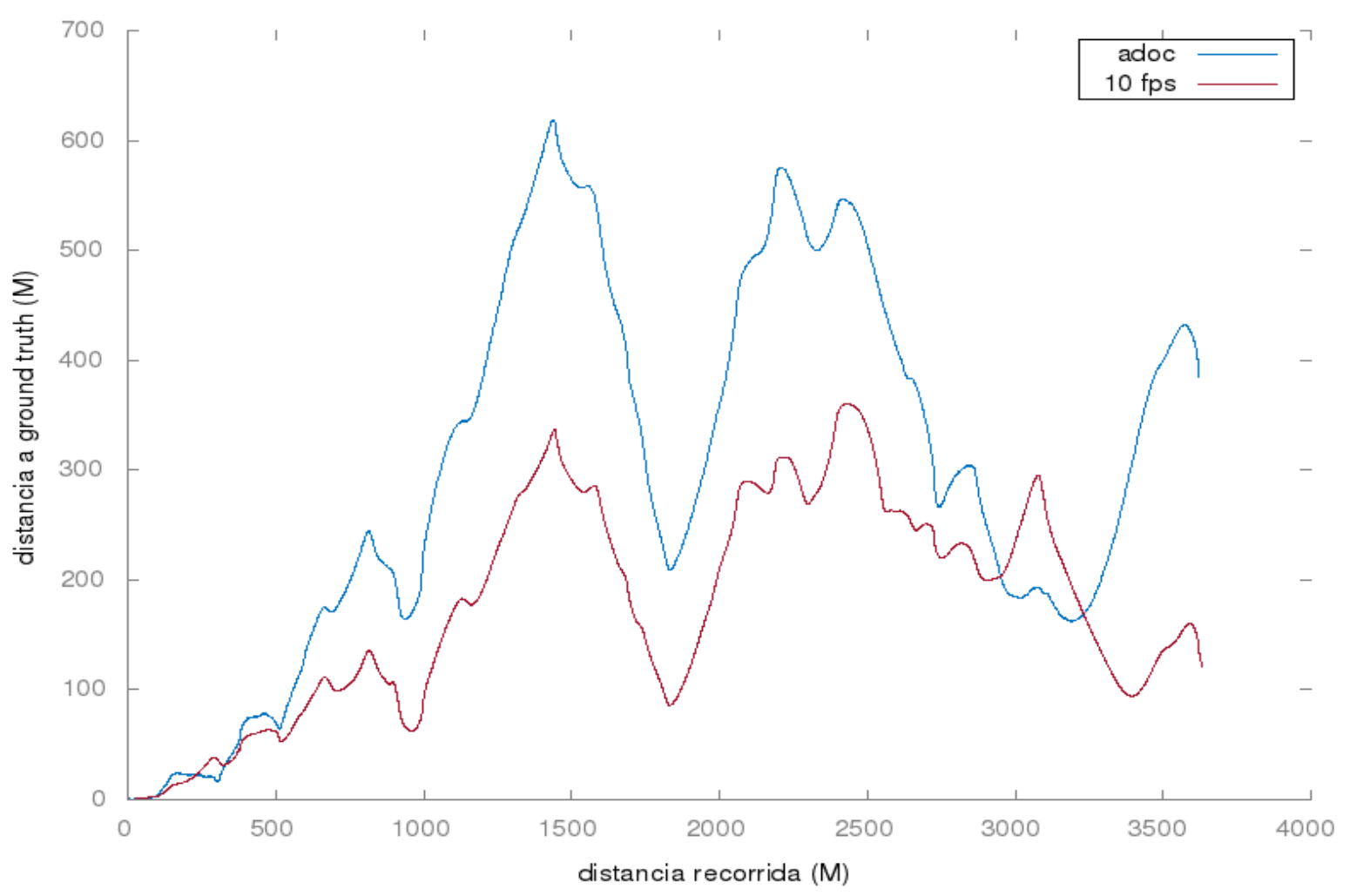
Resultados: Experimento 3

- Parámetros para un rendimiento de 10 fps (sin pérdida):



Resultados: Experimento 4

- Configuración “ad hoc” vs configuración para 10 fps:



Qué es el rendimiento ?

- Hay que considerar costo, necesidades y contexto!

Qué es el rendimiento ?

	u$d	kg	watts	cm3	FPS
Zybo	189	0,250	2	204	7,42
Ultrabook	880	1,8	32	1890	20

Conclusiones

- Metodología
- Estudio de los parámetros y pérdida de FPS
- Grupos de componentes
- HLS
  - Caracterización del patrón
  - Estudio de las optimizaciones de forma general
  - Las optimizaciones propuestas
  - Herramientas
- Resultados del caso de estudio