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

Alumno: Miguel Ángel García < *miguel.garcia@gmail.com* >
Directora: Patricia Miriam Borensztejn < *patricia@dc.uba.ar* >

Departamento de Computación, Facultad de Ciencias Exactas y Naurales,
Universidad de Buenos Aires

Resumen En este trabajo estudiamos técnicas y una metodología para la construcción de sistemas que utilizan visión artificial sobre All Programmable Systems on a Chip (AP SoCs). Los AP SoCs incorporan lógica programable, permitiendo implementar parte de la solución en hardware. El principal aporte de este trabajo son técnicas de optimización para la construcción de hardware eficiente utilizando síntesis de alto nivel.

Keywords: All Programmable System on a Chip, High Level Synthesis, Programmable Logic, Artifical Vision

1. Introducción

Este documento resume el trabajo llevado a cabo en mi tesis de grado de la carrera Ciencias de la Computación[1], en la que analizamos técnicas, optimizaciones y una metodología para la construcción de sistemas embebidos que hacen uso de visión artificial, utilizando **co-diseño hardware-software**.

Los sistemas embebidos móviles se encuentran por lo general sujetos a restricciones de tamaño y consumo eléctrico. Esto produce un conflicto de requerimientos cuando es necesario procesar grandes volúmenes de datos para responder a **eventos en tiempo real**, pues el uso de procesadores de alta velocidad se ve impedido por las limitaciones de consumo y área.

Una forma de hacer frente a este problema es mediante el uso de **All Programmable Systems On A Chip** (AP SoCs), circuitos integrados que contienen procesadores, memoria y lógica programable. Esta tecnología hace posible la construcción de sistemas utilizando **co-diseño hardware-software**, donde ciertas tareas son delegadas por el procesador a Intelectual Property Cores (**IP Cores**), componentes hardware reutilizables diseñados para tareas especificas.

En este trabajo realizamos el análisis de la construcción de soluciones utilizando **AP SoCs**, combinando software y **síntesis de alto nivel** (HLS) para

[1] La tesis completa puede descargarse desde https://github.com/miguelgarcia/tesis-apsoc/raw/master/tesis.pdf

crear sistemas utilizando **co-diseño hardware/software**, donde ciertas tareas son delegadas por la CPU a aceleradores hardware.

El foco del trabajo está puesto en el estudio de una metodología que abarca el ciclo completo de desarrollo y la exploración de optimizaciones y técnicas de programación necesarias para obtener buenos resultados al utilizar HLS.

Decidimos usar como guía del trabajo un caso de estudio: la implementación de una solución de **Odometría Visual Estéreo**. La misma consiste en utilizar la información de dos cámaras montadas en un móvil para conocer su posición, sin el uso de otros sensores, como GPS.

Las soluciones de odometría visual son intensivas en procesamiento y requieren para su correcto funcionamiento procesar video de dos cámaras en tiempo real, es decir, sin perder cuadros. La solución que desarrollamos delega el procesamiento intensivo de video a IP Cores implementados utilizando HLS, para lograr un buen rendimiento.

Los resultados obtenidos en este trabajo resultan aplicables en proyectos que, mediante el uso de la tecnología y el uso de prácticas de ingeniería de software y ciencias de la computación, pueden representar una innovación en el negocio de los sistemas embebidos que emplean visión artificial.

En particular el sistema construido pretende ser un punto de partida para futuros proyectos, facilitando la implementación de prototipos para el mercado de los sistemas embebidos mediante practicas de ingeniería de co-diseño hardware-software.

2. Conceptos, Herramientas y Tecnologías

Los **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.

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

La sección **PS** provee la funcionalidad equivalente a la de otros **SoCs** utilizados en muchas **Single Board Computers**, como **Raspberry Pi**, implementando un sistema de cómputo completo.

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

Para la construcción de **IP Cores** empleamos Vivado HLS, una herramienta de **Síntesis de Alto Nivel** que permite especificar circuitos digitales a través de lenguajes de alto nivel, en particular C y C++.

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.

3. Caso de Estudio

El caso de estudio en el que centramos nuestro análisis es el desarrollo de una solución de Odometría Visual Estéreo sobre una placa de desarrollo Zybo [1] de Digilent. La misma cuenta con un **AP SoC** *Zynq-7000*, de *Xilinx*. Utilizamos la plataforma de desarrollo *Vivado*[2].

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 utilizamos como punto de partida la implementación de código abierto *LIBVISO2*[3], una solución de odometría visual estéreo para **PC**.

En este trabajo utilizamos el **dataset KITTI para Odometría Visual** [4] para llevar a cabo las pruebas y mediciones. El mismo esta formado por secuencias de imágenes en escala de grises capturadas desde un automóvil mediante dos cámaras de 1.4 Megapíxeles montadas en el mismo. Ambas cámaras se encuentran sincronizadas y toman 10 imágenes por segundo.

El dataset proporciona los datos de posición reales de cada recorrido realizado, haciendo posible compararlos con la trayectoria obtenida utilizando odometría visual.

3.1. Metodología de trabajo

El proceso de desarrollo fue guiado siguiendo 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.

Como dijimos anteriormente, uno de nuestros objetivos era evaluar una metodología que integre el ciclo completo de desarrollo.

4. Implementación en computadora de propósito general

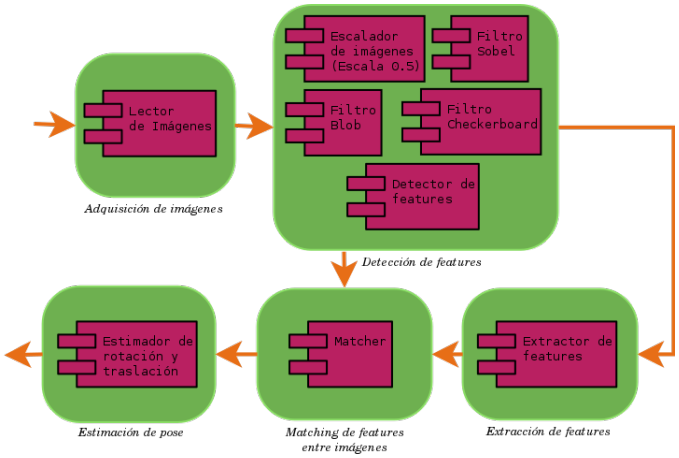
Siguiendo la metodología, anteriormente presentada, los primeros pasos del trabajo consistieron en "A. Diseño" y "B. Implementación y testing en procesador de propósito general".

Si bien partimos de un software existente, utilizamos la etapa de diseño para descomponerlo en componentes. Esta descomposición sirvió como guía en las etapas de implementación inicial en el **AP SoC** y optimización.

En esta etapa construimos LIBVISO en una PC de escritorio con procesador Intel Core i5, 6GB de RAM y sistema operativo Debian GNU/Linux "jessie".

IV

Figura 1. Componentes agrupados por tipo de tarea. Las flechas naranjas representan flujo de datos.



Además, creamos un programa, `run_test_case`, para facilitar el testing de la solución. 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.

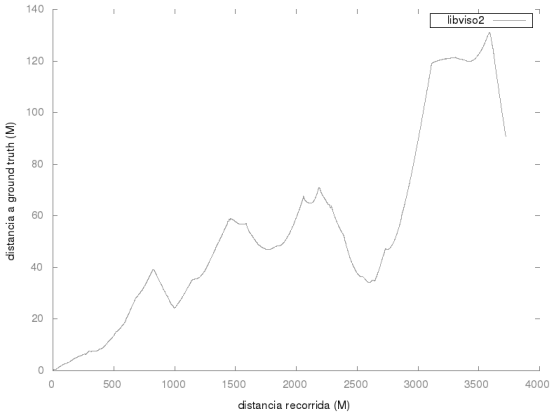


Figura 2. Gráfico de distancia entre la trayectoria estimada y la real, generado por `run_test_case` para la secuencia 00 del dataset KITTI.

Llamamos `libviso_gp` a la solución construida en la computadora de propósito general. La misma sirvió como referencia y punto de partida para las siguientes etapas del desarrollo.

También utilizamos la solución construida en esta etapa para medir el impacto de la pérdida de cuadros. Notamos que la calidad del mismo se ve sumamente afectada si se pierden la mitad o más de los cuadros.

5. Implementación inicial en el AP SoC

Luego de implementar la solución en una computadora de propósito general, procedimos a migrar la solución al **AP SoC**. Esta primera implementación fue realizada totalmente en software, haciendo uso solo del **PS** y sin usar lógica programable.

Tal como indica la metodología, configuramos la plataforma de hardware para que fuera 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

5.1. Plataforma 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.

Decidimos utilizar *PetaLinux*, una distribución de Linux diseñada para ser utilizada en productos de Xilinx, 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.
- Amplia documentación disponible.

5.2. Migración de libviso_gp al APSoC

Utilizando el SDK de PetaLinux creamos una nueva aplicación, llamada `viso`, usando el código fuente de `libviso_gp` como punto de partida.

Fue necesario reemplazar el uso de instrucciones *Streaming SIMD Extensions* (**SSE**), no disponibles en el procesador *ARM Cortex-A9* del **PS**, por instrucciones *Neon* equivalentes. Neon es un juego de instrucciones SIMD de propósito general para procesadores ARM.

Otro cambio realizado fue eliminar la dependencia de *libpng* para la carga de imágenes. 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**.

Al igual que para las pruebas en la computadora de propósito general, creamos un programa para facilitar esta tarea. Este programa, `run_test_case_remote`,

corre en la PC y utiliza NFS junto con SSH para disparar la solución en la placa, obtener el resultado y graficarlo.

Una vez verificado el funcionamiento de la solución en *Zybo*, medimos su rendimiento. Para descartar los costos propios de la transferencia de datos vía Ethernet, copiamos una secuencia de imágenes corta (100 cuadros) en el disco RAM, con una capacidad de 256MB.

De esta forma obtuvimos para `viso` un rendimiento de 2,55 cuadros por segundo (**fps**). Este rendimiento está lejos de la tasa de cuadros por segundo a la que fue capturado el dataset, 10 fps. Como vimos anteriormente, al perder más de la mitad de los cuadros la trayectoria estimada se aleja de la real rápidamente.

6. Aplicación de optimizaciones Software

Dado que el rendimiento de la solución construida, `viso`, resultó menor al requerido, y antes de buscar mejorarlo delegando tareas al hardware, aplicamos optimizaciones al software.

El primer paso fue realizar un análisis detallado del rendimiento de la solución. Medimos el porcentaje del tiempo total de procesamiento utilizado por cada componente y utilizamos la información obtenida para guiar el proceso de optimización.

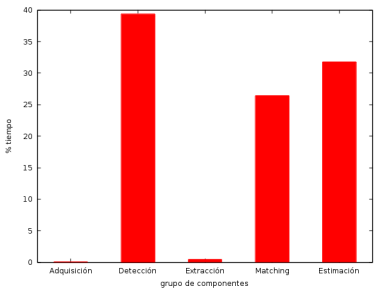


Figura 3. Porcentaje tiempos de procesamiento por grupo de componentes.

A partir de la información obtenida, 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".

Dado que el **AP SoC** cuenta con un procesador *ARM Cortex-A9* de doble núcleo, para cada grupo de componentes realizamos un estudio de sus posibilidades de paralelización, empleando para esto el diagrama de actividad. Luego, modificamos el código fuente para explotar el paralelismo utilizando **threads**.

Llamamos `viso_s` a la solución obtenida luego de aplicar optimizaciones por software.

Al igual que para `viso`, medimos el rendimiento de la solución optimizada. De esta forma obtuvimos el siguiente resultado:

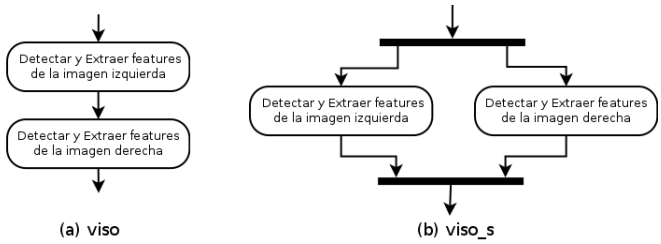


Figura 4. Diagrama de actividad del proceso de detección y extracción de ***features*** en `viso` y `viso_s`.

solución	$\frac{cuadros}{segundo}$
viso	2,55
viso_s	4,23

Cuadro 1. Comparativa de rendimiento entre `viso` y `viso_s`.

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

7. Optimización mediante aceleración por hardware

Esta sección describe el principal aporte de nuestro trabajo, en ella caracterizamos un patrón de diseño aplicable a **IP Cores** para procesamiento de imágenes y estudiamos optimizaciones aplicables al mismo.

En primer lugar, al igual que en la sección anterior, construimos un gráfico agrupando los porcentajes de tiempo de procesamiento real por grupo de componentes al que pertenece cada función:

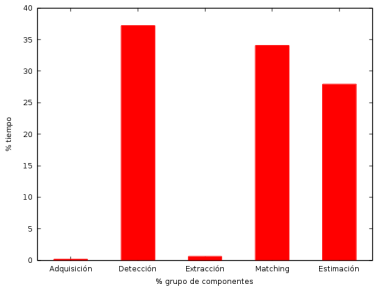


Figura 5. Porcentaje tiempos de procesamiento real por grupo de componentes en `viso_s`.

La información obtenida nos permitió evaluar qué tareas era conveniente delegar al hardware. Decidimos delegar la detección de ***features*** y el cálculo de

los filtros *Sobel* utilizados en la extracción de descriptores. Es decir, **delegamos al hardware las tareas que requieren el procesamiento de cada uno de los píxeles de las imágenes**.

7.1. Vivado HLS

Como dijimos *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.

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**, **Block RAM**, memoria distribuida en **Look Up Tables** (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.
- **Comunicación entre componentes:** Al estar creando circuitos las conexiones son señales, por lo tanto el diseñador debe decidir qué señales usar para comunicar distintos componentes, 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, pues no pueden basarse en el comportamiento de una máquina ya existente para tomar decisiones. Además, deben considerarse tres factores críticos: velocidad, área ocupada 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.

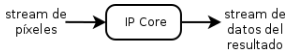
7.2. Patrón de diseño para procesamiento secuencial de píxeles

A partir de un artículo de Vallina [6], que propone estructuras de memoria para trabajo con imágenes al usar HLS, caracterizamos un patrón de diseño

aplicable al desarrollo de IP Cores que realizan un **procesamiento secuencial de los píxeles de una imagen**.

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 6. IP Core con ***streams*** como entrada y salida.



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**x**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 a continuación ejemplifica la aplicación de este patrón para procesar todos los píxeles de una imagen con una ventana de **N**x**M** píxeles como contexto.

Algoritmo 1 Procesamiento de imagen utilizando line buffer

```
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
```

X

7.3. Optimizaciones aplicables al patrón de diseño

Antes de aplicar el patrón de diseño en la construcción de los **IP Cores** requeridos para acelerar el caso de estudio, analizamos de forma genérica optimizaciones aplicables al patrón en Vivado HLS para incrementar el rendimiento y reducir el consumo de recursos.

En primer lugar el diseñador debe tomar decisiones sobre la alocación de recursos, seleccionando el tipo de memoria a utilizar para cada buffer. En este sentido el diseñador construye una jerarquía de memoria.

Vallina [6] propone guiar esta decisión estudiando la cantidad de accesos simultáneos a cada buffer necesarios para minimizar la cantidad total de ciclos de procesamiento. Por ese motivo propone utilizar un **Block RAM** independiente para cada línea del **Line Buffer** y registros independientes (en **LUTs**) para cada celda de **Window**.

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 con la lógica necesaria para su manejo. Esta biblioteca ya incluye las directivas de Vivado HLS [7] necesarias para implementar los buffers sobre los tipos de memoria antes mencionados.

Las **directivas** son opciones aplicables a elementos del código que Vivado HLS tiene en cuenta a la hora de sintetizarlo.

A continuación analizamos el impacto de la directiva **PIPELINE**. La misma indica a Vivado HLS que intente paralelizar todas las operaciones de un bucle sin esperar a que termine una iteración para comenzar la siguiente. Aplicando esta directiva logramos IP Cores con una productividad de $\sim 1 \frac{pixeles}{ciclo}$.[2]

Luego estudiamos optimizaciones que no pueden lograrse solo con directivas de HLS, sino que requieren modificaciones al algoritmo.

Optimización "K píxeles por ciclo": Analizando el algoritmo anteriormente presentado y el código fuente de su implementación en Vivado HLS, 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 de reloj para leer o escribir un dato en ellos.

Por lo tanto probamos una optimización que consiste en procesar en cada ciclo **K** píxeles, utilizando para esto **streams** más anchos. Para lograrlo aplicamos las siguientes modificaciones al algoritmo:

- Multiplicamos el ancho de los **streams** de entrada y salida por **K**.
- A nivel **Line Buffer** 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 Nx(M+K-1) píxeles.
- En el bucle de columnas realizar $\frac{ancho}{k}$ iteraciones, procesando en cada una **K** píxeles.

[2] El código fuente de esta implementación puede consultarse en el apéndice

Aplicando esta optimización, en combinación con la directiva **PIPELINE**, obtuvimos una productividad de $\sim \frac{K\ pixeles}{ciclo}$. K veces más rápido. El valor máximo de **K** está acotado por el ancho de los buffers de donde se lee la imagen, en el caso de la memoria externa en Zybo, el valor máximo es 8.

En la sección de apéndices se puede consultar el código fuente de esta implementación.

Optimización "Fusión de IP Cores": Está optimización es aplicable cuando en un diseño existen dos o más IP Cores que procesan una misma imagen siguiendo el patrón antes descrito.

Consiste en combinar la funcionalidad de todos los IP Cores en uno solo, compartiendo **Line Buffers**, **Window**, **stream** de entrada y parte de la lógica de control. De esta forma se ahorran recursos sin perder rendimiento.

En nuestras pruebas el consumo de recursos obtenido al combinar dos IP Cores fue $\sim 60\,\%$ que si hubiéramos implementado los dos IP Cores de manera independiente.

En la sección de apéndices se puede consultar el código fuente de esta implementación.

7.4. Aplicación de optimizaciones al caso de estudio

Luego de estudiadas las optimizaciones de manera genérica, las aplicamos al caso de estudio para mejorar su rendimiento.

El primer paso fue realizar la partición hardware/software de la funcionalidad, siguiente el paso "C. Partición Hardware/Software" de la metodología.

Decidimos implementar en hardware las secciones de LIBVISO2 que hacen un procesamiento intensivo de las imágenes, estas son:

- Filtro Sobel
- Filtro Blob
- Filtro Checkerboard
- Detector de features, utilizando **non-maximum supression**
- Escalador de imágenes

Como parte del paso "D. Implementación de hardware, testing e integración." aplicando el patrón de diseño y optimizaciones antes descritas construimos los IP Cores:

- Sobel16: utilizamos la optimización **K** píxeles por ciclo para procesar 2 píxeles en cada iteración. Además, aplicamos la fusión de IP Cores, pues se calculan simultáneamente el filtro Sobel vertical y horizontal de una misma imagen.
- Blob+Checkerboard: aplicando la fusión de IP Cores, construimos un IP Core que calcula los filtros Blob y Checkerboard de una imagen.
- Fetures: este IP Core aplica **non-maximum supression** a los filtros Blob y Checkerboard, en simultáneo, para detectar puntos sobresalientes.

XII

- HalfResolution16: Este IP Core procesa 2 píxeles por ciclo para obtener una copia de la imagen de la mitad del tamaño.

La decisión sobre las optimizaciones a aplicar obedeció a que los IP Cores forman un **pipeline de procesamiento**, donde para evitar cuellos de botella eran necesarios los siguientes rendimientos:

IP Core	velocidad de consumo	productividad
Sobel16	$\sim 2\frac{pixel}{ciclo}$	$\sim 2\frac{pixel}{ciclo}$
HalfResolution16	$\sim 2\frac{pixel}{ciclo}$	$\sim 0{,}5\frac{pixel}{ciclo}$
Blob + Checkerboard	$\sim 0{,}5\frac{pixel}{ciclo}$	$\sim 0{,}5\frac{pixel}{ciclo}$
Features	$\sim 0{,}5\frac{pixel}{ciclo}$	$\sim 1\frac{feature}{ciclo}$

Cuadro 2. Velocidades de consumo de datos y productividades mínimas de los IP Cores para evitar bloquear el ***pipeline***

7.5. Integración de IP Cores a la solución

Una vez implementados los IP Cores y probados de manera independiente, los integramos a la solución, modificando la plataforma de hardware como muestra la imagen a continuación:

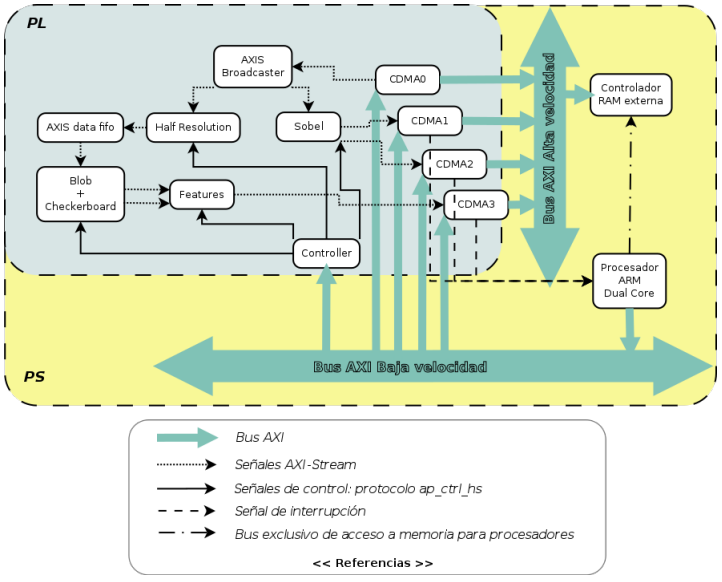


Figura 7. Plataforma de hardware

En el proceso, creamos el IP Core **Controller**, encargado de orquestar a los IP Cores del pipeline de procesamiento de imágenes y único nexo con las ordenes de los procesadores.

A continuación desarrollamos el controlador del dispositivo, un módulo del kernel que hace posible a las aplicaciones de usuario aprovechar las nuevas características de la plataforma hardware construida. El controlador proporciona dos servicios a las aplicaciones:

- Mapeo de espacios de memoria físicamente contiguos. Esto es necesario pues se utiliza DMA para alimentar al pipeline de procesamiento de imágenes y para guardar sus resultados en memoria. Utilizamos mmap para mapear los buffers en el espacio de memoria de los procesos de usuario.
- Procesamiento de imágenes utilizando el pipeline de IP Cores implementado en PL. Se utiliza IOCTL para la comunicación entre el controlador y los procesos de usuario.

7.6. Nueva aplicación delegando parte del trabajo al hardware

Partiendo del código fuente de `viso_s` creamos una nueva aplicación `viso_h` que utiliza el nuevo hardware, a través del controlador, para realizar la detección de ***features*** y el calculo de los filtros *Sobel*.

Al igual que para `viso` y `viso_s` medimos el rendimiento, obteniendo los siguientes resultados:

solución	$\frac{cuadros}{segundo}$
viso	2,55
viso_s	4,23
viso_h	7,42

Cuadro 3. Comparativa de rendimiento entre `viso`, `viso_s` y `viso_h`.

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`.

8. Pruebas y Resultados

Llevamos a cabo pruebas comparativas entre las soluciones construidas: `viso`, `viso_s` y `viso_h`.

El primer experimento consistió en procesar las secuencias de imágenes del dataset con las tres soluciones construidas, simulando en cada una la pérdida de cuadros acorde a su rendimiento. Para cada una medimos su distancia máxima a la trayectoria real y su distancia promedio.

Observamos que la media de la distancias promedio de `viso_s` a ***ground truth*** como porcentajes de las distancias promedio de la solución inicial, `viso`, es del 61 %. Análogamente para `viso_h` con respecto a `viso` este valor es 20,55 %.

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

Otro de los experimentos consistió en variar parámetros de configuración, entre ellos el tamaño de los rangos de búsqueda de coincidencias entre ***features*** y la cantidad de iteraciones a utilizar para hallar la traslación y rotación entre dos cuadros, para lograr rendimientos de 10 cuadros por segundo tanto para `viso_s` como para `viso_h`.

Observamos que con los nuevos parámetros `viso_h` otorga mejores resultados que con los anteriores, mientras que `viso_s` no. Esto se debe a que para lograr tasas de 10 cuadros por segundo en `viso_s` fue necesario relajar mucho los parámetros, mientras que en `viso_h` el mayor rendimiento permitió variar los parámetros para mejorar la velocidad sin impactar negativamente en el calculo de resultados.

Los resultados completos de la experimentación se puede hallar en el apéndice.

9. Conclusiones

En primer lugar, nos parece importante resaltar que la **metodología** elegida permitió llevar a cabo este trabajo con éxito. Contar con una metodología que integra todas las etapas del desarrollo resultó clave para ordenar y guiar el trabajo.

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 propuestas, **"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.

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

Creemos que hay varios elementos de este trabajo que pueden ser utilizados como base para distintos proyectos innovadores que pueden desembocar en emprendimientos tecnológicos redituables.

En primer lugar, la plataforma de hardware construida puede ser utilizada de forma directa en la construcción de sistemas embebidos que hagan uso de visión artificial o procesamiento de imágenes en tiempo real, por ejemplo en los campos de: realidad aumentada, video vigilancia y asistencia en la conducción de vehículos.

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.

En este punto es importante resaltar que la plataforma construida permite un buen rendimiento con costos y consumos menores que la implementación en una computadora de propósito general.

	u$d	kg	watts	cm3	fps
Zybo	189	0,250	2	204	7.42
Ultrabook	880	1,8	32	1890	20

Cuadro 4. Comparativa entre la implementación en Zybo y Ultrabook i5 para la misma configuración.

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 patrón de diseño para el desarrollo de **IP Cores** para procesamiento de imágenes utilizando **HLS**.
- Optimizaciones aplicables a **IP Cores** para procesamiento de imágenes.

Referencias

1. Digilent Inc: ZYBO Reference Manual (2014)
2. Xilinx: Vivado Design Suite User Guide. (2014)
3. Andreas Geiger and Julius Ziegler and Christoph Stiller: StereoScan: Dense 3D Reconstruction in Real-time. Intelligent Vehicles Symposium (IV) (2011)
4. Andreas Geiger and Philip Lenz and Raquel Urtasun: Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. Conference on Computer Vision and Pattern Recognition (CVPR) (2012)
5. Sol Pedre and Tomás Krajník and Elías Todorovich and Patricia Borensztejn: A co-design methodology for processor-centric embedded systems with hardware acceleration using FPGA. 2012 VIII Southern Conference on Programmable Logic (SPL)
6. Fernando Martinez Vallina: Implementing Memory Structures for Video Processing in the Vivado HLS Tool. XAPP793 (2012)
7. Xilinx: Vivado Design Suite User Guide: High-Level Synthesis, Vivado Design Suite User Guide. (2014)