Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

Trabajo Fin de Grado

Aceleración de algoritmos implementados sobre SoCs

Autor: Laura Montalvo Rodrigo

Tutor/es: Ignacio Bravo Muñoz

2020

Laura Montalvo Rodrigo

UNIVERSIDAD DE ALCALÁ DE HENARES

Escuela Politécnica Superior

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE LA TELECOMUNICACIÓN

Trabajo de Fin de Grado

Aceleración de algoritmos implementados sobre SoCs

Autor: Laura Montalvo Rodrigo

Tutor: Ignacio Bravo Muñoz

TRIBUNAL:

Presidente: Carlos Julián Martín Arguedas

Primer vocal: Esther Palomar González

Segundo vocal: Ignacio Bravo Muñoz

Fecha: 23 de julio de 2020

Índice:

1. Índice de figuras

2. Resumen breve.

El presente Trabajo Fin de Grado (TFG) realiza la implementación del algoritmo PCA (Principal Component Analysis) aplicado a procesamiento de imágenes, sobre una plataforma SoC (System on Chip) Zybo Z720 de Xilinx. El objetivo es el uso de la herramienta SdSoC que permite implementar todo el sistema que irá a bordo del SoC a través de una única fuente codificada en lenguaje C/C++. Con ello se pretende obtener una eficiente tasa de procesado de imágenes aplicando diferentes directivas de la herramienta tanto a la parte que irá embarcada sobre el microcontrolador del SoC como en su parte FPGA Para ello se realizará un análisis de distintas directivas que proporciona el software, SDx y HLS, con la intención de aprovechar la concurrencia y paralelismo que proporciona el hardware de un SoC. Tras encontrar el método que optimice esta tasa y la acelere lo máximo posible, se trasladará el algoritmo a un sistema de tiempo real que procese las imágenes según se capturen por una cámara conectada a la plataforma.

Palabras clave: PCA, SoC, tasa de procesado, tiempo real, SDx, HLS.

Summary:

This Final Degree Project implements the PCA (Principal Component Analysis) algorithm applied to image processing, on a Xilinx Zybo Z720 SoC (System on Chip) platform. The objective is the use of the SdSoC tool that allows to implement the entire system that will be on board the SoC through a single source encoded in C / C ++ language. The aim is to obtain an efficient image processing rate by applying different directives of the tool both to the part that will be shipped on the SoC microcontroller and to its hardware part. For this, an analysis of different directives provided by the software, SDx and HLS, will be carried out, with the intention of taking advantage of the concurrency and parallelism provided by the hardware of a SoC. After finding the method that optimizes this rate and accelerates it as much as possible, the algorithm will be transferred to a real-time system that processes the images as they are captured by a camera connected to the platform.

Key words: PCA, SoC, image processing rate, real-time system, SDx, HLS.

3. Glosario de Abreviaturas.

- BRAM – Block RAM
- CPU – Unidad Central de Procesamiento
- DRM – Direct Rendering Manager – Gestor de Renderizado Directo
- DSP – Digital Signal Procesor
- FF – Flip Flops
- FPGA – Field Programmable Gate Array
- HDL – Hardware Description Language – Lenguaje de Descripción Hardware
- HLS – High Level Synthesis
- HW – Hardware
- LUT – Look Up Table
- Matriz A – matriz que representa el valor de las imágenes de fondo con media nula
- Matriz U – matriz denominada matriz de transformación que representa al espacio transformado de dimensionalidad reducida.
- MM – Multiplicación de matrices
- PCA – Principal Component Analysis
- RMSE – Raíz del error cuadrático medio
- RTL – Register Transfer Level
- SDx – Software Design Enviroment – Entorno de Diseño Software
- SDK – Software Development Kit
- SoC – System on Chip – Sistema en un Chip
- SO – Sistema Operativo
- SW – Software
- V4L2 – Video for Linux

4. Memoria del proyecto.

Tras el breve resumen inicial sobre los temas a tratar en este proyecto, a continuación, se va a realizar una puesta en contexto al lector sobre los motivos que han llevado al desarrollo de estos aspectos.

4.1 Introducción:

La pregunta inicial que se debe responder para poder entender este trabajo es: ¿Qué es un SoC? ¿Qué beneficios puede aportar en el diseño de un sistema electrónico?

Un SoC ("*System on a Chip", Sistema en chip*) es un circuito integrado que integra los componentes fundamentales de un sistema electrónico digital en un único circuito, lo que se denomina "*chip"*. Es decir, es un circuito que integra tanto la arquitectura del microprocesador como la arquitectura del FPGA en una única pieza de silicio.

Hasta la creación de esta tecnología, microprocesadores y FPGA's se trataban separadamente, diseñando cada una de manera distinta y en distintos chips, lo que añadía complejidad y coste.

Los microprocesadores se encargaban de dotar la inteligencia de determinado sistema, encargándose de la ejecución de los programas desde un software principal y de la gestión de las aplicaciones de usuario (si este consta de un sistema operativo), mientras que las FPGAs ("*Field Programmable Gate Array"*) implementaban un sistema digital empleando un lenguaje específico de descripción hardware cuyas características internas permitían la reconfiguración de estos.

La introducción de esta tecnología resulta novedosa puesto que, a diferencia de la arquitectura de placa base de un computador, la cual aloja y conecta componentes desmontables, un SoC integra todos estos componentes en una única pieza. Esto permite la implementación de distintas aplicaciones cuya ejecución en arquitecturas convencionales (basadas en microprocesadores) no alcanzan las expectativas de velocidad, y la reducción de tiempos de diseño en aquellas aplicaciones implementadas íntegramente en FPGAs.

Además, los *System on Chip* presentan ciertas ventajas que hacen que el desarrollo de aplicaciones basadas en este componente sea interesante:

- Requiere poco espacio, puesto que los componentes están internamente conectados.
- Mayor rendimiento gracias a la cantidad de circuitos dentro del chip.
- Requiere menores cantidades de energía, es decir, menos consumo.
- Menor coste.

Algunas de las aplicaciones para estos sistemas son el procesado de señales digitales, los teléfonos móviles, los decodificadores de vídeo y los sistemas embebidos, donde estos últimos son sistemas diseñados para realizar funciones dedicadas frecuentemente en sistemas de computación en tiempo real.

Esta tecnología ha dado lugar a una nueva metodología de diseño, denominada codiseño HW/SW. Cuando microprocesadores y FPGAs se tratan separadamente, cada una dispone de su metodología de diseño y de su lenguaje de programación asociado. Los microprocesadores permiten un mayor nivel de abstracción al usuario gracias a la codificación en lenguajes de alto nivel como C++ o Java mientras que los FPGAs, a pesar de mostrar mayor grado de abstracción a nivel de registros, se codifican en lenguaje de descripción hardware (codificación más compleja al incluir la noción del tiempo).

Esta nueva metodología de diseño surge gracias al desarrollo de ciertas herramientas que permiten la programación de un SoC en lenguajes de alto nivel, realizando las transformaciones a HDL para la parte de FPGA's internamente. Esto permite que un usuario que normalmente se dedique al desarrollo de aplicaciones software pueda acelerar la velocidad de sus diseños gracias a la lógica programable de las FPGA's abstrayéndole del verdadero funcionamiento de estas (sin necesidad de conocer las herramientas de diseño de estilo ASIC) y de la codificación en HDL ("*Hardware Description Language"*), mientras que permite a un desarrollador de aplicaciones HW disminuir el tiempo de diseño, disminuyendo además el "*Time to Market"*.

La codificación en lenguajes de alto nivel resulta beneficiosa en muchos sentidos, ya que son lenguajes de codificación que se asimilan más al lenguaje hablado, siendo más simples de entender. Toda esta simplificación y abstracción que proporcionan resultan en menor tiempo de diseño lo cual resulta beneficioso a cualquier diseñador cuya finalidad es ser el primero en poner su diseño en el mercado.

Además, la codificación en lenguaje alto nivel presenta la ventaja del uso de librerías de código abierto que pueden facilitar la implementación del algoritmo, como lo pueden ser las librerías OpenCV, librería de visión, u otras librerías que eviten la implementación de ciertas operaciones como "*Eigen*" para el cálculo matemático.

Son varias las grandes empresas que han lanzado esta tecnología al mercado, entre ellas Intel, NDIVIA y Xilinx.

Cada una proporciona sistemas con distintas características, tanto en la parte de lógica programable (FPGA's) donde Xilinx emplea su gama "*7 series*" mientras que Intel emplea las FPGA de Altera *"Cyclone V"* como para el procesador donde ARM destaca con sus distintos microcontroladores.

Una de las aplicaciones, que se ha omitido nombrar antes, de los *System on Chip* es el procesado de imágenes. El procesado de imágenes surge para adecuar las imágenes para el análisis de estas con intención de extraer cierta información para una aplicación específica. Esto deriva en que el procesado varía fuertemente en función del problema a abordar. Este procesado puede ir desde la mejora y restauración de una imagen hasta la descripción y segmentación de los objetos que la componen, siendo estos últimos interesantes para aplicaciones de visión artificial.

El hecho de desarrollar estas aplicaciones en un SoC en lugar de en una arquitectura basada en un microprocesador es que el procesado de imágenes en HW proporciona ciertos beneficios. Muchas veces al implementar estas aplicaciones en un microcontrolador, el tiempo que tarda en procesar las imágenes no cumple con los requerimientos de las aplicaciones en tiempo real, proporcionando una tasa ("*fps, frames por segundo"*) muy baja.

El procesado de imágenes en HW proporciona una paralelización en la ejecución de los algoritmos que requieren de un gran número de operaciones, de manera que acelera estas aplicaciones permitiendo alcanzar los objetivos de aplicaciones en tiempo real, como lo es la detección de un objeto dentro de una imagen cuando este aparece, no con cierto retraso.

Además, otros beneficios que proporciona el procesado de imágenes en hardware son [1]:

- La implementación de arquitecturas específicas para cada tipo de algoritmo.
- La capacidad de trabajo con flujos de datos elevados.
- Frecuencias de reloj más bajas que las usadas por DSP.
- No es necesario el almacenamiento de la información en memoria antes de su procesado, permitiendo un procesamiento "*On the Fly"*.
- Permite la creación de sistemas reconfigurables.

Para concluir, cabe destacar que esta tecnología, junto con la nueva metodología de diseño, ha hecho que ciertas empresas que ponen estos dispositivos en el mercado también pongan a disposición del usuario las herramientas con las que programar a gusto del consumidor.

El ejemplo de fabricante en el que se centra este proyecto es Xilinx, el cual pone en el mercado sus dispositivos SoC Zynq-7000 y las herramientas en función del nivel de abstracción deseado por el usuario: "*Vivado Design Suite"* junto con "*SDK*" para aquellos usuarios que quieran una baja abstracción, acercándose lo máximo posible a la programación del hardware, "*HLS: High Level Synthesis"* para aquellos usuarios que deseen una baja abstracción pero una aceleración de ciertas funciones gracias a la implementación de partes en más alto nivel, y *"SDx: Software Design Tools"* para los usuarios que deseen abstraerse "*completamente"* del lenguaje de descripción hardware y de la programación a nivel de registros, centrándose más en el desarrollo de la aplicación.

4.1.1 Objetivos de este proyecto:

La finalidad de este proyecto es lograr esa aceleración que se ha mencionado anteriormente que proporciona el uso de la lógica programable de las FPGAs en los SoC gracias a la paralelización en el cómputo para lograr desarrollar un sistema de detección de objetos que cumpla las expectativas del cómputo en tiempo real. Para ello se va a emplear la plataforma de desarrollo proporcionada por Digilent *Zybo Z7-20* la cual incorpora un SoC Zynq-7000 de Xilinx.

Más que en el desarrollo del propio algoritmo de detección de objetos "*PCA, Análisis de Componentes Principales"*, el cual se explicará más adelante, se va a enfocar este proyecto en la optimización de este gracias a las herramientas proporcionadas por Xilinx nombradas anteriormente.

Esta optimización del algoritmo nombrada en el párrafo anterior consiste en obtener una tasa de procesado de imágenes *(frames)* por segundo, tolerable para un sistema de tiempo real. Para poder realizar esta optimización, inicialmente se aplicará el algoritmo sobre un conjunto de vídeos grabados con un teléfono móvil. Estos se almacenarán en una tarjeta de memoria externa que se insertará en la plataforma. Finalmente, una vez obtenida la velocidad de procesado deseada, se trasladará el algoritmo a un sistema en tiempo real basado en la interconexión de la plataforma con un módulo cámara PCam 5C y un cable HDMI conectado a un monitor.

Para la comprobación del cumplimiento de los objetivos se dispondrán, a lo largo del proyecto, evidencias tanto visuales como numéricas obtenidas gracias a herramientas de perfilado que se nombrarán más adelante. El resultado que se desea alcanzar de procesado de las imágenes, para que sea válido para sistemas en tiempo real, es de aproximadamente 25-30 *fps* dado que esta tasa es considerada como buena en numerosas aplicaciones de visión artificial, además de estar establecida por los estándares de formato de video europeos (NTSC "*National Television Standard Comitee – 29 fps"* y PAL "*Phase Alternate Line – 25 fps")*.

4.4.2 Estructuración del trabajo:

A lo largo de este trabajo se van a encontrar distintas secciones dedicadas a la comprensión de los distintos aspectos que la implementación del diseño final requiere.

Tras el breve resumen de la introducción, en la que se presenta la finalidad del proyecto y los objetivos que se pretenden cumplir, se comienza lo que es la memoria con una breve puesta en

contexto, *Estado del Arte,* capítulo segundo de este libro. En este apartado se van a comentar ciertos trabajos previos de los cuales se ha recabado información tanto del uso de las herramientas que se emplean como de los conceptos teóricos que se aplican en la implementación del diseño.

En este apartado, además, se introduce lo que fue el trabajo previo a este proyecto y el cual se pretendía conseguir los futuros trabajos [6].

Tras la puesta en contexto, se dedicará un capítulo del libro a introducir los distintos conceptos teóricos que se abarcan. Puesto que este proyecto pretende optimizar y acelerar un algoritmo de procesado de imágenes concreto, en esta parte de *Fundamentos teóricos,* capítulo tercero, se explica la matemática detrás de él y el resultado de su aplicación.

Este capítulo no solo va a abarcar el algoritmo por optimizar, sino que también explicará las distintas herramientas tanto a nivel software como hardware mediante las cuales se va a poner el funcionamiento la implementación del diseño. Brevemente se introduce también teoría sobre el concepto de la coma fija, puesto que en el capítulo cuarto de *Optimización del Algoritmo PCA y resultados* se le hace referencia y se desea que el lector lo conozca.

El cuarto capítulo se divide en dos ramas de investigación, puesto que narra lo que es la implementación del diseño y la finalidad del proyecto. La primera rama, *Optimización del Algoritmo PCA y resultados*, explica los distintos métodos que se proponen para obtener la aceleración deseada del algoritmo PCA implementado en [6] y se muestran los resultados obtenidos para cada uno de los escenarios. Esta primera parte finaliza con una comparativa de todos los métodos y sus resultados, analizando cuáles de todos ellos serían los métodos válidos.

La segunda rama, *Traslación del Algoritmo PCA a tiempo real,* narra cómo se introduce este algoritmo en un sistema de tiempo real, que capta las imágenes por una cámara y las muestra procesadas en un monitor instantáneamente. La finalidad de la primera rama es obtener una tasa de procesado lo suficientemente elevada que permita que la traslación del algoritmo a este nuevo sistema no provoque una latencia excesiva entre la captura de la imagen y lo que se observa en la pantalla.

El quinto capítulo de este libro, *Pliego de condiciones,* narra los requisitos para la puesta en marcha de este sistema implementado en cualquier dispositivo además de la configuración de estos y de los materiales requeridos.

El sexto capítulo de *Presupuesto* estima cual es el coste de la realización del proyecto, y el séptimo, *Manual de usuario,* es una guía de cómo usar el software SDx para evitar que la recreación de este proyecto suponga al diseñador ciertos problemas que ha causado a lo largo de este proyecto.

El noveno capítulo muestra las *Conclusiones y trabajos futuros* de este proyecto realizado y el octavo capítulo, *Planos,* introduce al usuario en la parte más relacionada con la programación del algoritmo, explicando las distintas funciones que se han codificado para llevar el algoritmo al sistema de tiempo real. Se finaliza con un décimo capítulo que engloba la *Bibliograafía* empleada para la elaboración de este trabajo.

4.2 Estado del arte:

Gracias a la integración en un único circuito de silicio de la FPGA y varias CPU junto a los periféricos, los SoC'S brindan una gran versatilidad en lo relativo a las aplicaciones bajo estudio. Todas aquellas aplicaciones que requieran ver su tiempo de cómputo reducido son las más interesantes para desarrollar en un SoC, siendo candidatas las aplicaciones de visión artificial o procesado de imagen, entre otras.

Particularizando en estas aplicaciones candidatas, los trabajos de investigación realizados sobre la optimización de algoritmos que aceleren el procesado de las imágenes gracias a la tecnología SoC va en aumento, aunque algunos todavía prefieren implementar la aplicación específica basando su diseño únicamente en FPGAs. Alguno de los ejemplos de aplicaciones puede ser el control de vehículos con las cámaras de la DGT o bien el sistema de vigilancia de un centro comercial, donde se puede detectar a ciertas personas y realizar un seguimiento sobre ellas.

El estudio sobre la detección de caras implementado sobre FPGA [2] demuestra como el uso de técnicas codiseño hardware-software permite aplicar el desarrollo de aceleradores hardware sobre SoC's en aquellas partes del algoritmo que requieren mayor consumo computacional obteniendo tiempos de respuesta que hacen que el sistema sea adecuado en aplicaciones biométricas en tiempo real.

Mientras que algunos estudios como el nombrado en el párrafo anterior y el estudio realizado para la detección de distintos elementos [3] basan su metodología de codiseño HW/SW en un nivel de abstracción muy próximo al hardware, realizando la implementación de los propios módulos IP que se van a emplear, otros estudios empiezan a desarrollarse en la línea del codiseño en un nivel de abstracción mucho mayor.

Esto surge gracias a las herramientas de desarrollo que, como proporciona Xilinx con SDSoC, permiten programar el SoC en lenguajes de alto nivel. Uno de los trabajos que demuestran la eficiencia de esta herramienta es [4] donde se resalta que la codificación en este alto nivel resulta peor en términos de planificación pero sin embargo ahorra tiempo de trabajo y diseño. Otros, sin embargo, exponen los beneficios que proporciona la técnica de reconfiguración dinámica parcial en los SoC mediante ciertos test usando esta herramienta de alto nivel [5].

Lo novedosa que es esta herramienta influye en que haya poca información sobre cómo usarla, siendo una buena guía para el aprendizaje el trabajo [6], el cual no solo sirvió como manual sino que ha sido la inspiración de este trabajo. En este se explica la herramienta SDSoC y se explica un algoritmo de procesado de imágenes empleado para la detección de objetos en una escena de fondo estático, *"PCA"*. Se presenta una pequeña aproximación a este algoritmo para presentar lo simple que es trasladar la ejecución de una función desde el software para su ejecución en el hardware y los beneficios que esta puede introducir en los tiempos de ejecución. Presenta además resultados de los distintos tiempos de ejecución conseguidos, los cuales, a lo largo de este trabajo se pretenden optimizar.

[6] no es el único trabajo que introduce la implementación del algoritmo "*PCA"* para su ejecución en HW sino que [7] desarrolla este completamente en el nivel de abstracción más bajo. En este segundo se presentan ciertas limitaciones que la implementación del algoritmo supone en la FPGA por falta de ciertos recursos, y presenta una descripción del "*Análisis de las Componentes Principales"* en profundidad, al que se hará referencia más adelante.

Este último trabajo nombrado también ha servido para influenciar ciertos trabajos que permiten acelerar, en vez del algoritmo al completo, ciertas partes como lo puede ser el cálculo de la

covarianza que requiere, [8], mostrando además la facilidad que proporciona el desarrollo en lenguajes de alto nivel, siendo exportable a herramientas de menor abstracción.

Es observable como, en función del consumidor y su aplicación bajo estudio, las metodologías de diseño de los SoC varían. Particularizando en el vendedor de *Sytem on Chip*, Xilinx, este proporciona distintas herramientas de estudio que han servido para el desarrollo de distintos de los estudios anteriores como [4][5][6][8].

La idea de particularizar en estas herramientas de Xilinx se debe a que proporcionan cada una de ellas distintos niveles de abstracción en función de la metodología de diseño que se desea emplear.

La primera que se desarrolló, para el diseño en lenguaje de descripción hardware, *Vivado Design Suite* permite al consumidor una metodología de diseño que no presenta abstracción al usuario, permitiéndole encargarse del más bajo nivel a la hora de codificar la FPGA. Esta herramienta permite emplear la metodología de codiseño hardware-software siempre que se emplee junto a ella *SDK*, lo que supone al usuario el hecho de emplear dos programas para esta metodología.

La segunda desarrollada por Xilinx fue HLS, la que permite la creación de ciertos módulos IP integrables en *Vivado Design Suite* en lenguaje de más alto nivel. Una vez creados estos módulos son exportables a *Vivado* pudiendo emplear la metodología de codiseño de la misma forma que se ha explicado en el párrafo anterior, junto a *SDK* en dos programas distintos.

La tercera desarrollada en este ámbito es la nombrada anteriormente SDSoC, la cual el grado de abstracción que presenta al usuario es máximo. Este no conoce de las transformaciones internas realizadas para obtener los módulos IP de las funciones que se aceleran en HW, pero si permite al usuario optimizarlas gracias a la integración de *HLS*. En este caso, la metodología codiseño se emplea con un único programa.

Cabe destacar que Xilinx no solo proporciona una herramienta que presenta un nivel de abstracción en lenguajes de alto nivel como C o C++, sino que también proporciona al usuario una herramienta con un intérprete de PYTHON a VHDL, permitiendo a programadores de PYTHON realizar diseños en SoC's. Esta plataforma de diseño está pensada para aquellos arquitectos de sistemas que deseen una interfaz software y un espacio de trabajo que permita un prototipado y desarrollo rápido. [9]

4.3 Fundamentos teóricos:

En este capítulo se van a abordar las explicaciones en el ámbito teórico del algoritmo elegido para ser optimizado gracias a los aceleradores hardware, *PCA,* y el software utilizado. Además, se explicará cómo fue la toma de contacto con las distintas herramientas y otros aspectos teóricos relacionados.

4.3.1 Algoritmo PCA:

Puesto que este trabajo pretende ser la continuación de [6], el algoritmo bajo estudio es el mismo. Este es el algoritmo *PCA ("Principal Component Analysis")* o bien "Análisis de las Componentes Principales". Este es un método estadístico que permite simplificar la complejidad de espacios de varias dimensiones dotados de una amplia información muestral a la vez que conserva su información principal.

Es ampliamente utilizado en campos muy variados como la estadística, el procesado de señales, y la visión artificial ya que permite eliminar la información redundante, es decir, la información que presenta correlación, mientras que preserva aquella información que representa lo fundamental, denominado como "componentes principales". Dado que la contribución de este trabajo no está en la mejora de este algoritmo, sino en su implementación en un SoC, esta es la razón por la que no se va a hacer demasiado hincapié en sus fundamentos ofreciendo una visión del algoritmo muy generalista.

Una de las ventajas que proporciona este algoritmo, como ya se ha nombrado, es la reducción de la dimensionalidad, lo que supone una reducción del volumen de información a manejar cuando se trabaja con grandes cantidades de variables con alta correlación. De manera que, este método, permite crear un conjunto de variables incorreladas, independientes entre sí, que son combinación lineal de las variables originales, las nombradas previamente "componentes principales".

En este trabajo, el algoritmo PCA se aplica a imágenes. En este caso cada una de ellas está compuesta por un número elevado de píxeles donde, normalmente, los adyacentes presentan una alta correlación. De manera más concreta, este algoritmo se aplicará a la detección de elementos en un fondo estático.

Esta detección se realizará mediante la determinación de la similitud de un conjunto de imágenes, denominado "imágenes de fondo", con una nueva imagen recibida, donde si estás presentan componentes principales similares (entendiendo como similares que la diferencia esté acotada a cierto umbral) se dirá que no hay objetos nuevos en la imagen recibida. Si, por el contrario, la diferencia entre la imagen y las imágenes de fondo supera el umbral, se podrá decir con gran probabilidad que en la nueva imagen se ha detectado un elemento.

Este algoritmo se compone por dos procesos como se observa en la [figura 1]:

- Primer proceso: obtención de las componentes principales de las imágenes que conforman el fondo, rama superior del diagrama de la [figura 1].
- Segundo proceso: determinación de si hay un nuevo elemento en el transcurso de las imágenes recibidas, rama inferior del diagrama de la [figura 1].

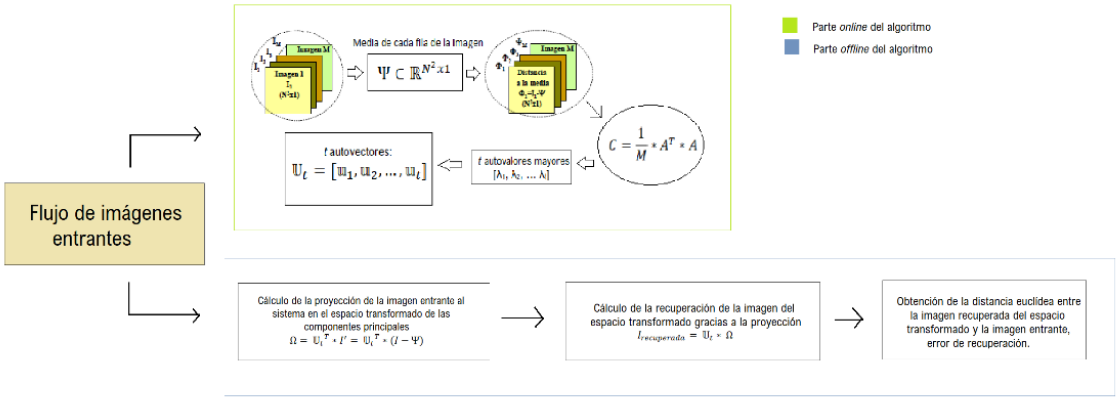


Figura 1: Diagrama del flujo del algoritmo PCA

El primer proceso se denomina como parte "*offline*" del algoritmo y el segundo como parte "*online*". Ambos procesos se describen y explican más detalladamente a continuación. En [7] se ofrece una descripción detallada del algoritmo PCA..

Como introducción al apartado siguiente, la parte *offline* (rama superior de la [figura 1]) lee un conjunto de imágenes a las que extrae la media, y mediante el cálculo de los autovalores y autovectores de la matriz de covarianza asociada se obtiene el espacio de componentes principales mencionado en los párrafos anteriores.

Una vez se obtiene el espacio transformado, se pasa a ejecutar la rama inferior de la [figura 1], la parte *online*, donde se obtiene la proyección de la imagen en el espacio transformado y se compara la recuperación con la entrada. Es a partir de esta comparación como se determinará si existe un nuevo objeto en la escena.

Particularizando el algoritmo para la lectura de un video, el cual se compone de diferentes *frames*, los dos procesos brevemente introducidos previamente son los siguientes:

- **Parte "*offline*" del algoritmo:**

Paso primero: Obtención del modelo de *escena de fondo.*

Para poder obtener las componentes principales de un escenario, lo primero es determinar cuántas imágenes se desea que compongan este fondo estático con el que se compararán posteriores imágenes, es decir, el número de muestras inicial usado para describir el escenario bajo estudio.

Este conjunto de imágenes que describen un mismo escenario será el empleado para determinar las componentes principales. El número de imágenes que se emplearán se denominarán *Mfondo* y el conjunto de imágenes se representara como $I_i \in \mathbb{R}^{NxN}$ donde i = 1,…*Mfondo.*

El algoritmo PCA requiere como entrada que las imágenes sean vectores de manera que cada imagen será un vector de dimensiones N^2x1. De manera que la *escena de fondo* será una matriz donde cada columna representará una imagen, siendo pues la entrada al algoritmo una matriz de dimensiones: [N^2x1,*Mfondo*].

Paso segundo: Obtención de la media de la matriz de entrada.

Otro de los requisitos de entrada del algoritmo PCA es que los datos introducidos deben poseer media nula. Para cumplir este requisito, lo que se debe hacer es obtener la media de cada fila y crear un vector columna con estas ($\Psi \in \mathbb{R}^{N^2x1}$).

La media se realiza con los pixeles de una fila puesto que son los que representan un mismo punto en las *Mfondo* imágenes elegidas para crear la escena de fondo.

Una vez obtenido este vector media, se le restará a la matriz que se introduce como entrada donde se encuentran las *Mfondo* imágenes vectorizadas. Así pues se obtendrá una matriz de entrada (A) donde cada uno de sus elementos tiene media nula:

$$A = \begin{bmatrix} I_1 - \Psi_1 & I_2 - \Psi_1 & \cdots & I_{Mfondo} - \Psi_1 \\ I_1 - \Psi_2 & I_2 - \Psi_2 & \cdots & I_{Mfondo} - \Psi_2 \\ \vdots & \vdots & \ddots & \vdots \\ I_1 - \Psi_{N^2} & I_2 - \Psi_{N^2} & \cdots & I_{Mfondo} - \Psi_{N^2} \end{bmatrix} (1)$$

Paso tercero: Obtención de la matriz de covarianza.

Una vez se obtiene la matriz de entrada a este algoritmo (A), se calculara su matriz de covarianza. La definición del cálculo de esta matriz es:

$$C = \frac{1}{M} * A * A^T \ (2)$$

Este producto matricial, según la definición de multiplicación de matrices, dará como resultado una matriz de tamaño [N^2, N^2]. Esto implica que cuando el tamaño de la imagen sea grande, el número de operaciones aritméticas a realizar aumentará. Por ello, puesto que el paso siguiente es la búsqueda de los autovalores con los que conformar la matriz de autovectores, y estos coinciden para la conmutatividad de la multiplicación (¡Cuidado! Coinciden los autovalores, no el resultado), la operación que se realizará será:

$$C = \frac{1}{M} * A^T * A \ (3)$$

Gracias a este cambio, la matriz de covarianza obtenida tendrá una dimensión cuadrada de *Mfondo* x *Mfondo*.

Paso cuarto: Obtención de la matriz de autovalores.

Para obtener la matriz de autovectores ($\mathbb{U} \in \mathbb{R}^{N^2 x M}$) lo primero que se debe realizar es el cálculo de los autovectores aplicando:

$$C * \mathbb{U} = \lambda I * \mathbb{U} \ (4)$$

Donde el vector fila λ representa el vector de autovalores ($[\lambda_1, \lambda_2, \dots, \lambda_{Mfondo}]$). Sustituyendo la matriz de covarianza por su definición, donde V son los autovectores de $A^T * A$:

$$A^T * A * V = \lambda I * V \ (5)$$

Multiplicando a ambos lados por la matriz A:

$$(A * A^T) * A * V = \lambda I * A * V \ (6)$$

Si se comparan la fórmula (6) con la fórmula (4) se observa que la matriz de autovectores $\mathbb{U}$ se obtiene a partir de V. De manera que: $\mathbb{U} = A * V$, denominándose a esta "matriz de transformación".

Paso quinto: Reducción de componentes principales.

Una vez se ha obtenido la matriz de transformación $\mathbb{U}$, se debe determinar cuántos autovectores son necesarios. La matriz $\mathbb{U}$ tiene el mismo tamaño que la matriz A, [N^2x1, *Mfondo*], poseyendo *Mfondo* autovectores de tamaño N^2x1. Si para pasos posteriores se emplease la matriz de transformación $\mathbb{U}$ al completo, no se estaría disminuyendo la dimensionalidad trabajando con todas las componentes de la imagen. Es por esto por lo que se requiere reducir el número de autovectores que componen esta matriz de transformación.

De los *Mfondo* autovectores se deberán coger aquellos que se asocien a los "*t*" autovalores de mayor peso, donde el valor de esta variable "*t"* se determina en función del criterio denominado error cuadrático medio residual normalizado (RMSE).

Este criterio se define como la relación entre el sumatorio de los autovalores elegidos entre el total de autovalores. Para que este RMSE se considere aceptable, esta relación debe ser superior a 0.95 (es decir, un 95%).

$$RMSE = \frac{\sum_{i=t+1}^{Mfondo} \lambda_i}{\sum_{i=1}^{Mfondo} \lambda_i} * 100 \ > 95\ (\%)$$

Será gracias a esto que se obtenga una matriz de transformación reducida $\mathbb{U}_t = [\mathbb{u}_1, \mathbb{u}_2, \dots, \mathbb{u}_t]$, donde cada una de las columnas corresponde a uno de los autovectores seleccionados.

De manera más gráfica, la parte *offline* del algoritmo se puede ver a continuación en la [Figura 1]:

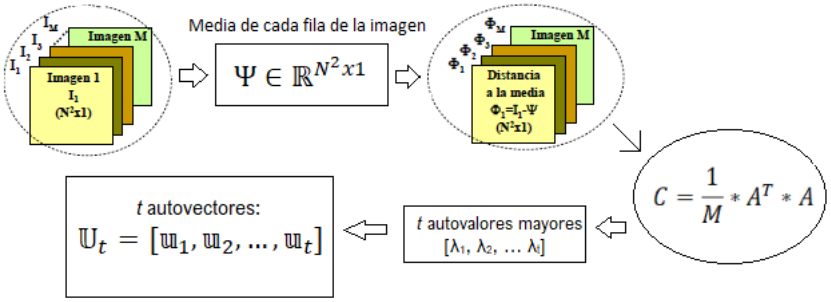


Figura 2: Parte offline del algoritmo PCA

Particularizando el algoritmo a este proyecto, se decidió, partiendo de la base de estudios anteriores como [7] y [6], que el número de imágenes empleadas para definir el fondo estático será *Mfondo* = 8. Un número superior de imágenes supone un error práctico similar de forma que con intención de reducir la carga computacional se justifica el uso de ocho imágenes.

- **Parte "*online*" del algoritmo:**

Una vez obtenida la matriz de transformación con los autovectores asociados a los autovalores de mayor peso, $\mathbb{U}_t$, se procede a determinar si en nuevas imágenes recibidas se detecta la aparición de algún objeto. Los pasos de esta parte del algoritmo son:

Paso primero: Preprocesado de la imagen recibida.

Al igual que se hizo con las imágenes que han conformado el fondo estático, para poder tratar una imagen como entrada al algoritmo PCA se debe:

- Transformar la imagen de matriz de pixeles a un vector columna.
- Convertir en una imagen de media nula.

De manera que, lo primero que se debe hacer es transformar la imagen I $\in \mathbb{R}^{NxN}$ en un vector I′ $\in \mathbb{R}^{N^2x1}$. Tras esto, para que esta imagen recibida tenga media nula, se le va a restar la media de las imágenes que conformaban el fondo estático, obteniendo así la imagen de entrada al algoritmo:

$$I' = \begin{bmatrix} I_1 - \Psi_1 \\ I_1 - \Psi_2 \\ \vdots \\ I_1 - \Psi_{N^2} \end{bmatrix}$$

Paso segundo: Cálculo de la proyección sobre el espacio transformado.

Una vez obtenida la imagen de entrada, lo que se debe hacer es obtener la proyección en el espacio de las componentes principales que se ha calculado de manera *offline*. Para ello se realiza la multiplicación matricial entre la matriz de transformación y la nueva imagen de media nula. Puesto que se debe cumplir el requisito de dimensiones entre las columnas de la primera matriz y las filas de la segunda, se empleará la transpuesta de la matriz de transformación. Matemáticamente:

$$\Omega = \mathbb{U}_t{}^T * I' = \mathbb{U}_t{}^T * (I - \Psi)$$

De esta manera se va a obtener un vector $\Omega \in \mathbb{R}^{tx1}$ donde cada componente representa la contribución de cada autovector de la matriz de transformación en la representación de I'.

Paso tercero: Recuperación de la imagen proyectada.

La recuperación de la imagen proyectada en el espacio transformado se realiza gracias a la proyección obtenida en el paso anterior:

$$I_{recuperada} = \mathbb{U}_t * \Omega$$

Paso cuarto: Determinación de la existencia de un nuevo objeto dentro de la imagen.

Para determinar si existe un objeto dentro de la imagen, se compara la imagen recibida una vez preprocesada con la imagen recuperada del espacio de las componentes principales. Gracias a esta comparación se puede calcular el error de recuperación, pudiendo determinar si existe o no un objeto en la imagen.

Para ello, el error de recuperación se va a representar mediante la distancia euclídea entre los elementos de ambas imágenes, y la determinación de si existe objeto o no se realizará al comparar el error con un cierto umbral experimental.

Si la diferencia entre la imagen recibida y la imagen recuperada es inferior al umbral, entonces, no se han detectado objetos en la imagen. Si por el contrario esta diferencia es superior al umbral, hay nuevos objetos no pertenecientes al fondo estático.

Se dice que el umbral es experimental puesto que será obtenido y ajustado en función de las condiciones de la escena.

Más adelante, en el siguiente capítulo, se realizará una particularización del algoritmo indicando las imágenes usadas, los tamaños y el diseño para la aplicación.

4.3.2 Codificación en coma fija:

En este apartado se va a presentar la codificación en coma fija. Habitualmente el método más empleado es el de la codificación en coma flotante debido a la precisión proporcionada en el cómputo, a pesar de que esta requiera más recursos, y a que las unidades de procesado empleadas incluyen unidades aritmético-lógicas específicas para ello. El tipo de variables empleadas habitualmente son "*float*" y "*double*" para aquellas situaciones que requieren decimales.

Dado que en este trabajo hay dos elementos de cómputo (FPGA y microprocesador) el escenario ideal es que ambos operaran con el mismo tipo de codificación. Así como las FPGAs poseen recursos limitados y la codificación en coma fija emplea menos, se antoja como el método óptimo para ello. Por su parte el microprocesador, emplea habitualmente coma flotante debido a sus unidades internas de cómputo. Por lo que se presenta un problema de compatibilidades y optimizaciones de datos.

La codificación en coma fija es un tipo de representación que destina ciertos bits del total para la representación de la parte entera y el resto para la representación de la parte fraccionaria. Esta frontera denominada como "*coma virtual"* es una frontera virtual que permanecerá fija, es decir, no permite que varíe el número de bits empleados para la representación de la parte entera ni de la decimal.

La definición de la coma y la longitud de la palabra requiere un estudio previo de los errores, como se comenta en el siguiente capítulo. Dado el potencial que ofrece Matlab se ha optado por emplear la Toolbox de Fixed Point como herramienta de cálculo de errores entre la codificación en coma fija y coma flotante.

Lo complejo de la coma fija no es la obtención del valor decimal si lo que se tiene es la representación binaria, sino el paso opuesto. Este paso supone complejidad puesto que las operaciones en coma fija pueden producir resultados con mayor número de bits que los operandos suponiendo cierta pérdida de información. Por ello se emplean técnicas de redondeo y truncamiento, las cuales cometen un error que debe tomarse en consideración.

Además de esto, al existir operaciones aritméticas entre números en coma fija, puede ocurrir "*overflow"* lo que implica sobredimensionar los datos con bits de guarda para evitarlo.

La decisión del número de bits a emplear para la representación de la parte entera viene dada por:

Ecuación 1: Expresión para determinar el número de bits para la parte entera

$$\text{Integer Bits Required} = Ceil\left(\frac{LOG^{10}\,\text{Integer\_Maximum}}{LOG^{10}2}\right)$$

Mientras que la elección de la parte fraccional dependerá de la precisión deseada. Se presenta este aspecto teórico puesto que más adelante se barajeará la posibilidad de implementar la multiplicación matricial usando esta representación, determinando si el ahorro de recursos supone o no una aceleración.

Como conclusión de este apartado se desea destacar la paradoja que se presente entre precisión y recursos consumidos. Se observa que ambas están claramente vinculadas, relacionándose una mayor precisión con un mayor consumo de recursos.

4.3.3 Software empleado:

A continuación, se va a hacer una introducción a las distintas herramientas software que se han empleado para llevar a cabo la realización de este proyecto. En algunas se hará más énfasis al ser

más principales mientras que otras simplemente se nombrarán puesto que han sido utilizadas. Estas herramientas son Matlab, Eclipse IDE, SDSoC, HLS, Vivado y Petalinux.

- **Sobre "Matlab":**

Matlab es un sistema de cálculo numérico que ofrece un entorno de desarrollo integrado con un lenguaje de programación propio. Este es un lenguaje interpretado que permite operaciones de vectores y matrices, funciones y programación orientada a objetos. Una de sus prestaciones fundamentales es la manipulación matricial, la representación de datos y funciones y la implementación de algoritmos, entre otras cosas. Este es un entorno de escritorio optimizado para la exploración iterativa, el diseño y la solución de problemas.

Permite obtener gráficas para visualizar datos y dispone de herramientas para la creación de diagramas personalizados. Proporciona además interfaces para lenguajes como C/C++,Java, .NET y otras.

Matlab es una herramienta que proporciona resultados tangibles lo que facilita la observación de si estos son los deseados y esperados o si se comete algún error por el camino. Esto permite implementar algoritmos en su lenguaje de programación, obtener los resultados deseados, y exportarlos al lenguaje deseado a posteriori para su aplicación deseada.

Puesto que es un lenguaje cuya prestación fundamental es la manipulación matricial, permite realizar multiplicaciones, divisiones, sumas, restas y el resto de los cálculos sobre matrices gracias a la utilización de los símbolos que la representan: '*',*/','+','-', mientras que el resto de los lenguajes requieren de codificación elemento a elemento para estos cálculos. Esto permite obtener de manera sencilla los resultados esperados y la comprobación del correcto funcionamiento de la aplicación implementada.

En este trabajo, Matlab se emplea como herramienta con la que obtener los "*Golden Data*", es decir, la herramienta que va a proporcionar los datos deseados/ideales que se esperan obtener al implementar el algoritmo posteriormente. Por ello Matlab se denomina herramienta de análisis y diseño.

Además, como se describirá más adelante, la codificación del algoritmo en SDSoC se realiza en lenguaje de alto nivel C/C++, y puesto que una de las propuestas de optimización del algoritmo es la implementación de la coma fija, Matlab presenta la posibilidad de determinar el número de bits que se requerirán para representar la parte fija y el número de bits que representarán la parte decimal gracias a su operador "fi". Estos son objetos numéricos que representan una especie de estructura con diferentes parámetros que permiten ajustar el tamaño, la forma de redondeo, los bits dedicados a la parte fraccional…

Gracias a la codificación del algoritmo en coma flotante y al estudio del error cuadrático entre la esta codificación y la codificación con el operador "fi", se puede determinar el número necesario de bits.

- **Sobre "Eclipse IDE":**

Eclipse es una plataforma de software que se compone por herramientas de programación en código abierto que típicamente se usa para desarrollar entornos de desarrollo integrados ("IDE: *Integration Design Enviroment*"). Eclipse dispone de un editor de textos con un analizador sintáctico, con compilación en tiempo real. No tiene en mente ningún lenguaje específico, sino que es un IDE genérico. Eclipse proporciona un depurador de código, de uso fácil e intuitivo que ayuda a obtener mejoras en el código.

La justificación del uso de esta herramienta en este trabajo se debe a que el software que se empleará para obtener resultados y poder optimizar y acelerar los algoritmos, SDSoC, se basa en su IDE. Por ello se emplea para obtener la implementación en lenguaje de alto nivel del algoritmo que se desea acelerar (el algoritmo PCA). De esta forma se asegura que el algoritmo funciona según lo deseado.

Esta herramienta dispone de un proceso de depuración donde se pueden introducir puntos de ruptura y observar el valor de las variables hasta ese punto. El problema que presenta esta IDE es que, cuando la variable es una matriz de grandes cantidades de elementos, no muestra los valores que toman cada uno de ellos de forma que no se puede comparar algunos valores con los "*Golden Data"* que proporciona la herramienta de MATLAB. A pesar de esto proporciona al usuario resultados visibles y permite establecer en lenguaje de alto nivel la programación de algoritmos, pudiendo ejecutarlos y observar, mediante la impresión por consola o bien mediante la impresión de imágenes (si este algoritmo es visual, como es nuestro caso), que se ejecuta según lo previsto.

- **Sobre "SDSoC":**

SDSoC ("*Software Design Development Enviroment*") es un entorno de desarrollo de la familia Xilinx, que se focaliza en los conocidos todo programables SoC's y MPSoC's Zynq, cuya finalidad es proporcionar una experiencia de programación enormemente simplificada para ASSP (del inglés "*Application Specific Estándar Product*", producto estándar para una aplicación específica). Esto es proporcionar al usuario una experiencia de programación en lenguaje de alto nivel (C/C++/OpenCL) gracias al uso del entorno de desarrollo basado en la IDE ("Entorno de desarrollo integrado") de Eclipse y una plataforma de desarrollo integral para el despliegue heterogéneo de la plataforma Zynq.

Esta herramienta es la base de este trabajo de fin de grado, permitiendo emplear la metodología de codiseño HW/SW que se nombraba en apartados previos.

La familia de productos Zynq proporciona al usuario la opción de construir soluciones SoC's bastante potentes que aprovechan la potencia combinada del sistema de procesado ARM y la lógica programable, permitiendo una amplia diferenciación, integración y flexibilidad para la programación de HW/SW y E/S.

Esto se debe a que SDSoC es un entorno de desarrollo de diseño software que permite al usuario definir la lógica programable de la FPGA mediante desarrollo software en C/C++, o incluso OpenCL, lo que lo hace diferente del flujo híbrido tradicional donde la definición de la FPGA se realiza usando el flujo de hardware tradicional (en HDL) y el sistema de procesamiento se desarrolla usando un flujo de software tradicional. Esto mejora significativamente la eficiencia del usuario permitiéndole disminuir el TTM ("Time To Market"), es decir, llevar los productos al mercado más rápido.

Es gracias al uso de la IDE de Eclipse que el usuario puede elegir mover las funciones de software a la lógica programable de la FPGA gracias a un simple clic de un botón, puesto que es el flujo de SDSoC el que se encarga de hacer que esto funcione sin que el usuario realmente intervenga, abstrayéndolo de las transformaciones realizadas. Esto incluye la configuración de conexiones, puesto que se requieren transportar datos al FPGA para el cálculo, la construcción de los paquetes de soporte de placas y, finalmente, la generación de "bitstreams" para el FPGA.

SDSoC viene equipado además con potentes herramientas de perfilado del sistema y de estimación de rendimiento que permiten al desarrollador tomar decisiones inteligentes en base a los distintos escenarios, las cuales llevan a mejores implementaciones en términos de aceleración y eficiencia.

La finalidad de SDSoC es pues permitir a los desarrolladores de software definir hardware permitiendo múltiples escenarios al poder moverse, simultáneamente, entre hardware y software.

Los lenguajes que soporta esta aplicación son C, C++ y OpenCL, permitiendo el uso de librerías de código abierto de OpenCV y funciones específicas creadas por Xilinx diseñadas para optimizar el rendimiento y consumo de recursos de estas al introducirlas en HW, librerías xfOpenCV. Estas segundas realmente usan funciones OpenCV estándar lo que implica la inclusión de las cabeceras de ambos conjuntos para que el diseño funcione.

Algunos de los aspectos importantes de esta aplicación se enumeran a continuación:

- Permite, para la modificación y optimización de aquellas funciones cuyo destino de ejecución es el hardware, la invocación de los programas de Xilinx: Vivado y HLS ("*High Level Synthesis*"), permitiendo la creación de IP cores (o incorporación de alguno ya creado) de forma que se incluyan en el diseño final influyendo en la mejora de su rendimiento.
- Permite distintas opciones de sistema operativo, a correr en el procesador, para construir los proyectos desarrollados: *standalone* (programa que no requiere cargar módulos externos, ni funciones ni librerías diseñado para arrancar con el procedimiento de arranque del procesador destino -*baremetal*-), PetaLinux (versión reducida del sistema operativo Linux) o FreeRTOS, que es un sistema de tiempo real. Si se elige que el programa compilado corra sobre un sistema operativo, SDSoC genera los archivos necesarios para arrancar la placa desde una SD que los almacena.
- Permite el uso de unas directivas denominadas "pragmas" que permiten indicar al compilador como desea el usuario hacer la asignación de recursos con la finalidad de acelerar la ejecución, como lo es desenrollar un bucle o realizar un pipeline, o bien como realizar los accesos a memoria para disminuir la latencia, optimizando el diseño.

Además de los propios pragmas del entorno SDSoC, denominados como "pragma SDS", esta aplicación permite el uso de algunos de los pragmas del programa HLS del mismo entorno de desarrollo de Xilinx. Se debe tener especial cuidado con esto puesto que SDSoC no admite los pragmas de HLS referidos a interfaces ("#pragma HLS interface"). Este entorno no genera una directiva de interfaz para este argumento indicado y además provoca mensajes de error de compilación bastante encriptados de forma que se recomienda evitar su uso.

- Permite acceder al diseño por bloques del sistema generado a partir del código creando un archivo ".xpr" que se abre desde Vivado.
- Como se ha nombrado brevemente anteriormente, cuenta con una herramienta de perfilado ("*TCF profiler*") que da los resultados de tiempos de uso de CPU con la finalidad de proporcionar al usuario cuáles son las funciones candidatas a ser aceleradas. Esta herramienta mide el tiempo generando muestras periódicas y en cada una de estas observa dónde está el programa ejecutándose. Además de la herramienta de perfilado, cuenta con un estimador de recursos y ejecución, debiendo activarse esta opción en las opciones generales del proyecto.

La finalidad de este estimador es mostrar una estimación de cuanto se tarda en procesar el programa si se emplea bien solo el software o paralelamente software y alguna función en hardware. Estos datos que da permiten obtener gran información sobre la supuesta ejecución del programa, pudiendo ver el "*speed-up*" que sufre cuándo se mandan algunas de las funciones para que las realice la lógica programable, y una estimación del uso de los recursos dentro de la FPGA.

Se emplea el adjetivo "supuesta" para referirse a la ejecución puesto que es tan solo una aproximación, debiendo ejecutar la aplicación en la placa para obtener los resultados de ejecución reales (cabe decir que son aproximaciones bastante próximas).

- Incluye un analizador de eventos que permite ver el tiempo que tardan en moverse los datos dentro del sistema y cuánto tardan en ejecutarse las operaciones, pudiendo observar también la paralelización de estas. Realmente útil para el proceso de depuración puesto que permite ver que está fallando si hay problemas de tiempos. Estas depuraciones se pueden realizar tanto descargando en placa como sin descargar. Para aplicaciones *standalone* esta depuración se puede realizar mediante un JTAG y un emulador y para el caso de aplicaciones con sistema operativo se debe conectar la placa al ordenador mediante una conexión Ethernet y configurar el "*TCF Agent*" para que se conecte a la dirección asignada a la placa.

Sobre SDSoC, cabe destacar que permite la creación de plataformas hardware personalizadas ("*Custom Hardware Platforms*"). Esta característica es realmente importante puesto que la placa de desarrollo que se va a emplear en este proyecto (Zybo Z7-20) no es una de las plataformas que incluye la aplicación por defecto.

Cabe destacar que SDSoC, a pesar de llevar ciertos años a disposición de los consumidores y desarrolladores, no es una herramienta muy popular entre ellos ni conocida por el resto. Esto se debe a que aquellos desarrolladores de aplicación que quieren sacar ventaja del hardware consideran que le nivel de abstracción que proporciona SDSoC es demasiado grande si realmente se quiere sacar beneficio de la aceleración del hardware y prefieren el codiseño HW/SW mediante herramientas con menor abstracción como es *Vivado Design Suite*, y su kit de desarrollo software asociado, *SDK*.

Para terminar la introducción a esta herramienta, se presenta en la siguiente [figura 3] una captura de la interfaz del programa.

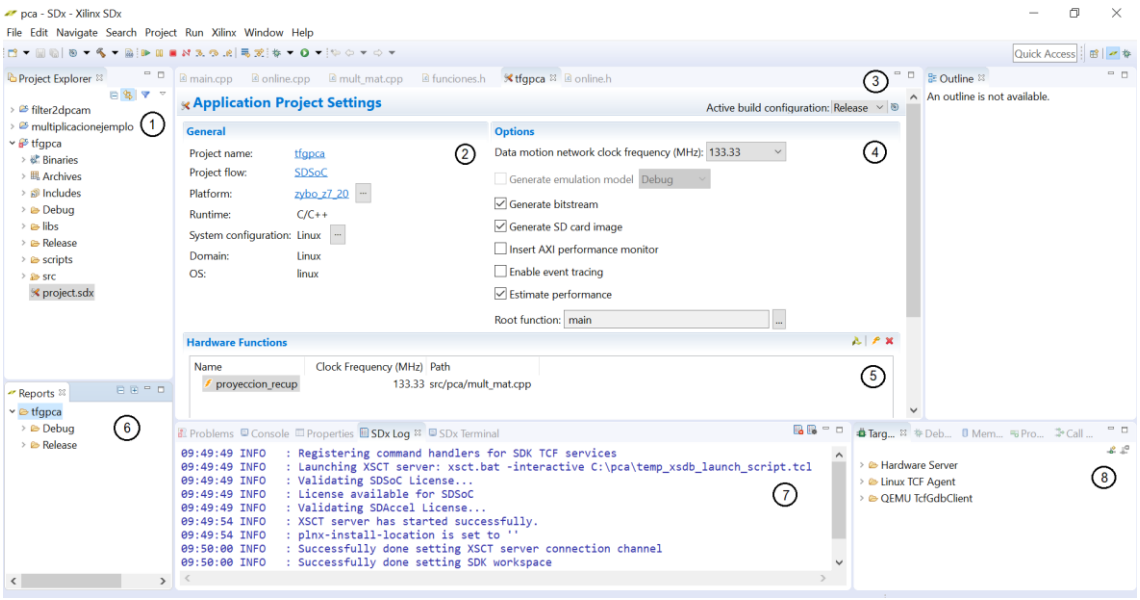


Figura 3: Interfaz del software SDSoC

En esta captura se puede observar que se han numerado las distintas secciones que se encuentran al iniciar el software. La sección 1 muestra de manera esquemática los distintos trabajos que se encuentran en el espacio de trabajo. Cada uno de ellos es independiente de los demás y cada uno de ellos dispone de las carpetas que se observan desplegadas.

Las carpetas "*include*","*libs*", "*src*" y *"scrips"* son las que contienen los archivos fuente necesarios para el trabajo. En las dos primeras se incluyen las distintas librerías que se requieren para la compilación, en "*src*" se incluyen los ficheros "*.c*" donde se describen las distintas

funciones del trabajo y los archivos de cabecera de las distintas funciones y finalmente en "*scripts"* se incluirán, si fuesen necesarios, los distintos archivos que se requieren al ejecutar la aplicación para que sean almacenados en la memoria externa SD.

Dentro de las carpetas "*release"* y "*debug"* se almacenan los ficheros que se crean cuando se compila el proyecto. Se encuentran tanto los archivos "*report"* donde se informa de cómo ha ido el proceso de compilación, los errores que hayan podido ocurrir o los avisos que hayan surgido, como los archivos que se deben copiar en la SD para ejecutar la aplicación sobre la plataforma.

El fichero de nombre "*Project.sdx"* es lo que se muestra numerado como sección 2 de la imagen. En esta se muestra un breve resumen de las características del proyecto: nombre, flujo, plataforma para la cual se diseña, y las configuraciones del sistema (si corre en un sistema operativo se indica cual, y si no se mostrará si emplea una configuración "*standalone"*).

La sección 3 muestra la pestaña donde se elige el tipo de compilación deseada, a elegir entre el modo "*release"* y el modo "*debug"* nombrados anteriormente.

El número 4 y el 5 están relacionados puesto que la sección 5 muestra aquellas funciones que se han destinado para la ejecución en HW mientras que la sección 4 muestra los resultados que se desean obtener de estas funciones. La frecuencia que se encuentran en ambas secciones representa la frecuencia de funcionamiento del reloj externo de la Zybo para la ejecución de la aplicación.

La opción de generación de *bitstream* debe estar siempre activada puesto que se desea emplear el hardware y es la manera de indicar a la FPGA como debe configurarse.

La generación de la imagen de la SD card representa la creación de los ficheros para cuando se desee que la aplicación arranque desde una memoria SD externa. Si se desea que la aplicación arranque mediante JTAG, la selección de esta opción no es necesaria y de hecho es aconsejable no marcarla puesto que requiere tiempo de compilación no necesario.

La generación de modelo de emulación es una opción que solo está disponible para aquellas plataformas que el propio software proporciona, y no para plataformas personalizadas añadidas.

El resto de las opciones de esta sección que se pueden seleccionar muestran los distintos resultados de la ejecución de la aplicación: la monitorización de la ejecución de la interfaz AXI, un seguimiento de los eventos, y una estimación de los recursos HW de la plataforma para observar, previa a la traslación sobre la plataforma, si la escasez de recursos es un problema en la aplicación.

La sección 6 muestra las distintas carpetas con los resultados de la compilación. Dentro de estas se pueden encontrar los resultados de las opciones de la sección 4 que se han marcado y la lista de informes obtenida durante la compilación. Además muestra la fecha y hora de la última compilación realizada, y avisa sobre su desactualización si se ha realizado alguna modificación desde la última vez que se compiló.

La sección 7 muestra la consola del programa donde se observa por dónde va la compilación cuando se está ejecutando (pestaña "*console"*), muestra una pestaña *warnings* donde acceder de manera rápida a la lista de informes de errores y avisos, una pestaña *propierties* donde se muestra el resumen de las características de la función que se traslada a HW (frecuencia del reloj externo, ruta al fichero) y una pestaña *SDx Terminal*, empleada para la comunicación puerto serie con la plataforma.

Finalmente, la sección 8 muestra otras de las características de la compilación, puesto que como se nombraba anteriormente, para la depuración se pueden emplear distintos analizadores en función de la configuración del sistema. Si se emplea un sistema *standalone* se encuentra el

Hardware server, si se emplea un sistema operativo Linux, el *"TCF Agent"* (el cual es un depurador de aplicación paso a paso que contiene a su vez un perfilador para observar la ejecución de cada función), y si se emplea un emulador QEMU, el *QEMU TcfGdbClient*. Al expandir estas carpetas aparece el archivo donde configurar estos analizadores según las preferencias de la aplicación.

Para el caso del "*TCF Agent"*, que es el que se emplea en este trabajo, se da la opción de establecer una dirección IP. Esta será con la cuál la plataforma establecerá comunicación por lo que se requiere asignar la misma dirección IP y conectar el ordenador y la plataforma mediante un cable de red Ethernet.

- **Sobre "Vivado Design Suite":**

Vivado ("Vivado Design Suite") es otro de los paquetes software producidos por Xilinx utilizados en este trabajo Esta es un herramienta para la síntesis y el análisis de diseños en HDL ("*Hardware Description Language*"). Este engloba todo el proceso de implementación de un diseño en un SoC desde el punto de vista con menor abstracción de los nombrados.

Esta herramienta permite el diseño de la implementación hardware hasta el punto de poder elegir a que pines se va a conectar la entrada y salida del sistema final, poder conexionar cada uno de los bloques que integran el diseño a nivel hardware y pudiendo crearse bloques ("IP cores") dedicados a un fin especifico.

Estos últimos se crean con lenguaje HDL, de descripción hardware, en vez de con lenguaje de alto nivel como proporcionaban las otras herramientas software de Xilinx. Este lenguaje puede ser VHDL o Verilog, siendo más complejo de programar, en especial para usuarios software.

Además de la posible creación de bloques, Vivado pone a disposición del usuario lo denominado como "IP catalog" donde se encuentran ya desarrollados por Xilinx (o bien por otros usuarios) gran variedad de diseños para incluir y usar en el diseño.

Vivado no solo permite diseñar la implementación del hardware sino que en lo referido al software incluye la herramienta SDK ("Software Development Kit"). Esta permite la creación de aplicaciones integradas para cualquiera de los microprocesadores empleados por Xilinx, proporcionando además su depuración y análisis del rendimiento.

Esto permite que, una vez desarrollado el diseño hardware (refiriéndonos por esto a la configuración de los relojes, interconexión de los bloques y la elección de los propios bloques que integran el diseño) y obtenido su "bitstream", se pueda trasladar el proyecto a este entorno de desarrollo software desde el mismo Vivado, y comprobar que todo funciona según lo deseado gracias a la simulación paso a paso.

- **Sobre "Vivado High Level Synthesis (HLS)":**

Vivado HLS ("High-Level Synthesis") es una herramienta proporcionada por la familia Xilinx que realiza la transformación de especificaciones en lenguaje de alto nivel a implementaciones RTL ("*Register Transfer Level*") las cuales son sintetizables en el campo de Xilinx FPGA's. Esta herramienta permite las especificaciones de IP cores escritas tanto en C como en C++, SystemC, o bien como OpenCL, y la FPGA proporciona una fuerte arquitectura paralela con mejoras en rendimiento, coste y potencia con respecto a procesadores tradicionales. Estos IP cores son bloques que se implementan en hardware para implementar una función concreta.

Esta herramienta de síntesis de alto nivel crea un puente entre dominios software y hardware proporcionando a los diseñadores de hardware mayor nivel de abstracción al trabajar creando hardware de alto rendimiento y proporciona a los diseñadores software la opción de acelerar las partes de cómputo intensivo de sus algoritmos mediante la compilación/ejecución sobre FPGA.

Usar la metodología de HLS permite desarrollar algoritmos en alto nivel, disminuyendo el tiempo de desarrollo y validando la correcta funcionalidad del diseño más rápidamente, controlar la síntesis mediante directivas de optimización (los pragmas HLS nombrados previamente) y la creación de códigos fuente en C comprensibles y portables a diferentes dispositivos.

El método de trabajo con esta herramienta es el siguiente:

- Se debe elegir cuál será la placa sobre la cual se va a implementar la aplicación, con la finalidad de aproximar/estimar los recursos utilizados de esta y obtener resultados óptimos.
- Se debe crear un fichero que será el módulo de nivel superior donde se implementa la función que se realiza en el HW en lenguaje de alto nivel.
- Se debe añadir al proyecto un fichero '.c' que será el testbench de la función. En este se encontrará la función principal del proyecto (la función "main") y es donde se realizará la llamada a la función del módulo superior implementada en hardware. Este fichero puede contener funciones de impresión o de comprobación con la finalidad de asegurar el correcto funcionamiento de la función HW (p.e si esta realiza un producto matricial, introducir en el programa principal la multiplicación matricial software y comprobar que se obtiene los mismo con ambas).
- Asociada a estos dos ficheros se adjuntarán las cabeceras de las librerías necesarias.
- Una vez especificada la función de nivel superior, su llamada dentro del programa principal, y sus cabeceras, se realiza o bien una implementación (para comprobar su correcto funcionamiento y ver si existen problemas de compilación) o bien la síntesis del proyecto, gracias a la cual se obtienen las aproximaciones del uso de recursos de la FPGA que se ataca y la latencia. Esta síntesis permite obtener información suficiente como para saber que entradas a la función deben ser atacadas y que directivas se pueden emplear para optimizar y acelerar la aplicación final.
- Una vez comprobada y verificada la implementación RTL, esta se puede empaquetar en una selección de paquetes IP exportables a otras plataformas de desarrollo de Xilinx.

Es por esto por lo que esta herramienta no obtiene una implementación final como tal sino que sirve como herramienta de apoyo a otras de la familia Xilinx como Vivado Design Suite o SDSoC, ya que lo que hace es crear IP cores personalizadas para incorporar a aplicaciones en las que se desea obtener una optimización de recursos o aceleración, es decir, reducción de latencia.

Además de una estimación de los recursos y de la latencia, HLS muestra el tipo de interfaz que tendrán cada una de las entradas y salidas del bloque generado. Cabe recordar que es importante no usar directivas de interfaz si la finalidad del bloque IP implementado con HLS es ser usado en SDSoC, puesto que causa errores de depuración encriptados y no genera la interfaz deseada.

Estas estimaciones de rendimiento se pueden ver dentro de la ventana de rendimiento que proporciona la denominada "Perspectiva de Análisis" ("*Analysis Perspective*"), la cual permite al usuario analizar interactivamente los resultados al detalle gracias a la visualización de manera esquematizada de las operaciones de lectura, escritura, bucles… en los registros.

Otra de las grandes ventajas que proporciona esta herramienta de síntesis son los ficheros de informes de C/RTL co-simulación. Esto es un proceso de post-síntesis que reúsa el banco de pruebas en C de la presíntesis para verificar la salida RTL. El proceso consiste de tres fases en las que se ejecuta la simulación C guardando las entradas como vectores de entrada, se emplean estos

en una simulación RTL usando el fichero RTL creado por Vivado HLS y se guardan las salidas como vectores de salida, y se usan finalmente estos vectores del salida al banco de pruebas (tras la síntesis) para verificar que los resultados son correctos.

Finalmente, sobre HLS, cabe destacar, de nuevo, que se emplea el uso de directivas (sentencias pragma) para dar instrucciones al compilador de cómo realizar la implementación en HW. Suelen estar dentro de la función y afectan solo a algunas partes del código o bien a alguna de las variables. Las más útiles y empleadas son: *pipeline* (que permite la reducción del intervalo de inicialización de una función permitiendo la ejecución concurrente de operaciones), *unroll* (que permite aumentar el acceso a datos y rendimiento), las de optimización de bucle (que permite indicar el número total de iteraciones de este) o arrays (combina pequeños arrays en uno mayor para ayudar a reducir los recursos de la RAM) y las de optimización de Kernel (enfocadas a latencia y recursos). Una muy curiosa que se expondrá más adelante en este trabajo es "pragma HLS stream" que permite una paralelización a nivel de tarea.

Este programa se va a emplear en el trabajo para realizar el denominado "*Fine Tunning*" de optimización de las funciones que se manden a HW en el entorno SDSoC.

Para la comprensión del funcionamiento del programa, HLS proporciona ciertos archivos ejemplo que demuestran la implementación de algunas aplicaciones. Estas engloban desde operaciones de álgebra lineal como es el cálculo matricial hasta demostraciones de cómo usar el formato de tipo en coma fija "*ap_fixed"*.

Estos dos fueron los ejemplos que se examinaron y sobre los cuales se practicó para observar la aceleración que suponía el uso de las sentencias *pragma*, de donde se sacó como conclusión que para la multiplicación matricial lo óptimo era realizar una paralelización de cómputo mediante tuberías ("*pragma HLS PIPELINE"*) y la división de las matrices de entrada en arrays más pequeños ("*pragma HLS ARRAY_PARTITION o ARRAY_RESHAPE"*). Más adelante, en el siguiente capítulo, se comentarán las sentencias *pragma* seleccionadas.

- **Sobre "PetaLinux":**

PetaLinux es otra de las herramientas de desarrollo para software embebido que proporciona Xilinx en su catálogo. Esta ofrece todo lo necesario para personalizar, crear y desplegar una solución que corra en Linux en la PS ("Processing System") de los sistemas embebidos de Xilinx.

Esta herramienta SW, junto con las herramientas de diseño hardware que Xilinx proporciona, facilita el desarrollo de sistemas operativos Linux en los SoC's. Desde el arranque del sistema a la ejecución, permitiendo personalizar la carga de arranque, el Kernel de Linux o las propias aplicaciones del sistema.

Permite, además, la introducción de nuevos drivers de dispositivos, aplicaciones, librerías e incluso pruebas del software vía JTAG, entre otras.

Esta herramienta generará automáticamente un paquete de soporte en placa personalizado que incluye los drivers de dispositivo para los cores de procesado de los embebidos de Xilinx, además de las configuraciones de carga tanto de arranque como de Kernel. Todo gracias a que permite sincronizar el diseño HW generado con Vivado Design Suite con la plataforma software.

PetaLinux se emplea en este proyecto puesto que, como se ha comentado previamente al nombrar la ventaja de SDSoC de incorporar plataformas personalizadas, se requirió el uso de una plataforma, Zybo Z7-20 de Digilent, que no pertenecía a las plataformas por defecto que proporciona la herramienta software. Por ello se requirió la implementación con la herramienta *Vivado Design Suite* la implementación HW de la plataforma, introduciendo los distintos elementos a los que se iban a dar uso de esta, como el puerto HDMI o el I2C para la conexión del

módulo de cámara, y la implementación del SW y el sistema operativo con la herramienta PetaLinux.

Como se explica más detalladamente en el capítulo de "*Pliego de condiciones"*, este proceso de integración vino guiado gracias a que la propia empresa fabricante de la plataforma Zybo Z7-20 ponía a disposición del usuario una guía de como integrarlo.

Para poder emplear esta herramienta, cabe destacar, que se debe usar uno de los siguientes sistemas operativos, no estando Windows entre ellos: Ubuntu 16.04.1, CentOS 7.2/7.3 o RHEL 7.2/7.3. si la versión del programa es la 2017.4.

4.3.4 Hardware empleado:

En este trabajo, puesto que lo que se desea es ver como la implementación de funciones inicialmente destinadas a software sufren una aceleración al ejecutarlas en la lógica programable del FPGA de un SoC, se van a realizar las pruebas sobre la placa Zybo Z7-20 de Digilent, la cual dispone de un System on chip de la familia Zynq-7000 de Xilinx.

A) Tarjeta Zybo Z7-20:

Esta tarjeta sobre la que se realizarán las pruebas sobre la aceleración obtenida del algoritmo bajo estudio es una placa de desarrollo de circuito digital y software integrado construida con un SoC de la familia Zynq-7000 de Xilinx. Esta placa desarrollada por Digilent se basa en la arquitectura todo programables "System on Chip" que integra un procesador doble núcleo Cortex A9 de la familia ARM con una lógica programable, FPGA, de las 7 series de Xilinx.

La plataforma Zybo Z7 rodea el SoC con un gran conjunto de periféricos de comunicación y multimedia junto con la flexibilidad y potencia que proporciona el FPGA. Esta incluye un conjunto de funciones que permiten video (como un conector FFC, al que se le puede conectar una PCam, compatible con MIPI CSI-2, una entrada HDMI, una salida HDMI y un alto ancho de banda DDR3L) que se eligió con la intención de que esta placa fuese una solución asequible para aplicaciones de visión en sistemas integrados.

La Zybo Z7 dispone de un diseño similar a su predecesora la Zybo, pero esta proporciona muchas más características y mejora de rendimiento.

Esta plataforma Zybo Z7 dispone de dos variantes, Zybo Z7-20 y Zybo Z7-10, siendo la primera la que se empleará para este proyecto. La diferencia principal entre ambas es el tamaño de la FPGA dentro del AP SoC. Ambos procesadores tienen la misma capacidad pero la variante -7020 tiene 3 veces más FPGA interna que la -7010. Además de esto, esta segunda dispone de menor número de pines que la primero, lo que significa que múltiples de las características que la -7020 proporciona, no están disponibles en la -7010. Por ejemplo, la -7010 dispone de menos de la mitad de memoria BRAM que la -7020 (270KB frente a 630KB), de tres veces menos LUTs ("Look Up Tables", 17600 frente a 53200) y tres veces menos de Flip-Flops (35200 frente a 106400) y de puerto HDMI solo para transmisión (mientras que la -7020 dispone tanto de transmisión como de recepción), entre otras características.

Además, la variante Z710 no hubiese sido compatible en este proyecto puesto que el propio fabricante recomienda no emplearla para sistemas de visión embebidos debido al tamaño pequeño de FPGA. Lo que si recomienda es que para aquellas aplicaciones interesadas en procesado de video, sea la variante 7020 la que se emplee.

La Zybo Z7 es completamente compatible con las herramientas de diseño de Xilinx, Vivado. Esto es lo que la hace la candidata perfecta a emplear en nuestro proyecto, además del hecho de ser más pequeña que otras placas de desarrollo como la Zedboard.

En un aspecto más técnico, la Zybo Z720 dispone, más concretamente, del SoC XC7Z020-1CLG400C de los Zynq nombrados anteriormente. Dispone de un ADC de 1MSPS ("*Megasample per second*"), de 53200 Look-up Tables, de 106400 registros ("*Flip Flops*"), una memoria en bloque RAM de 630KB, de 4 CMT ("*Clock Managment Tiles*"), y algo que lo hace muy interesante: "*Zynq Heat Sink*", un disipador de calor. Esto último está pensado para disipar el calor extra generado por los recursos adicionales del FPGA cuando se ejecutan diseños de cambio rápido complejos.

Soporta distintos tipos de alimentación (mediante USB, alimentación externa o mediante batería) y distintos tipos de arranque (microSD, Quad SPI Flash y JTAG), pudiendo seleccionarse estos ajustando los jumper asociados. Incluye además dos componentes de memoria Micron MT41K256M16HA-125 DDR3L que crean un único rango de interfaz de 32 bit de ancho con un total de GiB de capacidad.

La Zybo proporciona un reloj de 33.33MHz al reloj de entrada al PS usado para generar los relojes para cada uno de los subsistemas permitiendo al procesador operar a una frecuencia máxima de 667MHz y al controlador de memoria DDR3 a una tasa máxima de reloj de 533MHz. Los ficheros Zybo que proporciona el Centro de Recursos Zybo Z7 configuro automáticamente el IP core del PS del Zynq en Vivado para poder trabajar con todos los dispositivos incluidos en la PS.

Finalmente cabe destacar que la PS dispone de un PLL dedicado capaz de generar hasta cuatro relojes de referencia, cada uno con frecuencias ajustables, que se pueden emplear para personalizar los relojes de la lógica implementada en la PL.

Figura 4: Plataforma Zybo Z7-20

En este proyecto se empleará el modo de arranque mediante microSD donde se introducirán los ficheros necesarios para arrancar el SO PetaLinux en la Zybo Z720.

B) Zynq 7000 de la familia Xilinx:

En lo referido a la arquitectura del APSoC Zynq, esta se puede dividir en dos subsistemas distintos: la parte de procesado del sistema definida como PS ("Processing System", coloreado en verde en la figura inferior) y la parte de lógica programable que referencia el FPGA ("*Programmable Logic*", representada en amarillo).

En este caso el SoC Zynq no proporciona el controlador PCIe Gen2 ni los transceptores Multi-gigabit en las arquitecturas de la plataforma Zybo, ni Z720 ni Z710.

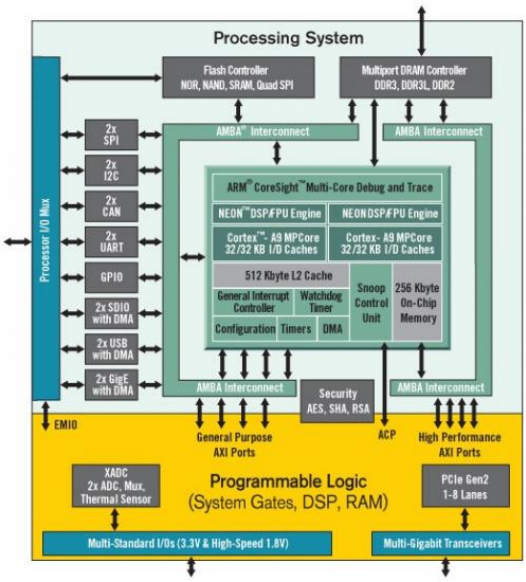


Figura 5: Esquema del SoC Zynq 7000

Para esta plataforma la lógica programable es casi idéntica a la del FPGA de las 7-series Artix, a diferencia de que contiene múltiples puertos dedicados y buses fuertemente unidos al PS. Además, no contiene la misma configuración HW de las FPGA 7-series y debe ser configurada directamente a través del procesador o mediante un puerto JTAG.

Por otro lado, la parte de PS dispone de diversos componentes, entre ellos la unidad de procesado de aplicación (APU, "*Application Processing Unit*") que incluye el doble-núcleo con procesadores Cortex A9, una arquitectura de interconexión avanzada (AMBA, "*Advanced Microcontroller Bus Architecture*"), controlador de memoria DDR3 y varios controladores de periféricos con sus entradas y salidas multiplexadas.

La conexión con los periféricos se realiza mediante un protocolo maestro-esclavo y lo mismo para conectar la FPGA. Además, los núcleos implementados en la PL pueden provocar interrupciones al procesador o bien realizar accesos vía DMA a memoria DDR3.

Para configurar de manera correcta la PS para funcionar con los periféricos se pueden usar los ficheros que proporciona Vivado en el centro de recursos Zybo Z7.

C) Módulo PCam 5C de Digilent:

El módulo PCam 5C es un módulo de imagen diseñado para usarse con placas de desarrollo FPGAs. Este incorpora un sensor de imagen en color Omnivision OV5640 de 5 megapíxeles (MP) y dispone de funciones de procesado interno para mejorar la calidad de la imagen. Es un módulo pequeño donde el PCB tiene 40 mm de largo y 25 mm de ancho.

Figura 6: Módulo PCam5C

Los datos se transfieren mediante una interfaz MIPI CSI-2 doble carril que proporciona el ancho de banda suficiente para admitir los formatos comunes de transmisión de video como 1080p y 720p.

La conexión a la placa de desarrollo Zybo Z7-20 se realiza mediante un cable plano flexible de 15 pines (FFC, "*Flexible Flat Cable"*), el cual viene incluido. Además de incluir el cable FFC, el módulo PCam 5C dispone de una lente de enfoque fijo con montura de lente M12, lo que permite su uso inmediato.

La conexión es sencilla puesto que la Zybo Z7-20 dispone de un adaptador para la conexión mediante FFC, por lo que sencillamente hay que insertar el cable con la cara azul apuntando al centro de la plataforma como se muestra en la [figura 7].

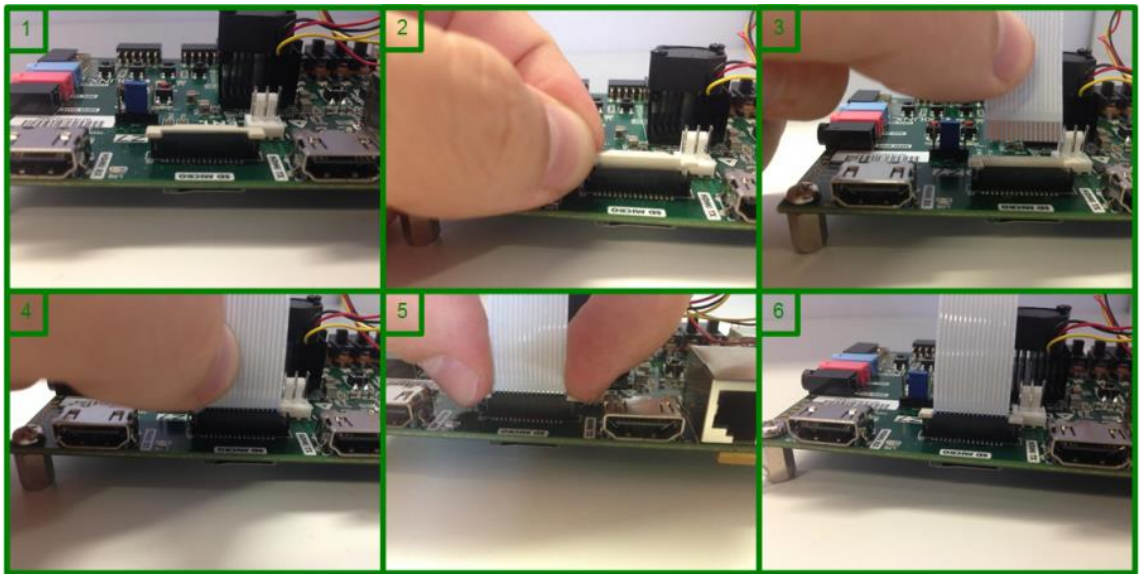


Figura 7: Montaje del módulo PCam5C en la Zybo Z7-20 [17]

Otras de las características de este módulo es que los formatos de Salida incluyen RGB565, YUV422/420 y YCbCr422 y compresión JPEG.

La particularidad de este módulo es que solo es controlable mediante un SoC con controlador MIPI CSI-2 en hardware, requiriendo además un software complicado para la configuración del controlador y el sensor. Por suerte, para facilitar al usuario el trabajo con este módulo, Digilent ha creado ciertos conjuntos de núcleos IP de Vivado de código abierto para aquellas FPGAs que constan de un SoC Zynq.

D) Sistema completo hardware:

En esta sección se introduce el sistema hardware de tiempo real al completo. Este se compone por el módulo de cámara PCam5C introducido en el apartado anterior, conectado mediante un cable FFC al puerto PCam MIPI CSI-2, un cable HDMI conectado al puerto de salida HDMI de la plataforma Zybo Z720 y un cable conectado desde el puerto USB/JTAG de la Zybo al USB del ordenador para poder establecer posteriormente la comunicación puerto-serie.

En la [figura 8] se observa el montaje del sistema al completo. En esta se observan numerados los distintos elementos que lo componen:

- El ❶ es el monitor por el cual se reproducirán las imágenes capturadas por el módulo cámara PCam5C.

- El indicado con ❷ representa la conexión entre el cable HDMI del monitor y el puerto HDMI de salida de la plataforma Zybo Z720. Es importante no equivocarse de puerto HDMI puesto que la plataforma dispone de dos puertos, uno para el flujo de entrada y otro para flujo saliente, y en este trabajo, puesto que se van a reproducir las imágenes capturadas por la cámara, el puerto HDMI que interesa es el de salida.
- El ❸ representa el conexionado entre el módulo PCam5C y el puerto MIPI CSI-2 mediante el cable FFC incorporado a este primero.
- Con el ❹ se muestra la conexión entre el puerto JTAG/USB de la Zybo y el puerto USB del ordenador, empleado para establecer comunicación entre el ordenador y la plataforma.
- La plataforma se puede alimentar de dos formas distintas: mediante la conexión JTAG/USB de la plataforma al USB del ordenador o bien mediante una alimentación externa de 5V. El fabricante recomienda el uso de la alimentación externa puesto que la máxima corriente que da la alimentación mediante USB son 0.5 A y la plataforma requiere más. Por ello se emplea un alimentador externo centro positivo de 2.1mm de diametro de conexionado interno que proporciona entre 4.5 y 5 voltios DC ("*Direct Current"*), al menos 2.5 amperios y 12.5 watios, con la finalidad de proyectos de alto consumo de energía en la plataforma y la lectura de periféricos externos. ❺

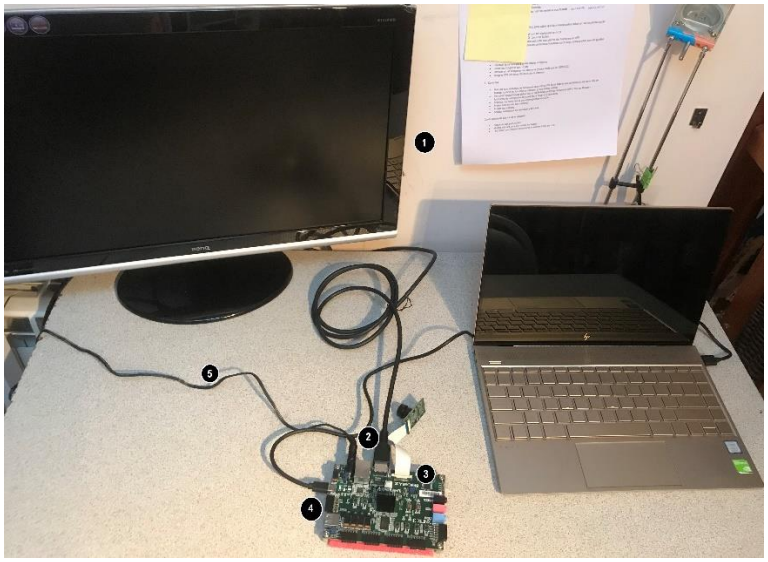


Figura 8: Montaje del sistema completo hardware

4.4 Diseño realizado y propuesta:

Una vez expuesto el contorno de trabajo así como las herramientas software y el hardware a emplear, en esta sección se va a exponer el diseño realizado para la implementación del algoritmo PCA aplicado a la detección de objetos en movimiento en un dispositivo SoC. Se va a aplicar una vista general *top-down* explicando lo que se ha realizado: optimizar el algoritmo para un video pregrabado, realizar la captura de imágenes por el módulo PCam 5C y extraerlas por el HDMI, trasladar el algoritmo PCA a esta captura de imágenes en tiempo real, como se observa en la [figura 9]. Se introduce al algoritmo un video pregrabado y un conjunto de imágenes con intención de obtener un punto de partida, observando cómo se comporta el sistema de manera inicial. Se le aplicarán distintos métodos para optimizar este resultado inicial y tras esto se pasará a usar el algoritmo optimizado obteniendo las imágenes de una cámara en tiempo real. Además, para observar el funcionamiento del sistema, las imágenes una vez procesadas se extraerán hacia un monitor. De esta manera, si se detecta un nuevo objeto en la escena, se apreciará en el monitor.

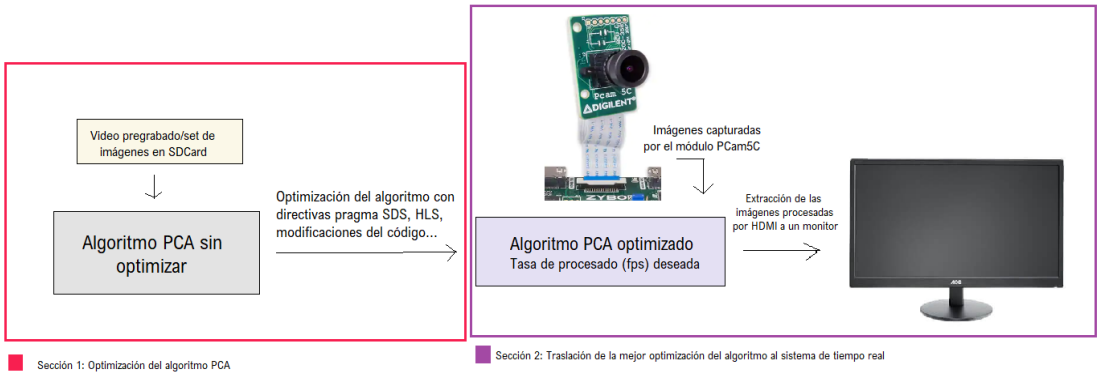


Figura 9: Diagrama de flujo de la visión top-down del diseño propuesto

Este capítulo se va a dividir en dos secciones. A lo largo de la primera sección se explicarán uno a uno los distintos métodos que se han empleado para alcanzar la meta de optimización del algoritmo, que como se explicaba en el capítulo 4.1.1, esta es la obtención de una tasa de procesado de las imágenes de 25-30 *frames* por segundo (*fps*). Al finalizar esta sección se mostrará una comparativa de los distintos métodos y las tasas obtenidas para cada uno y la opción con la cual se obtuvo mejor resultado se elegirá como candidata para la aplicación del algoritmo en el sistema de tiempo real.

Esta elección se trasladará a la sección 2, [figura 9], donde se utilizará para observar los resultados en el sistema de tiempo real. Introduciendo en este caso las imágenes capturadas por el módulo PCam5C y tras aplicar el algoritmo se muestran las imágenes procesadas en un monitor, gracias al conexionado HDMI entre este y la plataforma.

4.4.1 Optimización del algoritmo PCA y resultados:

Previa a la optimización de algoritmo, se requiere su codificación, su obtención de los denominados "*Golden Data"* y su comprobación del correcto funcionamiento. Para ello, basándose en el trabajo previo [6], se realizó una codificación inicial en Matlab. Puesto que es parte del trabajo [6], la codificación del algoritmo no es trivial, pero se requiere para obtener los resultados particularizados en este escenario. Además, tras la codificación en Matlab, el escenario se trasladará a una nueva interfaz antes de llevarse al software final de trabajo, SDSoC. Esta interfaz es Eclipe IDE. La elección de esta interfaz se debe a dos razones. Estas son que, al igual

que Matlab, proporciona resultados visuales en la ejecución del algoritmo, cosa que SDSoC no proporciona, y que la interfaz de SDSoC emplea una adaptación de la IDE de Eclipse, de manera que la exportación del código fuente entre Eclipse y SDSoC es directa.

La [figura 10] muestra de manera gráfica el flujo explicado en el párrafo anterior, que describe los pasos seguidos antes de realizar las pruebas y sumergirse en el entorno de SDSoC.

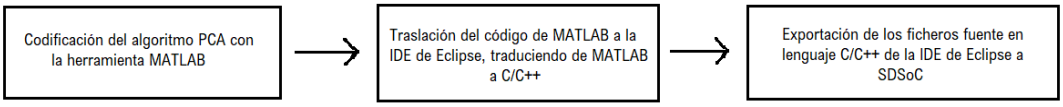


Figura 10: Flujo de la secuencia de trabajo previa a SDSoC

A diferencia de [6], puesto que lo que se desea es obtener una optimización del algoritmo dónde la latencia disminuya y la velocidad de procesado de imagen aumente, se omite en este trabajo la actualización de fondo dejándolo como una futura línea. Esta actualización de fondo se realiza puesto que un escenario de fondo puede verse sometido a ciertos cambios, como la iluminación o la introducción de nuevos elementos estáticos, que modifican las componentes principales que lo componen.

Como lo que se pretende es la aceleración de la parte *offline* y la parte *online* del algoritmo, la cual se explicó previamente en el capítulo 4.3.1, la actualización de fondo es solo un añadido en el que se volvería a ejecutar tras cierto tiempo la parte *offline*.

La codificación con la herramienta Matlab resulta ser muy importante puesto que esta permite la visualización de los resultados intermedios permitiendo así asegurar que los valores que se están obteniendo pertenecen a los rangos que se tienen que obtener como resultado, permite observar las dimensiones de las matrices de datos, y facilita ver el punto en el que se pueda estar cometiendo el error. Además, proporciona una primera aproximación visual del resultado esperado del algoritmo.

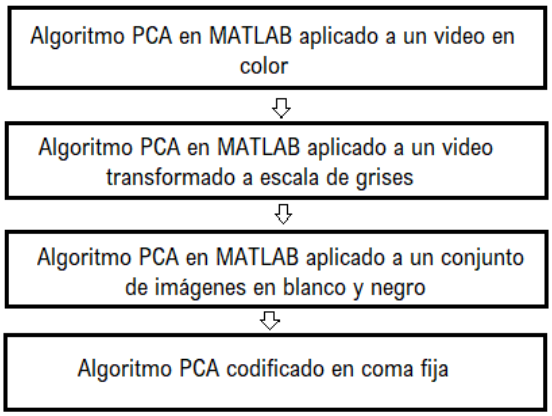


Figura 11: Diagrama de flujo de la codificación con la herramienta MATLAB

En la [figura 11] se adjunta un diagrama que muestra los pasos seguidos en la codificación con la herramienta Matlab. Dado que la cámara escogida posee la opción de trabajar con el video en color, la primera opción que se consideró y probó en esta herramienta fue el tratamiento de imágenes en color. Por ello se comenzó trabajando con un video en color grabado con un teléfono móvil, con dimensiones 1920x1080. En este video se observa un fondo estático de un jardín formado por césped, arbustos y árboles, y será a lo largo del video que se observe una persona paseando por este, en varias ocasiones [figura 14].

Esta primera opción fue finalmente descartada al requerir el triple de tiempo de cómputo al tener que tratar los 3 canales RGB individualmente. Es decir, para poder aplicar el algoritmo PCA en imágenes en color se requería separar los tres canales y someter cada uno de ellos a la parte *offline* y *online.* La comprobación se realizó con Matlab, observando que el error cuadrático medio entre la utilización de PCA en color y en escala de grises era inferior al 5%, por lo que no compensaba el aumento computacional. Es por esta razón por la que se realiza una conversión del video del espacio de color a escala de grises.

Las dimensiones del video también resultan en un aumento computacional puesto que dan lugar a matrices de dimensiones excesivamente grandes, por lo que se cambió el tamaño a 256x256. La elección de estas dimensiones se debe a [6], junto con la elección de utilizar un conjunto de 8 imágenes de fondo. De esta manera, a pesar de procesar un video en lugar de un *set* de imágenes, el punto de partida de optimización del algoritmo será [6].

El siguiente paso dentro del flujo [figura 11] fue la codificación del algoritmo PCA realizando una conversión a escala de grises de los *frames* del video según se leían.

Al codificar el código con Matlab, a pesar de no ser algo trivial, se tomaron ciertas medidas y consideraciones que variaron con respecto a [6]:

- Para mayor control sobre el algoritmo se desea tener acotados los datos y evitar que tomen valores demasiado altos o tengan tamaños excesivos. Por ello se realiza una normalización. Dos son las posibles opciones a la hora de normalizar: con respecto al máximo del rango de valores o bien con respecto al máximo de la matriz de transformación [7, punto 4.2.5, página 176]. Ambas se evaluaron con Matlab, decidiendo finalmente normalizar con respecto al máximo de la matriz de transformación al obtener resultados con un error cuadrático medio entre ellos de 2,97% e imágenes con visualmente menos píxeles de ruido.
- Al observar la codificación del algoritmo de [6], se observó que a la hora de calcular los autovectores de la matriz de covarianza se realizaba una modificación de algunos de los signos de estos. Investigando las funciones que se usan para el cálculo de los autovalores y autovectores de una matriz, se descubrió que Matlab dispone de varias funciones para el cálculo de estos, entre ellas "*Eig"* y "*SVD"*. Ambas devuelven la matriz de autovalores (la diagonal) y la matriz de autovectores. Al comparar ambas matrices, la matriz de autovalores solo difiere en el orden de los mismos (*"Eig"* devuelve la matriz ordenada de menor a mayor mientras que "*SVD*" la ordena de mayor a menor) mientras que las matrices de autovectores difieren en el signo de alguno de estos. Este cambio de signos entre funciones de cálculo de autovectores no afecta realmente puesto que es simplemente la multiplicación por un entero, -1 en este caso. Es decir, ambas funciones difieren en el hecho de que una es una versión escalada de la otra. Matemáticamente, un escalado de los autovectores no afecta al resultado final, y para demostrarlo se realizó la la comparación con Matlab de los dos escenarios para tres videos distintos. En esta compobación se observó el error cuadrático medio que se obtenía entre la versión del algoritmo realizando la comprobación de los signos de los autovectores modificándolos de manera que los signos de SVD y "*Eig"* fuesen los mismos y la versión del algoritmo sin esta comprobación ni modificación.
  La única diferencia entre estos dos escenarios residía en estos signos de los autovectores puesto que el RMSE ("*Root-mean square error",* Raíz del error cuadrático medio), el error de reconstrucción obtenido (distancia euclídea entre pixeles) y los valores de la imagen proyectada en la dimensión transformada resultaban ser los mismos para ambos

escenarios. Es decir, el error cuadrático medio entre escenarios (con comprobación y sin comprobación) era nulo.
Esto resultó interesante comprobarlo puesto que los signos de los autovectores varían según el video que se capture, no difiriendo siempre el signo de las mismas columnas de la matriz. Hubiese causado mayor complejidad al exportar el algoritmo a la captura de video en tiempo real al requerir hacer esta comprobación para cualquier nuevo escenario de fondo.

Tras obtener la codificación del algoritmo para un video pregrabado, se decidió probar a codificar el algoritmo para la lectura de un conjunto de imágenes, con la intención de observar el algoritmo para distintas entradas (tercer escalón del *"Diagrama de flujo de la herramienta MATLAB"* [figura 11]). El hecho de codificar el algoritmo de esta forma resultó beneficioso a posteriori como se explica más adelante al explicar los distintos métodos de optimización al usar la herramienta SDSoC.

El último paso del flujo con Matlab fue la codificación del algoritmo empleando el tipo de dato en coma fija puesto que, a pesar de que este tipo de codificación implica pérdida de precisión, se considera que puede ser una opción viable para optimizar el algoritmo. No solo en términos de la utilización de recursos, ya que la coma fija requiere menos, sino también para observar si produce optimización de la latencia.

Como se explica en el capítulo 4.3.2, la coma fija requiere determinar previamente el número de bits dedicados para la parte decimal y para la parte entera, ya que una elección errónea de estos puede llevar a errores. La codificación con Matlab del algoritmo en coma fija resultó trivial, a la par de complejo, puesto que como se expone en párrafos anteriores esta herramienta permite la visualización numérica de las variables intermedias empleadas en el algoritmo, lo cual sirve como primera aproximación para observar el error cometido.

Gracias al segundo paso del flujo de la [figura 11] se han obtenido los "*Golden data"* y estos serán el punto de comparación al codificar con coma fija. La idea es obtener entre los resultados deseados, *"Golden data"* y los resultados obtenidos con la coma fija un error cuadrático medio tolerable, siendo este error el criterio final para decidir el número de bits asignado a cada parte.

Para implementar el algoritmo en coma fija se duplicaron todas las variables creadas. A la copia duplicada de cada una de las variables se les cambió el nombre introduciendo un sufijo "*-fixed*" y se cambió el tipo de dato gracias al operador "*fi*" de la toolbox de Matlab "*Fixed point"*.

Este operador "*fi"* [20] se emplea para asignar un tipo de dato en coma fija a un número o una variable. Este operador devuelve un objeto "*a*" de manera que:

```
a = fi(v,s,w,f)
```

Le asigna el valor indicado en "*v*", con signo o sin signo en función de lo indicado en la variable "*s*", con la longitud de palabra (en bits) dada por "*w*" y con número de bits para la parte fraccional indicados en "*f*".

De manera teórica, como se indica en el capítulo 4.3.2, se obtuvo el número de bits requeridos para determinar las distintas variables que se deseaban declarar en coma fija a lo largo del algoritmo. Esto se hizo determinando el rango de valores que representaba cada una de ellas y usando la [ecuación 1], capítulo 4.3.2.

Para determinar el error entre ambos formatos de cada una de las variables se empleó el error medio porcentual de la diferencia, y posteriormente la elección del número de bits se realizó en función del error cuadrático medio. La elección de longitud de palabra se tomó respetando los rangos permitidos por los multiplicadores internos de la parte de la FPGA de la Zybo. Estos no

se han nombrado previamente pero las longitudes de sus operandos son de 18 y 25, dando un máximo de bits de representación a la salida de 43.

Los resultados obtenidos de error porcentual para las distintas variables bajo evaluación fueron los siguientes:

- La matriz de media nula, A, para la cual se obtuvo un error porcentual del 0% con una longitud de palabra de 18 bits con signo y una parte fraccional de 4 bits.
- La matriz de covarianza, C, para la cual se obtuvo un error porcentual del 5,12% con una longitud de palabra de 20 bits con signo y una parte fraccional de 2 bits.
- El valor del RMSE para el cual se obtuvo un error porcentual del 0.0013%.
- El error entre imágenes proyectadas para el cual se obtuvo un error porcentual del 2,76% con una longitud de 39 bits con signo y 12 bits de parte fraccional.

Con estos errores porcentuales se obtuvo el error cuadrático medio, el cual resultó ser de 2,6186%. Esta diferencia entre los valores codificados en coma flotante y los valores codificados en coma fija resultó ser muy bajo y tolerable, de manera que se emplearon las longitudes nombradas en la lista anterior al implementar esas variables en coma fija.

Para todos estos escenarios planteados con Matlab no solo se observan los resultados numéricos sino que también se ha decidido imprimir por pantalla ciertos resultados visuales [figura 12]. En la columna de la izquierda se muestran la denominada como "*imagen original"* siendo esta el *frame* que se está introduciendo a la parte *online* del algoritmo (el *frame* que sobre el cual se examina si existe nuevo objeto o no), y la denominada como *"imagen reconstruida"* la cual representa la escena de fondo con la cual se realiza la comparación.

Por otro lado la columna de la derecha muestra otras dos imágenes, "*imagen umbralizada"* e "*imagen umbralizada y filtrada"*. Como bien indica el nombre, ambas imágenes están umbralizadas, esto es que solo se representa en blanco aquellos píxeles para los cuales la distancia euclídea, o bien error de reconstrucción (capítulo 4.3.1, "*Parte online del algoritmo"*) supera cierto umbral. Este umbral se decide en función del escenario de fondo, en este caso se introdujo un umbral de 60 en la escala de grises. Es decir que en ambas dos imágenes se observará con píxeles en blanco el objeto detectado.

La diferencia entre estas dos imágenes es que la denominada como "*filtrada"* aplica un filtrado morfológico de erosión y dilatación para recalcar la figura del elemento detectado y eliminar pixeles ruidosos no pertenecientes a este. Este filtrado elimina parte del objeto detectado como se puede observar comparando ambas imágenes, pero al haber establecido un umbral de detección de objeto basado en el error de reconstrucción, esta diferencia de píxeles no afecta a la decisión.

Una vez codificadas con Matlab las distintas versiones del algoritmo PCA y obtenidos los resultados parciales y finales, tanto de manera numérica como visual [figura 12], como se muestra en el diagrama de flujo de la [figura 10], al disponer SDSoC de una interfaz basada en la IDE de Eclipse, se decidió traspasar el código de Matlab a C/C++ en una máquina virtual que disponía de Eclipse. Para traspasar este código se tuvieron que implementar aquellas funciones específicas de Matlab que no están disponibles en C. Estas funciones no disponibles de manera directa en lenguaje C/C++ son todas aquellas que requieren de cálculo matricial puesto que, a diferencia de Matlab, en este lenguaje de alto nivel se requieren realizar las operaciones elemento a elemento. Es decir que se implementaron todas las funciones cuyas variables de entrada eran matrices:

- Cálculo de la media de la matriz de imágenes de fondo (A).
- Cálculo de la matriz de elementos de media nula: esta es la resta de la media de la matriz de imágenes de fondo a la propia matriz de imágenes de fondo.
- Cálculo de la matriz transpuesta.

- Cálculo de la multiplicación entre dos matrices.
- Cálculo de la distancia euclídea entre dos imágenes, puesto que estas vienen representadas por matrices.

Para el cálculo de los autovalores y autovectores se aprovechó una función ya implementada del algoritmo "*SDV"*. Esta fue usada también en [6]. Se valoró también usar la librería de código abierto "*Eigen"* que proporciona una variación de funciones de algebra lineal pero se descartó la idea puesto que suponía complicar la codificación del algoritmo al tener que declarar nuevas plantillas para las variables y como se comentó al inicio del capítulo: la codificación del algoritmo no es lo trivial del trabajo.

El resto de las funciones como la apertura del video y la lectura de cada uno de los *frames* que lo conforman, y la representación de las imágenes de manera visual, además de la aplicación de la umbralización y el filtrado de erosión y dilatación, se trasladan a Eclipse de manera directa puesto que existe la librería de visión artificial de código abierto "*OpenCV"*.

Esta librería de visión se emplea en una amplia variedad de áreas como la realidad aumentada, el reconocimiento de objetos, la interacción persona-computadora y el seguimiento de objetos, entre otras. "*OpenCV"* se desarrolla completamente en lenguaje de alto nivel C++ pero incluye conectores para otros lenguajes como lo es Matlab. Es esta la razón por la que se emplean las mismas llamadas a las funciones de visión tanto en Matlab como en Eclipse.

Como resumen, la exportación de Matlab a Eclipse es directa para aquellas funciones de la librería "*OpenCV"* e indirecta para aquellas funciones que requieren cálculo algebraico y matricial, es decir, el resto de las funciones que realizan los cálculos de los procesos *offline* y *online* del algoritmo [capítulo 4.3.1] requieren su codificación manual.

Al realizarse aprovechó este traspaso para ordenar el algoritmo mediante la agrupación en funciones de los distintos pasos [capítulo 4.3] para facilitar posteriormente en SDx el movimiento de estas entre la ejecución en SW y ejecución en HW. Esta agrupación se hizo de manera que la codificación final quedaba simplificada mediante llamadas a funciones declaradas en otros ficheros fuente. El hecho de que esto facilite el movimiento en SDx se debe a que al estar dividido el algoritmo en funciones se puede seleccionar cada una de ellas de manera individual para destinar su ejecución bien a la lógica programable del HW o al microcontolador (SW).

Aquí es donde el hecho de haber obtenido los *"Golden Data"* con Matlab toma protagonismo. La IDE de Eclipse en el modo depuración permite observar los resultados parciales que se obtienen de las distintas variables lo que, al comparar con los *"Golden Data"*, sirve como comprobación del correcto funcionamiento.

Algo que se apreció mediante la comparación de los "*Golden Data"* de Matlab con los resultados parciales obtenidos mediante la IDE de Eclipe fue el hecho de que inicialmente no se obtenía la misma matriz que conformaba las imágenes de fondo. Resultó curioso puesto que el video que se estaba introduciendo era el mismo y se leía de la "*misma forma"*. Ambas interfaces empleaban la función de lectura de video "*VideoCapture"* pero se observó que esta función devolvía dos *frames* leídos ligeramente diferentes. El *frame* devuelto por Eclipse era en una escala de grises que difería dos pixeles por debajo de la escala de grises del *frame* devuelto por Matlab. Fue importante darse cuenta de esta diferencia puesto que una pequeña variación en esta matriz supone como resultado matrices de covarianza distintas que dan lugar a distintos autovalores y autovectores. Por ello, se exportó la matriz de covarianza obtenida con el IDE de Eclipse y se llevó de vuelta a Matlab con intención de obtener los "*Golden Data"* para esa matriz de covarianza.

Una vez obtenidos los nuevos *"Golden Data"* de Matlab para la matriz de covarianza obtenida con la IDE de Eclipse, estos se compararon con el resto de las variables obtenidas en esta segunda

interfaz. El error cuadrático medio obtenido entre ambos resultados (los "*Golden Data"* de Matlab y los obtenidos con la IDE de Eclipse) resultó ser inferior al 3%, permitiendo exportar los ficheros fuente a la herramienta final SDSoC.

Al igual que para Matlab, Eclipse permite obtener resultado visuales gracias al empleo de la librería de visión "*OpenCV*", como se nombraba en párrafos anteriores.

Los resultados visuales que proporciona Eclipse se muestran en la [figura 13]. En esta se muestran las mismas imágenes que las mostradas para Matlab [figura 12], observando que los resultados son aproximadamente iguales, con un error inferior al 3% como se comentaba previamente.

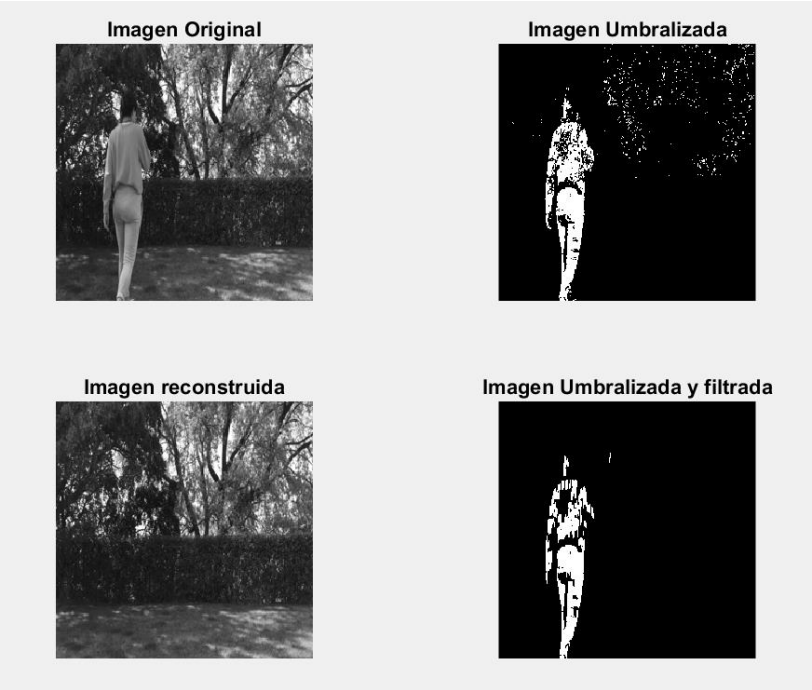


Figura 12: Resultados de PCA con Matlab

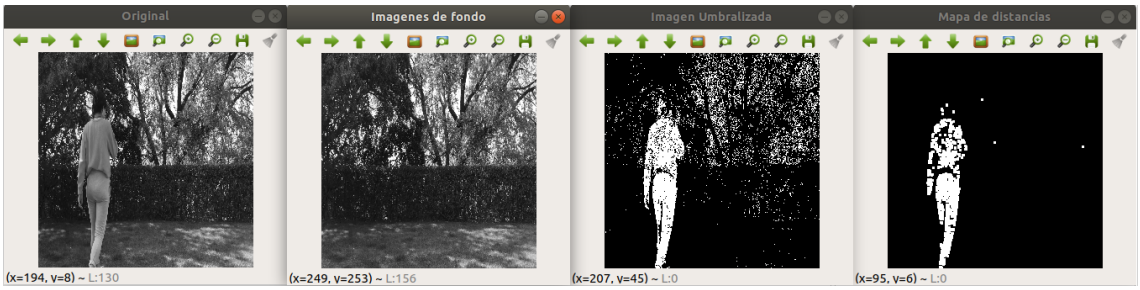


Figura 13: Resultados de PCA con la IDE de Eclipse

Una vez obtenida la codificación del algoritmo en C/C++ con Eclipse, se exportó el proyecto al software SDx, siguiendo los pasos explicados en el [capítulo 5, *Creación de un nuevo proyecto*].

Una vez trasladado a SDSoC el código funcional del algoritmo PCA que se ha implementado en la IDE de Eclipse es cuando realmente comienza el objetivo de este trabajo.

Como se comentaba en la introducción, y de manera breve se ha nombrado a lo largo del trabajo hasta este punto, la primera meta por conseguir de este trabajo es conseguir acelerar el algoritmo PCA. El término "acelerar" hace referencia a conseguir una velocidad de procesado de imágenes superior a la conseguida en una primera aproximación, donde no se realiza ninguna modificación del algoritmo, ni se incluyen las directivas pragmas [capítulo 4.3.3, "*SDSoC*"], y no se traslada

ninguna aplicación a HW para aprovechar su paralelismo.Esta primera aproximación dará el punto de partida, es decir la velocidad de procesado inicial que se pretende aumentar.

Algo sobre lo que no se ha hecho demasiado hincapié previamente es el hecho de que la plataforma Zybo permite la implementación de un sistema operativo (SS.OO) que arranque desde la SD Card. Esto se consigue gracias a la copia de la imagen del SS.OO y el fichero de arranque dentro de la memoria externa SD Card.

Como se comentaba al describir los distintos SW que se utilizan en este trabajo, SDSoC no dispone de la plataforma Zybo Z720 dentro de las plataformas básicas del programa. Esta fue la razón por la que se recurrió a una plataforma pre-diseñada que el propio fabricante de la plataforma, Digilent, pone a disposición del usuario.

Esta plataforma no solo dispone de las características HW y sus restricciones creadas con Vivado Design Suite sino que también dispone de un sistema de archivos de un sistema operativo Linux, de manera que este trabajo se aprovecharán las ventajas que proporciona, siendo la más importante la comunicación directa con la plataforma mediante la comunicación serie UART con el terminal de Linux. La introducción de la plataforma dentro del software es un aspecto más técnico que viene detalladamente explicado en el [capítulo 5, "*Pliego de condiciones"*]. Otra de las razones por la que se decide el uso del sistema operativo Linux que contiene la plataforma prediseñada en lugar de la utilización de una versión *standalone* se debe a que si no se tuviese sistema operativo habría que montar un sistema de archivos propio o introducir el video/conjunto de imágenes a memoria interna mediante algún otro procedimiento. La implementación de este sistema de archivos para el método *standalone* requeriría mucho tiempo y puesto que la finalidad real del trabajo es la optimización de un algoritmo dentro de la plataforma, esto se deja como trabajo futuro.

Cabe destacar en este punto también que dentro del sistema operativo de la plataforma se incopora una versión de la librería "*OpenCV*" pensada para ejecutar estas funciones de visión artificial en la lógica programable del HW, permitiendo acelerar su ejecución. Esta librería se llama "*xfOpenCV"* y es una librería que Xilinx proporciona para las plataformas que incluyen sus SoC's. No todas las funciones de "*OpenCV*" están implementadas en la librería "*xfOpenCV"* pero muchas de ellas si se pueden encontrar. Algunas de estas son la lectura y escritura de imágenes, la detección de esquinas usando "*Harris",* el filtrado mediante operaciones morfológicas de dilatación y erosión, conversiones de color, y redimensionado.

Volviendo al tema del trabajo, para poder determinar la velocidad de procesado del algoritmo se ha decidido emplear el método empleado en [6].Este es el uso de la comunicación mediante el puerto serie de la UART de la Zybo y el terminal de SDx y el comando "*time"* del que dispone el sistema operativo Linux empotrado dentro del SoC. Se introducirá en el terminal el comando *"time -p ./<proyecto>.elf IMG_xxxx.MOV"* de manera que se obtendrá, tras la ejecución de la aplicación, tres medidas:

- Real: Esta medida representa el tiempo transcurrido desde el inicio hasta el final de la llamada, incluye los segmentos de tiempo utilizados por otros procesaos y el tiempo que el proceso está bloqueado. Es la medida que se empleará para observar la aceleración.
- User: Esta medida representa la cantidad de tiempo de CPU real consumido en la ejecución del proceso.
- Sys: Esta medida representa la cantidad de tiempo de CPU que se pasa en el núcleo dentro del proceso, es decir el tiempo de CPU gastado en las llamadas al sistema *dentro del núcleo.*

Una vez obtenida estas medidas de tiempo, para obtener la velocidad de procesado en *frames* por segundo, se realizará la división entre el conjunto total de *frames* leídas a las que se ha aplicado el algoritmo y el tiempo real.

El conjunto total de *frames* leídas viene representado solo por aquellas *frames* a las que se le aplica la parte *online* del algoritmo, ya que esta es la parte que se pretende optimizar. Por ello se realizaron 10 ejecuciones de la parte *offline* del algoritmo y con esto se calculo la media de tiempo que tardaba en ejecutar esta: 1,56 segundos. Este tiempo se restará a la media del tiempo total de ejecución del algoritmo en las pruebas que se encuentran más adelante para poder obtener la tasa de procesado de la parte *online* únicamente.

Se incluyó además una sentencia dentro del código que imprimía el número de *frame* sobre el cual se estaba aplicando el procesado con la intención de observar el número de *frames* totales que conformaban el video/conjunto de imágenes que se ha usado. El total de *frames* sobre las que se aplica el algoritmo son 2181 si se emplea el video y 2173 si se emplea el conjunto de imágenes.

Para los posteriores apartados cabe destacar que las condiciones entre [6] y este trabajo no son las mismas al trabajar con recursos diferentes, tanto la plataforma (se emplea la Zybo Z720 en vez de la ZedBoard) como en el conjunto de imágenes (extraídas de un video en vez de ser un set de imágenes directamente en escala de grises.), por lo que los resultados no tienen que ver.

Además de la velocidad de procesado, se puede observar gracias a la herramienta de perfilado que proporciona la depuración mediante el *"TCF Agent"*, el "*TCF Profiler"*, cuál de las funciones implementadas requiere más tiempo en ejecutarse en la CPU del microprocesador. Esto va a servir para sacar las candidatas para la aceleración mediante hardware, puesto que aquellas que más tiempo de CPU consuman serán las que se desee paralelizar para evitar bloqueos muy largos de las tareas en la ejecución. Si se evitan estas esperas largas, se obtiene una aceleración del algoritmo.

Para observar los resultados del "*TCF Profiler"* se debe compilarel programa y, pasándole el video como entrada al algoritmo, se realiza la depuración, indicando un *"breakpoint"* antes del "*return"* que finaliza la aplicación. Esto es necesario puesto que sino los resultados proporcionados por el *TCF Profiler* no son observables puesto que da por finalizada la depuración.

Para aquellos escenarios donde se realiza la ejecución de ciertas funciones en la lógica programable del FPGA, se mostrarán además la estimación de recursos empleados y la aceleración que proporciona para la propia función y el conjunto total del algoritmo.

De esta manera se introducen las tres medidas que se han examinado y gracias a las cuales se han elegido distintas formas de optimizar el algoritmo:

- Velocidad de procesado de las imágenes, obtenida mediante la división del conjunto total de *frames* y el tiempo que tarda en ejecutar la parte *online* del algoritmo.
- Tiempo consumido por las distintas funciones en la CPU del microprocesador puesto que aquellas que más tiempo requieran serán las que se desee paralelizar en HW.
- Estimación de los recursos empleados por el HW y la aceleración estimada que proporciona la ejecución en la lógica programable en la ejecución total del algoritmo.

Llegados a este punto, y antes de empezar a mostrar los resultados obtenidos para los distintos métodos de optimización, se va a hacer una pequeña puntualización con respecto a la compilación que se ha empleado en SDx:

- SdSoC dispone, como se ha comentado al explicar el software, de dos modos de compilación: *"release"* y "*debug"*. Ambos métodos representan la forma en la que el

código software se compila, no afectando ni a la compilación ni a la implementación de las funciones en hardware.

La elección del método de compilación para este proyecto resulta trivial puesto que ambos métodos presentan beneficios diferentes:

- El modo de compilación *release* se emplea para reducir el tamaño del código y el tiempo de ejecución de la aplicación. Esto se consigue puesto que elimina la parte de código de depuración, de manera que con este modo de compilación no se puede realizar una ejecución paso a paso del algoritmo. Además, también elimina la información simbólica.
- El modo de compilación *debug* proporciona información adicional para facilitar la depuración de la aplicación y permitir recorrer el código paso a paso.

Es el primer modo, el modo *release*, el que se ha elegido para este proyecto, puesto que realmente se quiere ver la ejecución de la aplicación y obtener tiempos de ejecución menores, lo que proporciona mejor tasa de procesado. Si es cierto que, a lo largo del trabajo, ciertas pruebas han dado violación de segmento u otros errores para lo cual se ha requerido la versión de depuración.

Además, la prueba inicial requiere del uso de la versión *debug* de compilación para poder situar un punto de ruptura al final del algoritmo y poder visualizar gracias al *TCF Profiler* cuál de las funciones requiere más CPU.

Finalmente cabe destacar que todas estas pruebas se van a realizar a la frecuencia máxima de reloj de interfaz de memoria para el movimiento de datos permitida por la plataforma Zybo Z7-20 de133MHz.

Ahora sí, una vez explicadas las medidas que se van a realizar para observar la aceleración del algoritmo y el método de compilación empleado, se van a mostrar las distintas aproximaciones que van a ser examinadas para la finalidad de este trabajo, la aceleración y optimización del algoritmo PCA de procesado de imágenes, y finalmente se mostrará una tabla que contendrá los diferentes resultados de los distintos escenarios para comparar:

1. **Primera aproximación: Ejecución en software del algoritmo completo con *cv::imread***

En esta primera aproximación se van a obtener los resultados que se han denominado anteriormente como punto de partida. En este caso se va a ejecutar todo el algoritmo en la CPU del microcontrolador.

Cuando no se envía ninguna función para ser ejecutada mediante la lógica programable de la plataforma, no es posible realizar una estimación de recursos ni de aceleración conseguida con HW, de manera que en esta primera aproximación se realizará la toma de decisiones en función de la velocidad de procesado obtenida y las funciones que más tiempo consumen en CPU.

En este primer caso se introduce al algoritmo como entrada el video compuesto por 2181 *frames* ejecutando el procesado de cada uno en software en su totalidad. Recordamos que en este video se muestra un fondo estático de un jardín y una persona que de vez en cuando aparece paseando cruzándose por toda la escena [primera codificación del algoritmo realizada con Matlab], como se observa en la [figura 14].

Figura 14: Secuencia de frames del video sobre el que se aplica el algoritmo

De este total de *frames*, los ocho primeros se van a emplear para el cálculo de la parte *offline*, la extracción de las componentes principales, y el resto se emplearán para la parte *online*. Cada uno de estos, tras ser capturados y previo al algoritmo, requerirán cierto procesado [figura 15]:

- Paso primero: El video es en color, y como se nombra anteriormente, trabajar con color no proporciona ninguna ventaja adicional, de hecho produce mayor latencia en el algoritmo, siendo el propósito lo contrario, minimizarla. Por ello se realiza una conversión del *frame* capturado a escala de grises.
- Paso segundo: Puesto que el video se ha grabado con un dispositivo móvil, las dimensiones son demasiado grandes para aplicar directamente el algoritmo PCA. Por ello, se realiza un cambio de dimensión. Inicialmente, cada *frame*, tenía dimensiones 1920x1080, y, tras el redimensionado mediante la función de *OpenCV* "*resize"*, las dimensiones finales de cada *frame* son 256x256.

Tras este procesado inicial se obtiene un *frame* en escala de grises y con dimensiones 256x256, pudiendo introducirse en el algoritmo PCA.

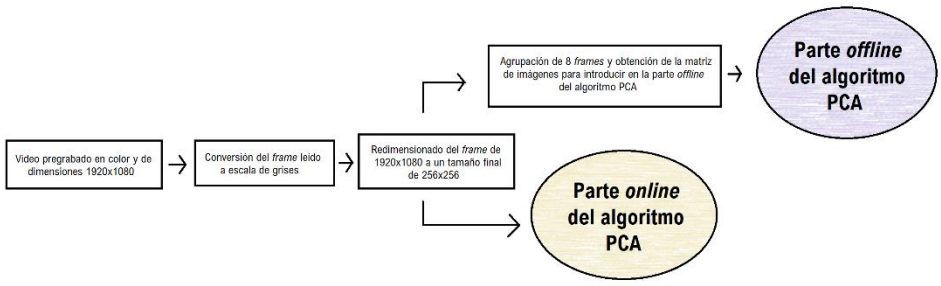
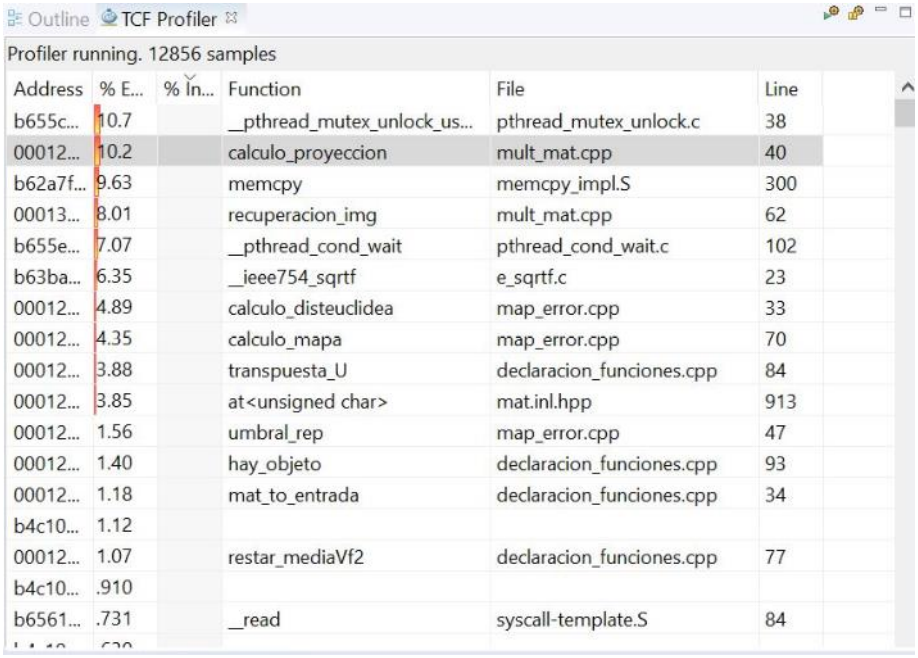


Figura 15: Flujo de preprocesado de los frames de entrada al algoritmo PCA

Se obtienen dos resultados: el tiempo que tarda en procesar el video en su totalidad, con el que se obtiene la velocidad de procesadoy las funciones que más tiempo de CPU requieren.

Los resultados obtenidos en la depuración con el "*TCF profiler"* se muestran en la [figura 16]. En esta se observan una serie de funciones (bajo la columna de título "*function"*), el fichero donde se encuentran y la línea de este donde se declaran, y el porcentaje del tiempo total que se han estado ejecutando en la CPU ("*%Execution"*). Aquellas funciones que más porcentaje de tiempo han consumido son las primeras que se encuentran en la figura. Se saca como conclusión que las funciones que más tiempo consumen son:



Address	% E...	% In...	Function	File	Line
b655c...	10.7		__pthread_mutex_unlock_us...	pthread_mutex_unlock.c	38
00012...	10.2		calculo_proyeccion	mult_mat.cpp	40
b62a7f...	9.63		memcpy	memcpy_impl.S	300
00013...	8.01		recuperacion_img	mult_mat.cpp	62
b655e...	7.07		__pthread_cond_wait	pthread_cond_wait.c	102
b63ba...	6.35		__ieee754_sqrtf	e_sqrtf.c	23
00012...	4.89		calculo_disteuclidea	map_error.cpp	33
00012...	4.35		calculo_mapa	map_error.cpp	70
00012...	3.88		transpuesta_U	declaracion_funciones.cpp	84
00012...	3.85		at<unsigned char>	mat.inl.hpp	913
00012...	1.56		umbral_rep	map_error.cpp	47
00012...	1.40		hay_objeto	declaracion_funciones.cpp	93
00012...	1.18		mat_to_entrada	declaracion_funciones.cpp	34
b4c10...	1.12				
00012...	1.07		restar_mediaVf2	declaracion_funciones.cpp	77
b4c10...	.910				
b6561...	.731		__read	syscall-template.S	84

Figura 16: Resultados de la ejecución del TCF Profiler con video

- El cálculo de la imagen proyectada en el espacio transformado de las componentes principales, que requiere una multiplicación matricial, de nombre "*calculo_proyección"*.
- El cálculo de la recuperación de la imagen proyectada en el espacio transformado, la cual también requiere una multiplicación matricial, de nombre "*recuperación_img*".

Estas dos funciones son aquellas del algoritmo que consumen más de un 5% de tiempo de CPU y por ello son las que se consideran candidatas para su aceleración. Se observa en la [figura 16] que en la lista de funciones se encuentran las funciones "*__pthread_muyex_unlock"*, "*memcpy"* y "*__pthread_con_wait"* entre las 5 funciones que más porcentaje de tiempo en CPU consumen. Estas funciones no se han definido ni codificado para el algoritmo sino que son funciones internas del sistema operativo que corre en el microcontrolador. Son funciones de sincronización de la CPU que controlan la ejecución en esta y el acceso a posiciones de memoria, y al ser internas del microcontrolador no son funciones que se puedan optimizar ni trasladar al HW y por ello no son candidatas, a pesar de consumir mayor porcentaje de tiempo de CPU que las que si son candidatas.

Para la obtención de la velocidad de procesado de las imágenes, tras lanzar la aplicación del algoritmo 10 veces, se obtuvo la media de tiempo de ejecución del algoritmo. Esta resultó de 410'15 segundos para procesar las 2181 *frames* que componían la parte *online* del algoritmo, traduciéndose en una tasa de procesado de 5,29 *fps* ("*frames* por segundo").

La obtención de esta tasa de procesado de imágenes tan baja junto con el hecho que se nombraba dos párrafos antes de que la función que más porcentaje de tiempo de CPU consume está asociada a la sincronización de esta , causó cierta incertidumbre sobre lo que podía estar pasando al ejecutar esta aplicación en su totalidad en software.

Fue por ello qué se decidió realizar la prueba de la ejecución del algoritmo evitando el procesado realizado en la parte *online*, es decir:

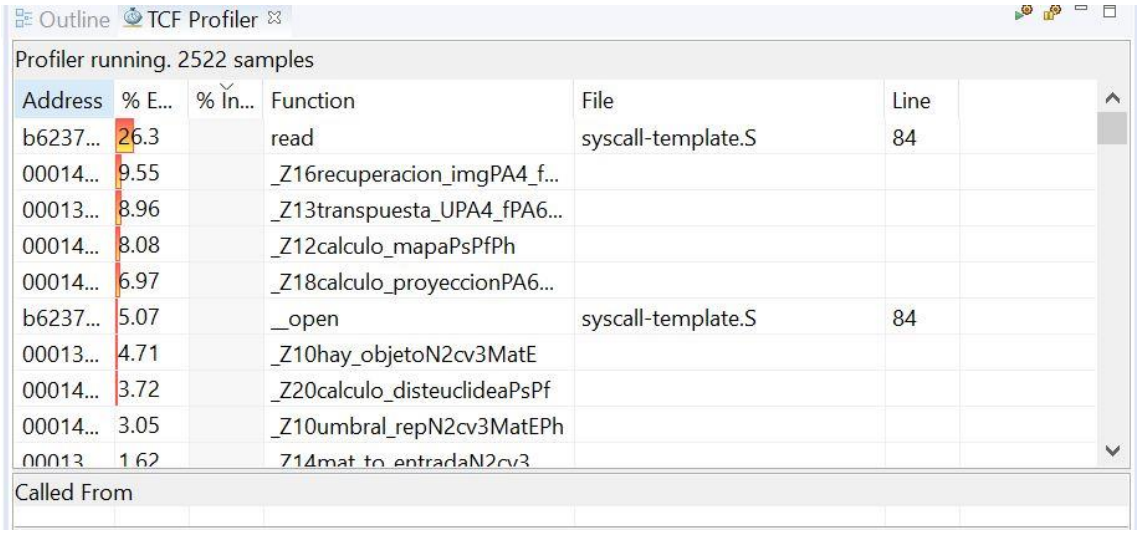
- Tras la ejecución de la parte *offline* del algoritmo, la cual es indiferente tener en cuenta o no puesto que el tiempo que tarda en ejecutarse es despreciable, lo que se realiza es la captura de cada *frame* y el preprocesado, y no se aplica el cálculo online. Tan solo se lee el *frame* capturado del video, se convierte a escala de grises y se redimensiona, y tras esto se desprecia y se lee el siguiente, así hasta que el video finaliza

Gracias a la ejecución del algoritmo solo leyendo y preprocesando las 2181 *frames*, sin aplicar PCA como tal, se observó que el tiempo medio de lectura y preprocesado era en media de 315,65 segundos, de manera que realmente el procesado de las imágenes de la parte *online* de PCA era la diferencia (en media 94,15 segundos). El hecho de observar que tan sólo la captura de un *frame* del video, su conversión a escala de grises y su redimensionado llevaba tanto tiempo de CPU, fue el motivo para cambiar el método de trabajo, al no ser realmente funciones pertenecientes al algoritmo PCA y reducir tanto la tasa de procesado.

Es aquí donde se realiza la modificación de código y se utiliza la codificación en C/C++ de la segunda versión codificada en Matlab, donde recordemos que se introducía como entrada un conjunto de imágenes de tamaño 256x256 en color en lugar de un video que requiere redimensionado. La decisión de no solo realizar el fraccionado del video sino también realizar un dimensionado se debe a que el conjunto total de las imágenes en su tamaño original superaban los 500MB de almacenamiento de la microSD.

El hecho de modificar la entrada al algoritmo del video al conjunto de imágenes modifica el punto de partida. Por ello se requiere volver a realizar la ejecución del algoritmo PCA en su totalidad en software, tratando en este caso un conjunto de 2173 imágenes de tamaño 256x256 en color que describen la misma escena que el video. Compilando y depurando para obtener los resultados del "*TCF Profiler"* [figura 17] se observa que en este caso la función que mayor porcentaje de tiempo de CPU es la lectura del *frame* con la función de *OpenCV* con un 26,3% y tras esta, con porcentajes mayor al 5%, se encuentran las que son candidatas para la ejecución en HW:

- La recuperación de la imagen del espacio transformado, "*recuperación_img"* con un 9,55% de tiempo.
- El cálculo de la transpuesta de la matriz de transformación de los componentes principales, "*transpuesta_U"* con un 8,96%.
- El cálculo del mapa de distancias euclídeas entre la imagen recuperada del espacio de las componentes principales y la imagen de entrada, "*calculo_mapa"*, con un 8,08%.
- El cálculo de la imagen proyectada en el espacio transformado, "*calculo_proyección"*, con un 6,97%.


Outline TCF Profiler

Profiler running. 2522 samples

Address	% E...	% In...	Function	File	Line
b6237...	26.3		read	syscall-template.S	84
00014...	9.55		_Z16recuperacion_imgPA4_f...		
00013...	8.96		_Z13transpuesta_UPA4_fPA6...		
00014...	8.08		_Z12calculo_mapaPsPfPh		
00014...	6.97		_Z18calculo_proyeccionPA6...		
b6237...	5.07		__open	syscall-template.S	84
00013...	4.71		_Z10hay_objetoN2cv3MatE		
00014...	3.72		_Z20calculo_disteuclideaPsPf		
00014...	3.05		_Z10umbral_repN2cv3MatEPh		
00013	1.62		_Z14mat_to_entradaN2cv3		

Called From

Figura 17: Resultados de la ejecución del TCF Profiler con imágenes

Haciendo un breve inciso se va a demostrar la diferencia de la compilación mediante el método *release* y el método *debug* para justificar el uso del primero. Recordemos que el método *release* reduce el tamaño del código y el tiempo de ejecución pero no proporciona información de

depuración y que el método *debug* proporciona esta información de depuración permitiendo recorrer el código paso a paso a costa de un mayor tamaño.Se adjunta a continuación la media de tiempos de ejecución de obtenidos con ambos modos, observable en la [figura 19]. El tiempo que se muestra en esta figura es el del algoritmo completo por lo que se le debe extraer los 1,56 segundos pertenecientes a la parte *offline* antes de calcular la velocidad de procesado:

- Como se observa, en el modo *release*, se obtiene un tiempo de ejecución de 111,76 segundos (113,32-1,56). La división de los 2173 *frames* que conforman el conjunto de imágenes entre este tiempo proporciona una velocidad de procesado de 19,44 *fps*.
- En el caso del modo *debug* se observa que el tiempo de ejecución es de 312,49 (314,05-1,56) segundos, lo que da una tasa de procesado notablemente inferior de 6,95 *fps*.

Release		Debug	
Real	113,32	Real	314,05
User	73,32	User	271,43
Sys	11,76	Sys	11,47

Figura 18: Tiempos medios de ejecución escenario 1

Como se esperaba, con el primero se obtiene menor tiempo de ejecución y mayor optimización de la aplicación. Puesto que la información de depuración del algoritmo paso a paso solo es necesaria si se requiere descubrir algún fallo en ejecución, se justifica el uso de las medidas gracias a la compilación con el modo *release*.

Así, como resumen de este primer escenario se puede concluir que:

- Se obtiene una tasa de procesado del conjunto de 2173 imágenes de 19,44 *fps* obtenida gracias a la ejecución del algoritmo en su totalidad en software con el modo de compilación *release*. Esta cifra se empleará en los apartados siguientes como lo que se denomina como "punto de partida".
- Como se ha listado anteriormente las funciones candidatas a la aceleración hardware son:
    - Lectura de imágenes, que se recuerda consumía el mayor porcentaje de tiempo de CPU con un 26,3%, "*read"*.
    - La función que recupera la imagen del espacio transformado de las componentes principales, "*recuperacion_img"*, con 9,55%.
    - La función que calcula a traspuesta de la matriz de transformación, U, "*transpuesta_U",* con 8,96%.
    - La función que calcula el mapa de distancias entre la imagen enviada al espacio transformado y la recibida, "*calculo_mapa"*, con un 8,08%.
    - La función que calcula la proyección de la imagen en el espacio transformado de las componentes principales, "*calculo_proyeccion"*, con 6,97%.

Antes de continuar a la segunda aproximación, una de estas candidatas es descartada: el cálculo de la matriz transpuesta de U, "*transpuesta_U"*. Esto se debe a que esta función tiene dos parámetros, la matriz U y la matriz U transpuesta que se devuelve. El excesivo tamaño de la matriz U impide que esta se pueda mandar al HW mediante el bus de interfaz el exceder las dimensiones de este, por ello se propone usar el siguiente pragma:

- *Pragma SDS data access_pattern(<var>:SEQUENTIAL):* Este pragma especifica el patrón de acceso de datos en la función hardware. El hecho de emplear el acceso secuencial implica que se genera una interfaz de flujo como lo es una FIFO, lo que es beneficioso porque por defecto con acceso aleatorio se genera una interfaz RAM, siendo esto lo que da problemas de dimensiones.

Este pragma implica el acceso a los datos de manera secuencial de manera que permite acceder a estas matrices de altas dimensiones secuencialmente. El problema es que se debería declarar

ambas de esta forma para poder trasladarlo a HW pero solo una de ellas es accedida realmente manera secuencial dando error al intentar acceder y almacenar los datos en la transpuesta.

Las funciones que se optimizaran en las siguientes aproximaciones serán finalmente:

- Lectura de imágenes, que se recuerda consumía el mayor porcentaje de tiempo de CPU con un 26,3%, "*read"*.
- La función que recupera la imagen del espacio transformado de las componentes principales, "*recuperacion_img"*, con 9,55%.
- La función que calcula el mapa de distancias entre la imagen enviada al espacio transformado y la recibida, "*calculo_mapa"*, con un 8,08%.
- La función que calcula la proyección de la imagen en el espacio transformado de las componentes principales, "*calculo_proyeccion"*, con 6,97%.

2. **Segunda aproximación: Ejecución en software del algoritmo completo con *xf::imread***

Como se ha observado en la primera aproximación, la función de la librería "*OpenCV"* que se emplea para la lectura de las imágenes es la que más consume CPU (26,3% del tiempo). La lectura de estas imágenes es necesaria por lo que se requiere usar una función que las lea. Las únicas dos opciones que existen para leer las imágenes son la lectura mediante software con la librería "*OpenCV"*, que es lo que se ha hecho hasta ahora, o bien la lectura mediante hardware con la versión de "*OpenCV"* implementada por Xilinx para su ejecución con la lógica programable, "*xfOpenCV"*, la cual como se explicaba previamente viene incluida en las librerías del sistema operativo Linux de la plataforma.

Por ello en esta segunda aproximación se realiza una modificación leve de la codificación del algoritmo. Se mantiene la ejecución del algoritmo PCA completo en software, pero como se en lugar de emplear la función *read* (lectura de las imágenes de la microSD) de la librería software"*OpenCV*"se propone la implementación del algoritmo usando la función equivalente que proporciona la librería "*xfOpenCV"* para observar si se obtiene mejora en el tiempo de ejecución del algoritmo (si se observa disminución de este) [figura 20] .

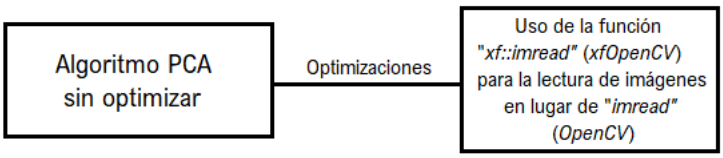


Figura 20: Aproximación 2

La recodificación del algoritmo sustituyendo la función "*imread"* por "*xfimread"* se realiza previo a mandar más funciones en HW con intención de observar que método de lectura resulta más eficiente en términos de latencia.

Las sentencias de código que empleaban la función "*imread"* de la librería de "*OpenCV"* se sustituyen por sentencias de código que emplean la función "*xf::imread"*, de la librería "*xfOpenCV"*. Puesto que el resto del algoritmo se mantiene, se requiere tras la lectura de la imagen la conversión del tipo "*xf::Mat"* de la librería "*xfOpenCV"* a "*cv::Mat"* de la librería "*OpenCV*", puesto que el resto de las funciones del algoritmo emplean este segundo tipo de datos.

Para la realización de esta conversión se requiere la creación de una matriz que represente la imagen "*cv::Mat"* que tenga el tamaño de la imagen obtenida con "*xf::imread"* y tras esto la copia de los datos, como se observa en la [figura 21]:

```
xf::Mat<XF_8UC1,MAT_SIZE,MAT_SIZE,XF_NPPC1> frame2;
frame2= xf::imread<0,MAT_SIZE,MAT_SIZE,XF_NPPC1>(file,0);
Mat framegray2(frame2.rows,frame2.cols,CV_8UC1);
framegray2.data = frame2.copyFrom();
```

Figura 21: Lectura de una imagen con xf::imread y conversión a tipo cv::Mat

En esta figura se adjunta el trozo de código que realiza la lectura de la imagen con la función hardware y su posterior asignación a la variable del tipo de dato "*Mat"* de la librería software. La primera línea declara la variable tipo "*xf::Mat"* "*frame2"* de un solo canal (escala de grises), dimensiones 256x256 ("*MAT_SIZE")* y con operaciones de un píxel por palabra.

La segunda línea le asigna a esta variable la lectura realizada por la función "*xfimread"* leída en blanco y negro (indicado por el primer parámetro '0') y donde "*file"* representa el fichero que se está leyendo.

La tercera línea crea la variable del tipo "*Mat"* con nombre "*framegray2"* asignándole el número de filas y columnas que tiene la variable *"xf::Mat"* "*frame2"* en escala de grises. Finalmente se realiza la copia de los datos de la variable hardware a la recién creada en software de manera que finalmente tanto "*frame2"* como "*framegray2"* contienen la misma información, empleando esta segunda para el resto del algoritmo.

Tras modificar esa sentencia de código, se lanza la compilación y la posterior ejecución en la plataforma Zybo, obteniendo los tiempos de ejecución de la [figura 22]:

Release		Debug		Release		Debug	
Real	113,32	Real	314,05	Real	168,98	Real	414,7
User	73,32	User	271,43	User	92,57	User	331,76
Sys	11,76	Sys	11,47	Sys	16,2	Sys	15,75

Figura 22: Comparativa de tiempos de cv::imread (izquierda) y xf::imread (derecha).

En las dos columnas de la izquierda se observan los resultados obtenidos al realizar la lectura con las funciones de las librerías "*OpenCV* " mediante software, y en las dos columnas de la derecha se observan los resultados obtenidos para la lectura de las imágenes con las funciones de la librería "*xfOpenCV"* y la lógica programable FPGA.

Centrando la vista en los resultados obtenidos para el método de compilación *release* se observa que la lectura de imágenes en hardware es relativamente más lenta, concretamente 55,66 segundos, proporcionando una tasa de 12,98 *fps* frente a las 19,44 *fps* que proporciona el software, obteniendo una ralentización del algoritmo del 33,34%.

Debido a que la lectura de imágenes con "*xfOpenCV"* es más lenta que con "*OpenCV"* proporcionando una tasa de procesado de las imágenes inferiorse decide que para los próximos escenarios, el método de lectura de las imágenes se realizará mediante software y la librería "*OpenCV"*, a pesar de ser la función que más tiempo de CPU consume.

3. **Tercera aproximación: Funciones de recuperación de la imagen y cálculo de la proyección en HW.**

En esta tercera aproximación se realiza la primera prueba con funciones trasladadas a la ejecución en hardware. Más concretamente las funciones que se mandan en esta aproximación son el cálculo de la proyección (“*calculo_proyeccion”*) y la recuperación de la imagen proyectada (“*recuperación_img”*).

Recordamos que junto a estas se encuentra la función del cálculo del mapa de distancias (“*calculo_mapa”*) como candidata a la aceleración, pero se eligió comenzar optimizando estas dos funciones ya que ambas implementan una misma operación matemática: la multiplicación de matrices. La diferencia entre ambas funciones son los tipos de datos y los tamaños de las funciones de entrada, pero la optimización aplicada es la misma a ambas.

Al ser la primera toma de contacto con la optimización de matrices se plantean dos ramas de optimización como se muestra en la [figura 23]:

- La primera de ellas fue la búsqueda de un algoritmo de multiplicación de matrices más eficiente que la multiplicación matricial estándar.
- La segunda fue la aplicación de pragmas de HLS y SDS (“*SDSoC pragmas”*) para optimizar y reducir la latencia.

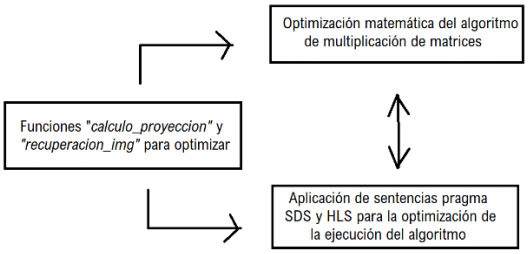


Figura 23: Diagrama de flujo de la tercera aproximación

La primera corriente de optimización (rama superior del diagrama de la [figura 23]) se basó en una investigación de los distintos métodos existentes de la multiplicación de matrices, realizando esta de manera individual mediante HLS y observando los recursos empleados y el tiempo que tardaba en ejecutarse. Para ello se creó un proyecto individual con la herramienta HLS y se trasladó la función que realizaba la multiplicación de matrices.

En este proyecto de HLS se crean dos ficheros fuente “*multiplicación_top.cpp*” y “*multiplicación_top_tb.cpp”*, donde en el primero se realiza la multiplicación matricial y en el segundo se le pasan los valores de las matrices por multiplicar.

Uno de los métodos que se observó como posible solución fue el método de multiplicación de matrices de *Strassen* [21], el cual proporcionaba una complejidad de cómputo de $O(4.7m^{log2(7)})$ frente a la complejidad $O(m^3)$ del algoritmo de multiplicación de matrices estándar. Éste se descartó debido a que se requería rellenar la matriz de ceros para que cumpliese la condición de matriz cuadrada, el cuál es requisito indispensable de *Strassen*. Además, este algoritmo requiere de llamadas a funciones de manera recursiva, lo cual no está permitido implementar en funciones con destino de ejecución el HW.

Otro método que se examinó fue el presentado por [19], multiplicación de matrices por bloques. En este escenario se plantea un código que divide las matrices a multiplicar en bloques de vectores, de manera que, como para obtener los elementos de cierta columna de la matriz final se requiere multiplicar por el mismo conjunto vectorial, tan solo se realiza la copia de este cuando se renueva. Este proceso de multiplicación se observa en la [figura 24] de manera más gráfica, viendo como hay bloques de elementos de la primera matriz que permanecen fijos al multiplicarse por los distintos elementos.

En la primera iteración del algoritmo, a) y b) de la [figura 24] se observa que permanecen fijos en la multiplicación todos los elementos de las dos primeras filas de la matriz A. Al multiplicarse en a) todos los elementos de las dos primeras filas de A por todos los elementos de las dos primeras columnas de B se obtienen los elementos de las dos primeras filas y de las dos primeras columnas de AB.

En b) al solo modificarse las columnas de B, en este caso se emplean las columnas tercera y cuarta, se obtienen los elementos de las dos primeras filas y de las tercera y cuarta columna de AB. Lo mismo ocurre cuando se cambian a las dos últimas filas de A y se multiplican por las columnas de B.

A pesar de que este método parecía ser viable, requería el uso del tipo de datos "*HLS stream"* el cual crea una matriz de vectores que contiene los elementos de las filas y las columnas, y este tipo de dato viene indicado en el manual de referencia de SDSoC que no está permitido como tipo de dato de argumento de ninguna función destinada a hardware.

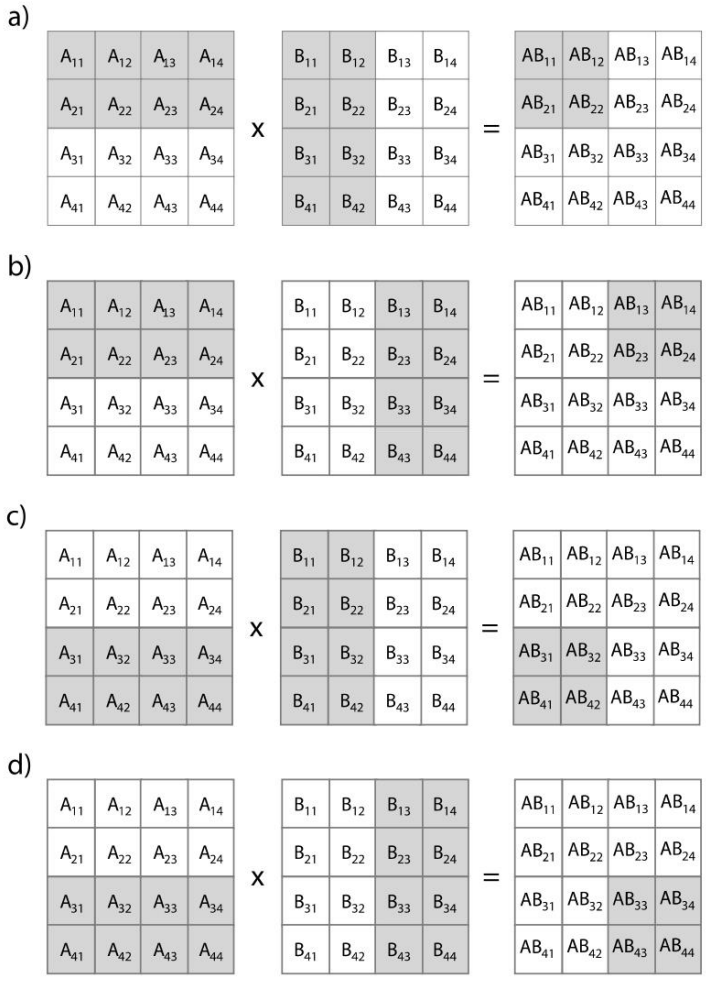


Figura 24: Explicación gráfica de la multiplicación de matrices por bloques explicada en la página 135 de [19].

En este se observa una posible descomposición por bloques de la multiplicación de dos matrices 4x4. El producto completo AB se descompone en 4 operaciones de multiplicación de matrices, operado en un bloque de A de tamaño 2x4 y en un bloque de B de tamaño 4x2.

Finalmente, la optimización que se encontró para la multiplicación de matrices en su forma matemática es la reducción de uno de los tres bucles "*for*" que componen la codificación de la

multiplicación al tratarse de una multiplicación de matriz por vector. Para que sea más visual se muestra a continuación los tres bucles "*for*" que conforman la multiplicación:.

```
Primer bucle "for": for(i=0;i<A_ROWS;i++){
        Segundo bucle "for": for(j=0;j<B_COLS;j++){
                        A[i][j] = 0;
                Tercer bucle "for": for (k=0;k<A_COLS/B_ROWS;k++){
                                C[i][j] = C[i][j] + A[i][k]*B[k][j];
                        }
                }
        }
```

En esta optimización se elimina el bucle dependiente de la variable "j" puesto que esta no toma nunca un valor superior a uno. Para eliminarlo, se sustituye en las matrices la variable "j" por el elemento "0" que hace referencia a la única columna que contiene la matriz, quedando:

```
Primer bucle "for": for(i=0;i<A_ROWS;i++){
                        A[i][0] = 0;
        Segundo bucle "for": for (k=0;k<A_COLS/B_ROWS;k++){
                                C[i][0] = C[i][0] + A[i][k]*B[k][0];
                }
        }
```

A pesar de que se plantea la siguiente modificación en la codificación del algoritmo de multiplicación de matrices también se realizan aproximaciones más adelante que emplean la codificación del algoritmo estándar. En esta tercera aproximación, sin ir más lejos. Para la segunda corriente de optimización de estas funciones basadas en la multiplicación matricial [rama inferior de la figura 23] se recurrió a los pragmas que proporciona HLS. Recordemos que esto es uno de los beneficios que proporciona SDx, permitiendo optimizar las funciones a ejecutar en hardware con la herramienta HLS.

En esta primera aproximación se plantea el uso de dos de los pragmas HLS:

- *Pragma HLS INLINE*: Este pragma permite eliminar una función como entidad individual, separada, en la jerarquía. Tras la aplicación de este pragma de alineación, en algunos casos, permite compartir y optimizar las operaciones de dentro de la función de manera más efectiva con las operaciones circundantes.
- *Pragma HLS PIPELINE II=1 rewind*: Este pragma reduce el intervalo de inicialización para una función o un bucle permitiendo la ejecución concurrente de operaciones. Este pragma permite que un bucle pueda procesar nuevas entradas cada N ciclos de reloj donde viene representado por el II (intervalo de inicialización). Realizar el *pipelining* en un buce permite que las operaciones de este se implementen de manera concurrente. Se introduce además la característica *rewind* la cual permite la tubería de bucles continuos sin pausa entre la iteración final de un bucle y el inicio del siguiente.

Puesto que la idea de trasladar las funciones de hardware a software es beneficiarse de esta concurrencia que proporciona el FPGA a la hora de realizar operaciones, el empleo de la tubería resulta el pragma más eficiente.

Además, puesto que las dimensiones de la matriz son demasiado grandes (262.144, 4x65536, elementos) exceden las dimensiones del bus de acceso al puerto RAM, de manera que se requiere indicar al SoC como se desea hacer el acceso de datos.

Las variables que causan estos problemas son las siguientes:

Tabla 1: Variables problemáticas

Input	Vector fila que representa la imagen de entrada que se va a proyectar en el espacio transformado.	Tamaño del vector: [256x256] Tipo de dato: SHORT
Recup_img	Vector fila que representa la imagen recuperada del espacio transformado.	Tamaño del vector: [256x256] Tipo de dato: FLOAT
Matrices U y U transpuesta	Estas matrices representan la matriz de transformación del espacio de las componentes principales.	Tamaño de las matrices: [256x256][4] [4][256x256] Tipo de dato: FLOAT

Para solucionar estos problemas se van a emplear los siguientes pragmas que proporciona SDS para indicar el acceso a los datos:

- *Pragma SDS data data_mover(<var>:AXIDMA_SIMPLE):* Este es un pragma del cual no se recomienda su uso a no ser que no se cumplan los requisitos de diseño, como es el caso. En este caso lo que hace el compilador automáticamente es asignar la instancia de IP de transporte de datos para transferir el array asociado. El hecho de emplear AXIDMA_SIMPLE se debe a que proporciona el mayor ancho de banda.
- *Pragma SDS data access_pattern(<var>:SEQUENTIAL):* Este pragma especifica el patrón de acceso de datos en la función hardware. El hecho de emplear el acceso secuencial implica que se genera una interfaz de flujo como lo es una FIFO, lo que es beneficioso porque por defecto con acceso aleatorio se genera una interfaz RAM, siendo esto lo que da problemas de dimensiones.
- *Pragma SDS data zero_copy(<var>[tamaño]):* Este pragma implica que los accesos a los datos se hacen de manera directa a través de una interfaz bus AXI maestro. Por defecto el compilador SDSoC asume el pragma contrario *data copy* para un argumento con forma de array, lo que implica una copia explicita desde el procesador al acelerador a través transportador de datos.

Además, tanto el primer como el tercer pragma requieren que las variables que se indiquen sean previamente asignadas usando *sds_alloc()*. Esto proporciona a las funciones el mapeo de espacios de memoria, más concretamente *sds_alloc()* proporciona una asignación de memoria física continua, lo que facilita la lectura de los datos disminuyendo la latencia.

Entonces, como resumen, esta tercera aproximación ejecuta en hardware las funciones de multiplicación matricial que calculan la proyección y recuperan la imagen en/del espacio transformado, empleando para el transporte y acceso a los datos los pragmas SDS ("*data_mover", "zero_copy" y "access_pattern")* y empleando los pragmas HLS ("*pipeline"* e "*inline"*) para crear una paralelización de las operaciones realizadas en los bucles. Además emplea el método de multiplicación de matrices estándar [figura 25].

Además, la sentencia *Pragma HLS PIPELINE II=1 rewind* se coloca en el bucle más interno puesto que lleva asociado un desenrollado del bucle y, si se coloca en los bucles externos, el desenrollado de los bucles que contiene lleva excesivo tiempo y SDx indica que la síntesis ha fallado.

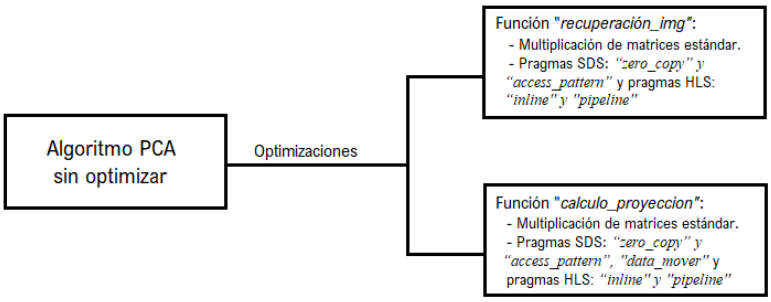


Figura 25: Aproximación 3

En este caso al trasladar funciones a hardware la opción de estimación de recursos si está disponible, siendo esta y la velocidad de procesado las medidas que llevarán a tomar la decisión de cuánto ha conseguido acelerar esta aproximación el algoritmo.

Los resultados obtenidos para esta aproximación se muestra en las siguientes figuras [23,24], tanto la tasa de procesado como la de recursos empleados por la lógica programable y la aceleración obtenida:

- En la [figura 26] se observa la aceleración estimada por SDx para la función '*main*' y para las funciones ejecutadas por la lógica programable son los siguientes, indicando el número de ciclos que se miden al ejecutar la función en software ("*SW-only(Measured cycles)"*) y el número de ciclos que se estima que va a tardar en ejecutar la función en hardware (*"Hardware accelerated (Estimated cycles)"*:

Performance estimates for 'main' function

SW-only (Measured cycles)	210456866032
Hardware accelerated (Estimated cycles	152259797184
Estimated speedup	1.38

Details

Performance estimates for functions 'recuperacion_img in ...

SW-only (Measured cycles)	28205320
Hardware accelerated (Estimated cycles	25141602
Estimated speedup	1.12

Figura 26: Aceleración HW de la aproximación 3

Se observa que realizar las funciones de recuperación y cálculo de la proyección mediante hardware suponen una aceleración del 12% ("*Estimated speedup"*, tabla inferior de la [figura 26]) con respecto a la ejecución de estas en software, obteniendo como resultado final una aceleración del 38% con respecto a la aproximación inicial ("*Estimated speedup"*, tabla superior de la [figura 26]).

Estas medidas son estimadas siendo la manera de ver la aceleración lograda realmente la ejecución en la propia plataforma del algoritmo.

Al lanzar la ejecución en la plataforma Zybo Z720, se obtiene una velocidad de procesado media de ejecución de 19,77 *fps*, ligeramente superior a la que se obtenía al ejecutar todo en software en

la primera aproximación (punto de partida), 19,44 *fps*. Esto supone una aceleración real de aproximadamente el 2%.

Resource	Used	Total	% Utilization
DSP	10	220	4.55
BRAM	25	140	17.86
LUT	15905	53200	29.9
FF	19174	106400	18.02

Figura 27: Recursos empleados por el FPGA en la aproximación 3

Otra de las medidas que se obtiene al realizar la estimación de la aceleración que se va a obtener en HW es la estimación de los recursos hardware que se van a emplear para ejecutar las funciones. Esta medida es interesante observarla para no llevar a ejecutar ninguna aplicación falta de recursos.

Para esta aproximación, los recursos estimados se observan en la [figura 27]: usa 4.55% de los DSP disponible, un 17.86% de la memoria BRAM disponible, un 29.9% de las LUTs disponibles y un 18.02% de los registros (FF, "*Flip-Flop"*). Se observa pues que la escasez de recursos no es punto de preocupación en esta aproximación planteada.

4. **Cuarta aproximación: Funciones de recuperación de la imagen y cálculo de la proyección en HW, en este caso con un bucle menos.**

Para esta cuarta aproximación las funciones que se destinan a ser ejecutadas en hardware son las mismas que las planteadas en la tercera aproximación y con las mismas condiciones. La única diferencia entre estas dos aproximaciones es el método de multiplicación matricial empleado [figura 28].

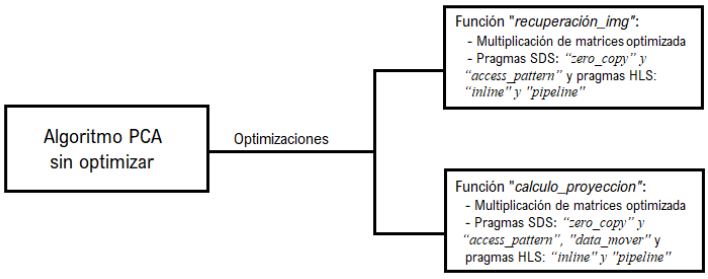


Figura 28: Aproximación 4

Como bien se comentaba antes, al describir la rama superior de la [figura 23], el método de multiplicación de matrices estándar, codificado en lenguaje de alto nivel C, viene descrito como la agrupación de tres bucles "*for"*. El más externo recorre las filas de la primera matriz, denominémosla A (*"loop1"*), el segundo bucle recorre las columnas de la segunda matriz, B, ("*loop2"*) y finalmente un tercer bucle recorre el número de elementos de las columnas de A y filas de B (ya que es condición necesaria para la multiplicación de matrices que estos coincidan, "*loop3"*):

```
loop1: for(i=0;i<A_ROWS;i++){
       loop2: for(j=0;j<B_COLS;j++){
                      A[i][j] = 0;
              loop3: for (k=0;k<A_COLS/B_ROWS;k++){
```

```
                        C[i][j] = C[i][j] + A[i][k]*B[k][j];
                }
        }
}
```

En estas funciones que realizan multiplicación matricial, tanto para el cálculo de la proyección como para el cálculo de la recuperación de la imagen, la matriz "B" de la multiplicación es realmente un vector fila, al ser B_COLS = 1 siempre, obteniendo como resultado de la multiplicación también un vector fila. Gracias a esto, para este caso de multiplicación específico, se puede transformar la multiplicación de manera que se elimina el bucle 2 ("*loop2"*). Realmente supone la ejecución de una línea de código, pero es una línea de código que se ejecuta una cantidad significativa de veces de manera que, *inicialmente*, parece que si se elimina se reducirán los ciclos empleados para la ejecución de la multiplicación.

Se recalca la palabra *inicialmente* puesto que llevado a la práctica esto no tiene porqué cumplirse. La multiplicación queda entonces como:

```
loop1: for(i=0;i<A_ROWS;i++){
                A[i][0] = 0;
        loop2: for (k=0;k<A_COLS/B_ROWS;k++){
                        C[i][0] = C[i][0] + A[i][k]*B[k][0];
                }
        }
```

Donde se observa que desaparece un bucle y en su lugar se indica como indicativo de las columnas de C (resultado) y B el dígito '0', una única columna para esas matrices.

Los resultados obtenidos para este escenario se muestran en las figuras [29,30], tanto de tasa de procesado como de recursos empleados por la lógica programable y la aceleración obtenida:

- La aceleración estimada por SDx para la función '*main*' y para las funciones ejecutadas por la lógica programable son las que se muestran en la [figura 29]:

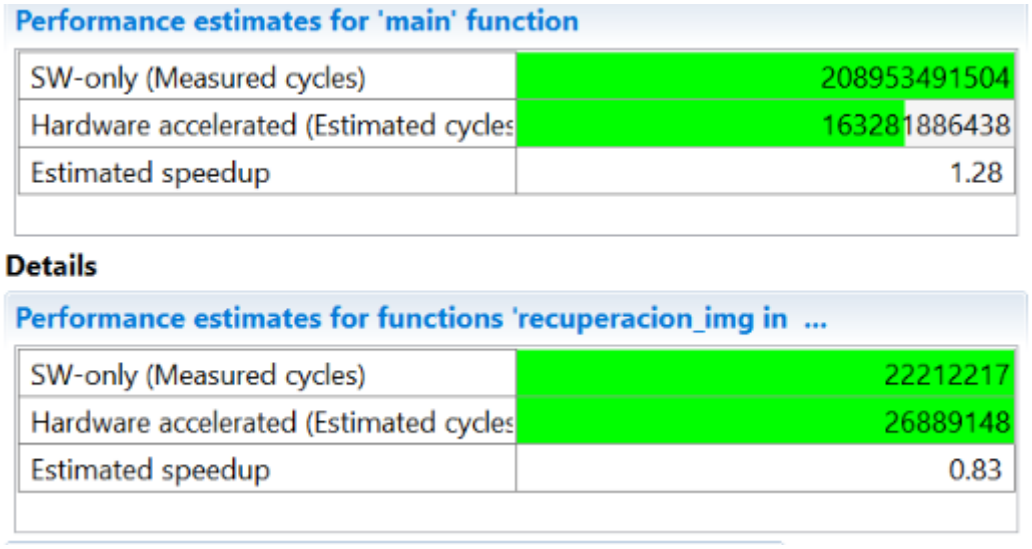


Figura 29: Aceleración HW de la aproximación 4

Se observa que realizar las funciones de recuperación y cálculo de la proyección mediante hardware [tabla inferior de la figura 29] no suponen una aceleración con respecto a la ejecución de estas en software, lo contrario, individualmente se ejecutan en mayor número de ciclos de reloj. Aun así, trasladar la ejecución de estas sigue proporcionado una paralelización, obteniendo como resultado final una aceleración del 28% con respecto al escenario inicial, en términos estimados de ejecución [tabla superior de la figura 29].

Para observar de manera real la aceleración conseguida, de nuevo se requiere ejecutar la aplicación en la plataforma.

- Al lanzar esta ejecución en la plataforma Zybo, se obtiene una velocidad media de procesado de las imágenes de 20,07 *fps*, ligeramente superior a la que se obtenía al ejecutar la multiplicación con un bucle más, aproximación tercera, 19,77 *fps* y al ejecutar todo en software, 19,44 *fps*. Esto supone una aceleración real de aproximadamente el 3,1% con respecto a la aproximación del punto de partida, primera aproximación.
- Los recursos empleados de hardware, lo cual también resulta interesante tener controlados puesto que puede que se excedan y el escenario no sea implementable son los siguientes [figura 30]:

Resource	Used	Total	% Utilization
DSP	10	220	4.55
BRAM	25	140	17.86
LUT	15905	53200	29.9
FF	19174	106400	18.02

Figura 30: Recursos empleados por el FPGA en la aproximación 4

Se observa pues que la escasez de recursos tampoco es punto de preocupación en esta cuarta aproximación, puesto que son los mismos recursos que los empleados en la tercera aproximación, no se ahorran recursos hardware, pero si se observa aceleración con respecto a esta.

5. **Quinta aproximación: Funciones de recuperación de la imagen y cálculo de la proyección en HW con alisado de bucles.**

Al igual que las dos aproximaciones anteriores, esta aproximación ejecuta en la FPGA las dos funciones de cálculo de proyección y de recuperación de la imagen basadas en bucles para la multiplicación de matrices.

En este caso la diferencia con las aproximaciones anteriores es la adición de una nueva sentencia pragma de HLS denominada "*Loop_Flatten"*.

Esta sentencia pragma permite que los bucles anidados se alisen en un único bucle de jerarquía única mejorando la latencia. En una implementación a nivel de registros (RTL) se requiere un ciclo de reloj para realizar el paso de un bucle externo a un bucle interno y viceversa, de manera que el alisado de bucles anidados permite optimizarlo en un único bucle, ahorrando ciclos de reloj y permitiendo potencialmente una mayor optimización de la lógica del cuerpo del bucle.

Esta sentencia se puede aplicar puesto que se encuentra un nido de bucles semi-perfecto donde todos los límites son constantes y solo el bucle más interno tiene contenido de cuerpo. En función de donde se coloque la sentencia pragma, el nido de bucles tendrá una jerarquía un otra, lo cual se observará en la aproximación siguiente.

En este caso se sitúa el pragma dentro del bucle más interno, de manera que creará una única jerarquía con los dos bucles por encima.

Entonces esta aproximación recoge las sentencias pragma de las aproximaciones anteriores y le añade la sentencia HLS ("*Loop flatten"*) manteniendo la codificación matemática de la multiplicación matricial en su versión estándar [figura 31] .

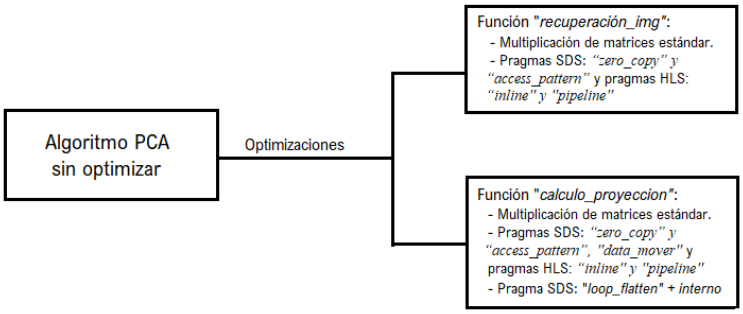


Figura 31: Aproximación 5

Los resultados obtenidos para esta aproximación de muestran en la [figura 32]:

- La aceleración estimada según el software SDx es de un 4% al comparar la ejecución en HW y en SW de las funciones de manera individual [tabla inferior de la figura], obteniendo aceleración aunque esta sea pequeña. Sin embargo sí parece observarse una mayor aceleración para la ejecución del algoritmo en su totalidad en términos de ciclos de reloj, en este caso del 36% [tabla superior de la figura].

Performance estimates for 'main' function

SW-only (Measured cycles)	221531373602
Hardware accelerated (Estimated cycles	163135369948
Estimated speedup	1.36

Details

Performance estimates for functions 'recuperacion_img in ...

SW-only (Measured cycles)	28067899
Hardware accelerated (Estimated cycles	26889148
Estimated speedup	1.04

Figura 32: Aceleración HW de la aproximación 5

Para obtener la velocidad de procesado se traslada la ejecución de la aproximación al SoC de la Zybo y ejecutarlo, al medir el tiempo de ejecución, se obtiene de media 19,19 *fps* que es un valor inferior al obtenido en el punto de partida, 19,44 *fps*. Esta aproximación no consigue acelerar el algoritmo sino lo contrario, lo decelera en un 1,286%, de manera que esta se descarta directamente.

En la [figura 33] se adjuntan los recursos empleados por el hardware para la ejecución de las funciones: 10 DSP, 25 BRAM, 15905 LUTs y 19174 FF. Si la opción fuese viable y proporcionase una aceleración razonable, la escasez de recursos no serían motivo para no llevar a cabo su uso.

Resource	Used	Total	% Utilization
DSP	10	220	4.55
BRAM	25	140	17.86
LUT	15905	53200	29.9
FF	19174	106400	18.02

Figura 33: Recursos empleados por el FPGA en la aproximación 5

6. **Sexta aproximación: Funciones de recuperación de la imagen y cálculo de la proyección en HW, en este caso con alisado de bucles en el primer bucle.**

Como se comentaba en la aproximación anterior la posición de la sentencia pragma "*Loop flatten"* dentro del código modifica los resultados obtenidos al aplicarla, por ello en esta aproximación se varía su localización. En la aproximación anterior se introduce la sentencia en el bucle más interno, alisando los dos bucles de mayor jerarquía. En este caso la sentencia se introduce por debajo del primer bucle, con la intención de demostrar cómo afecta la posición de este pragma en la ejecución de la aplicación y en su aceleración.

Esta aproximación consta pues de las sentencias pragma SDS y HLS descritas en apartados anteriores ("*zero_copy", "access_pattern"* y "*data_mover"* para el acceso a los datos y "*inline", "pipeline"* y "*loop_flatten"*) [figura 34].

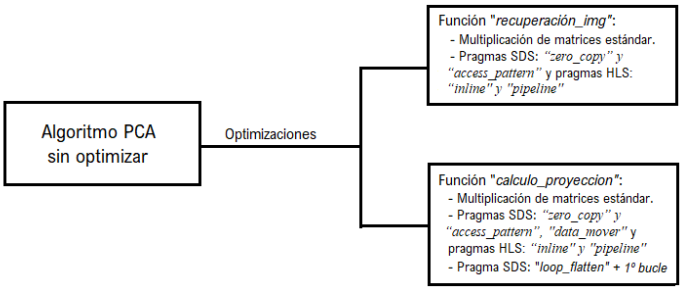


Figura 34: Aproximación 6

En la [figura 35] se adjuntan los resultados obtenidos de la estimación de los ciclos de reloj que tarda en ejecutarse esta aproximación. Como se observa en la tabla inferior, la aceleración estimada obtenida de la ejecución de las funciones de manera aislada es la misma que para la aproximación anterior en la que la sentencia pragma *"loop_flatten"* se introducía en el bucle más interno, 4%, mientras que si se observa la tabla superior donde se indica la aceleración del algoritmo completo esta aproximación presenta una aceleración estimada un 2% superior, del 38%.

Performance estimates for 'main' function

SW-only (Measured cycles)	213526655932
Hardware accelerated (Estimated cycles	155183426922
Estimated speedup	1.38

Details

Performance estimates for functions 'recuperacion_img in ...

SW-only (Measured cycles)	28043613
Hardware accelerated (Estimated cycles	26889148
Estimated speedup	1.04

Figura 35: Aceleración HW de la aproximación 6

En este caso, al ejecutar la aplicación en la plataforma Zybo se obtiene un valor medio de 19,73 *fps* valor superior a los 19,19 *fps* obtenidos en la aproximación anterior donde la única variación era la posición de la sentencia, obteniendo un 2,736% de aceleración con esta ligera modificación de la sentencia pragma. Con respecto a la aproximación que establece el punto de partida, 19,44 *fps*, se obtiene una aceleración real de procesado obtenida de un 1,5%.

En términos de recursos empleados por el hardware se obtienen los siguientes resultados, [figura 36],utilixando tan solo un 4,55% de los DSPs disponibles, un 17,86% de BRAM, un 29,9% de LUTs y un 18,02% de *flip-flops*. De hecho, a pesar de modificar la colocación de la sentencia, la utilización de los recursos de la lógica programable sigue siendo la misma que para los casos anteriores, no habiendo escasez de estos.

Resource	Used	Total	% Utilization
DSP	10	220	4.55
BRAM	25	140	17.86
LUT	15905	53200	29.9
FF	19174	106400	18.02

Figura 36: Recursos empleados por el FPGA en la aproximación 6

Hasta este punto solo la tercera, la cuarta y la sexta aproximación proporcionan aceleración sobre el algoritmo aunque esta es pequeña (inferior al 5%) y no proporciona resultados de velocidad de procesado considerables para la aplicación de estas a la aplicación del sistema en tiempo real (tasas de procesado inferiores a los 20 *fps*).

7. **Séptima aproximación: Función de recuperación de la imagen en HW.**

En esta séptima aproximación, puesto que como se había observado para el punto de partida (primera aproximación), la función (tras la lectura de la imagen) que mayor tiempo de CPU consumía era la recuperación de la imagen en el espacio transformado, se plantea observar cuánto se aceleraría el algoritmo si solo fuese esta la función que se ejecutase en hardware, pasando la ejecución del cálculo de la proyección a la CPU del microcontrolador.

Puesto que esta función sigue siendo una multiplicación matricial, la optimización de las aproximaciones tercera y cuarta se mantienen. Es decir, se siguen aplicando los pragmas SDS de acceso de datos de manera secuencial y se sigue realizando un alineación y tuberías (*HLS INLINE y HLS PIPELINE)*. En esta aproximación se omite la optimización con el pragma "*Loop_Flatten"* puesto que comparando la sexta aproximación con la tercera y la cuarta (1,5% frente al 3,1% de la tercera y 2% de la cuarta) se obtiene mayor aceleración sin esta [figura 37].

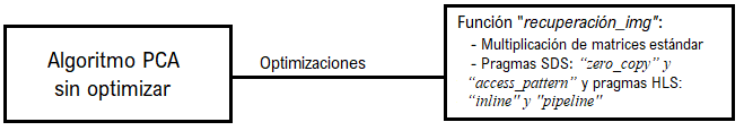


Figura 37: Aproximacion 7

En este caso los resultados obtenidos presentan menor utilización de recursos hardware como es obvio al destinar menor número de funciones:

Los recursos hardware son los que se muestran en la [figura 38], observando que son menor cantidad de recursos que los empleados en las aproximaciones anteriores. Esto se debe a que en

lugar de ejecutar dos funciones mediante la lógica programable del FPGA tan solo se ejecuta una. Se pasa de utilizar un 4,55% de los DSPs totales a un 2,27%, una reducción de la mitad, de un 17,86% de memoria BRAM a un 11,43%, de un 29,9% a un 17,66% de LUTs y de un 18,02% de *flip-flops* a un 11,14%. Esto implica que la preocupación por la escasez de recursos al implementar esta aproximación es incluso inferior a la mostrada en escenarios anteriores.

Resource	Used	Total	% Utilization
DSP	5	220	2.27
BRAM	16	140	11.43
LUT	9395	53200	17.66
FF	11854	106400	11.14

Figura 38: Recursos empleados por el FPGA en la aproximación 7

Observando que los recursos permiten su implementación en hardware, se observa ahora la aceleración que estima el propio software SDSoC en este caso. Esta aceleración se muestra en el [figura 39] siendo observable en la tabla inferior que realmente la ejecución de la función aislada en hardware no supone la ejecución de un número inferior de ciclos ni tampoco superior (no existe variación en su ejecución, aceleración del 0%), pero deja libre la CPU para la ejecución de otros procesos paralelamente provocando una estimación de la aceleración de la aplicación al completo del 12%, tabla superior de la figura.

Performance estimates for 'main' function

SW-only (Measured cycles)	282993198648
Hardware accelerated (Estimated cycles	252928731387
Estimated speedup	1.12

Details

Performance estimates for 'recuperacion_img in main.cpp:1 ...

SW-only (Measured cycles)	14849361
Hardware accelerated (Estimated cycles	14849501
Estimated speedup	1

Figura 39: Aceleración HW de la aproximación 7

Hasta este punto los resultados obtenidos resultan favorables a falta de ejecutar el criterio de decisión definitivo que implica la ejecución del algoritmo en la propia plataforma. Al descargar la aproximación sobre el SoC de la plataforma Zybo se obtiene una media de procesado de las imágenes del 19,80 *fps* lo que supone una mejoría (aceleración) frente al escenario del que se partía, 19,44 *fps*, del 1,89%. Este resultado que se obtiene sigue siendo inferior a los obtenidos en las aproximaciones tercera y cuarta de manera que tampoco resultará la optimización elegida.

8. **Octava aproximación: Función de recuperación de la imagen en HW, introduciendo la sentencia "*Loop_Flatten*" .**

Puesto que la sentencia "*loop_flatten"* se había probado para las aproximaciones anteriores con la traslación de las funciones "*calculo_proyeccion"* y "*recuperación_img"*, en esta aproximación

se introduce de nuevo la sentencia en la función de la recuperación de la imagen, a pesar de no haber supuesto excesiva aceleración en los apartados anteriores.

Esta decisión se toma puesto que el número de funciones que se han destinado a hardware es distinto (una función en vez de dos) y no se sabe cómo puede afectar realmente ese cambio a la ejecución de la aplicación. La posición donde se introduce esta sentencia se elige en función de los resultados obtenidos en las aproximaciones quinta y sexta, eligiendo finalmente situar la sentencia por debajo del bucle exterior, ya que en el bucle más interno no proporcionaba aceleración en el algoritmo, lo contrario, lo ralentizaba.

Se manda entonces de manera aislada la función que recupera la imagen del espacio transformado para su ejecución en hardware y se optimiza mediante las sentencias pragma HLS: *pipeline*, *inline* y *loop_flatten*, empleando para el acceso de datos las mismas sentencias pragma SDS que para aproximaciones anteriores [figura 40].

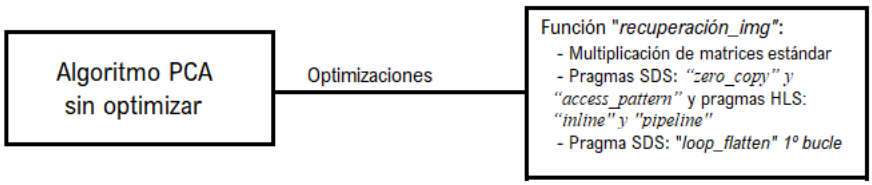


Figura 40: Aproximación 8

Se muestra en la [figura 41] la estimación de la aceleración que SDx proporciona previa a la descarga en la plataforma. En la tabla inferior se observa que la ejecución de la función en hardware requiere de un número de ciclos de reloj un 2% superior al número de ciclos de reloj que requiere la ejecución en la CPU del microcontrolador. Lo importante es que a pesar de consumir un mayor número de ciclos de reloj en HW se sigue obteniendo un número total de ciclos de reloj en la ejecución del algoritmo completo inferior al del escenario inicial, obteniendo una aceleración global del 16%.

Performance estimates for 'main' function

SW-only (Measured cycles)	213526655932
Hardware accelerated (Estimated cycles	184176863327
Estimated speedup	1.16

Details

Performance estimates for 'recuperacion_img in main.cpp:1 ...

SW-only (Measured cycles)	14520472
Hardware accelerated (Estimated cycles	14849501
Estimated speedup	0.98

Figura 41: Aceleración HW para el escenario 8

Para observar la aceleración real en medidas de tiempo en lugar de ciclos de reloj se descarga la aplicación en la plataforma Zybo y se ejecuta el comando "*time*" siendo el resultado obtenido, en media, 19,36 *fps*, inferior al del punto de partida, 19,44 *fps*. Al igual que para la quinta

aproximación, este proporciona una deceleración del 0,4% descartando su implementación de manera inmediata.

Puesto que es una opción que se descarta no se presta excesiva atención a los recursos empleados, aunque estos (observables en la [figura 38], los mismos que para la séptima aproximación), al igual que para la quinta aproximación, no serían problema para su uso.

9. **Novena aproximación: Creación de una nueva función que realiza de manera conjunta el cálculo de la proyección y la recuperación de la imagen.**

Para este caso, se recuperan los conceptos del algoritmo PCA [capítulo 4.3.1]. Al hablar del proceso *online* hablamos de dos operaciones de multiplicación de matrices que implican el cálculo de la proyección de la imagen que entra al algoritmo en el espacio de transformación de las componentes principales y la recuperación de esta del mismo espacio. Para obtener la recuperación de la imagen del espacio transformado se requiere introducir la imagen proyectada que se obtiene como resultado de la función del cálculo de la proyección, es decir, la función de recuperación de la imagen del espacio transformado "*recuperación_img"* emplea como parámetro de entrada la salida de la función del cálculo de la proyección "*calculo_proyeccion"*.

Es por esto por lo que en esta aproximación lo que se plantea es la implementación de una única función que contiene dos nidos de bucles (cada uno realizando una multiplicación matricial) y que realiza como primer paso la obtención de la proyección y como segundo, de manera inmediata sin abandonar la función, la obtención de la imagen recuperada. De esta manera, introduciendo a la función la imagen que llega a la parte *online* del algoritmo se obtiene de manera directa, como salida de la función, la imagen recuperada del espacio de transformación, lista para la comparación y la obtención del error de reconstrucción.

Es decir, el segundo nido de bucles usa el resultado del primer nido de bucles, evitando la extracción de la variable intermedia al programa principal y la copia en la memoria del micro de ésta.

Además de la creación de esta nueva función, se introduce junto a los demás pragmas una nueva sentencia de HLS: "*array_partition"*:

Esta directiva particiona un array en arrays más pequeños, e incluso en elementos individuales. Este particionado resulta a nivel de registro en múltiples pequeñas memorias (o múltiples pequeños registros) en vez de una larga. El empleo de esta directiva aumenta de manera significativa la cantidad de puertos de lectura y escritura para el almacenamiento, y requiere mayor cantidad de instancias de memoria o registros, pero es el hecho de que potencialmente mejora el rendimiento del diseño lo que lo convierte en una directiva interesante a probar.

Al usar esta directiva se requiere indicar los siguientes parámetros:

- Variable =<*name*>, indicando el nombre del array sobre el cual se realiza el particionado.
- <*tipo*> = indica de manera opcional el tipo de particionado. Si no se indica por defecto realiza un particionado completo (realiza un particionado del array en elementos individuales). También están el tipo cíclico donde se crean arrays más pequeños entrelazando elementos de la matriz original y el tipo bloque que crea arrays más pequeños a partir de bloques consecutivos del array original.
- factor=<*int*>, el cual especifica el número de arrays más pequeños que se van a crear.
- dim =<*int*>, que especifica que dimensión de un array multidimensional se particiona.

En este caso se ha probado a realizar un particionado de tipo bloque, de factor 16, a la dimensión 1 del array de entrada que representa la imagen de entrada. Para poder realizar la optimización con esta directiva, se requiere copiar de manera interna la variable ya que sino SDx no permite la síntesis del pragma.

Para mejor compresión de lo que se relata se adjunta a continuación la codificación de la función:

```
#pragma SDS data zero_copy(input[0:totpix])
#pragma SDS data access_pattern(Utrans:SEQUENTIAL)
#pragma SDS data access_pattern(U:SEQUENTIAL)
#pragma SDS data access_pattern(recup_img:SEQUENTIAL)

void proyeccion_recup(float Utrans[tComp][totpix],short input[totpix],float
U[totpix][tComp],float recup_img[totpix]){
	int l;
	int i,k;
	int a,c;
	float omega[tComp][1];
	short in[totpix][1];
	for(l=0;l<totpix;l++){
			in[l][0] = input[l];
	}
	#pragma HLS ARRAY_PARTITION variable=in block factor=16 dim=1
	#pragma HLS INLINE
	loop1: for(i=0;i<tComp;i++){
			omega[i][0] = 0;
			for (k=0;k<totpix;k++){
				#pragma HLS PIPELINE II=1 rewind
				omega[i][0] = omega[i][0] + Utrans[i][k]*in[k][0];
			}

		}
	loop2: for(a=0;a<totpix;a++){
			recup_img[a] = 0;
			for (c=0;c<tComp;c++){
				#pragma HLS PIPELINE II=1 rewind
				recup_img[a] = recup_img[a] + U[a][c]*omega[c][0];
			}
		}

}
```

Como se puede observar se emplean las mismas sentencias pragma SDS para el acceso a los datos que se emplean en las aproximaciones anteriores. Como recordatorio el uso de estas se debe a las dimensiones de los parámetros de entrada ya que exceden las dimensiones del bus de interconexión BRAM.

Dentro de las variables que se declaran se observa que se ha declarado la variable "*omega[tComp][1]"* que representa el vector columna que contiene la proyección de la imagen en el espacio de transformación de las componentes principales. Esta variable permanece local a la función, no siendo una variable que se extraiga al programa principal. Como se puede observar esta una vez conformada pasa como parámetro al segundo bucle para realizar la multiplicación elemento a elemento con intención de obtener la imagen recuperada del espacio transformado.

Se observa también la declaración de la variable "*in[totpix][1]*" la representa una copia en la memoria de la lógica programable del parámetro de entrada "*input[totpix]"* evitando el acceso a la memoria del microcontrolador para la búsqueda de este en la multiplicación elemento a elemento. En su lugar consulta directamente la memoria BRAM de la lógica programable, permitiendo además aplicar la sentencia "*array_partition"* sobre la variable "*in[totpix][1]"*. El

resto de las sentencias pragma HLS son las mismas que para las aproximaciones anteriores: "*inline"* y "*pipeline"*, [figura 42].

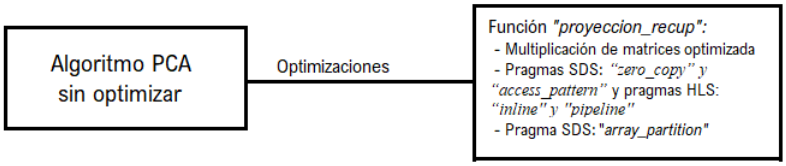


Figura 42: Aproximación 9

En esta aproximación se modifica también la codificación de la multiplicación de matrices, pasando de usar la codificación estándar a la eliminación de uno de los bucles interiores, como se explicaba previamente.

Los resultados obtenidos para esta aproximación se muestran en las figuras [34,35]. En la [figura 43] se muestra la aceleración estimada por SDx. En este caso la ejecución de la función aislada por la lógica programable del hardware tampoco supone la reducción del número de ciclos que requiere su ejecución, por el contrario este número aumenta en un 5%. A pesar de tardar un número de ciclos de reloj ligeramente superior al ejecutarse en el FPGA en lugar de en la CPU, libera esta segunda y reduce el número de ciclos de reloj totales, permitiendo una aceleración de la aplicación en su conjunto total del 28%.

Performance estimates for 'main' function

SW-only (Measured cycles)	202776694478
Hardware accelerated (Estimated cycles	158169144048
Estimated speedup	1.28

Details

Performance estimates for 'proyeccion_recup in main.cpp:1 ...

SW-only (Measured cycles)	21524632
Hardware accelerated (Estimated cycles	22738204
Estimated speedup	0.95

Figura 43: Aceleración en HW de la aproximación 9

Al llevarlo de manera practica a la plataforma, y ejecutarlo, como para el resto de los casos, se obtiene una tasa de procesado media de *30,06 fps* lo que supone una gran aceleración con respecto al escenario del que se partía de 19,44 *fps*, primera aproximación. Se consigue entonces de manera real en la ejecución una aceleración del 54,62%. Este resultado alcanza lo que se esperaba conseguir al optimizar este algoritmo de procesado de imágenes, lo que permite descargar todas las aproximaciones obtenidas hasta este punto.

Al haber alcanzado una opción de implementación viable, se requiere observar que la escasez de recursos no sea una limitación cuando se lleve a cabo. Por ello se observa la [figura 44]. En este caso la cantidad de memoria interna, BRAM, empleada aumenta, usándose un 40,71% del total. Este porcentaje es superior a los obtenidos anteriormente y se debe a la copia de las variables "*input[totpix][1]"* y "*omega[tComp][1]"*. En caso de las DSP se utiliza el mismo porcentaje que para la octava aproximación, mientras que la utilización de las LUTs y las FF aumenta (2,27%,

30,12% y 18,06%, respectivamente). Con esto se concluye que la escasez de recursos no es limitante para la implementación.

Resource	Used	Total	% Utilization
DSP	5	220	2.27
BRAM	57	140	40.71
LUT	16023	53200	30.12
FF	19216	106400	18.06

Figura 44: Recursos empleados en la aproximación 9

10. **Décima aproximación: Funciones de recuperación de la imagen y cálculo de la proyección en HW, en este caso con particionado del array de la imagen de entrada**

Puesto que el uso de la sentencia pragma de partición del array, junto con la implementación de la nueva función "*proyección_recup"*, proporcionó resultados que cumplían los objetivos planteados, se decidió comprobar cómo funcionaba esta sentencia a la tasa de procesado y a la aceleración del algoritmo si se aplicaba a las funciones de cálculo de la proyección y de recuperación de la imagen proyectada nuevamente separadas y ambas ejecutadas en HW.

Se vuelve entonces a la codificación de la tercera aproximación y se envían las mismas funciones para ejecutarlas en la lógica programable: "*calculo_proyeccion"* y "*recuperación_img"*. Además, se emplean las mismas directivas SDS de acceso a los datos: "*zero_copy"*, "*access_pattern"* y "*data_mover"*. Y para optimizar la ejecución en hardware se emplean tanto las directivas "*pipeline"* e "*inline"* en ambas funciones, mientras que la directiva "*array_partition"* se aplica tan solo en el cálculo de la proyección.

Para esta aproximación se modifica también la codificación matemática del algoritmo de multiplicación matricial puesto que para casos anteriores se observa que la supresión del bucle intermedio ofrece una tasa de procesado de imágenes superior [figura 45].

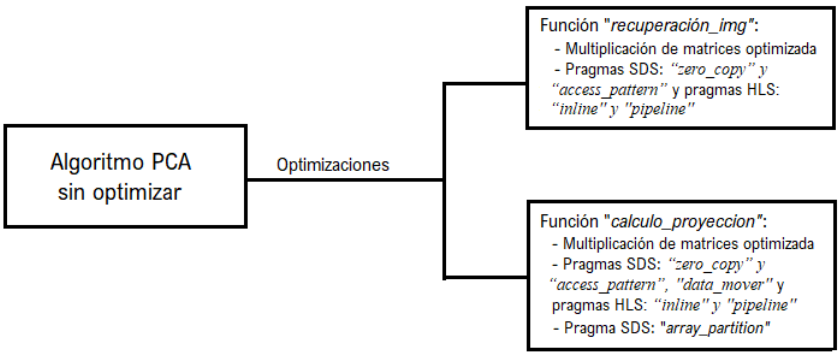


Figura 45: Aproximación 10

En términos de aceleración estimada por SDSoC con respecto a la aproximación anterior, ambas de las aproximaciones suponen un aumento del 5% de los ciclos de reloj al ejecutar las funciones en hardware con respecto a su ejecución en software, pero a diferencia del anterior este proporciona una disminución del número de ciclos totales superior, acelerando el conjunto en un 35% frente al 28% anterior.

Performance estimates for 'main' function

SW-only (Measured cycles)	223470419450
Hardware accelerated (Estimated cycles	166075063547
Estimated speedup	1.35

Details

Performance estimates for functions 'recuperacion_img in ...

SW-only (Measured cycles)	26613868
Hardware accelerated (Estimated cycles	27872123
Estimated speedup	0.95

Figura 46: Aceleración HW en la aproximación 10

Al ejecutar la aplicación en la plataforma se calculó que, de media, la tasa de procesado volvía a disminuir a 19,25 *fps*, de nuevo inferior que la tasa del punto de partida 19,44 *fps*, volviendo a decelerar la ejecución del algoritmo un 0,9%. De nuevo se descarta esta aproximación al no cumplir con el propósito de la aproximación.

En términos de recursos [figura 47], al realizar una copia interna de la variable (vector columna) que representa la imagen de entrada, la memoria BRAM vuelve a rondar la mitad de su capacidad (40,71%). Aun así, puesto que la aproximación es descartable no se enfatiza en esta utilización de recursos.

Resource	Used	Total	% Utilization
DSP	10	220	4.55
BRAM	57	140	40.71
LUT	16305	53200	30.65
FF	19428	106400	18.26

Figura 47: Recursos empleados en la aproximación 10

11. **Onceava aproximación: Funciones de recuperación de la imagen y cálculo de la proyección en HW, en este caso con implementadas con coma fija.**

En esta onceava aproximación, antes de descartar por completo optimizar únicamente aquellas funciones que requieren multiplicación matricial (“*calculo_proyeccion”* y “*recuperación_img”*), se plantea la realización de la multiplicación matricial en coma fija [figura 48].

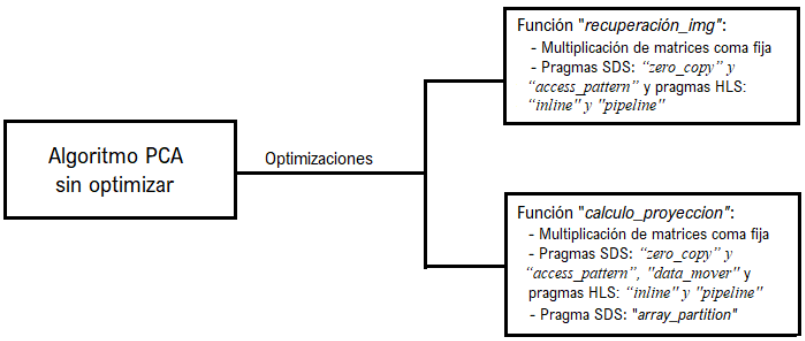


Figura 48: Aproximación 11

Cabe destacar que el tipo de dato en coma fija solo se puede implementar en HW puesto que el microcontrolador no soporta la codificación en coma fija. Por ello para poder implementarlo se

requiere crear nuevas variables del tipo de dato coma fija dentro de la propia función destinada a HW y asignarles su valor correspondiente de coma flotante.

Como se explicaba en la introducción de este capítulo, al hablar de las distintas formas de codificar el algoritmo con Matlab, la elección de la longitud de la palabra y la cantidad de bits que representan cada una de las partes es trivial para que los resultados no sean erróneos.

Es en este punto donde se emplean los resultados obtenidos de la codificación con Matlab en coma fija del algoritmo. Lo que se realizó en esta herramienta, recordemos, fue copiar el algoritmo en un nuevo fichero y duplicar las variables creadas pero usando el operador *"fi"* en su creación. Esto significaba representar la variable con una longitud de palabra fija y un número de bits para la representación de la parte decimal fija.

Recordando los resultados que se obtuvieron y los que se exportaron a la codificación en C/C++ fueron, para un error cuadrático medio del 2,6186%:

- La matriz de media nula, A, para la cual se obtuvo un error porcentual del 0% con una longitud de palabra de 18 bits con signo y una parte fraccional de 4 bits.
- La matriz de covarianza, C, para la cual se obtuvo un error porcentual del 5,12% con una longitud de palabra de 20 bits con signo y una parte fraccional de 2 bits.
- El valor del RMSE para el cual se obtuvo un error porcentual del 0.0013%.
- El error entre imágenes proyectadas para el cual se obtuvo un error porcentual del 2,76% con una longitud de 39 bits con signo y 12 bits de parte fraccional.

Para poder exportar esto a SDx asegurando su correcto funcionamiento (resultados correctos de la multiplicación de matrices), considerando que su optimización se iba a realizar con HLS, se decidió implementar de manera individual con esta segunda herramienta una IP que se encargase de realizar este tipo de multiplicación con coma fija. Para ello se planteó un banco de pruebas en el que se creaban las matrices de las dimensiones que se requieren en SDx para el algoritmo, (dimensiones de la multiplicación para el cálculo de la proyección: A[4][65536] y B[65536][1]) del tipo de datos coma flotante, empleando la librería HLS "*ap_fixed"*, y se pasan con este tipo de dato a la función que realiza la multiplicación matricial.

La diferencia entre la codificación en coma fija de Matlab y la coma fija en lenguaje de alto nivel C/C++ es mínima, puesto que la única diferencia es que en vez de indicar como parámetro el número de bits que se emplean para la parte fraccional (Matlab) se indican el número de bits que se emplean para la parte entera (C/C++).

En esta función se crean las nuevas variables de tipo de dato fijo con la longitud de palabra que se ha calculado previamente: 18 bits con signo de los cuales 4 representan la parte fraccional para A, lo mismo para B que representa la matriz de transformación y 39 bits con signo y 12 de parte fraccional para el resultado. De esta manera se aseguran resultados fiables, con menor precisión, pero ciertos y sin *overflow*.

Tras obtener una IP funcional, esta se traslada a SDx mediante la copia de la función en el fichero fuente que contiene el resto de las funciones multiplicativas y se compila.

Para esta aproximación en coma fija no se pudo pasar de la compilación, puesto que al extraer los resultados de la estimación de recursos se pudo observar que se requería muchísima más memoria (más del doble) interna BRAM de la que la plataforma disponía [figura 49].

Esto tenía sentido puesto que se requiere crear una copia interna de cada una de las variables que se pasan como parámetros en la memoria BRAM, y las dimensiones de las matrices son muy elevadas, de [4][65536] y [65536][1].

Llegados a este punto se decide que de aquí en adelante, puesto que ninguna de las aproximaciones anteriores en las que enviando las dos funciones de multiplicación de matrices "*calculo_proyeccion"* y "*recuperación_img"* consiguen obtener una tasa de procesado de imágenes que se acerque a lo deseado, se descarta el envío de ambas de manera simultánea para su ejecución en hardware ya que no parecen no ser compatibles.

Resource	Used	Total	% Utilization
DSP	11	220	5
BRAM	283	140	202.14
LUT	31921	53200	60
FF	37490	106400	35.23

Figura 49: Recursos de la aproximación 11 en coma fija

12. **<u>Duodécima aproximación: Funciones de recuperación de la imagen y cálculo del mapa de distancias en HW.</u>**

Para este escenario se vuelve a echar un vistazo a los resultados obtenidos por el *"TCF Profiler"* [figura 17], observando que la segunda función candidata a la optimización en HW es la referida al cálculo del mapa de distancias. Esta función se basa sencillamente en dos bucles individuales que realizan el cálculo de la distancia euclídea entre la imagen recuperada del espacio transformado y la imagen original. El primer bucle realiza el cuadrado de la diferencia, y el segundo realiza la raíz cuadrada.

Como bien se ha visto en los casos anteriores, para optimizar un bucle, la directiva óptima es la que proporciona la concurrencia mediante un desenrollado de bucle y minimiza el intervalo de inicialización: *HLS PIPELINE.*

Por ello en esta aproximación se envían para ejecutarse en la lógica programable las dos funciones que más porcentaje de tiempo de CPU requieren, "*recuperacion_img"* y "*calculo_mapa"*. En ambas se emplean las directivas pragma SDS de acceso a los datos: "*access_pattern"* y "*zero_copy",* y las directivas pragma de HLS *"PIPELINE"* e "*INLINE"*. En este caso, para la multiplicación de matrices de la recuperación de la imagen se emplea la multiplicación estándar [figura 50].

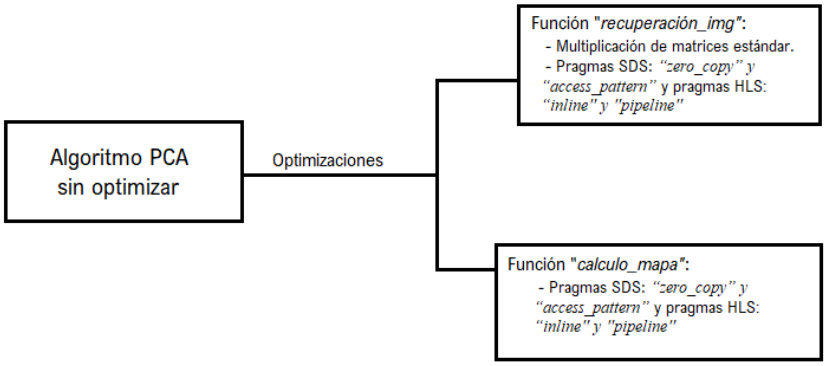


Figura 50: Aproximación 12

La aceleración estimada por la aplicación en este caso [figura 51] proporciona una aceleración increíble en el número de ciclos necesarios para ejecutar el cálculo del mapa de distancias en

hardware frente al número de ciclos que se requiere en software. Como se puede observar en la tabla intermedia de la figura se ejecutan casi 10 veces menos ciclos en hardware que en software, mientras que la función que recupera la imagen, tabla inferior de la figura, no se ve acelerada sino que ejecuta un 28% más de ciclos de reloj en hardware que en software. La liberación de la ejecución de estas dos funciones de la CPU del microcontrolador supone una aceleración del programa principal de casi el 50%, un 49%, siendo el menor número de ciclos de ejecución que se ha obtenido hasta ahora en la prueba de todas las aproximaciones.

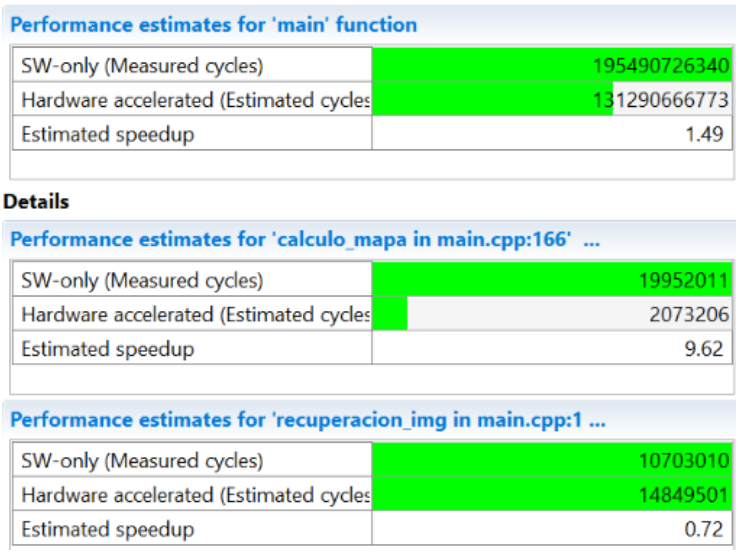


Figura 51: Aceleración HW de la aproximación 12

Al ejecutar el algoritmo en la plataforma Zybo, se obtuvo una media de tasa de procesado de 29,85 *fps*, obteniendo una aceleración con respecto al punto de partida obtenido con la primera aproximación de 19,44 *fps*. Es decir, se obtiene un valor real de aceleración del 53,54%.

En este caso a la hora de adjuntar la captura realizada de la pantalla de SDx donde deberían observarse los recursos estos no aparecen, pero, puesto que se realizó una copia en Excel de los recursos empleados, se adjuntan estos [figura 52]. La utilización de memoria BRAM excede el 50% del total, con un 63,67%, mientras que los DSPs, las LUTs y las FF se siguen manteniendo debajo del 50%. Esta optimización no solo es viable al alcanzar los objetivos de tasa de procesado sino que la utilización de los recursos no resulta limitante.

Resource		Total	% Utilization
DSP	10	220	4.55
BRAM	89	140	63.57
LUT	18984	53200	35.68
FF	23032	106400	21.65

Figura 52: Recursos empleados en la aproximación 12

13. **Treceava aproximación: Funciones de recuperación de la imagen y cálculo del mapa de distancias en HW, junto con el cálculo de la proyección.**

Como última aproximación se juntan las tres funciones que se han enviado para su ejecución a HW a lo largo de este trabajo, para observar si se obtiene o no aceleración.

Para este caso se envían a ejecutar en HW las funciones "*calculo_proyeccion"*, "*calculo_mapa"* y "*recuperación_img"*. Las optimizaciones con sentencias pragma que se aplican a cada una de ellas se resumen a continuación [figura 53]:

- A las funciones que realizan el cálculo de la proyección y recuperación de la imagen, es decir las dos que multiplican matrices, se le aplican las directivas de acceso a datos SDS "*zero_copy"*, *"access_pattern"* y "*data_mover"*, y las directivas de optimización HW de HLS: "*PIPELINE"* e "*INLINE"*.Además, la codificación matemática es la de la multiplicación de matrices estándar.
- A la función que realiza el cálculo del mapa de distancias se le aplican las directivas de acceso a datos SDS: "*zero_copy"* y "*access_pattern"*, y la directiva de optimización HW HLS "*PIPELINE"*.

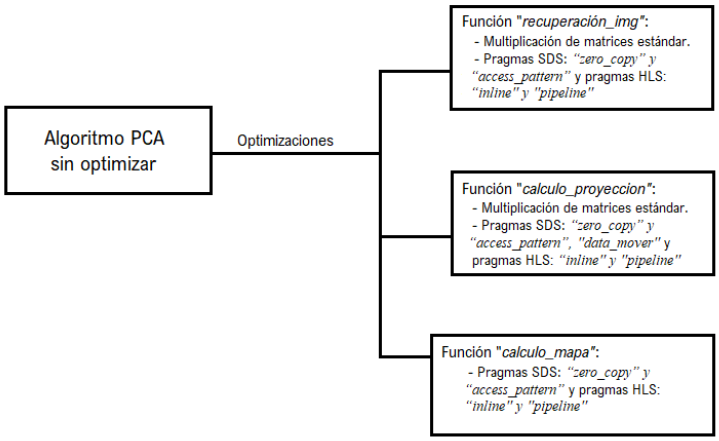


Figura 53: Aproximación 13

Observando la [figura 54], la aceleración obtenida para el programa principal es de un 102%, mientras que las funciones individuales de recuperación de la imagen y cálculo de la proyección se ejecutaban en un número superior de ciclos en HW que en SW, un 4% más. El cálculo del mapa de distancias si presenta esta aceleración ejecutándose en 6 veces menos ciclos que si se ejecutase en software:

SW-only (Measured cycles)	195490726340
Hardware accelerated (Estimated cycles	96966244112
Estimated speedup	2.02

Details

Performance estimates for 'calculo_mapa in main.cpp:166' ...

SW-only (Measured cycles)	19952011
Hardware accelerated (Estimated cycles	3157337
Estimated speedup	6.32

Performance estimates for functions 'recuperacion_img in ...

SW-only (Measured cycles)	26770026
Hardware accelerated (Estimated cycles	27872123
Estimated speedup	0.96

Figura 54: Aceleración HW de la aproximación 13

Ejecutando el algoritmo en la plataforma, la tasa media de procesado obtenida es de 29,70 *fps* siendo también un valor de tasa deseado, suponiendo una aceleración real del 52,78% con respecto al punto de partida inicial, donde se obtenía una tasa media de procesado de 19,44 *fps*.

Esta aproximación no es tan deseada en términos de recursos [figura 55], puesto que el hecho de mandar tantas funciones para ser ejecutados en HW consume casi en su totalidad la memoria BRAM, un 98,57%, y hace que la utilización de las LUTs que hasta entonces se mantenía por debajo del 50% exceda este porcentaje. Las LUTs y las DSPs siguen por debajo del 50%. Puesto que es una estimación de los recursos empleados y la utilización de memoria BRAM roza el límite, se decide descartar esta implementación para su posterior uso.

Resource		Total	% Utilization
DSP	15	220	6.82
BRAM	138	140	98.57
LUT	30101	53200	56.58
FF	36204	106400	34.03

Figura 55: Recursos empleados en la aproximación 13

14. **Conclusiones:**

Una vez expuestos las trece distintas aproximaciones que se han planteado para encontrar una optimización y aceleración del algoritmo que proporcionase una tasa de *fps* en el rango de 25-30 *fps* que se deseaba, se muestra una tabla comparativa como resumen y apoyo para la decisión de la optimización que se va a emplear para trasladarse al sistema real:

Tabla 2: Comparativa de las aproximaciones de optimización

Resultados de las tasas	Release (*fps*)	Debug (*fps*)	Aceleración obtenida (%)
Optimización 1	19,4434503	6,9538225	Se compara con este escenario
Optimización 2	12,9793334	5,2597183	No se obtiene aceleración
Optimización 3	19,7707215	8,0849797	1,683195342
Optimización 4	20,0683413	7,5127921	3,213889915
Optimización 5	19,1910271	7,0101297	No se obtiene aceleración
Optimización 6	19,7276441	7,0339559	1,461643214
Optimización 7	19,8049581	7,9640828	1,859278163
Optimización 8	19,3672014	7,1092063	No se obtiene aceleración
Optimización 9	30,0636414	7,5714286	54,62091865
Optimización 10	19,2590623	7,1832336	No se obtiene aceleración
Optimización 11	COMA FIJA		No se puede llevar a la práctica
Optimización 12	29,8489011	8,2341796	53,51648352
Optimización 13	29,7020230	7,4519890	52,76107162

Como se puede observar en la [tabla 2] las aproximaciones óptimas: la novena, la duodécima y la treceava, que proporcionan una aceleración del algoritmo superior al 50%, es decir, logra ejecutarse a más del doble de velocidad que el escenario original del que se partía. Puesto que la treceava roza los límites de los recursos BRAM disponibles por la plataforma, y que no es la que mayor aceleración proporciona, se descarta, y la que se elige como función a implementar en el sistema real es la optimización 9. En esta optimización, cabe recordar, se creaba una nueva función que calculaba conjuntamente la proyección de la imagen en el espacio de las componentes principales y la imagen que se recuperaba, devolviendo esta última como parámetro de salida de

la función. Empleaba la codificación matemática del algoritmo de multiplicación matricial estándar, aplicaba las sentencias de acceso a los datos SDS "*zero_copy"* y "*access_pattern"* y optimizaba la ejecución de la función en hardware mediante las directivas pragma HLS "*array_partition"*, "*pipeline"*, e "*inline"*.

Se adjunta, además, resultados gráficos de la aplicación del algoritmo en la plataforma con intención de mostrar el correcto funcionamiento de este, puesto que parte del código indica que se almacenen en la microSD aquellas imágenes en las que se ha detectado un objeto, empleando la función de la librería "*OpenCV"* nombrada anteriormente, siendo el resultado el que se observa en la [figura 56]:

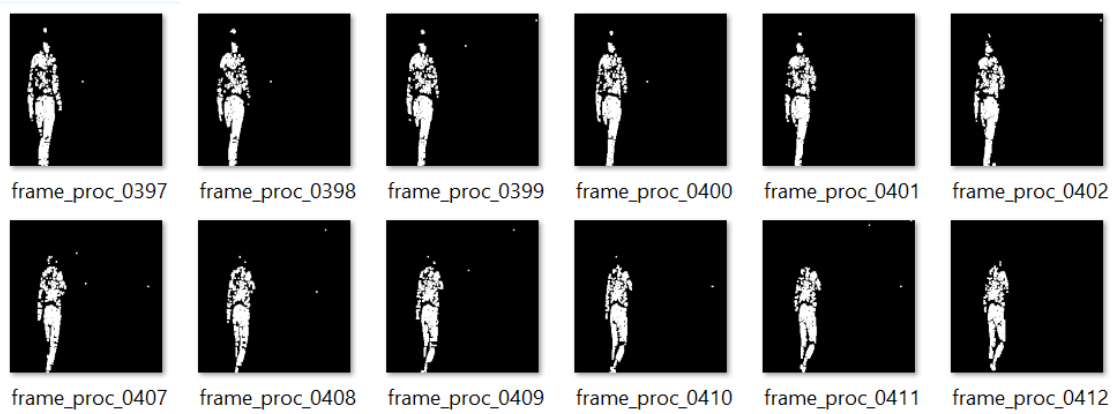


Figura 56: Resultados tras el procesado en la Zybo de las imágenes con objeto detectado

Como se puede observar el resultado es el mismo que el obtenido en la implementación con Matlab y con Eclipse que se muestra en las figuras [12] y [13], respectivamente, evidenciando el correcto funcionamiento del algoritmo PCA sobre la plataforma Zybo Z720.

4.4.2 Traslación del algoritmo PCA a tiempo real:

En el capítulo anterior se narraban las distintas aproximaciones que se han empleado para conseguir optimizar el algoritmo PCA de manera que sea capaz de procesar las imágenes entrantes a una tasa de 20-30 *fps*. Este análisis realizado concluyó con una aproximación que conseguía procesar 30,036 *fps* gracias a la sustitución de las funciones de "*calculo_proyeccion"* y "*recuperación_img"* por una única función que realiza las dos multiplicaciones matemáticas en una mediante dos nidos de bucles [capítulo 4.4.1, "*Aproximación 9*"]. Además de esta modificación, cabe recordar que empleaba la codificación matemática del algoritmo de multiplicación matricial estándar, aplicaba las sentencias de acceso a los datos SDS "*zero_copy"* y "*access_pattern"* y optimizaba la ejecución de la función en hardware mediante las directivas pragma HLS "*array_partition"*, "*pipeline"*, e "*inline"*.

Una vez obtenida la aproximación que cumple los objetivos, la finalidad es la ejecución de este algoritmo en tiempo real mediante la plataforma expuesta en la [figura 8]. Esto es capturando un fondo estático y detectando si se introduce un nuevo objeto en la escena a tiempo real. La captura de imágenes se realiza mediante el módulo Pcam 5C [capítulo 4.3.3, "*Elementos hardware"*].

Este sistema de tiempo real se constituye, además del módulo PCam5C, por un monitor conectado al HDMI de salida de la plataforma Zybo Z720, de manera que toda imagen procesada será extraída por este y presentada en el monitor. Esto permitirá observar la introducción de nuevos elementos en la escena establecida como fondo.

Para poder llevar a cabo este sistema de tiempo real se debe realizar la combinación del algoritmo PCA con una aplicación que sea capaz de realizar un *streaming* de las imágenes capturadas por la cámara hacia el monitor. Por ello se plantea, antes de introducir el procesado, conseguir que las imágenes capturadas por la cámara se observen según se capturan por la pantalla, y tras esto introducir entre la presentación y la captura de imágenes el procesado mediante el algoritmo PCA.

Cabe recalcar que este módulo PCam 5C solo puede controlarse mediante un SoC con un controlador MIPI CSI-2 en hardware y requiere un software de configuración complejo que se encargue de configurar de manera correcta el controlador y el sensor de la cámara. Es por esto por lo que se encuentran ciertas complicaciones al intentar implementar una aplicación que requiera de este módulo. Para solucionar esto, el fabricante de la cámara propone hacer lo siguiente:

- O bien pagar una licencia para obtener un MIPI CSI-2 IP diseñado para el funcionamiento sobre la FPGA, lo cual proporciona una solución sólida y un buen soporte de software con los controladores Linux integrados,
- O bien desarrollar el hardware y el software desde cero, lo cual requiere una gran inversión de tiempo de alguien con habilidades avanzadas y con acceso a información que solo Digilent (el fabricante) proporciona, como las especificaciones cerradas y las hojas de datos.

Puesto que estas son dos opciones que requieren de expertos en la materia, para permitir a un usuario con conocimientos medios-básicos usar este módulo, Digilent ha creado un conjunto de núcleos IP de Vivado de código abierto que funcionan con el módulo PCam 5C en las placas de FPGA y en las placas que emplean un SoC Zynq. Usar estos núcleos prediseñados limitan el funcionamiento del módulo, además de que su software no aprovecha al máximo todas las características del sensor, afectando a la calidad de la imagen producida.

Para este trabajo las características del sensor del módulo PCam5C y su funcionamiento con el núcleo IP de Vivado diseñado para la plataforma Zybo Z720 proporciona resultados lo suficientemente válidos como para no requerir de una licencia ni de un desarrollo hardware/software desde cero, agilizando el desarrollo de esta aplicación.

Este núcleo IP de Vivado, como se nombra en el párrafo superior, tiene una versión diseñada para la plataforma Zybo Z720. Esta versión viene denominada como plataforma "*ZyboZ720-reVISION"*. En esta, Digilent, proporciona tanto la configuración y las restricciones de puertos (habilitando el puerto MIPI CSI-2 para el conexionado de la cámara) para el FPGA como la implementación del sistema operativo con las librerías de visión "*OpenCV"* y *"xfOpenCV"* (por ello el nombre de "*reVISION*"), como se comentaba en el apartado 4.4.1 de este libro, tras la explicación de la codificación con Matlab. Además, el diseño de la plataforma proporciona proyectos de ejemplo en los que basar aplicaciones de visión empleando estas dos librerías.

Dentro de este conjunto de código abierto de ejemplos que Digilent proporciona, existe un denominado "*Filter 2D for PCam 5c"*, el cual se basa en aplicar un filtrado con la función "*filter2d"* de la librería "*OpenCV"* y extraer la imagen filtrada por un HDMI hacia una pantalla. En este trabajo se va a modificar y aprovechar este código abierto con la intención de usar la captura de video de la cámara y el *streaming* hacia la pantalla, realizando el procesado del algoritmo PCA en lugar de un filtrado.

Este ejemplo de filtrado emplea una interfaz de programación de aplicaciones denominada V4L2 (*"Video for Linux2"*) que permite la captura de video para un sistema operativo Linux, el cual viene implementado en la plataforma. Esta interfaz captura el video como una secuencia de *frames* que va almacenando en un buffer de memoria DDR de la plataforma. Para aplicar el procesado de estas *frames*, va leyendo el buffer DDR de entrada con el método FIFO (*"First In – First out"*),

mediante DMA, les aplica el filtrado y almacena la imagen en un nuevo buffer, esta vez de salida, que el módulo PCam5C va leyendo gracias a la interfaz V4L2. Este proceso se describe de manera gráfica en la [figura 57].

De esta demo/código ejemplo se eliminan las dos carpetas asociadas a los filtros puesto que no se van a usar, y se crea un nuevo fichero que va a introducir las funciones codificadas según la novena aproximación del algoritmo PCA. Del código ejemplo, el único fichero que se modifica es el "*main.cpp"*, el resto permanecen según se dan.

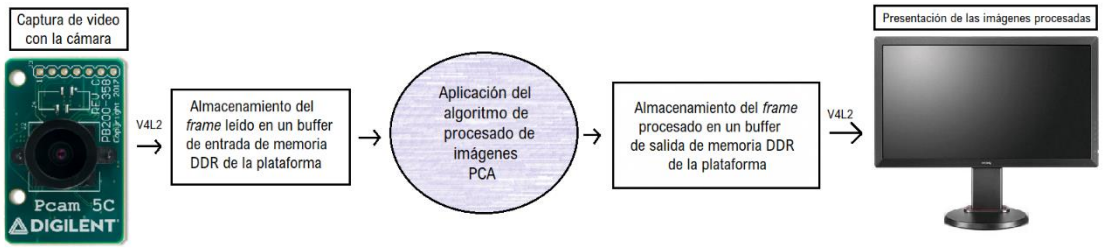


Figura 57: Diagrama de flujo del algoritmo PCA en el sistema de tiempo real

Haciendo un poco más de hincapié en el código fuente del archivo "*main.cpp"* que se va a modificar del ejemplo de Digilent que se usa como base del sistema en tiempo real, este realiza la inicialización de la entrada de video y asigna a esta distintos buffers de entrada de memoria DDR en el microcontrolador. La resolución de la entrada y salida de video es uno de los parámetros que se controlan en la inicialización de la entrada y la elección de estos se controla mediante la llamada, previa a la ejecución del ejecutable en la plataforma Zybo, a uno de los *scripts* que se encuentran dentro de la carpeta. Estos *scripts* configuran la resolución a 1080p o 720p, pudiendo elegir. Para poder realizar esta elección se debe hacer uso de la línea de comandos del sistema operativo escribiendo *"./config_pcam_xxxp.sh*" antes de ejecutar de la misma forma el fichero *".elf*" de la aplicación, sustituyendo "*xxx"* por la resolución deseada. De estos formatos, las pruebas se han realizado con 1080p, puesto que el monitor permitía mostrar las imágenes de salida en esta resolución.

Tras esta inicialización en el código, lo siguiente que se hace es tomar control del dispositivo por el que se van a mostrar las imágenes capturadas y se ajusta la resolución elegida previamente para que esta se corresponda con la entrada y se crean múltiples buffers en memoria DDR para almacenar los *frames* de salida.

Una vez realizada la configuración inicial de las características de la captación y representación de las imágenes, se comienza a capturar imágenes del módulo de la cámara. Es tras esta captura donde se introducen las modificaciones en el código. En este código se introduce la lectura de los ocho primeros *frames* que van a conformar el vector del fondo estático, determinando la escena en la que se desea detectar la aparición de nuevos objetos, y se detiene momentáneamente la captura de nuevas *frames* para realizar la parte del proceso *offline*. La detección de la captura de imágenes se debe a que estas no se van a emplear para detectar objetos en esas nuevas imágenes que se reciban, puesto que aún no está determinado cuales son las características principales que conforman el fondo.

Esta parte del proceso *offline* es sencillamente un copia-pega del fichero del programa principal "*main.cpp"* que se ha empleado para realizar el análisis del algoritmo de manera individual. Tras obtener el valor del RMSE se comienza el proceso *online*. En esta parte del trabajo, para los tres escenarios que se han probado y ejecutado en la plataforma, se obtiene un RMSE superior al 96%, en media de 96,92%.

Antes de comenzar el proceso *online* se reanuda la captura de imágenes de la cámara y según se vayan capturando se le introducen como parámetro.

Para este proceso *online* ocurre como con el proceso *offline*, se realiza un copia-pega de las llamadas a las funciones que se han empleado en la parte *online* del algoritmo en la codificación individual del análisis del capítulo anterior. Toda la codificación de estas funciones y sus llamadas en el código se pueden encontrar en el capítulo 8 ["*Planos"*].

Al iniciar ambos de los procesos, *offline* y *online,* se introduce, previo a las llamadas a las funciones, una leve modificación con respecto a los planteados en apartados anteriores. Esto se debe a que se observó que este código fuente captura las imágenes directamente en formato del espacio de colores "*YCbCr"* y que solo se empleaba el plano Y para la aplicación del procesado del *frame*, lo cual representa una imagen en escala de grises. En la codificación del algoritmo PCA en apartados anteriores se realizaba la conversión del espacio de color RGB a escala de grises como procesado previo a la aplicación de este.

Puesto que el resultado final de ambos es una imagen en escala de grises se decide observar si, en vez de convertir la imagen de "*YCbCr"* a RGB para su posterior conversión a otra escala de grises, se puede usar la imagen en escala de grises que proporciona el plano Y de este espacio de color. Este plano Y se corresponde con la luminancia, que es la escala de grises de la imagen original.

Para la comprobación de esta igualdad teórica, se decidió retornar a Matlab y realizar la lectura de la imagen mediante la función de lectura de "*OpenCV"* y tras esta convertirla al espacio de color YCbCr y realizar el algoritmo con el plano Y. Para observar la diferencia entre la aplicación del algoritmo con la imagen en escala de grises de la conversión con "*RGB2Gray"* y el plano Y de la conversión "*RGB2YCBCR"* se vuelve a dar uso al error cuadrático medio, siendo este del 0%, implicando que ambas escalas de grises son iguales.

Otra de las ventajas que proporciona este código fuente que se emplea como base es que presenta el uso de las funciones "*sds_clock_counter"*, lo que permite observar la tasa de procesado que se obtiene al ejecutar el algoritmo en tiempo real.

Una vez realizada la comprobación del *streaming* de imágenes sin aplicar el procesado de estas y tras haber incluido las modificaciones al código para poder aplicar el algoritmo PCA y la optimización de este determinada en el apartado anterior, se procede a realizar la ejecución en la plataforma y a observar los resultados que se obtienen. Como recapitulación, esta optimización unía las funciones que realizaban el cálculo de la proyección y la recuperación del espacio transformado, empleaba la multiplicación de matrices estándar e incluía las sentencias pragma SDS para el acceso de datos "*zero_copy"* y "*access_pattern",* y las sentencias de HLS "*PIPELINE" e "INLINE".*

Los resultados obtenidos para este escenario de tasa de procesado de imagen son en media de 14,17 *fps*, un resultado estimado cada 30 *frames* capturados que supone una velocidad un 52,8% por debajo de lo esperado. Comparar la velocidad obtenida en tiempo real con la velocidad obtenida en el algoritmo aislado no sería adecuado al usar un número de imágenes y un método temporal distinto para hallar esta tasa.

Aun así los resultados obtenidos son buenos ya que, a pesar de cierta latencia inicial que se debe a que se requiere accesos a buffers de memoria para almacenar las imágenes y leerlas antes de procesarlas y extraerlas hacia el monitor tras aplicarles el procesado, las imágenes se observan como si estuviesen a velocidad real (es decir, no se observa la escena a cámara lenta).

En la [figura 58] se muestran los resultados intermedios que se han observado en el monitor. Empezando de izquierda a derecha, y de arriba abajo, se muestran la imagen que se ha capturado y elegido para ser el fondo estático, una nueva imagen recibida donde se introduce un objeto como

lo es una mano, la imagen que conforma el mapa de distancias entre la imagen recibida y la proyectada en el espacio de las componentes principales, y finalmente la imagen con el objeto detectado en blanco pues se ha empleado el mismo método de umbralización que en los apartados anteriores.

En la parte inferior de esta última captura se puede observar el reflejo de los dedos puesto que también detecta el objeto en el reflejo de la pantalla del ordenador. Además, se observa en la captura el escenario en el que la cámara está capturando la imagen de la mano, y no solo lo que se ve en el monitor.

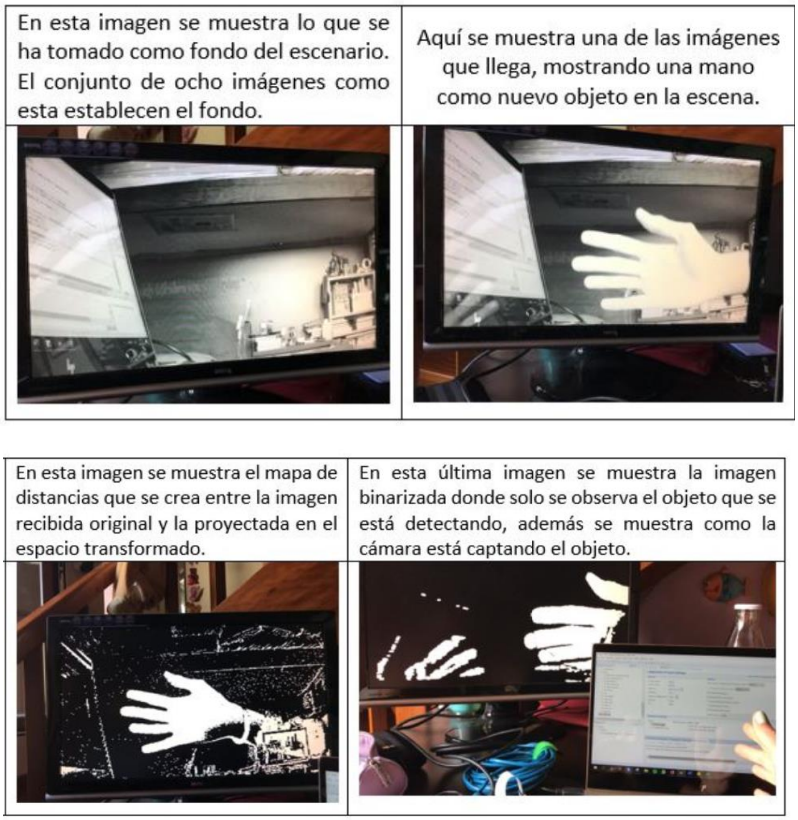


Figura 58: Demostración de los resultados de aplicar PCA con imágenes capturadas en tiempo real

Se adjunta a continuación otro ejemplo con un escenario de fondo distinto donde se va a demostrar la sensibilidad del algoritmo a los cambios de iluminación de la escena.

Comenzando de izquierda a derecha y de arriba a abajo [figura 59]:

- En la primera imagen se muestra la escena de fondo estático, donde se observa una habitación iluminada.
- En la segunda imagen se muestra una nueva *frame* capturada en el proceso *online* donde una persona comienza a entrar en la habitación.
- En la tercera imagen, primera de la segunda fila, se muestra como el algoritmo detecta a esa persona introduciéndose en la habitación.
- En la cuarta escena se muestra a la persona una vez está quieta en la habitación transcurrido un espacio de tiempo. La figura de la persona sigue siendo apreciable pero se observa que ahora casi toda la imagen contiene píxeles blancos y ciertos objetos que inicialmente formaban parte del fondo ahora pertenecen a la casilla de objetos detectados.

Figura 59: Segundo escenario de demostración de PCA en un sistema de tiempo real

Este problema tiene como solución algo que en [6] se prueba pero que como se nombraba anteriormente se deja como trabajo futuro al no ser necesario para acelerar el algoritmo PCA. Esto es la necesidad de una actualización de fondo.

Se sabe que al capturar una escena y conformar un escenario con 8 *frames* consecutivos la variación entre ellos es mínima por lo que la media dará una imagen con la misma luminosidad que la media de los 8 *frames.*

Si inicialmente se captura un escenario de noche y se pretende detectar objetos en ese escenario, funcionará bien hasta que la luz comience a aumentar, puesto que la escena de fondo no será la misma cuando la luz aumente, detectando objetos que realmente pertenecen al fondo.

Por ello cada cierto tiempo se requiere una actualización de fondo. Como se decía, en este trabajo no se abarca puesto que la finalidad es obtener una tasa optimizada del proceso *online* del algoritmo y la actualización de fondo requeriría realizar de nuevo la parte *offline*.

Se concluye este capítulo haciendo una recapitulación de lo que se ha realizado. En este se ha aplicado el algoritmo PCA a un sistema que captura imágenes y las envía para un procesado de detección de nuevos objetos en ellas en tiempo real. En este procesado se emplea el método de optimización que en el capítulo anterior se observó que daba la mejor tasa, siendo esta 30,063 *frames* por segundo.

A pesar de usar esta optimización y aprovechar los beneficios de la concurrencia hardware, los accesos a memoria DDR presentan ciertas limitaciones que resultan en una disminución del 52,8% de esta tasa en tiempo real.

5. Pliego de Condiciones.

En este capítulo se va a realizar una explicación de los requisitos necesarios para poner en marcha este proyecto si se quisiese exportar.

5.1 Requerimientos a nivel hardware:

Para la puesta en marcha de este proyecto se requiere una plataforma Zybo Z7-20 de Digilent, la cuál de debe conectar a un ordenador mediante dos conexiones: un cable USB a micro USB para establecer las comunicaciones serie asíncronas por la UART y un cable de conexión Ethernet. Además, al comprar la plataforma esta no dispone de alimentador, por lo que se requiere una fuente de alimentación con conector de salida centro positivo de 2,1x5,5x12 milímetros. La plataforma Zybo consta con un modo de alimentación mediante cable USB al ordenador, pero no es válido para aplicaciones complejas y con requerimientos de potencia.

Se requiere además un ordenador con al menos 0GB de memoria disponible para la instalación del software de Xilinx, ya que este requiere 37GB. Además, para la ejecución de este se requiere además tener al menos 10GB de memoria extra para poder correr estos a la vez de otros programas. Para la realización de este proyecto se emplearon dos ordenadores, un portátil donde se ejecutaron los programas de SDSoC, HLS, Matlab, y la máquina virtual que contenía Eclipse, y un ordenador de sobremesa con sistema operativo Ubuntu, aunque en este trabajo se ha empleado una versión de nombre "*Kubuntu*", donde se ejecutaron Vivado y Petalinux.

El ordenador portátil dispone de un procesador Intel Core i7 de octava generación, un procesador inferior puede producir que el ordenador se quede pillado o vaya mucho más lento. Además, dispone de una memoria gráfica NVIDIA GEFORCE. Por otro lado, el ordenador de sobremesa tiene un procesador AMD RYZEN 2700x y una memoria gráfica NVIDIA GEFORCE GTX180. Estas condiciones resultaron suficiente para poder llevar a cabo este proyecto.

Se requiere también el módulo PCam 5C, también de Digilent, el cual incluye el cable FFC para la conexión con la plataforma y un cable HDMI para la conexión con un monitor. En este caso se ha utilizado un monitor BENQ E2220HD el cual permite el formato de reproducción 1920x1080 y dispone de entrada de video HDMI.

Finalmente, como último requerimiento hardware, una tarjeta microSDHC de al menos 8GB con una clasificación de clase de velocidad 10.

5.2 Requerimientos a nivel software:

Como se explica más detalladamente a lo largo del capítulo 4, se requiere el uso de diversas herramientas software para la puesta en marcha de este proyecto. La versión empleada de todas las herramientas de Xilinx, aunque desactualizada, es la versión 2017.4. Esto se debe a que, como muy por encima se ha nombrado, a partir de la versión de SDSoC 2016.1, la plataforma Zybo dejó de estar entre las plataformas incluidas por defecto en el software. Por ello se debió introducir como plataforma personalizada. Para evitar la construcción de esta plataforma y facilitar la finalidad de este trabajo, puesto que la incorporación de la plataforma no era la finalidad, se decidió usar una plataforma predefinida que Digilent publicó para la versión 2017.4 en su repositorio de GitHub. Esta no es trasladable a versiones más recientes y es por ello por lo que se emplea la versión 2017.4 de todas las herramientas de Xilinx.

Al descargar de la página oficial de Xilinx, dentro de *Vitis SW*, la versión de SDx 2017.4, las herramientas de Vivado Design Suite y de Vivado HLS vienen incorporadas, de manera que se

instalan a la par que SDSoC. PetaLinux sin embargo que debe descargar por separado, misma versión 2017.4.

Para el uso de todas las herramientas de Xilinx se necesita una licencia. Existe una versión de prueba de 60 días gratuita, pero al tardar más tiempo en realizar el proyecto, se solicitó a la Universidad de Alcalá de Henares una licencia flotante para el uso de la herramienta.

Con respecto a la herramienta de Matlab, la versión instalada fue la R2017b, con licencia de estudiante proporcionada por la Universidad de Alcalá de Henares, y para la IDE de Eclipse, versión 2019-06, se requiere una máquina virtual (en este caso se empleó "*Virtual Box"*) donde correr un Ubuntu 18.04 LTS.

Los presentados en los párrafos anteriores son las herramientas empleadas a lo largo de este proyecto, pero para poder replicarlo y que no de problemas de compatibilidad lo más complejo es la introducción de la plataforma Zybo Z7-20 dentro de la herramienta SDSoC. Con respecto a la plataforma Zybo Z7-20 se requiere descargar de [18] los ficheros para la creación de la plataforma reVISION para aplicaciones de visión.

El proceso de introducción de la plataforma se explica en el capítulo 7 ("Manual de Usuario") pero para ello el software PetaLinux y Vivado Design Suite deben instalarse en Ubuntu 16.04 o superior, debe tener instalado Python en la versión 2.7.5, siendo esto indispensable ya que el código de ejecución de la orden *"petalinux-build"* que permite el compilado del sistema requiere esa versión (*), y se debe trabajar como"*/bin/bash"* en el Terminal y no como "*root"*. Además, los directorios de instalación de Vivado y PetaLinux deben ser *"/opt/Xilinx/SDx/"* y *"/opt/pkg/petalinux/"* respectivamente.

> (*) Para el caso de este proyecto se tenía instalada por defecto la versión 3.4 de Python, y puesto que cambiar a la 2.7.5 y que esta fuese la que se usase por defecto resultó imposible, se optó por crear un entorno temporal donde se usase la distribución 2.7.5, denominado como "*virtualenv"*.

6. Presupuesto.

En este capítulo se va a realizar una estimación de los costes tanto de los materiales empleados como de la mano de obra. En este caso el material a su vez se va a dividir en dos ramas: materiales software y materiales físicos para la realización del proyecto.

Costes del material físico				
Dispositivo	**Uso**	**Duración**	**Precio**	**Coste**
Portátil con procesador Intel Core i7 (8th Gen)	6 meses	5 años	999,00 €	99,90 €
Módulo PCam 5C	2 meses	4 años	42,18 €	1,76 €
Zybo Z720	6 meses	5 años	265,00 €	26,50 €
Cables de conexión:				
Cable de corriente	6 meses	3 años	17,75 €	2,96 €
Cable Ethernet	6 meses	3 años	10,11 €	1,69 €
Adaptador Ethernet	6 meses	3 años	16,99 €	2,83 €
Cable HDMI	6 meses	3 años	10,99 €	1,83 €
Montitor BENQ	2 meses	5 años	149,00 €	4,97 €
MicroSD	6 meses	5 años	18,80 €	1,88 €
Total del conjunto de material físico				**144,31 €**

Costes del software				
Programa	**Uso**	**Duración**	**Precio**	**Coste**
MATLAB 2017Rb	6 meses	4 años	1200 €	150 €
SDx 2017.4 y PetaLinux	6 meses	5 años	1000€	100 €
Office 2016	6 meses	3 años	270 €	45 €
Windows 10	6 meses	3 años	140 €	23,4 €
Total del conjunto de material software				**318,4 €**

El coste total del material hace un total de:

Total físico	144,31 €
Total software	318,4 €
Total materiales	**462,71 €**

Para la estimación de los costes de la mano de obra se calculan a partir del número de horas trabajadas, incluyendo los costes de la implementación del diseño y redacción del libro.

Mano de obra			
	Coste por hora	**Total de horas**	**Coste**
Implementación del diseño	40 €	600 horas	24.000 €
Redacción del libro del proyecto	20 €	192 horas	3.840 €
Total del conjunto de mano de obra			**27.840 €**

El coste total resulta en **veintiún mil doscientos sesenta y cuatro euros** de la mano de obra y **cuatrocientos sesenta y dos con setenta y uno** del material empleado en el proyecto.

El coste total de la realización del proyecto resulta, como la suma de ambos costes anteriores:

Coste del material del proyecto	462,71 €
Coste de la mano de obra	27.840 €
Total del coste del proyecto	**28.302,71 €**

Lo que hace un total de **veintiocho mil trescientos dos con setenta y un euros.** Ahora bien, para el presupuesto de ejecución por contrata se ha de incluir tanto el beneficio por la ejecución del proyecto como los gastos que ha supuesto.

Coste total del proyecto	28.302,71 €
Recargo del 18%	3.910,81 €
Presupuesto de ejecución por contrata	**32.213,52 €**

Para obtener el presupuesto final del proyecto, se deben añadir tanto los honorarios por la realización del proyecto, un 7% del total, como el IVA del 21%.

Presupuesto de ejecución por contrata	32.213,52 €
Honorarios realización proyecto (7%)	1.794,63 €
Presupuesto	34.008,15 €
IVA del 21%	5.760,75 €
Presupuesto FINAL del proyecto	**39.768,9 €**

El presupuesto total del proyecto resulta de **TREINTA Y NUEVE MIL SETECIENTOS SESENTA Y OCHO EUROS CON NOVENTA CÉNTIMOS.**

7. Manual de Usuario.

SDSoC es una herramienta poco empleada por lo que encontrar información sobre cómo utilizarla de manera correcta y poner en funcionamiento ciertas aplicaciones puede ser complicado. En este capítulo se explican paso a paso como se creó el proyecto y los problemas encontrados al crearlo, además de las soluciones encontradas para estos.

- **Primer paso:** Introducir la plataforma Zybo Z7-20 en la herramienta SDSoC.

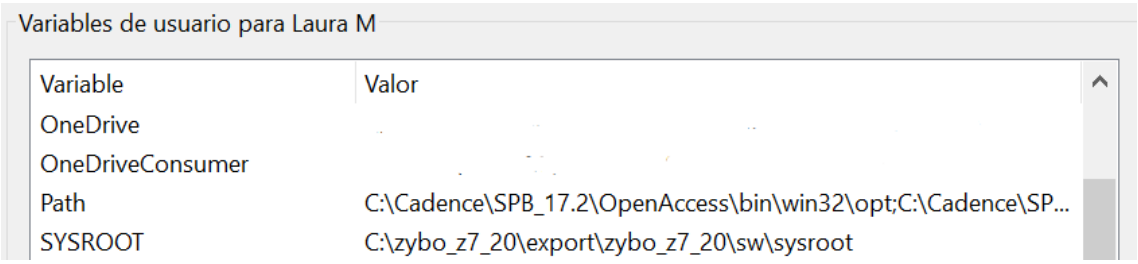
Para introducir esta plataforma se siguen los pasos del repositorio de GitHub de Digilent [18](*) donde se explica la creación de la plataforma a partir de la unión de un fichero "*.dsa"* y una imagen del sistema operativo "*.bif"* y el sistema de arranque.

En este enlace se explica detalladamente los pasos que se deben seguir, donde, como resumen, primero se empleará la herramienta Vivado para la creación de la plataforma y el fichero "*.dsa"* que contiene el hardware asociado con los "*Constraints"* requeridos para la plataforma y con los distintos elementos que van a ser usados, como la interfaz HDMI, y tras esto se emplea la herramienta PetaLinux para la creación de la imagen y el sistema de arranque. Es con PetaLinux con lo que se incluyen las librerías que la plataforma va a requerir.

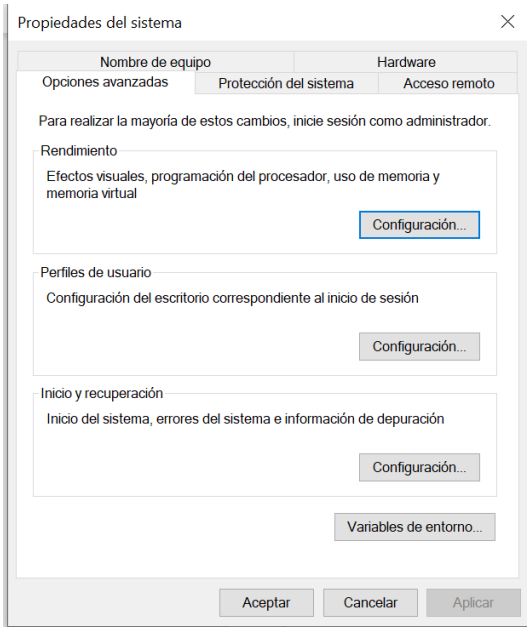
Los pasos por seguir son sencillos: Descargar el repositorio [18] de GitHub en Ubuntu 16.04 y crear la plataforma siguiendo paso a paso los pasos nombrados en el fichero "*ReadMe.txt"*. Una vez creada, se obtendrán distintas carpetas una de ellas con título "*SDSoC*". Esta carpeta se deberá comprimir y trasladar al ordenador donde se ha instalado SDx. Extraer en C:\ la plataforma Zybo de dentro de la carpeta "*SDSoC"* (puesto que los asistentes de Xilinx lo recomiendan para evitar nombres de ruta demasiado largos).

Tras esto comienza la incorporación de la plataforma en SDx:

1. Abrir "*Panel de Control*" → "*Sistema"* → "*Configuraciones Avanzadas"* → "*Variables de Entorno"* y crear una variable de entorno con nombre "*SYSROOT"* y cuya ruta apunte a la carpeta de mismo nombre que se crea dentro del SW de la plataforma. Este paso es importante para el uso de las librerías *xfOpenCV* que incorpora la plataforma creada, puesto que es una variable de entorno que va a apuntar al sistema de archivos de arranque del sistema operativo Linux dentro de la plataforma.

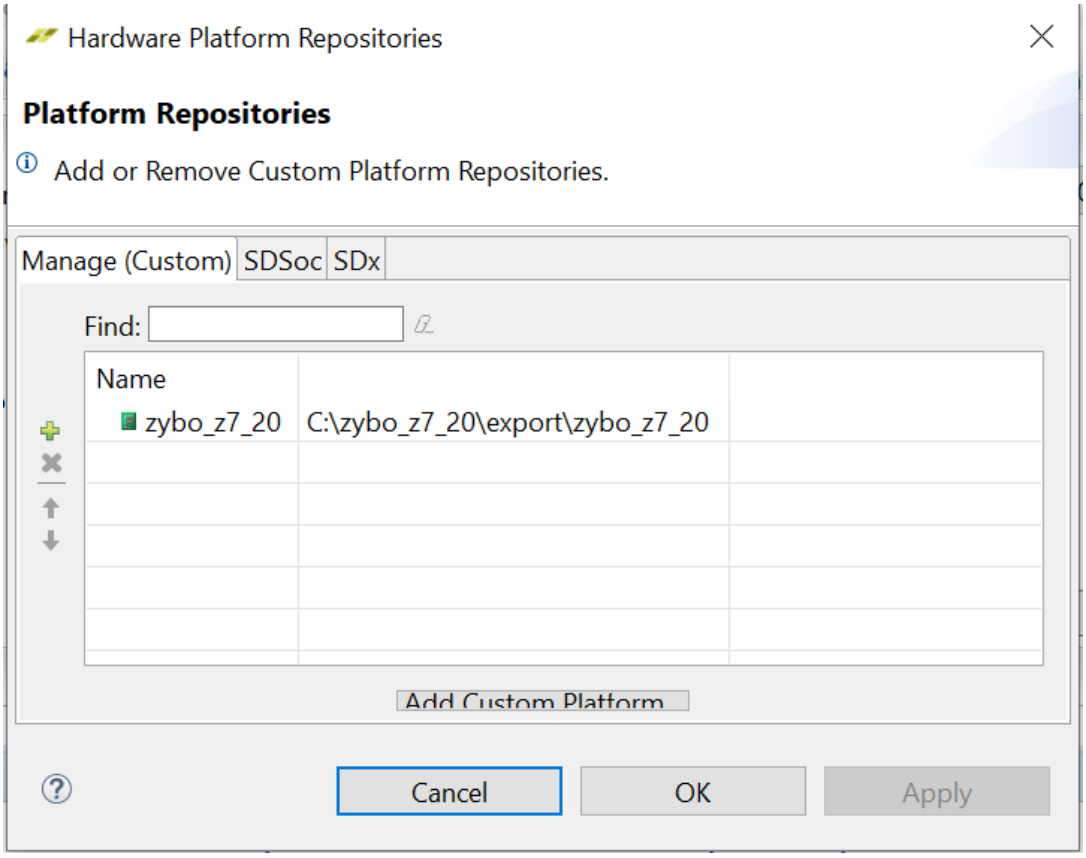


Captura 1: Variables de entorno del sistema



Captura 2: Menu de propiedades del sistema

2. Arrancar el software SDx y crear un nuevo entorno de trabajo.
3. Una vez abierto el programa, elegir la ventana "*Xilinx"* en la parte superior izquierda de la pantalla. Dentro de las opciones que se dan elegir "*Add custom platform"*
4. La siguiente pantalla aparece, debiendo introducir la ruta a la carpeta de la plataforma descomprimida que incluye las carpetas de nombre "*hw*", "*sw*", "*sd_image*" y "*samples*", que es donde se encuentra el fichero ".xpfm" que será la plataforma:



Captura 3: Ventana de adicción de plataformas personalizadas

(*) Digilent proporciona dos tipos de plataforma Zybo Z7-20 para usar con SDSoC. La creación de ambas es mediante los mismos pasos, pero la finalidad de uso de cada una difiere. Digilent proporciona una plataforma estándar de la Zybo para aplicaciones que no requieran de dispositivos externos, sin incluir librerías asociadas, y proporciona una plataforma dedicada a aplicaciones de visión, reVISION, la cual incluye las librerías de Xilinx "*xfOpenCV"* las cuales suelen dar problemas al incorporarlas en la plataforma estándar. La plataforma reVISION dispone además de los drivers para poder emplear la PCam 5C y la conexión de E/S de HDMI. Por ello es esta versión de plataforma la que se emplea en este trabajo.

- **Segundo paso:** Creación de un nuevo proyecto e introducción de las librerías de visión OpenCV.

Puesto que en este proyecto se va a trabajar con el procesado de video, se requiere la introducción de las herramientas de visión de código abierto OpenCV. Normalmente, las plataformas incluidas en SDx proporcionan estas librerías de OpenCV, pero al trabajar en este caso con una plataforma añadida, se deben proporcionar de manera "*externa"*.

El hecho de poner "*externa"* entre comillas y cursiva se deba que no se requiere descargar las librerías OpenCV y compilarlas, se puede realizar un pequeño truco para obtenerlas directamente compiladas.

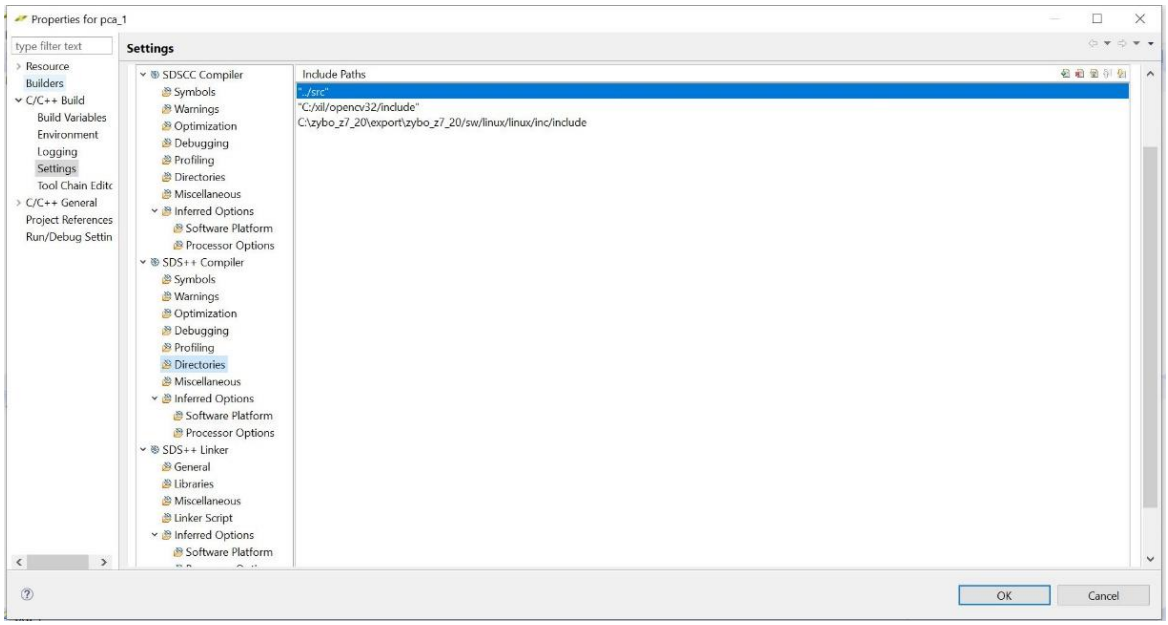
1. Introducirse en la carpeta "*SDK*" dentro del directorio raíz "*Xilinx"* creado en la instalación de las herramientas.
2. Desplazarse hasta: *C:\Xilinx\SDK\2017.4\data\embeddedsw\ThirdParty\opencv*
3. Copiar el contenido de la carpeta "*aarch32"* y pegarlo en una carpeta denominada con el nombre que se quiera (preferiblemente breve para evitar problemas).

Serán estas librerías las que se empleen, pero al no estar dentro de los directorios de Xilinx se pierde vinculación y se evitan los problemas de compilación.

Una vez realizado este paso, se siguen los siguientes para la creación del proyecto:

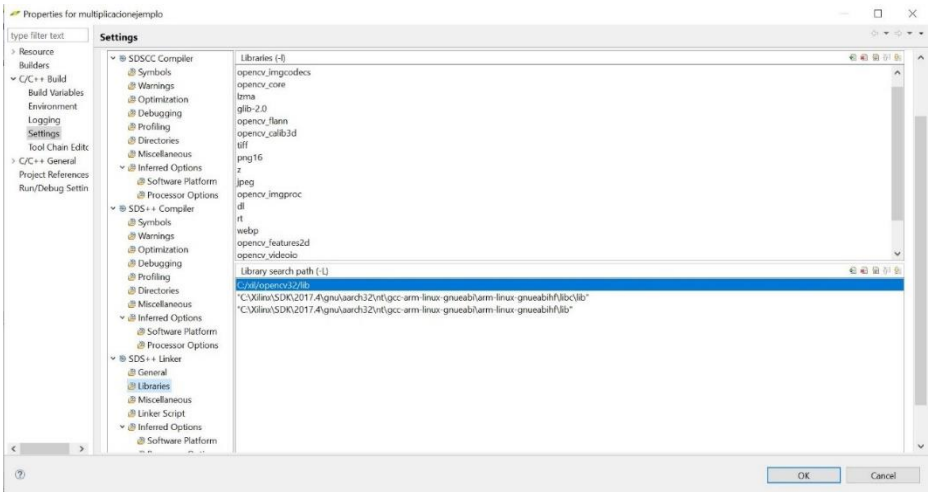
1. Se selecciona nuevo proyecto de aplicación, y tras esto, en la ventana de plataformas, se selecciona la plataforma Zybo Z7-20 introducida. En la siguiente ventana se selecciona la opción de usar un sistema operativo Linux.
2. En la siguiente ventana se muestra la opción de emplear un ejemplo de aplicación o una aplicación vacía. Para este caso, puesto que se emplean aplicaciones de visión y ciertas funciones se van a llevar a HW se emplea como base uno de los ejemplos proporcionados por la plataforma. Esta elección evita los problemas de compilación que las librerías de "*xfOpenCV"* causan. (*)
3. Eliminar los ficheros fuente por defecto e importar los ficheros fuente que contienen la aplicación a implementar, el algoritmo PCA en esta caso particular. Para introducirlos se realiza *"click"* derecho sobre la carpeta "*src"* y se selecciona importar archivos. Se navega hasta la carpeta que los contiene y se seleccionan tanto los ficheros "*.cpp"* como los "*.h"*. Además, en una carpeta que comparta directorio con la carpeta "*src"* de nombre "*Data"* se debe introducir el video que se va a usar. (**)
4. Se deben añadir las librerías que se requieren tanto como enlazar el compilador por lo que se debe hacer "*click"* derecho sobre la carpeta del proyecto y elegir la carpeta "*C/C++ build"*.
5. Dentro de esta se irá a la carpeta *"SDS++ Compiler → Directories"*, donde se deberá observar como la carpeta que contiene las librerías *xfOpenCV* viene indicada por defecto.

En esta ventana se deberá incluir la carpeta que contiene las librerías OpenCV que se ha creado anteriormente:



Captura 4- Directorios del compilador

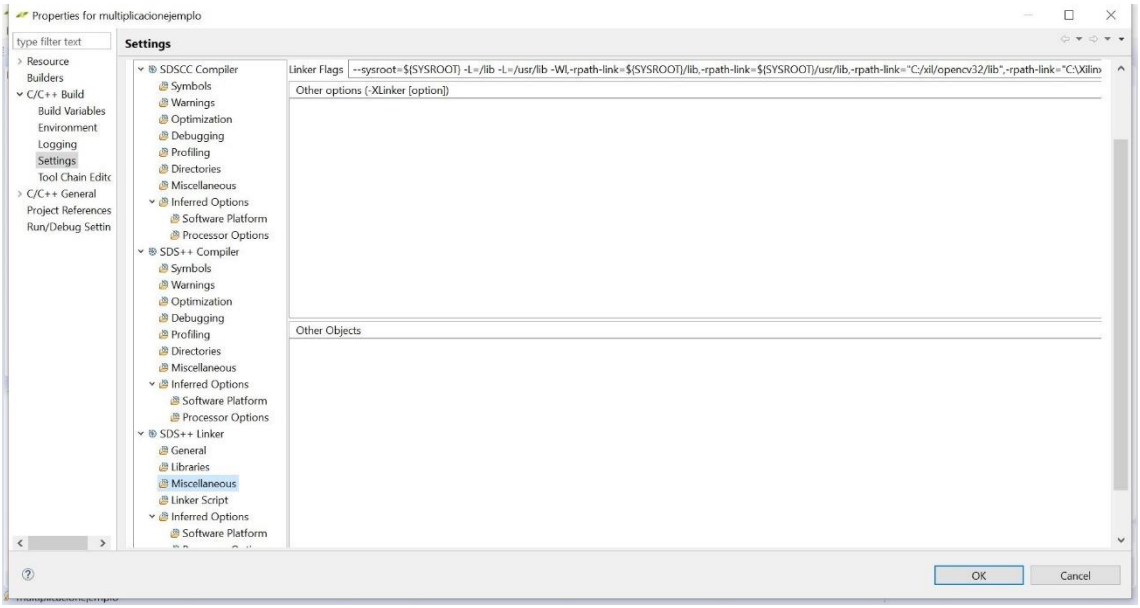
6. Tras esto se debe acceder a la carpeta *"SDS++ Linker → Libraries"*, e incluir las siguientes librerías y las rutas donde encontrarlas, además de la ruta del compilador:
    - opencv_core, opencv_imgproc, opencv_imgcodecs
    - opencv_features2d, opencv_calib3d, opencv_flann
    - lzma
    - tiff
    - png16
    - z
    - jpeg
    - dl
    - rt
    - webp
    - opencv_videoio
    - glib-2.0



Captura 5: Ventana de librerías del enlazador

7. Finalmente se debe acceder a la ventana "*SDS++ Linker → Micellaneous"* e introducir en los *"Linker Flags"* lo siguiente:

--sysroot=${SYSROOT} -L=/lib -L=/usr/lib -Wl,-rpath-link=${SYSROOT}/lib,-rpath-link=${SYSROOT}/usr/lib,-rpath-link="C:/xil/opencv32/lib",-rpath-link="C:\Xilinx\SDK\2017.4\gnu\aarch32\nt\gcc-arm-linux-gnueabi\arm-linux-gnueabihf\libc\lib".-rpath-link="C:\Xilinx\SDK\2017.4\gnu\aarch32\nt\gcc-arm-linux-gnueabi\arm-linux-gnueabihf\lib" -sdcard ../data



Captura 6: Ventana de "Miscellaneous" del enlazador

Donde se observan los directorios raíz de la plataforma, los directorios de compilación y los de la librería de visión junto con la dirección donde se van a almacenar los videos dentro de la tarjeta SD.

(*) Inicialmente se empleó el ejemplo que introduce la plataforma de "*Bilateral Filter"* dentro de la carpeta "*LiveIO"* de los ejemplos de *xfOpenCV*. Esta permitió usar las librerías de visión sin problema pero presentó problemas al usar la función "*Toggle HW/SW"* no permitiendo enviar funciones para que se ejecutasen en HW. Por ello copiaron los ajustes de compilación, las librerías y los ficheros fuente al ejemplo base de "*Matrix Multiplication HW"* el cual previamente se había comprobado que permitía mandar la función de multiplicación para que fuese ejecutada por HW. Tras introducir los mismos ajustes de compilación de las capturas anteriores, y las librerías asociadas, incorporando las librerías "*libs"* del ejemplo (estas no se deben borrar puesto que se hace uso de ellas al contener las librerías "*sds_malloc*"), la ejecución no dio ningún problema.

(**) Para la exportación del proyecto a tiempo real con la cámara, el proyecto que se usa de base es "*Filter2DPcam"* el cuál no requiere de almacenamiento en SD de las imágenes procesadas ni la lectura de video, por lo que esa información se omite.

8. Una vez compilado el programa, se habrá creado, tanto para el modo "*Debug*" como para el modo "*Release"*, una carpeta denominada "*sd_card"* la cual se deberá copiar y pegar dentro de la microSD. (***)
9. Una vez copiados los archivos, extraer la microSD e introducirla en la Zybo Z7-20, seleccionar el arranque por SD con el *jumper* y abrir el terminal de SDx para dirigir mediante la UART el SS.OO de Linux.

10. Encender la placa, y seleccionar en el terminal el "*/COMx"* asociado con las propiedades de comunicación por defecto.
11. Al estar usando una plataforma personalizada, una vez arranque el sistema operativo de la placa, se deberá montar la carpeta que contiene el ejecutable de la aplicación. Para ello se deberán ejecutar los siguientes comandos:
    - mount /dev/mmcblk0p1 /mnt/
    - cd /mnt
    - ./<project name>.elf

    Y cuando se quiera dejar de usar, para asegurar que el sistema de archivos se desmonta sin ningún problema, se debe ejecutar:
    - Poweroff

(***) Como se ha nombrado antes, la carpeta que contiene el ".xpfm" de la plataforma incluye para este caso una carpeta llamada "sd_image". Dentro de esta se encuentra un ".zip" que contiene la imagen del sistema de archivos del sistema operativo Linux creado para la plataforma. Por ello es muy importante realizar un particionado de manera que una parte contenga el sistema de archivos y la otra tanto la imagen del sistema como el fichero de arranque y el ejecutable de la aplicación (es decir, el contenido de la carpeta *sd_card* que crea SDx).
Para realizar este particionado, si se es usuario Windows, se debe descomprimir el contenido de la carpeta "sd_image" de la plataforma, y tras ello, se recomienda descargar la aplicación "Win32diskimager". Una vez descargada, ejecutarla y seleccionar el dispositivo sobre el cuál se quiere introducir la imagen del sistema de archivos.

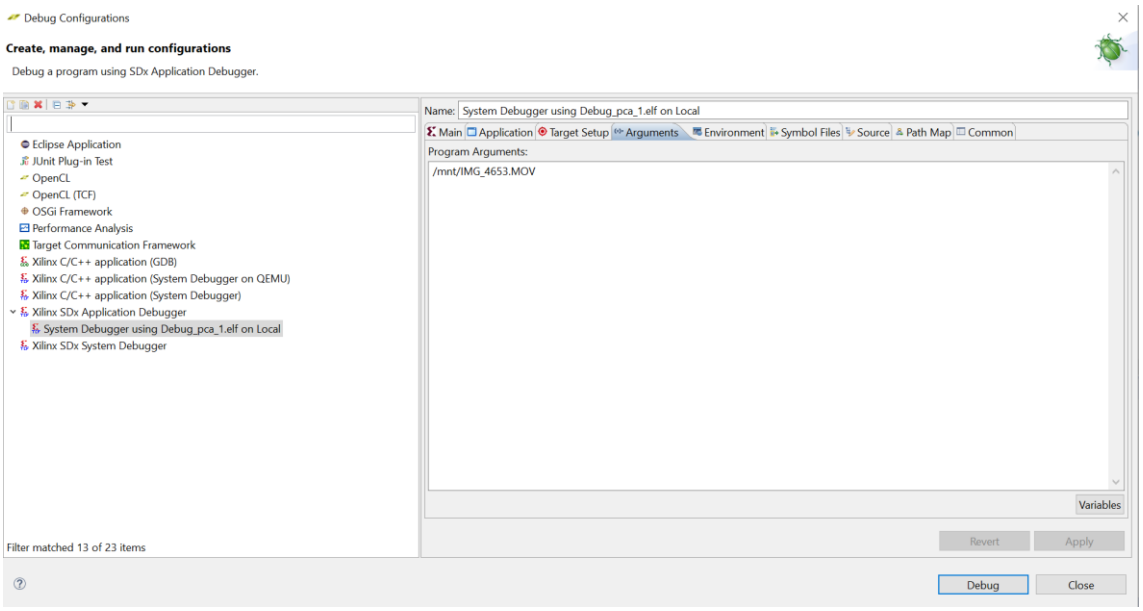
Esta aplicación realizará el particionado de manera que el sistema de archivos pertenecerá a la parte del particionado de formato "ext4" no visible en Windows y el resto, de tamaño 500MB, tendrá formato FAT32. Será en esta segunda parte donde se introducirán los ficheros obtenidos con SDx.

Nota: Si además se desea emplear la opción de depuración en hardware, se requiere el uso del TCF Agent de Linux. Este requiere la conexión mediante cable Ethernet y los siguientes pasos para una correcta detección de la IP:

- *Desde la configuración de redes de Windows, asignar una IP estática al "Ethernet" asociado a la placa (P.e: 192.168.1.114)*
- *Una vez arrancado el SS.OO de la plataforma ejecutar el siguiente comando que asigna una IP conocida a la placa:*
    - *Ifconfig eth0 xxx.xxx.x.xxx (para el ejemplo anterior: 192.168.1.115)*
- *Cliquear sobre la opción "TCF Agent" e introducir la IP asignada a la placa y aceptar.*
- *Ya se ha configurado el "TCF Agent" y se puede iniciar la depuración.*

En el caso de usar el TCF Agent para depurar una aplicación que requiere la lectura de imagen o video, se recomienda modificar un poco el código de manera que este se pase como argumento en el terminal (P.e ./<nombre_ejecutable>.elf IMGxxxx.jpg).

En este caso si se realiza la depuración en el terminal se introducirá por la consola de comandos el nombre del ejecutable a correr y el argumento que se le pasa, pero en el caso de depuración con el TCF Agent se deberá indicar en los parámetros de la depuración. Además, puesto que lo que estamos depurando se está ejecutando en la placa, se debe tener en cuenta que el archivo de lectura está dentro de la carpeta montada en el sistema operativo que corre en la placa, de manera que el argumento debe indicar su localización en esta de la siguiente manera:



Captura 7: Ajuste de los argumentos de la depuración

A la hora de poner en marcha la creación del proyecto y hacerlo funcionar se encontraron ciertos problemas. Estos y la solución asociada se enumeran a continuación:

8. Si se va a usar la aplicación para tratar video, en este caso fue un video copiado en memoria externa, y se desea usar las funciones definidas en la librería "opencv/videoio.hpp" se debe añadir a las librerías de la ventana "*Libraries*" del enlazador la librería "opencv_videoio", puesto que esta no viene por defecto.
9. Al emplear el "*TCF Agent"* ciertas direcciones virtuales daban problemas y causaban una violación de segmento. Mediante la depuración paso a paso, al visualizarlas en la memoria, se observó que estas aparecían escritas en color rojo. Se descubrió que se debía a que no se almacenaban en memoria de manera correcta, por lo que se usó de asignación de memoria dinámica al crear esas variables, empleando "*sds_alloc"*.

- **Tercer paso:** Puesta en marcha del módulo PCam5.

Para la puesta en marcha de este módulo se creó un nuevo proyecto siguiendo los pasos anteriores eligiendo en la ventana de aplicación el ejemplo "*Filter2DPcam"*. En este ejemplo se aplica un filtro de convolución simple. Requiere la conexión de un cable HDMI al puerto de Tx de la Zybo y el módulo PCam5C conectado al puerto MIPI-CSI2.

Una vez creado el proyecto, sin necesidad de modificar nada, se realiza la depuración. Puesto que ciertas funciones se destinan a la ejecución en HW, el tiempo de compilación es elevado. A finalizar esta, se copia la carpeta creada "*sd_card"* dentro de la microSD y una vez copiados esta se inserta en la plataforma.

Se inicia la Zybo y se establece la comunicación usando el terminal de SDx. Una vez iniciado el sistema operativo, tras montar el sistema de archivos y acceder a la carpeta "*mnt"*, se deberá ejecutar uno de los siguientes archivos para configurar las dimensiones de captura y reproducción de video:

- *"./config_pcam_1080p.sh"*
- *"./config_pcam_720p.sh"*

- *"./config_pcam_vga.sh"*

Una vez ejecutado este archivo, si no se han obtenido errores, se ejecutará el "*.elf"* asociado al proyecto. Si se han conectado la cámara y el HDMI correctamente, y se ha quitado el protector de la cámara, se deberá observar por el HDMI el video a tiempo real.

Puede ser que inicialmente la pantalla se vea en negro y esto se debe a que esta demo realiza un filtrado de dos maneras: en hardware con funciones *xfOpenC* y en software con funciones OpenCV. El filtrado por hardware se controla con los *switches* de la plataforma mientras que para controlar el filtrado software se debe mantener pulsado uno de los botones.

La opción por defecto para la codificación de los *switches* "0000", es decir todos abajo, es mostrar un fondo en negro, por lo que antes de pensar que no está funcionando, se recomienda tocar los interruptores para comprobar verdaderamente su funcionamiento.

Es importante hacer funcionar este ejemplo puesto que es el que se emplea para realizar la traslación de PCA a la captura de *frames* en tiempo real. A continuación se muestra los resultados obtenidos al ejecutar la aplicación "*filter2DPcam"*:

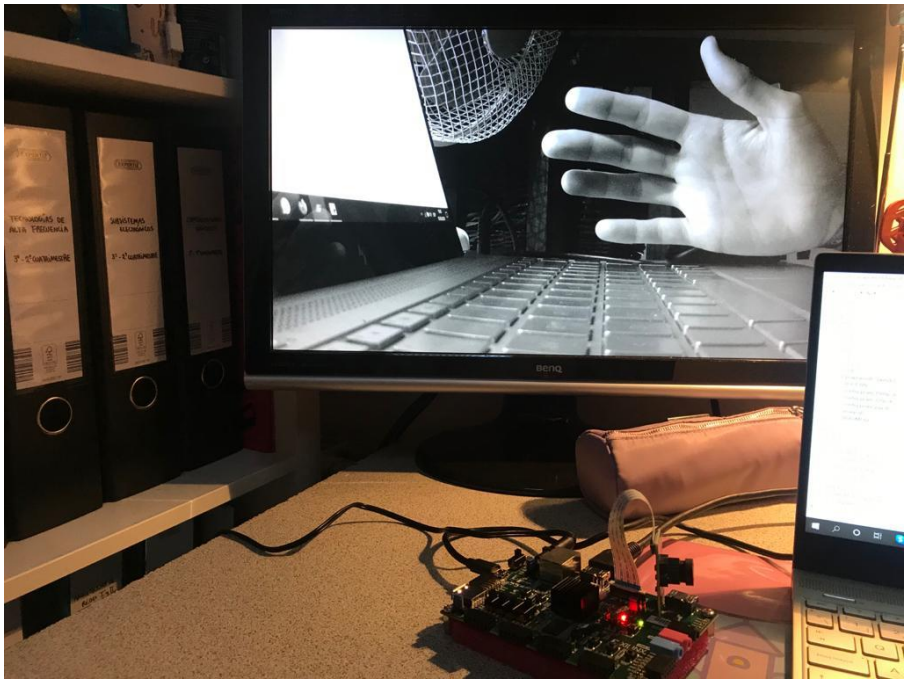


Figura 60: Demostración del funcionamiento del módulo PCam5C

8. Planos.

A lo largo de este capítulo, puesto que se ha modificado parte del código que se encuentra en el ejemplo "*Filter2D Pcam 5C"*, se van a explicar las distintas modificaciones realizadas y las distintas funciones de código en lenguaje C++ que se han implementado.

A. "online.cpp"

En este fichero se encuentra la declaración de las funciones que muestran la imagen seleccionada como fondo estático y la función que calcula el proceso online al completo.

En la primera función que muestra el fondo, sencillamente se hace una pasarela entra la imagen captada y usada para estimar el vector del fondo y el HDMI hacia la pantalla. La función lo único que hace es cambiar la imagen de entrada de escala de grises, que es el plano Y de la luminancia, a RGBA que es el formato de color de salida.

```
#include <opencv2/core.hpp>
#include <opencv2/imgproc.hpp>

#include <stdio.h>
#include "pca/funciones.h"
#include "online.h"


using namespace cv;

void mostrarfondo(uint32_t *frm_data_in, uint32_t *frm_data_out,
              int height, int width, int in_stride,int out_stride){

            cv::Mat src(height, width, CV_8UC1, frm_data_in,in_stride);
            cv::Mat dst(height, width, CV_8UC4, frm_data_out,out_stride);
          cv::cvtColor(src, dst, CV_GRAY2RGBA);

}
```

Esta función "*online"* realiza todas las llamadas al resto de funciones que realizan el proceso "*online"*. Estas llamadas vienen declaradas en "*funciones.h"*. Además se crean los datos que se van a usar a lo largo del proceso. Se observa que tanto "*input"* como "*recup_img"* se declaran como tipo de dato al que se le reserva memoria para ser almacenado de manera física usando *sds_alloc()*. Esto se realiza puesto que son las variables que se van a pasar como parámetros a las funciones que se van a ejecutar en hardware y sobre las cuales se van a aplicar los pragmas SDS. Finalmente, se debe liberar siempre ese espacio con un *sds_free()*, puesto que si esta sentencia no se pone, tras cierto tiempo de ejecución se consume la memoria y el programa se cierra al intentar acceder a una posición de memoria ilegal.

```
void online(uint32_t *frm_data_in, uint32_t *frm_data_out,
              int height, int width, int in_stride,int out_stride,float
U[totpix][tComp],short vectormedia[totpix][1]){

            short vf2[totpix][1];
            float norm_Ut[tComp][totpix], distancia;
            short *input = (short*)sds_alloc(totpix*sizeof(short));
            float omega[tComp][1];
            float *recup_img = (float*)sds_alloc(totpix*sizeof(float));
            unsigned char mapa_distancias[totpix]={0};
```

```
            cv::Mat src(height, width, CV_8UC1, frm_data_in,in_stride);
            cv::Mat dst(height, width, CV_8UC4, frm_data_out,out_stride);

          cv::Mat frame(256,256,CV_8UC1);
          cv::resize(src, frame, frame.size(), 0, 0);

          mat_to_entrada(frame,vf2);
          restar_mediaVf2(vf2,vectormedia,input);

          transpuesta_U(U,norm_Ut);

          calculo_proyeccion(norm_Ut,input,omega);
          recuperacion_img(U,omega,recup_img);

          distancia = calculo_disteuclidea(input,recup_img);
          calculo_mapa(input,recup_img,mapa_distancias); //,imgumbral);
          Mat imgbin(N,N,CV_8UC1);
          umbral_rep(imgbin,mapa_distancias);

          cv::Mat imgbinerode(N,N,CV_8UC1);
          erode(imgbin,imgbinerode,Mat());
          dilate(imgbinerode,imgbin,Mat());

        float porcentaje;
        porcentaje = hay_objeto(imgbin);

        printf( "Porcentaje de blanco del frame es %f\n ",porcentaje);

          if(porcentaje<0)
            porcentaje = 0;
          if(porcentaje >4){ //Este 0.5 es dinámico y va a depender
            printf("Existe un objeto en el frame con un porcentaje de blanco
%f\n",porcentaje);
          }else{
            printf("No existe un objeto en el frame\n ");
          }
          cv::resize(imgbin,imgbin,Size(width,height));
          cv::cvtColor(imgbin, dst, CV_GRAY2RGBA);
          sds_free(input);
          sds_free(recup_img);
}
```

B. “online.h”

En este primer archivo de cabecera se declaran las librerías de OpenCV que contienen el tipo de dato de matriz que se requieren como parámetros de entradas y la librería creada que incluye tanto las constantes como las funciones necesarias para el algoritmo PCA. Las dos funciones que se declaran en esta librería son “*mostrarfondo”* y “*online”*. A la primera se le pasa como parámetro la imagen de entrada recibida leída del buffer de memoria que la almacena, la posición del buffer de memoria para la extracción de la imagen por el HDMI, las dimensiones de las imágenes con las que se ha configurado la conexión HDMI. El quinto y el sexto parámetro representan los bits por línea de representación.

La segunda función realiza todo el proceso online del algoritmo PCA por ello se declara con parámetros de entrada la matriz de transformación U y el vector media, que se obtienen en la parte

offline, la imagen que se acaba de recibir del módulo de la cámara y los bits por línea de representación, y como parámetros un puntero a la posición del buffer de memoria de salida.

```
#ifndef _ONLINE_H_
#define _ONLINE_H_

#include "platform.h"
#include "pca/funciones.h"
#include <iostream>
#include <stdlib.h>
#include "../libs/sds_utils/sds_utils.h"

void online(uint32_t *frm_data_in, uint32_t *frm_data_out,
             int height, int width, int in_stride,int out_stride, float
U[totpix][tComp],short vectormedia[totpix][1]);
void mostrarfondo(uint32_t *frm_data_in, uint32_t *frm_data_out,
             int height, int width, int in_stride,int out_stride);


#endif
```

C. “main.cpp”

Este fichero muestra la función principal, viene por defecto en el ejemplo del filtro pero se ve modificado. Las primeras sentencias de código configuran la cámara y la captura de imágenes y se asegura de que las conexiones de los módulos son correctas, tanto la del HDMI como la del módulo PCam5c. Esta configuración se realiza con las funciones *v4l2_helper_open* y *v4l2_helper_init*. Tras esto se inicializa el buffer DMA y se asigna a dos variables el ancho y el alto de las dimensiones configuradas como entrada. Tras esto se evita que el terminal del SO de la plataforma se dibuje en el monitor y toma el control de este, configurando la resolución para que coincida con la entrada y se creen múltiples buffers para los frames. Tras esto se comienza a capturar frames, concretamente 8, y tras capturar y crear el vector fondo con estas 8 frames se detiene la captura. Se comienza el proceso offline y una vez calculada la matriz de transformación y el RMSE se crea un bucle que va a capturar las imágenes a tiempo real y las envía al proceso online. Este bucle finaliza cuando el usuario toca una tecla.

Además se aprovecha las funciones *sds_clock* que incluye el ejemplo para obtener la tasa de procesado en tiempo real. Finalmente, se libran todos los buffers y registros tanto del DRM como del V4L2.

```
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include "../libs/sds_utils/sds_utils.h"
#include <sys/ioctl.h>
#include <linux/fb.h>
#include <linux/kd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <ncurses.h>
#include "online.h"
#include "drm_helper/drm_helper.h"
#include "platform.h"
#include "uio/uio_ll.h"
#include "rgbpxl.h"
```

```
#include "v4l2_helper/v4l2_helper.h"
#include "opencv2/imgcodecs.hpp"
#include "opencv2/core.hpp"
#include "opencv2/videoio.hpp"
#include "opencv2/opencv.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "pca/funciones.h"
#include "pca/defs_and_types.h"

int main(int argc, char **argv){
    uint32_t hActiveIn, vActiveIn;
    int sw_fd, tty_fd, mem_fd;;
    uint8_t *swMem;
    struct drm_cntrl drm = {0};
    struct v4l2_helper v4l2_help = {0};


    int k,i,j;
    short vectorfondo[totpix][Mfondo],vectormedia[totpix][1];
    short A[totpix][Mfondo], Atrans[Mfondo][totpix];
    float covar[Mfondo][Mfondo], RMSE;
    float U[totpix][tComp], Uprev[totpix][Mfondo];
    float auto_val[Mfondo], auto_vect[Mfondo][Mfondo], autoval_tot=0,
autoval_max=0;


    /*Initialize V4L2 input and allocate buffers. Note that the resolution is
controlled by calling media-ctl prior to launching this program. Also, v4l2-
ctl must set the format to NM16 and yavta must be run at least for a single
frame with the equivalent format specifier (NV16M). */

    v4l2_helper_open(&v4l2_help, V4L2_VIDEO_PATH);
    v4l2_helper_init(&v4l2_help, NUM_V4L2_BUF);

    for (int i = 0; i < v4l2_help.num_buf; i++)
      for (int j = 0; j < v4l2_help.fmt.fmt.pix_mp.num_planes; j++)

      sds_register_dmabuf(v4l2_help.buffers[i].start[j],v4l2_help.buffers[i]
.dmabuf_fd[j]);

    hActiveIn = v4l2_help.fmt.fmt.pix_mp.width;
    vActiveIn = v4l2_help.fmt.fmt.pix_mp.height;
/* Stop the terminal from being drawn on the monitor */
      tty_fd = open("/dev/tty1", O_RDWR);
      ioctl(tty_fd,KDSETMODE,KD_GRAPHICS);
      close(tty_fd);

/* Take control of the display device, set the resolution to match the input,
and create multiple framebuffers. */
if (drmControlInit("/dev/dri/card0",hActiveIn,vActiveIn,&drm) != SUCCESS){
            printf("drmControlInit Failed");
            return -1;
      }

/* Initialize ncurses for framerate display and non-blocking user input */
      WINDOW *mywin = initscr();
      cbreak();
      nodelay(mywin, TRUE);
```

```
	wmove(mywin, 0, 0);
	printw("Filter2d is now running on the attached display\n");
	printw("Use the 4 onboard switches to control the filter\n");
	printw("Input Resolution: %dx%d\n",hActiveIn,vActiveIn);
	printw("Output Resolution:
%dx%d\n",drm.current.hdisplay,drm.current.vdisplay);
	printw("DRM FB Stride (bytes):
%d\n",drm.create_dumb[drm.current_fb].pitch);
	printw("Input Plane 0 Stride (bytes):
%d\n",v4l2_help.fmt.fmt.pix_mp.plane_fmt[0].bytesperline);
	printw("Input Plane 1 Stride (bytes):
%d\n",v4l2_help.fmt.fmt.pix_mp.plane_fmt[1].bytesperline);
	printw("\n");
	printw("Press any key to exit...");
	wrefresh(mywin);

/*Start capturing frames from V4L2 device */
	v4l2_helper_start_cap(&v4l2_help);

	unsigned int v4l2_buf_index;
	/* PARTE OFFLINE DEL ALGORITMO: Se realiza la lectura de un frame y
este se almacena dentro de una cv::MAT, que a su vez se procesa cambiando su
tamaño a 256x256 y se almacena en el vector fondo */

	for (k=0;k<Mfondo;k++){
		v4l2_buf_index = v4l2_helper_readframe(&v4l2_help);
		cv::Mat frameprev(vActiveIn, hActiveIn,CV_8UC1,(uint32_t *)
v4l2_help.buffers[v4l2_buf_index].start[V4L2_FORMAT_Y_PLANE],v4l2_help.fmt.fm
t.pix_mp.plane_fmt[V4L2_FORMAT_Y_PLANE].bytesperline);
		cv::Mat frame(256,256,CV_8UC1);
		cv::resize(frameprev,frame,frame.size(),0,0);
		mat_to_vector(frame,vectorfondo,k);
		v4l2_helper_queue(&v4l2_help, v4l2_buf_index);
	}
	mostrarfondo((uint32_t *)
v4l2_help.buffers[v4l2_buf_index].start[V4L2_FORMAT_Y_PLANE],(uint32_t *)
drm.fbMem[drm.current_fb],vActiveIn, hActiveIn,
v4l2_help.fmt.fmt.pix_mp.plane_fmt[V4L2_FORMAT_Y_PLANE].bytesperline,
drm.create_dumb[drm.current_fb].pitch);
    v4l2_helper_stop_cap(&v4l2_help);
	printf("Calculo de la media\n\n");
	calcular_media(vectorfondo,vectormedia);
	printf("Restar la media\n\n");
	restar_mediaA(vectorfondo,vectormedia,A);
	printf("Calculo de la transpuesta\n\n");
	calculo_transpuesta(A,Atrans);
	printf("Calculo de la covarianza\n\n");
	calculo_covarianza(Atrans,A,covar);
	printf("Calculo de los autovectores\n\n");
	dsvd(covar,Mfondo,Mfondo,auto_val,auto_vect);
	printf("Ordenar los autovectores\n\n");
	ordenar_auto(auto_val,auto_vect);
	printf("Calculo de la matriz de transformacion\n\n");
	calculo_mat_transformacion(A,auto_vect,Uprev);
	printf("Seleccion de las %d componentes principales\n\n", tComp);
	for(i=0;i<tComp;i++){
		for(j=0;j<totpix;j++){
			U[j][i] = Uprev[j][i];
		}
```

```
      }
      norm_U(U);
      printf("Calculo del RMSE\n\n");
    for(i=0;i<Mfondo;i++){
      autoval_tot =autoval_tot + auto_val[i];
    }
    for(i=0;i<tComp;i++){
      autoval_max =autoval_max + auto_val[i];
    }
    RMSE = autoval_max/autoval_tot*100;
    printf("El RMSE obtenido para %d componentes principales es: %f\n\n",
tComp, RMSE);
      printf("Comienza ahora la deteccion de objetos (Parte Online del
algoritmo PCA)\n\n");

      unsigned long long ticks = sds_clock_counter();
      unsigned long long timeElapsed;
      double frameRate = 0;
      int userInput = ERR;

      i=0;
      v4l2_helper_start_cap(&v4l2_help);
      while (userInput == ERR)
      {

            v4l2_buf_index = v4l2_helper_readframe(&v4l2_help);

            online((uint32_t *)
v4l2_help.buffers[v4l2_buf_index].start[V4L2_FORMAT_Y_PLANE],(uint32_t *)
drm.fbMem[drm.current_fb],vActiveIn, hActiveIn,
v4l2_help.fmt.fmt.pix_mp.plane_fmt[V4L2_FORMAT_Y_PLANE].bytesperline,
drm.create_dumb[drm.current_fb].pitch, U,vectormedia);

            v4l2_helper_queue(&v4l2_help, v4l2_buf_index);

            i++;
            if (i == 30)
            {
                  timeElapsed = (sds_clock_counter()- ticks);
                  frameRate = ((double) sds_clock_frequency())/((double)
timeElapsed) * 30.0;
                  printf("Framerate achieved = %3.3f", frameRate);
//                mvprintw(7,0,"Framerate = %3.3f     ", frameRate);
                  wrefresh(mywin);
                  ticks = sds_clock_counter();
                  i = 0;
            }
            userInput = getch();
      }


      endwin();

      /* Unregister v4l2 buffers with SDSoC driver */
    for (int i = 0; i < v4l2_help.num_buf; i++)
            for (int j = 0; j < v4l2_help.fmt.fmt.pix_mp.num_planes; j++)

      sds_unregister_dmabuf(v4l2_help.buffers[i].start[j],v4l2_help.buffers[
i].dmabuf_fd[j]);
```

```
    /* Free all v4l2 buffers and close all files*/
    v4l2_helper_stop_cap(&v4l2_help);
    v4l2_helper_uninit(&v4l2_help);
    v4l2_helper_close(&v4l2_help);

    /* Unregister and free DRM buffers and close DRM files */
    drmControlClose(&drm);

    /*Close /dev/mem and gpio switch files*/
      close(mem_fd);
      close(sw_fd);

      /*Reactivate terminal*/
      tty_fd = open("/dev/tty1", O_RDWR);
      ioctl(tty_fd,KDSETMODE,KD_TEXT);
      close(tty_fd);

    return 0;
}
```

D. "declaracionfunciones.cpp"

En este fichero se declaran las funciones que forman parte tanto del proceso online como del offline de PCA. Se encuentran las funciones que ordenan los autovalores de mayor a menor a la par que cambian de posición su autovector asociado, la función que convierte un tipo de dato matriz de OpenCV a un vector en C, tanto para la parte offline *mat_to_vector* como para la online *mat_to_entrada*.

Se encuentra también la función que calcula la media de cada una de las filas del vector fondo con intención de crear con ella la matriz A, la función que resta esta media, las que calculan la transpuesta y la que detecta si existe o no objeto en la imagen.

```
#include "funciones.h"
#include "opencv2/imgcodecs.hpp"
#include "opencv2/core.hpp"
#include "opencv2/videoio.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"

void ordenar_auto(float eig_values[Mfondo], float eig_vect[Mfondo][Mfondo]) {
	int i, j;
	float temp;
	for (i = 0; i < Mfondo; i++) {
		if (eig_values[i] < eig_values[i + 1]) {
			temp = eig_values[i];
			eig_values[i] = eig_values[i + 1];
			eig_values[i + 1] = temp;
			for (j = 0; j < Mfondo; j++) {
				temp = eig_vect[j][i];
				eig_vect[j][i] = eig_vect[j][i + 1];
				eig_vect[j][i + 1] = temp;
			}
		}
	}
}
void mat_to_vector(cv::Mat frame, short vectorfondo[totpix][Mfondo], int
posicioncolumna) {
```

```
	int i, j;
	for (i = 0; i < N; i++) {
		for (j = 0; j < N; j++) {
			vectorfondo[i * N + j][posicioncolumna] =
frame.at<uchar>(j, i);
		}
	}
}

void  mat_to_entrada(cv::Mat frame, short vf2[totpix][1]){
	int i, j;
	for (i = 0; i < N; i++) {
		for (j = 0; j < N; j++) {
			vf2[i * N + j][0] = frame.at<uchar>(j, i);
		}
	}
}

void calcular_media(short vectorfondo[totpix][Mfondo], short
vectormedia[totpix][1]){
	int i,j;
	for(i=0;i<totpix;i++){
		vectormedia[i][0] = 0;
		for(j=0;j<Mfondo;j++){
			vectormedia[i][0] = vectormedia[i][0] + vectorfondo[i][j];
		}
		vectormedia[i][0] = vectormedia[i][0]/Mfondo;
	}
}
void restar_mediaA(short vectorfondo[totpix][Mfondo], short
vectormedia[totpix][1],short A[totpix][Mfondo]){
	int i,j;
    for (i=0;i<totpix;i++){
		for(j=0;j<Mfondo;j++){
			A[i][j] = vectorfondo[i][j] - vectormedia[i][0];
		}
    }
}

void calculo_transpuesta(short A[totpix][Mfondo], short
Atrans[Mfondo][totpix]){
	int i,j;
	for (i=0;i<totpix;i++){
		for(j=0;j<Mfondo;j++){
			Atrans[j][i] = A[i][j];
		}
	}
}

void restar_mediaVf2(short vf2[totpix][1], short vectormedia[totpix][1],short
input[totpix]){
	int i;
    for (i=0;i<totpix;i++){
      input[i] = vf2[i][0] - vectormedia[i][0];
    }
}

void transpuesta_U(float U[totpix][tComp], float Utrans[tComp][totpix]){
	int i,j;
```

```
	for(i=0;i<totpix;i++){
		for(j=0;j<tComp;j++){
			Utrans[j][i] = U[i][j];
		}
	}
}
float hay_objeto(cv::Mat frame){
	int i, j;
	unsigned int blanco = 0;
	float porcentaje;
	for (i = 0; i < N; i++) {
		for (j = 0; j < N; j++) {
			if (frame.at<uchar>(j, i) ==255){
				blanco++;
			}
		}
	}
	porcentaje = ((float)blanco/(N*N))*100;
	return porcentaje;
}
```

E. “funciones.h”

En este archivo de cabecera se encuentra la definición de cada una de las funciones que pertenecen al algoritmo PCA.

```
#ifndef FUNCIONES_H
#define	FUNCIONES_H

#include <stdio.h>
#include <math.h>
#include "opencv2/imgcodecs.hpp"
#include "opencv2/core.hpp"
#include "opencv2/videoio.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdlib.h>
#include "../libs/sds_utils/sds_utils.h"

#define N 256
#define Mfondo 8
#define totpix N*N
#define tComp 4
#define umbral 45

int dsvd(float a[][Mfondo], int m, int n, float w[], float v[][Mfondo]);
void ordenar_auto(float eig_values[Mfondo], float eig_vect[Mfondo][Mfondo]);
void mat_to_vector(cv::Mat frame, short vectorfondo[totpix][Mfondo],int
posicioncolumna);
void calcular_media(short vectorfondo[totpix][Mfondo], short
vectormedia[totpix][1]);
void restar_mediaA(short vectorfondo[totpix][Mfondo], short
vectormedia[totpix][1],short A[totpix][Mfondo]);
void calculo_transpuesta(short A[totpix][Mfondo], short
Atrans[Mfondo][totpix]);
void calculo_covarianza(short Atrans[Mfondo][totpix],short
A[totpix][Mfondo],float C[Mfondo][Mfondo]);
void calculo_mat_transformacion(short A[totpix][Mfondo], float
auto_vect[Mfondo][Mfondo], float U[totpix][Mfondo]);
```

```
void norm_U(float U[totpix][tComp]);
void  mat_to_entrada(cv::Mat frame, short vf2[totpix][1]);
void restar_mediaVf2(short vf2[totpix][1], short vectormedia[totpix][1],short
input[totpix]);
void calculo_proyeccion(float Utrans[tComp][totpix],short input[totpix],float
omega[tComp][1]);
void transpuesta_U(float U[totpix][tComp], float Utrans[tComp][totpix]);
void recuperacion_img(float U[totpix][tComp],float omega[tComp][1],float
recup_img[totpix]);
float calculo_disteuclidea(short input[totpix],float recup_img[totpix]);
void calculo_mapa(short input[totpix],float recup_img[totpix],unsigned char
mapa_distancias[totpix]);
void umbral_rep(cv::Mat frame, unsigned char mapa_distancias[totpix]);
float hay_objeto(cv::Mat frame);

#endif // !FUNCIONES_H
```

F. "map_error.cpp"

En este fichero se encuentran las funciones:

- Normalización de la matriz de transformación: *norm_U*
- Calculan la distancia euclídea entre la imagen de entrada y la recibida del espacio proyectado: *calculo_disteuclidea.*
- La que crea la imagen binarizada en función de un umbral y muestra en blanco el objeto detectado, esto es cuando la distancia entre dos puntos del mapa de distancias supera cierto umbral: *umbral_rep*
- La función que crea el mapa de distancias gracias a la distancia euclídea: *calculo_mapa*. Esta función tiene, previa a su declaración, las sentencias pragmas SDS para el acceso a datos cuando se envía a ejecutarse en hardware. Además, dentro de la propia función, se encuentran la sentencias pragmas HLS para optimizar su ejecución.

```
#include "funciones.h"

void norm_U(float U[totpix][tComp]){ //, float Unorm[totpix][tComp]){
	int i, t;
	float suma[tComp];
	for (t=0;t<tComp;t++){
		suma[t] = 0;
		for(i=0; i<totpix;i++){
			suma[t] = suma[t] + U[i][t]*U[i][t];
		}
	}
	for (t=0;t<tComp;t++){
		for(i=0; i<totpix;i++){
			U[i][t] = U[i][t]/sqrt(suma[t]);
		}
	}
}

float calculo_disteuclidea(short input[totpix],float recup_img[totpix]){
	int i;
	float temp[totpix] = {0.0};
	float sum=0;
	for(i=0;i<totpix;i++){
		temp[i] = (input[i]-recup_img[i])*(input[i]-recup_img[i]);
	}
	for(i=0;i<totpix;i++){
```

```
			sum = sum + temp[i];
//			mapa_distancias[i] = sqrt(temp[i][0]);
		}
		return sqrt(sum);
}
void umbral_rep(cv::Mat frame, unsigned char mapa_distancias[totpix]) {
		int i, j;
		for (i = 0; i < N; i++) {
			for (j = 0; j < N; j++) {
				if (mapa_distancias[i*N+j]<umbral){
					frame.at<uchar>(j, i) = 0;
				}else{
					frame.at<uchar>(j, i) = 255;
				}
			}
		}
}

#pragma SDS data zero_copy(input[0:totpix])
#pragma SDS data access_pattern(recup_img:SEQUENTIAL)
#pragma SDS data access_pattern(mapa_distancias:SEQUENTIAL)
void calculo_mapa(short input[totpix],float recup_img[totpix],unsigned char
mapa_distancias[totpix]){ //, unsigned char imgumbral[N][N]){
		int i;
		float temp[totpix] = {0.0};
		for(i=0;i<totpix;i++){
			#pragma HLS PIPELINE II=1 rewind
			temp[i] = (input[i]-recup_img[i])*(input[i]-recup_img[i]);
		}
		for(i=0;i<totpix;i++){
			#pragma HLS PIPELINE II=1 rewind
			mapa_distancias[i] = sqrt(temp[i]);
		}
}
```

G. "mult_mat.cpp"

En este fichero "*mult_map.cpp*" se encuentra la definición de todas las funciones que realizan una multiplicación matricial en su interior. El *calculo_covarianza* y *calculo_mat_transformación* son funciones del proceso offline y no constan de ningún pragma porque no son funciones que se ejecuten en ningún momento en hardware, sin embargo *calculo_proyeccion* y *recuperación_img*, pertenecen al proceso online y ambas se prueban en hardware y por ello se indica previo a su declaración las pragmas que van a indicar el acceso de datos de SDx y los pragmas HLS para optimizar su ejecución, todos descritos en el capítulo cuarto de la memoria.

```
#include "funciones.h"
#include <iostream>
#include <stdlib.h>
#include "../libs/sds_utils/sds_utils.h"

void calculo_covarianza(short Atrans[Mfondo][totpix],short
A[totpix][Mfondo],float C[Mfondo][Mfondo]){
		int i,j,k;
		for(i = 0;i<Mfondo;i++){
			for (j=0;j<Mfondo;j++){
				C[i][j] = 0;
				for (k=0;k<totpix;k++){
					C[i][j] = C[i][j] + Atrans[i][k]*A[k][j];
```

```
                }
            }
        }
}
void calculo_mat_transformacion(short A[totpix][Mfondo], float
auto_vect[Mfondo][Mfondo], float U[totpix][Mfondo]){
        int i,j,k;
        for(i = 0;i<totpix;i++){
            for (j=0;j<Mfondo;j++){
                U[i][j] = 0;
                for (k=0;k<Mfondo;k++){
                    U[i][j] = U[i][j] + A[i][k]*auto_vect[k][j];
                }
            }
        }
}

#pragma SDS data zero_copy(input[0:totpix])
#pragma SDS data access_pattern(Utrans:SEQUENTIAL)
void calculo_proyeccion(float Utrans[tComp][totpix],short input[totpix],float
omega[tComp][1]){
        int i,k,l; //,j
        short in[totpix][1];
        for(l=0;l<totpix;l++){
                in[l][0] = input[l];
        }
        #pragma HLS ARRAY_PARTITION variable=in block factor=16 dim=1
        #pragma HLS INLINE
        loop1: for(i=0;i<tComp;i++){
                    omega[i][0] = 0;
                    loop3: for (k=0;k<totpix;k++){
                        #pragma HLS PIPELINE II=1 rewind
                        omega[i][0] = omega[i][0] +
Utrans[i][k]*in[k][0];
                    }
        }
}

#pragma SDS data data_mover(recup_img:AXIDMA_SIMPLE)
#pragma SDS data access_pattern(U:SEQUENTIAL)
#pragma SDS data access_pattern(recup_img:SEQUENTIAL)
void recuperacion_img(float U[totpix][tComp],float omega[tComp][1],float
recup_img[totpix]){
        int i,k;
        #pragma HLS INLINE
        for(i=0;i<totpix;i++){
                    recup_img[i] = 0;
                    for (k=0;k<tComp;k++){
                        #pragma HLS PIPELINE II=1 rewind
                        recup_img[i] = recup_img[i] +
U[i][k]*omega[k][0];
                    }
        }
}
```

9. Conclusiones y trabajos futuros.

En este proyecto se han comparado distintos métodos y directivas para optimizar el algoritmo de procesado de imágenes PCA. Este algoritmo se emplea para detectar nuevos elementos dentro de una escena estática donde el fondo no varía. La finalidad de comparación es obtener una tasa superior a la de un escenario inicial sin ninguna optimización. Se examinan distintos métodos de multiplicación matricial y se ponen en prueba distintos pragmas que el software SDx de Xilinx pone a disposición.

Se parte de un escenario inicial en el que se obtiene una tasa de procesado de 19,44 *fps*, sin ninguna directiva de optimización y empleando el método de multiplicación de matrices estándar. En los distintos escenarios se muestran la modificación del modo de multiplicación y los diferentes pragmas SDS y HLS. Finalmente, se obtienen ciertos escenarios que logran proporcionar una tasa de aproximadamente 30 *fps* lo que permite exportar el algoritmo a un escenario en tiempo real.

Finalmente se exportó el algoritmo a un sistema de captura de imágenes en tiempo real donde se procesa y se muestra el objeto detectado a través de un HDMI en un monitor. En este último escenario se muestra además la necesidad de introducir una actualización de fondo al ser observable como este algoritmo es sensible a los cambios de iluminación.

Como trabajo futuro queda entonces determinar cuándo es el momento óptimo para realizar la actualización de fondo y como realizarla. Eligiendo si se realiza tras la captura de determinado número de *frames* o si se realiza pasado cierto tiempo, además sería bueno optimizar el proceso offline para que no sea apreciable la detección de captura de imágenes en tiempo real.

Otro posible trabajo futuro sería la capacidad de determinar el umbral de detección en función del escenario de fondo y no de manera empírica, puesto que el umbral de detección para este proceso se ha determinado mediante fallo-acierto en los softwares previos a SDx, Matlab y Eclipse.

Finalmente, como trabajo futuro, se propone investigar reducir la latencia que se produce con los accesos a los buffers de memoria y que hacen que la imagen se vea con cierto retraso a la capturada. Es decir, se muestra cómo se cierra la mano tres segundos después de haberla cerrado. Mejorar esta latencia supone además mejorar la tasa de procesado de imágenes que como se ha observado, a pesar el mismo algoritmo por separado, la obtenida es relativamente inferior.

10. Bibliografía.

[1] Aguirre Dobernack, Nicolás. (TFG) "*Implementación de un sistema de detección de señales de tráfico mediante Visión Artificial basado en FPGA"* - Universidad de Sevilla, 2013. [http://bibing.us.es/proyectos/abreproy/12112/fichero/Documento_completo%252FProyecto+Fin+de+Carrera-Nicol%C3%A1s+Aguirre+Dobernack.pdf]

[2] Acasandrei, Laurentiu. Barriga, Ángel. "*Implementación sobre FPGA de un sistema de detección de caras basado en LEON3"* – Universidad de Sevilla. [http://digital.csic.es/bitstream/10261/84137/4/Implementaci%C3%B3n%20sobre%20FPGA.pdf]

[3] Llorente Aragón, Rodrigo. (TFM) "*Codiseño de un sistema HW/SW para el procesamiento de video en tiempo real"* – Universidad de Alcalá de Henares, 2019. [https://ebuah.uah.es/dspace/bitstream/handle/10017/39927/TFM_Llorente_Arag%c3%b3n_2019.pdf?sequence=1&isAllowed=y]

[4] Suriano, Leonardo. "*Analysis of a Heterogeneous Multi-Core, Multi-HW-Accelerator-Based System Designed Using PREESM and SDSoC*". [http://oa.upm.es/51778/1/recosoc17.pdf] Fecha última consulta: 12/06/20

[5] Kalb, Tobias. "*Enabling Dynamic and Partial Reconfiguration in Xilinx SDSoC*" [http://tulipp.eu/wp-content/uploads/2019/01/reconfig2016_kalb.pdf] Fecha última consulta: 12/06/20

[6] Merino, Daniel. (TFG) "*Aprendizaje en el funcionamiento de la herramienta de SDSoC de Xilinx para diseños basados en dispositivos SoC"* – Universidad de Alcalá de Henares, 2019. [https://ebuah.uah.es/dspace/bitstream/handle/10017/39346/TFG_Merino_Perez_2019.pdf?sequence=1&isAllowed=y]

[7] Bravo Muñoz, Ignacio. "*Arquitectura basada en FPGAs para la detección de objetos en movimiento utilizando visión computacional y técnicas PCA"* – Capítulo 4 - Universidad de Alcalá de Henares, 2007.

[https://ebuah.uah.es/dspace/bitstream/handle/10017/1381/tesis_ignacio_bravo.pdf?sequence=1&isAllowed=y]

[8] César Jiménez, Rodrigo. (TFG) "*Generador de Matrices de Covarianza basado en Vivado HLS"* – Universidad de Alcalá de Henares, 2016. [https://ebuah.uah.es/dspace/bitstream/handle/10017/26579/TFG-C%C3%A9sar-Jim%C3%A9nez-2016.pdf?sequence=1&isAllowed=y]

[9] PYNQ: PYTHON PRODUCTIVITY. [http://www.pynq.io/] Fecha última consulta: 12/06/20

[10] Guía de Referencia de la plataforma Zybo Z7 – Fecha de última consulta: 12/06/20 [https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/start]

[11] Hoja de Características del SoC Zynq 7000 de Xilinx – Fecha de última consulta: 12/06/20 [https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf]

[12] Introducción a la herramienta SDx, Xilinx. – Fecha de última consulta: 12/06/20. [https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/nal1504034315084.html]

[13] Petalinux Tools – Xilinx – Fecha de última consulta: 12/06/20 [https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html]

[14] Vivado Design Suite – High Level Synthesis – Fecha de última consulta: 12/06/20 [https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug902-vivado-high-level-synthesis.pdf]

[15] Vivado Design Suite – Xilinx – Fecha de última consulta: 12/06/20 [https://www.xilinx.com/support/university/vivado.html]

[16] PCam 5C Reference Manual – Digilent – Fecha de última consulta: 13/06/20 [https://reference.digilentinc.com/reference/add-ons/pcam-5c/reference-manual?_ga=2.250636998.1749145181.1591972512-2114177312.1538468879]

[17] Zybo Z7 PCam 5C Demo – Digilent – Fecha de última consulta: 13/06/20 [https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-z7-pcam-5c-demo/start]

[18] Zybo Z7 20 reVISION platform – Digilent – Fecha de última consulta: 15/06/20 [https://github.com/Digilent/reVISION-Zybo-Z7-20]

[19] Kastner,Ryan. Matai,Janarbek. Neuendorffer,Stephen – "*Parallel Programming for FPGA's"* Capítulo 7 - [11-05-2018] Fecha de última consulta – 25/06/2020 [https://arxiv.org/pdf/1805.03648.pdf]

[20] "*fi, Construct fixed-point numeric object*" - Matlab Help Center – Fecha de última consulta – 10/07/2020 [https://es.mathworks.com/help/fixedpoint/ref/embedded.fi.html]

[21] Tuan Nguyen, Alex Adamson, Andreas Santucci - "*Matrix Multiplication: Strassen's Algorithm"* – Standford University – Fecha de última consulta – 15/07/2020 [https://stanford.edu/~rezab/classes/cme323/S16/notes/Lecture03/cme323_lec3.pdf]

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR