Acelerando aplicaciones con el entorno unificado Vitis. Caso estudio: Vitis visión library

Manuel Jesús Rodríguez Valido
line 2: *dept. name of organization (of Affiliation)*
Universidad de La Laguna
San Cristóbal de La Laguna, España
mrvalido@ull.es

Eduardo Magdaleno Castello
line 2: *dept. name of organization (of Affiliation)*
Universidad de La Laguna
San Cristóbal de La Laguna, España
emagcas@ull.edu.es

Patricia Sánchez Medina
line 2: *dept. name of organization (of Affiliation)*
Universidad de La Laguna
San Cristóbal de La Laguna, España
alu0100981797@ull.edu.es

Argelio Mauro González
line 2*: dept. name of organization (of Affiliation)*
Universidad de La Laguna
San Cristóbal de La Laguna, España
alu0100070378@ull.edu.es

Resumen—*Actualmente, los aceleradores hardware constituyen un campo de investigación en pleno desarrollo, proponiendo constantemente nuevas aplicaciones y dispositivos, y explorando su influencia en diferentes campos. En el presente artículo se estudia, analiza y aprovecha la potencia que presenta para enseñar a los estudiantes de ingeniería la aceleración hardware, caso a estudio: procesado de imágenes. Valoraremos la capacidad que tiene el entorno de desarrollo Vitis y el uso de su librería de visión, para crear herramientas y aplicaciones de procesamiento de imágenes.*

Palabras clave—FPGA, acelerador hardware, Vitis, procesado de imágenes.

I. INTRODUCCIÓN

Vitis es una plataforma de software unificado que permite el desarrollo de aplicaciones aceleradas en hardware y software integrado para una gran variedad de plataformas Xilinx [1]. Es decir, Vitis ofrece un enfoque centrado en el software para desarrollar tanto hardware como software [2], pudiendo programar mediante C, C++ y *OpenCL* [3].

Las aplicaciones de visión por ordenador y el procesamiento de imágenes, omnipresentes hoy en día en una amplia gama de áreas, exigen una solución que pueda cumplir con el rendimiento en tiempo real y la flexibilidad para gestionar una gama de resoluciones a la vez que sean eficientes en términos de energía. Por estos motivos, el uso de la librería *Vitis Vision Library* (*VVL*) permite desarrollar aplicaciones aceleradas de visión por ordenador y procesamiento de imágenes, sin dejar de trabajar a nivel de aplicación. Las funciones de la librería ofrecen una interfaz familiar como las bibliotecas *OpenCV* y han sido optimizadas para el rendimiento y la utilización de recursos en las plataformas Xilinx.

Con este trabajo, pretendemos mostrar el poder de esta metodología, Vitis y el uso de la librería *VVL* conjuntamente con la librería OpenCV, para enseñar a desarrollar aceleradores hardware para las plataformas con FPGAs. La aceleración hardware aprovecha la posibilidad de estos dispositivos de ejecutar tareas simultáneas en software y hardware. Además, ofrecen ventajas en rendimiento, latencia y eficiencia energética para la implementación de aplicaciones de alta carga computacional y baja latencia en comparación con las CPUs (*Central Processing Unit*) de propósito general y las GPUs (*Graphics Processing Unit*) [3].

Para explorar esta metodología, hemos propuesto un Trabajo de Fin de Grado en el Grado en Ingeniería Electrónica Industrial y Automática en la Universidad de La Laguna. En él, se propone a la alumna que realice un estudio y análisis de la reciente lanzada plataforma, Vitis, para así valorar la capacidad que tiene el entorno de desarrollo mediante un caso a estudio centrado en los pormenores de la metodología. Se proporciona, además, la arquitectura de un algoritmo pipeline de procesado de imágenes, concretamente detección de centros en discos solares. Le proponemos que acelere por hardware una parte de esta arquitectura pipeline, el redimensionamiento de la imagen de entrada de 4096*4096 a 512*512 pixeles.

La librería *OpenCV*, permite de forma rápida obtener los centros de circunferencias con una simple llamada a la función: *HoughCicles*. Esta es una función con un alto coste computacional que, a su vez, se agrava cuando se ejecuta en sistemas empotrados. Una solución rápida del cálculo del centro de un círculo contenido en una imagen es: reducir el tamaño de la propia imagen de entrada antes de aplicar la transformada de *Hough*. Esta función (*resize* de la imagen) permite reducir la computación y si, además, se acelera por hardware, el tiempo de computación disminuye.

En este artículo se describirán las herramientas empleadas, así como la metodología y el procedimiento seguido para obtener la aplicación acelerada en hardware. Se detallará el flujo de trabajo empleado para la aplicación, además de los resultados y conclusiones obtenidos al término del proyecto.

II. MATERIALES Y MÉTODO

A. Materiales

Los materiales están asociados al flujo de trabajo que sigue la metodología de diseño creada por Xilinx. Distinguimos entre herramientas software, para el diseño de las aplicaciones, y herramientas hardware, necesarias como soporte tecnológico (placa de desarrollo basada en FPGA).

a) Software

1) Linux. Es muy importante tener en cuenta la versión del sistema operativo Linux que se va a emplear para el desarrollo. Es un requisito al que hay que prestar especial atención ya que las herramientas que se van a emplear son muy sensibles a estas variaciones y un problema de compatibilidad con los productos Xilinx derivará en retrasos relevantes en el desarrollo de la aplicación. Esta información está disponible en las páginas web oficiales de Xilinx [4].

2) Vitis. Como se comentó en la introducción de este documento, Vitis es una plataforma de software unificado análogo al anterior Xilinx SDK, Vivado *High-Level Synthesis* (*HLS*) y SDSoC. Hay una amplia variedad de librerías dentro de Vitis que permiten acceder a las funcionalidades ya creadas anteriormente en lugar de comenzar una aplicación desde cero. Existen dos bloques principales de librerías:

comunes y específicas. Las librerías comunes incluyen funciones estándar que se pueden necesitar en cualquier aplicación como son las operaciones matemáticas, algebraicas, estadísticas o manejo de datos. En las librerías específicas se encuentran paquetes aplicados a aplicaciones más particulares como análisis de datos o visión e imagen [1].

3) Vitis Vision Library (VVL). Estas librerías componen una versión reducida y optimizada para aceleración hardware de las librerías *OpenCV* (útiles para: procesamiento de imágenes, visión artificial, control de procesos, sistemas de seguridad con detección de movimiento, reconocimiento de objetos, robótica avanzada, etc. [2][5]) y permiten desarrollar e implementar aplicaciones de procesamiento de imágenes y visión por computador aceleradas en plataformas Xilinx, mientras se continúa trabajando a nivel de aplicación. Estas funciones ofrecen una interfaz familiar y se han optimizado para el rendimiento y la utilización de recursos en plataformas Xilinx [5].

4) Vivado. Vivado Design Suite es una herramienta de Xilinx para la síntesis de alto nivel y análisis de diseños HDL (*Hardware Description Language*). Vivado permite crear el diseño hardware de nuestra placa dirigido exclusivamente a las necesidades del proyecto que se vaya a realizar [2][6].

5) Petalinux Tools. Petalinux Tools son unas herramientas que permiten personalizar, crear e implementar soluciones Linux integradas en sistemas de procesamiento Xilinx (imagen Linux). Estas herramientas son proporcionadas por Xilinx y su objetivo es acelerar la productividad al admitir las herramientas de diseño hardware para facilitar el desarrollo de sistemas Linux para plataformas [7].

b) Hardware

1) Plataforma. Para ejecución del este proyecto se ha empleado el kit de evaluación ZCU102 (Fig. 1). Cuenta con una *FPGA Zynq UltraScale+ MPSoC*, correspondiente a la sección del procesador de la plataforma: *processing system* (*PS*). En este caso cuenta con: cuatro núcleos *Arm Cortex-A53* y procesadores en tiempo real *Cortex-R5F* de doble núcleo. Además, cuenta con una unidad de procesamiento gráfico *Mali-400 MP2*, basada en la lógica programable: *programmable logic* (*PL*) 16nm *FinFET+* de Xilinx.

Fig. 1. Zynq UltraScale+ MPSoC ZCU102 [3]

B. Método

La metodología propuesta por Xilinx para el diseño de sistemas embebidos y/o algoritmos para aceleración hardware está principalmente centrada en el software. Es decir, una vez codificado un algoritmo software, Vitis nos permite definir o identificar cuellos de botella en tiempo de ejecución y, con directrices dadas por la metodología, se decide qué parte del sistema se implementa en hardware y qué parte en software. Gracias a esta posibilidad, el ingeniero decide, atendiendo a criterios de velocidad de procesado, qué sección del código se ejecuta en *PS* y cuál en *PL*. Además, esta metodología, etiquetada por Xilinx como un software unificado, permite que el ingeniero desarrolle la aplicación en un único entorno y, de esta forma, se beneficien sin necesidad de tener un conocimiento profundo de los detalles del hardware.

Normalmente, y para ingenieros experimentados en diseño hardware, la aceleración comienza creando un diseño de la plataforma en Vivado desde cero. Para aquellos ingenieros no familiarizados con el diseño hardware, Vitis permite usar una plataforma base proporcionada por el fabricante: *board support package* (BSP) [8]. En ambos casos se precisa de un sistema operativo Linux que, al igual que para la plataforma base, existen dos opciones para su creación: se puede crear desde cero usando las herramientas *Petalinux* o se obtiene la imagen común adecuada para tu plataforma. En Fig. 2 podemos apreciar las dos posibilidades de entrada al entorno Vitis.

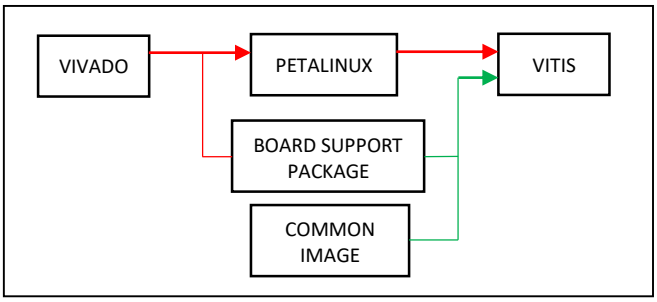

Fig. 2. Metodología de creación de sistemas empotrados

Si optásemos por diseñar la plataforma desde cero (camino en rojo en Fig. 2), haríamos uso del *block design* de la herramienta Vivado para definir la plataforma. En este paso se opta por una configuración de la FPGA ligada a las necesidades del proyecto, pudiendo incluir además del PS, diferentes componentes e interfaces básicos. Desde Vivado, se exporta un archivo XSA (*XelaSoft Archive*) que contiene toda la información que se ha incluido en el diagrama de bloques diseñado, es decir, el diseño hardware del sistema. Seguidamente, con las herramientas Petalinux, generamos la imagen Linux. Para esto es necesario importar tanto el diseño hardware (XSA) como el paquete de soporte de la placa (BSP). Como último paso en la creación y configuración del sistema operativo, Petalinux nos van guiado en la configuración de la imagen de este. Como se puede observar en Fig. 3, Petalinux generará una serie de archivos Linux que conforman esa imagen Linux y que, posteriormente, serán empleados en Vitis (Fig. 2).

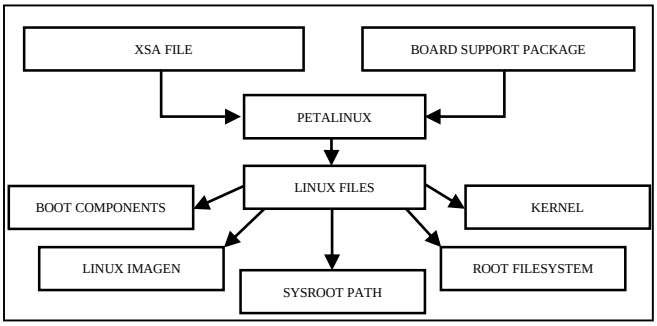

Fig. 3. Diagrama de bloques de Petalinux. Archivos de entrada y salida.

En caso de querer partir de la plataforma base y una imagen común cero (camino en verde en Fig. 2) no haremos uso de Petalinux y Vivado, pero el resultado obtenido será igualmente válido para la aplicación que se desee desarrollar.

Llegados a este punto, entramos en el entorno Vitis que integra todos los archivos en un paquete para que los diseñadores crear la aplicación y su algoritmo para ser acelerado (Fig. 4).

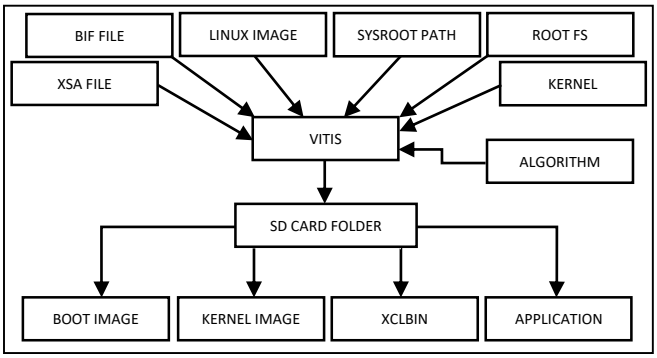

Fig. 4. Diagrama de bloques de Vitis. Archivos de entrada y salida del entorno.

Vitis proporciona, además, dos herramientas para realizar pruebas y depuraciones previas a la ejecución de la aplicación en la plataforma. Estas son: la emulación y la ejecución.

Por un lado, la emulación permite comprobar rápidamente si el funcionamiento del diseño es correcto tanto a nivel software como hardware. La emulación software consiste en la simulación puramente software de la aplicación creada, nos permite verificar con eficiencia y de forma rápida el funcionamiento de la aplicación acelerada. En la emulación hardware, el código (la aplicación) se compila en un modelo de comportamiento *RTL* (*Register Transfer Level*) que se ejecuta en Vivado para obtener una representación gráfica (formas de ondas) del funcionamiento del diseño.

Por otro lado, una vez revisado el correcto funcionamiento mediante las emulaciones software y hardware, nos resta realizar la ejecución hardware. Básicamente, la herramienta genera los archivos que se cargarán en una tarjeta *SD* para poder ejecutar la aplicación en la plataforma (FPGA).

III. CASO A ESTUDIO. CÁLCULO DE CENTROS CON LA TRANSFORMADA *HOUGH*

En nuestro caso a estudio, el desarrollo de la aplicación está formado por y hará uso de:

- Un algoritmo de detección de centros de discos solares mediante la transformada *Hough*. Bajo la metodología marcada por Vitis, todo proceso de aceleración hardware pasa por la creación de un algoritmo o descripción en software del diseño. Las imágenes del disco solar de entrada o de partida son tomadas del satélite *SDO* (*Solar Dynamics Observatory*) con el instrumento *HMI* (*Helioseismic and Magnetic Imager*). *SDO* es un satélite que cuenta con una herramienta diseñada para estudiar las oscilaciones y el campo magnético en la superficie solar (*HMI*), de él obtenemos imágenes de alta resolución (4096x4096) [9]. La transformada de *Hough* sobre estas imágenes conlleva un alto coste computacional, por lo que una reducción del tamaño de la imagen previa a la aplicación de la transformada mejora el tiempo de procesado. Esto es muy importante, sobre todo, en sistemas de baja capacidad computacional como puede ser el sistema de a bordo de un satélite.
- Uso de *VVL*. Como comentamos en los objetivos de este trabajo, utilizaremos la función acelerada *resize* de *VVL*. El resultado (imagen redimensionada) es la entrada de la función *HoughCircles*.
- Uso de *OpenCV*. Emplearemos la librería tanto para la función *HoughCircles* como para crear un banco de pruebas y comparar los resultados obtenidos por el acelerador y por *OpenCV*.

El algoritmo (C/C++) se diseña empleando el lenguaje e interfaz de comunicación de *OpenCL*. Con él, creamos un modelo enteramente software tanto del acelerador como su banco de pruebas. Este modelo de diseño está formado por dos bloques principales: *host code* y *kernel code* (Fig. 5). Son dos bloques diferenciados, pero se complementan ya que el ejecutable del *kernel* se llama desde el código *host* [8]. El *host code* se ejecuta en PS y el *kernel code* en PL.

En definitiva, el *host code* configura el entorno de ejecución, prepara los datos y llama al *kernel* que está destinado a ser acelerado (se ejecuta en la FPGA). Hay tres modalidades de ejecución del *kernel*: *secuential*, *pipelined* y *free-running*. En nuestro caso, sigue un modelo secuencial, el *host* ejecuta el *kernel* y tiene acceso a los resultados. El *host* finaliza el *kernel* y este último queda disponible para iniciarse de nuevo en su caso [10].

El *host code* está dividido en 4 secciones (Fig. 5) que etiquetamos como:

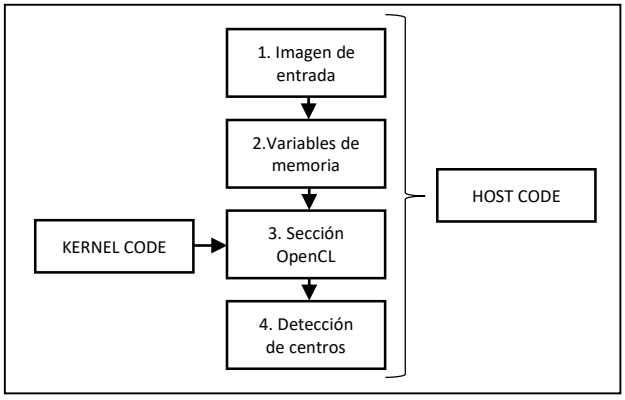

Fig. 5. Estructura de *host code*. Caso a estudio: detección de centros.

1) Imagen de entrada. En esta etapa se lee mediante los argumentos la imagen de entrada y se obtienen sus

dimensiones y características: escala de grises o RGB, así como el tamaño del redimensionamiento.

2) Variables de memoria. La reserva de memoria para variables en este tipo de diseños es muy importante debido a que, de no ser así, el sistema deriva en errores de dimensión fuera de rango.

3) Sección OpenCL. En esta sección se prepara los datos y llama al *kernel*, o *kernels* en su caso, que se ejecuta en la FPGA. Consta de los siguientes pasos [11]:

1º. Detectar el dispositivo acelerador. Dado que esta metodología es compatible con múltiples plataformas Xilinx, el primer paso consiste en detectar cuántos dispositivos aceleradores hay disponibles, su información (nombre, proveedor, etc.) y cuál es el que se va a emplear.
2º. Crear y configurar el contexto. Este paso inicializa la aplicación y define la memoria que necesita, servirá para ejecutar *kernels* en la plataforma o hacer transferencia de datos.
3º. Crear los buffers de entrada/salida. Los buffers son espacios de memoria de tamaño determinado, diseñados para que estén disponibles tanto para el host como para el *kernel* (plataforma).
4º. Configurar los argumentos del *kernel*. Cada función *kernel* (de *VVL*) tiene su lista de argumentos y se seleccionarán los necesarios para la aplicación a desarrollar.
5º. Configurar la transferencia de datos desde los buffers de entrada al dispositivo
6º. Llamar al *kernel* para que se ejecute
7º. Configurar la transferencia de datos desde los buffers de salida al host
8º. Esperar a que el *kernel* termine su tarea. Cuando el *kernel* termine su tarea, se obtendrá el resultado de la aceleración del redimensionamiento de la imagen realizado en hardware (*HLS*).

```
static constexpr int __XF_DEPTH = (HEIGHT * WIDTH *
(XF_PIXELWIDTH(TYPE, NPC_T)) / 8) / (INPUT_PTR_WIDTH / 8);
static constexpr int __XF_DEPTH_OUT = (NEWHEIGHT * NEWWIDTH *
(XF_PIXELWIDTH(TYPE, NPC_T)) / 8) / (OUTPUT_PTR_WIDTH / 8);
void resize_accel(ap_uint<INPUT_PTR_WIDTH>* img_inp,
                  ap_uint<OUTPUT_PTR_WIDTH>* img_out,
                  int rows_in,
                  int cols_in,
                  int rows_out,
                  int cols_out) {
            // clang-format off
    #pragma HLS INTERFACE m_axi     port=img_inp  offset=slave
bundle=gmem1 depth=__XF_DEPTH
    #pragma HLS INTERFACE m_axi     port=img_out  offset=slave
bundle=gmem2 depth=__XF_DEPTH_OUT
    #pragma HLS INTERFACE s_axilite port=rows_in
    #pragma HLS INTERFACE s_axilite port=cols_in
    #pragma HLS INTERFACE s_axilite port=rows_out
    #pragma HLS INTERFACE s_axilite port=cols_out
    #pragma HLS INTERFACE s_axilite port=return
    // clang-format on
    xf::cv::Mat<TYPE, HEIGHT, WIDTH, NPC_T> in_mat(rows_in,
cols_in);
    xf::cv::Mat<TYPE, NEWHEIGHT, NEWWIDTH, NPC_T>
out_mat(rows_out, cols_out);
            // clang-format off
    #pragma HLS DATAFLOW
    // clang-format on
    xf::cv::Array2xfMat<INPUT_PTR_WIDTH, TYPE, HEIGHT, WIDTH,
NPC_T>(img_inp, in_mat);
    xf::cv::resize<INTERPOLATION, TYPE, HEIGHT, WIDTH,
NEWHEIGHT, NEWWIDTH, NPC_T, MAXDOWNSCALE>(in_mat, out_mat);
    xf::cv::xfMat2Array<OUTPUT_PTR_WIDTH, TYPE, NEWHEIGHT,
NEWWIDTH, NPC_T>(out_mat, img_out);
}
```

Fig. 6. Código del *kernel code*. Función *resize*.

En nuestro caso a estudio, el *kernel code* (Fig. 6) se estructura como sigue:

1º. Como cualquier función en C/C++ la función *resize* tiene como argumentos dos punteros a imagen de entrada y a imagen de salida, además de un conjunto de argumentos de configuración. Estos argumentos, a través de directiva pragma HLS interface definen la interfaz de comunicaciones de datos y permite que el *kernel* lea y escriba datos en la memoria global [12]. Los punteros a memoria de entrada y salida son tipo m_axi, modo de transferencia de datos en ráfaga. Los otros cinco puertos son tipo *s_axilite*, interfaz que utiliza el host para configurar el kernel [13].
2º. Se crean dos variables tipo *xf::cv::Mat* (*in_mat* y *out_mat*). Estas son pasadas al *kernel* previamente a la conversión de sus tipos. La salida del *kernel* es de nuevo convertida al tipo *Array* y es devuelto al *host code*.

4) Detección de centros. Aplicamos la función, *HoughCircles*, sobre la imagen del disco solar

redimensionada, con unos determinados parámetros de entrada y salida, y obtendremos el resultado del centro detectado. Comprobaremos los resultados obtenidos pintando sobre la imagen las coordenadas del centro del disco solar y la circunferencia detectada (Fig. 7).

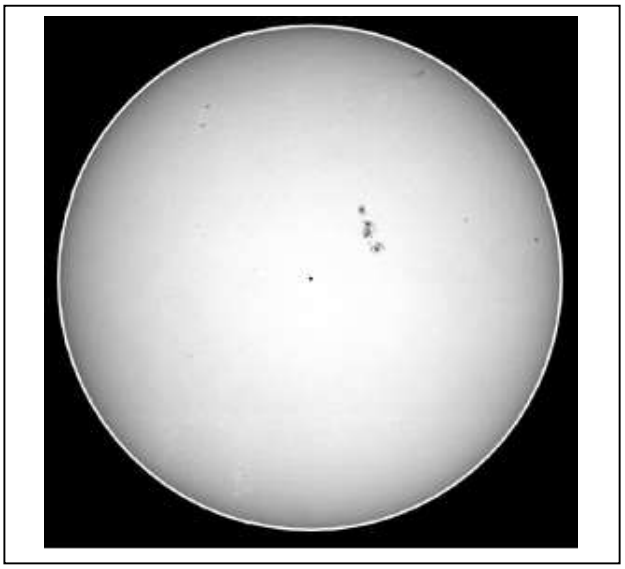

Fig. 7. Representación gráfica del centro y círculos calculado por *Hough*

IV. CONCLUSIONES

Hemos presentado la metodología basada en el entorno Vitis del fabricante de FPGA Xilinx, aclarando los pasos principales para desarrollar un acelerador. Además, hemos situado, en el proceso metodológico o flujo de trabajo otras herramientas complementarias tales como Petalinux y Vivado. Esta metodología y herramientas permiten reducir el esfuerzo de diseño, implementación y tiempos de ejecución del algoritmo de partida al completo.

Una vez adquiridos los conocimientos necesarios de las herramientas y la metodología a seguir para la realización del diseño de sistemas empotrados y aceleración de una función/algoritmo, ha sido posible desarrollar las aplicaciones empleando las herramientas Vitis, Vivado y Petalinux y haciendo uso de *Vitis Vision Library* y *OpenCV*. Este trabajo nos deja con una percepción clara de la versatilidad de Vitis. Ofreciéndonos un enfoque centrado en el software para desarrollar tanto hardware como software.

A pesar de que esta metodología de diseño tiene una curva de aprendizaje muy elevada, una vez familiarizados con ella, resulta en una forma muy útil y versátil de creación de sistemas empotrados con aceleradores hardware.

Hemos diseñado un algoritmo que acelera el proceso de reducción de una imagen con un acelerador procedente de la *VVL*. Conjuntamente hemos comparado resultados obtenidos con una solución basada enteramente en *OpenCV* y con la obtenida usando un acelerador para realizar una tarea del pipeline: detección centros. Esto ha sido posible gracias a la posibilidad que ofrece esta metodología de identificar qué funciones o tareas es preferible ejecutar en software o hardware. Permitiendo esto, paralelizar los procesos que de otra forma se realizarían secuencialmente y ralentizarían considerablemente la ejecución de la aplicación.

Además, y sin centrarnos en el algoritmo concreto de procesado de imagen, hemos presentado la estructura del *host code* así como el *kernel code*, ambos codificado en un lenguaje de alto nivel, C/C++ y también hemos visualizado como se usa e integra el lenguaje para sistemas heterogéneos *OpenCL*.

V. REFERENCIAS

[1] Xilinx, «Vitis Unified Software Platform», 2020. [En línea]. Disponible: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.

[2] Digilent Blog, «What's different between Vivado and Vitis?», 2021. [En línea]. Disponible: https://blog.digilentinc.com/whats-different-between-vivado-and-vitis/.

[3] Xilinx, «Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit», 2020. [En línea]. Disponible: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

[4] Xilinx, «Vitis Unified Software Development Platform 2020.2 Documentation», 2020. [En línea]. Disponible: https://www.xilinx.com/htmldocs/xilinx2020_2/vitis_doc/acceleration_installation.html#vhc1571429852245.

[5] Xilinx, «Vitis Vision Library», 2020. [En línea]. Disponible: https://www.xilinx.com/products/design-tools/vitis/vitis-libraries/vitis-vision.html.

[6] Xilinx, «Vivado», 2020. [En línea]. Disponible: https://www.xilinx.com/products/design-tools/vivado.html.

[7] Xilinx, «Petalinux Tools», 2020. [En línea]. Disponible: https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html.

[8] Xilinx, «Vitis Unified Software Development Platform 2020.2 Documentation» 2020. [En línea]. Disponible: https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/runemulation1.html#btg1600442263101.

[9] Standford Solar Group, «Helioseismic and Magnetic Imager». [En línea]. Disponible: http://hmi.stanford.edu/

[10] High-Level Synthesis & Embedded Systems, «Embedded Hardware Accelerator with Xilinx Vitis: Part 3: Programming Model», 2020. [En línea]. Disponible: https://highlevel-synthesis.com/2020/11/04/embedded-hardware-accelerator-with-xilinx-vitis-part-3-programming-model/.

[11] High-Level Synthesis & Embedded Systems, « Embedded Hardware Accelerator with Xilinx Vitis: Part 3: Programming Model», 2020. [En línea]. Disponible: https://highlevel-synthesis.com/2020/11/04/embedded-hardware-accelerator-with-xilinx-vitis-part-3-programming-model/

[12] Vitis High-Level Synthesis User Guide (UG1399), «AXI4 Master Interface», 2021. [En línea]. Disponible: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/AXI4-Master-Interface

[13] Xilinx, «Vitis Accel Examples», 2020. [En línea]. Disponible: https://xilinx.github.io/Vitis_Accel_Examples/2020.1/html/hello_world.html