CÁLCULO DE DESPLAZAMIENTOS PARA CORRECCIÓN DE FLATFIELD EN IMÁGENES SOLARES MEDIANTE FPGA

ESCUELA SUPERIOR DE INGENIERÍA Y TECNOLOGÍA

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

TRABAJO DE FIN DE GRADO

Estudiante: Patricia Sánchez Medina

Tutor: Manuel Jesús Rodríguez Valido

Fecha: Julio 2021

Resumen

Actualmente, los sistemas embebidos constituyen un campo de investigación en pleno desarrollo, proponiendo constantemente nuevas aplicaciones y dispositivos, y explorando su influencia en diferentes campos.

En el presente Trabajo de Fin de Grado se estudia, analiza y aprovecha la recientemente lanzada plataforma de software unificado de Xilinx para el diseño de sistemas empotrados, Vitis. El objetivo es valorar la capacidad que tiene el entorno de desarrollo, evaluar el diseño de un acelerador hardware y, haciendo uso de esta nueva metodología de diseño, crear herramientas y aplicaciones de procesamiento de imágenes.

Para ello usaremos y evaluaremos la Vitis Vision Library, aplicándolas a los cálculos de desplazamientos del disco solar. Estos desplazamientos relativos son necesarios para el cálculo del flatfield.

El flatfield consiste en un proceso de corrección/calibración de imágenes solares. Al realizar fotografías, en este caso al Sol, las imágenes resultantes tienen pérdidas debidas a ruidos, a la suciedad la óptica del telescopio, píxeles defectuosos del sensor de imagen y otros factores, esto hace que la calidad de la imagen obtenida no sea la misma que recibe originalmente la cámara SSD.

El proceso de aprendizaje del modo de funcionamiento de las herramientas necesarias para la ejecución de este sistema embebido y realización de pruebas previas a la ejecución del programa definitivo, nos han permitido obtener una clara visión de las posibilidades que nos ofrece esta metodología. Esto nos ha llevado a optimizar uno de los

pasos de la calibración de flatfield, concretamente el cálculo de desplazamientos de las imágenes.

Con este propósito, se ha decidido acelerar el proceso de redimensión de la imagen solar implementando esta tarea en hardware y empleando las librerías de aceleración, Vitis Vision Library, para posteriormente realizar el cálculo de desplazamientos de la imagen solar. Considerando los resultados obtenidos, se podrá comprobar si finalmente, al realizar la aceleración hardware e implementarlo en la FPGA, es posible acelerar el proceso completo comparado con su previo método de corrección.

Palabras clave

ZCU102, Zynq UltraScale+ MPSoC, FPGA, sistemas empotrados, aceleración por Hardware, Vivado, Vitis™, Vitis Vision Library, Petalinux, Flat-Field, OpenCV.

Abstract

Nowadays, embedded systems are a in full development research field. There are constantly emerging new applications and devices, and exploring their influence in different fields.

In this Final Degree Project we study, analyze and take advantage of the recently launched Xilinx unified software platform for the design of embedded systems. The objective is to assess the capability of the development environment and evaluate the design of a hardware accelerator. Therefore, use this new design methodology to create image processing tools and applications.

To do so, we will use and evaluate the Vitis Vision Library, applying it to the solar disc displacements calculations. These relative displacements are necessary for the estimation of the flatfield.

The flatfield is a process of correction/calibration of solar images. When taking pictures, in this case of the Sun, the resulting images have losses due to noise, dirty telescope optics, defective pixels in the image sensor and other factors. This means that the quality of the image obtained is not the same as the one originally received by the SSD camera.

The process of learning how the tools are necessary for the execution of this embedded system work and carrying out tests prior to the execution of the final programme has allowed us to obtain a clear vision of the possibilities offered by this methodology. This has led us to optimise one of the flatfield calibration steps, specifically the calculation of image displacements.

For this purpose, we have decided to accelerate the solar image resizing process by implementing this task in hardware and using the acceleration libraries, Vitis Vision Library, to subsequently perform the calculation of solar image displacements. Considering the results obtained, it will be possible to check if finally, by performing the hardware acceleration and implementing it in the FPGA, it is possible to accelerate the whole process compared to its previous correction method.

Agradecimientos

A mi tutor Manuel Jesús Rodríguez Valido, por estar dispuesto a sacar adelante este proyecto por muy complicado que se pusiera en algún momento. Gracias por estar esas eternas mañanas durante estos últimos 5 meses junto a los ordenadores haciendo ensayos de prueba y error, y por estar ahí siempre que lo he necesitado.

Quiero agradecer también a todos los profesores que he tenido, tanto en la universidad como fuera de ella. Gracias por haber confiado en mí, por haberme dado el empujón que necesitaba en el momento adecuado y por animarme a creer y valorar tanto mi trabajo como a mí misma.

A la gran familia que me ha dado esta etapa académica, sobre todo en este último año, sin ustedes no hubiera sido lo mismo. Por animarme, por estar ahí siempre, en las buenas y las malas, por formar parte de la que puedo decir que ha sido la mejor etapa de mi vida, por todo esto les estaré siempre agradecida.

Y por último a mi familia, mi motor. Mis padres, mi hermana y las hermanas que me ha dado la vida. Por el apoyo incondicional y constante motivación, por su paciencia y compresión, por todo, gracias.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

1. Introducción

Solar Orbiter es un satélite científico de observación solar lanzado el 10 de febrero de 2020. Su misión es obtener las imágenes más cercanas tomadas al Sol. Para ello se ha instalado en el satélite un total de seis instrumentos de teledetección para estudiar tanto la superficie como la capa más externa de la atmósfera solar [1].

Uno de los objetivos de esta misión espacial es conseguir imágenes solares. Para ello se hace uso del instrumento SOPHI (Polarimetric and Helioseismic Imager for Solar Orbiter) creado en un 40% por el IAC (Instituto de Astrofísica de Canarias). El instrumento cuenta con dos telescopios de los cuáles el de disco entero (FDT) es completamente responsabilidad española, al igual que la electrónica del instrumento. Su principal trabajo es la medida del campo magnético vectorial y de los flujos de velocidad del sol [2].

Cuando se captura una imagen a través de una cámara y un sistema óptico, siempre hay unas pérdidas/ruido que hacen que la imagen obtenida tenga errores. A todo esto, se añade la degradación que sufre la óptica y el sensor en el espacio debido a la radiación y la no uniformidad espacial de la ganancia del sensor. Normalmente el proceso de calibrado del sensor y su óptica (flatfield) se realiza en laboratorio antes del lanzamiento. Como se comentó anteriormente, la degradación a lo largo del tiempo va afectando al funcionamiento del sistema.

Kuhn, Lin y Loranz [3] desarrollan un algoritmo para calibrar la falta de uniformidad espacial (flatfield) de los detectores de matriz de imágenes (tipo CCD). La Figura 1 muestra los efectos de la no uniformidad espacial captados por la cámara.

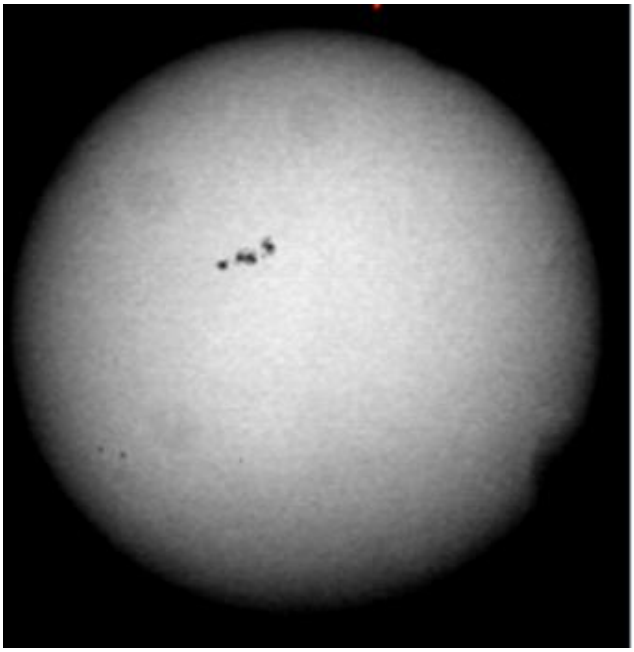


Figura 1: Imagen solar

El proceso de calibración de flatfield ofrecido por estos tres científicos está formado por varias etapas. Para verificar dicho algoritmo hemos seguido el proceso descrito por J.A. Bonet, cuyos pasos consisten añadir una ganancia sintética a la imagen incidente para obtener lo que sería la imagen observada por la cámara. De esta forma, y empleando esa misma imagen desplazada 8 veces (con desplazamientos conocidos), se procede a realizar la detección de centros para a continuación, aplicar el algoritmo de Kun, Lin y Loranz y obtener la imagen (flatfield image, con la ganancia píxel a píxel). Finalmente, esta flatfield se divide a la imagen centrada para eliminar esas pérdidas y calibar la imagen.

Lo idóneo para conseguir una optimización favorable del proceso de calibración sería acelerar el proceso descrito completo. La obtención de flatfield tiene un alto coste computacional, sobretodo si se realiza en sistemas empotrados como el que lleva el instumento SOPHI. En el presente Trabajo de Fin de Grado se ha decidido arrojar algo de luz a la posibilidad de acelerar uno de los pasos del proceso: el cálculo de desplazamientos de las imágenes. A partir de las coordenadas de los centros se pueden obtener los desplazamientos relativos de los discos solares.

Para abordar y obtener una solución para nuestro acelerador haremos uso de la nueva metodología de diseño de sistemas empotrados ofrecida por Xilinx, empleando la plataforma Vitis, y empleando la librería de aceleración y procesamiento de imágenes, Vitis Vision Library.

Este trabajo se estructura de la siguiente forma. Primero, se indicarán los objetivos que motivan a la realización del propio trabajo. Segundo, se describirán las herramientas empleadas, así como la metodología y el procedimiento seguido para obtener los resultados requeridos. Y, por último, se detallará el algoritmo de corrección de flatfield implementado y, los resultados y conclusiones obtenidos al término del proyecto.

2. Objetivos

El objetivo de este TFG es usar la metodología basada en la nueva herramienta Vitis y la Vitis Vision Library para acelerar por hardware un algoritmo de procesado de imágenes implementado en FPGA. Con esto, se pretende mejorar el rendimiento temporal del algoritmo para obtener los resultados en un tiempo menor. Esta metodología es compleja y por ello, requiere de una curva de aprendizaje alto. Es importante identificar y estudiar cada uno de los elementos que la forman, y para conseguir este propósito tenemos como fin:

- Entender la metodología para acelerar un proceso en Hardware.
    - Desarrollo de ejemplos para familiarizarnos con las herramientas y entender el flujo de diseño usado por la nueva metodología. En definitiva, ensayos de prueba y error.
- Diseñar e implementar una plataforma de partida para integrar el acelerador a partir de su creación y diseño en Vivado o empleando la plataforma general ofrecida por Xilinx (common image).
- Uso de las Vitis Vision Libraries.
- Desarrollar la aplicación software realizando las emulaciones necesarias para comprobar su funcionamiento:
    - Identificar qué tarea del código (kernel) se quiere implementar en Hardware para acelerar.
    - Uso de la función Resize de OpenCV para acelerar el redimensionamiento de la imagen y obtener los centros de los discos solares desplazados.

3. Herramientas

Para la realización del presente proyecto se ha empleado la nueva metodología de diseño de plataformas creada por Xilinx. Para ello se emplean tres herramientas principales: Vitis y sus librerías de procesamiento de imáegenes (Vitis Vision Library), Vivado y Petalinux, además de la plataforma ZCU102.

3.1. Vitis Unified Software Platform

Vitis es la plataforma de software unificado que permite el desarrollo de aplicaciones aceleradas y software integrado, análogo al anterior Xilinx SDK, Vivado High-Level Synthesis (HLS) y SDSoC, para una gran variedad de plataformas Xilinx [4]. Anteriormente, el diseño hardware se podría realizar únicamente en Vivado. Ahora, en Vitis es posible realizar diseños que se ejecuten en FPGA sin la necesidad de emplear Vivado. Este último ofrece un enfoque centrado en el hardware para diseñar hardware, mientras que Vitis ofrece un enfoque centrado en el software para desarrollar tanto hardware como software [5].

- *Vitis IDE*

Es el entorno de desarrollo integrado de Vitis, está diseñado para desarrollar aplicaciones software integrado empleando diseños hardware creados con Vivado Design Suite [4].

- *Librerías: "Vitis Accelerated Libraries"*

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 [4].

- *Vitis Vision Library (OpenCV)*

Una de las librerías de aceleración mencionadas anteriormente es la Vitis Vision. Estas librerías, que componen una versión optimizada de las librerías OpenCV, 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 [6].

OpenCV (Open Source Computer Vision) es una librería multiplataforma originalmente desarrollada como proyecto por Intel para apoyar a los primeros compiladores Intel C++ y Microsoft Visual C++ en x86. Es muy utilizada actualmente para el procesamiento de imágenes y la visión artificial, también se ha utilizado en infinidad de aplicaciones como control de procesos, sistemas de seguridad con detección de movimiento, reconocimiento de objetos, robótica avanzada, etc. [7].

- *Kit de desarrollo: Vitis Core*

El Vitis Core consiste en el conjunto de herramientas (compilador, analizador y depurador) de desarrollo gráfico que permite analizar los algoritmos desarrollados en C, C++ u OpenCL [4].

- *Xilinx Runtime (XRT)*

Este componente es la clave principal de la comunicación entre el código de la aplicación (el "host") y el kernel acelerador. Proporciona todas las librerías, APIs, drivers y utilidades que se necesitan para tener un completo control del acelerador a la hora transferir datos, sincronizar, ejecutar las funciones PL [4].

3.2. 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. Este diseño consiste en un conjunto de archivos de lenguaje de descripción de hardware (HDL, típicamente Verilog o VHDL), o en un diseño de bloque, que puede incluir una variedad de bloques de IP preconstruidos (que en su esencia abstraen el HDL prescrito). Si un diseño incluye un procesador, Vitis también escribirá el programa para que se ejecute en esa sección, ya que Vivado solo maneja la lógica programable (PL, Programmable Logic) y no el procesador (PS, Processing System) [8] [5].

3.3. Petalinux Tools

Petalinux Tools son unas herramientas que permiten personalizar, crear e implementar soluciones Linux integradas en sistemas de procesamiento Xilinx (imagen Linux). Su objetivo es la aceleración de la productividad al admitir las herramientas de diseño hardware de Xilinx para facilitar el desarrollo de sistemas Linux para plataformas como la Zynq® UltraScale+™ MPSoC ZCU102 [9].

3.4. FPGA. Plataforma de desarrollo ZCU102

Existen muchos tipos de FPGA y cada una de ellas proporciona unas capacidades y funciones que pueden ser beneficiosas a la hora de implementar en ellas una u otra tarea. A lo largo del tiempo, Xilinx ha creado una gran variedad de dispositivos programables. Concretamente las Zynq UltraScale+ MPSoC están formadas por un tejido programable FPGA con múltiples procesadores de núcleo duro. Además, integran la programabilidad de software de un procesador con la programabilidad de hardware de una FPGA, proporcionando a los diseñadores rendimiento, flexibilidad y escalabilidad al sistema [10].

Por todo esto, para ejecución del este proyecto se ha empleado el kit de evaluación ZCU102. Cuenta con un Zynq UltraScale+ MPSoC (sistema multiprocesador en chip) con cuatro núcleos Arm Cortex-A53, procesadores en tiempo real Cortex-R5F de doble núcleo y una unidad de procesamiento gráfico Mali-400 MP2 basada en la lógica programable 16nm FinFET+ de Xilinx. La ZCU102 (Figura 2 y Figura 3) es una placa muy flexible que permite el desarrollo e implementación de múltiples aplicaciones ya que es compatible con los principales periféricos e interfaces [11].

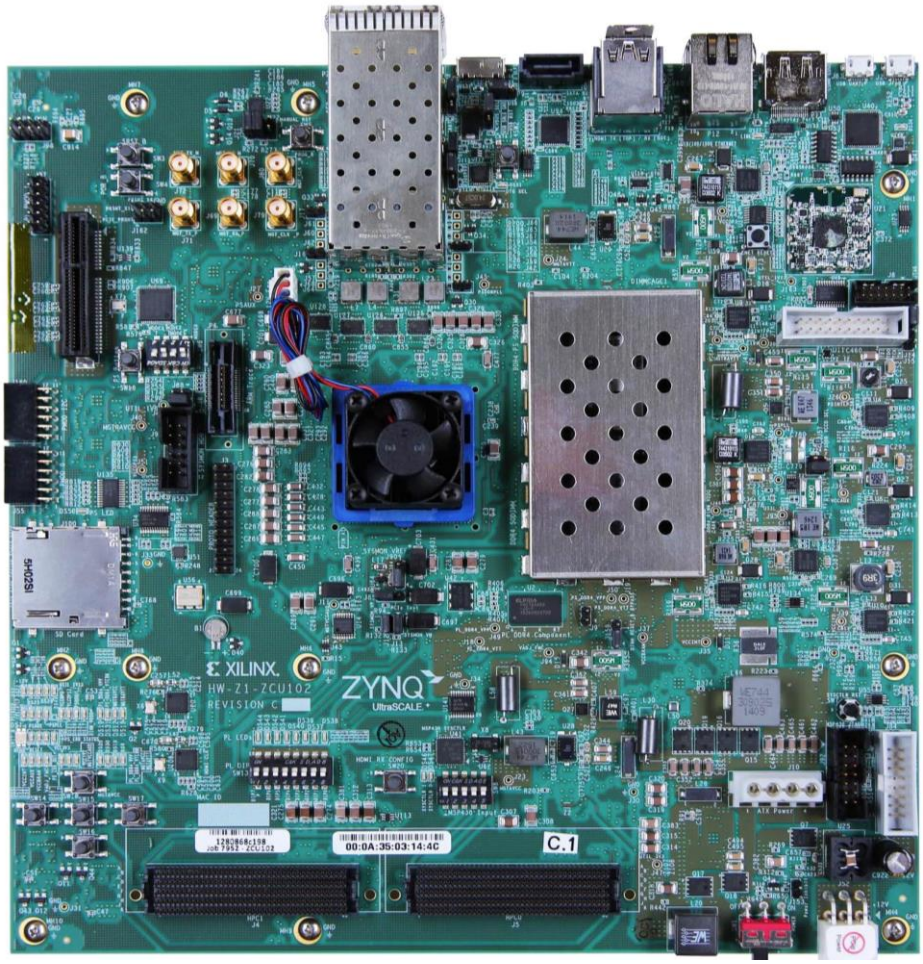


Figura 2: ZCU102 [11]

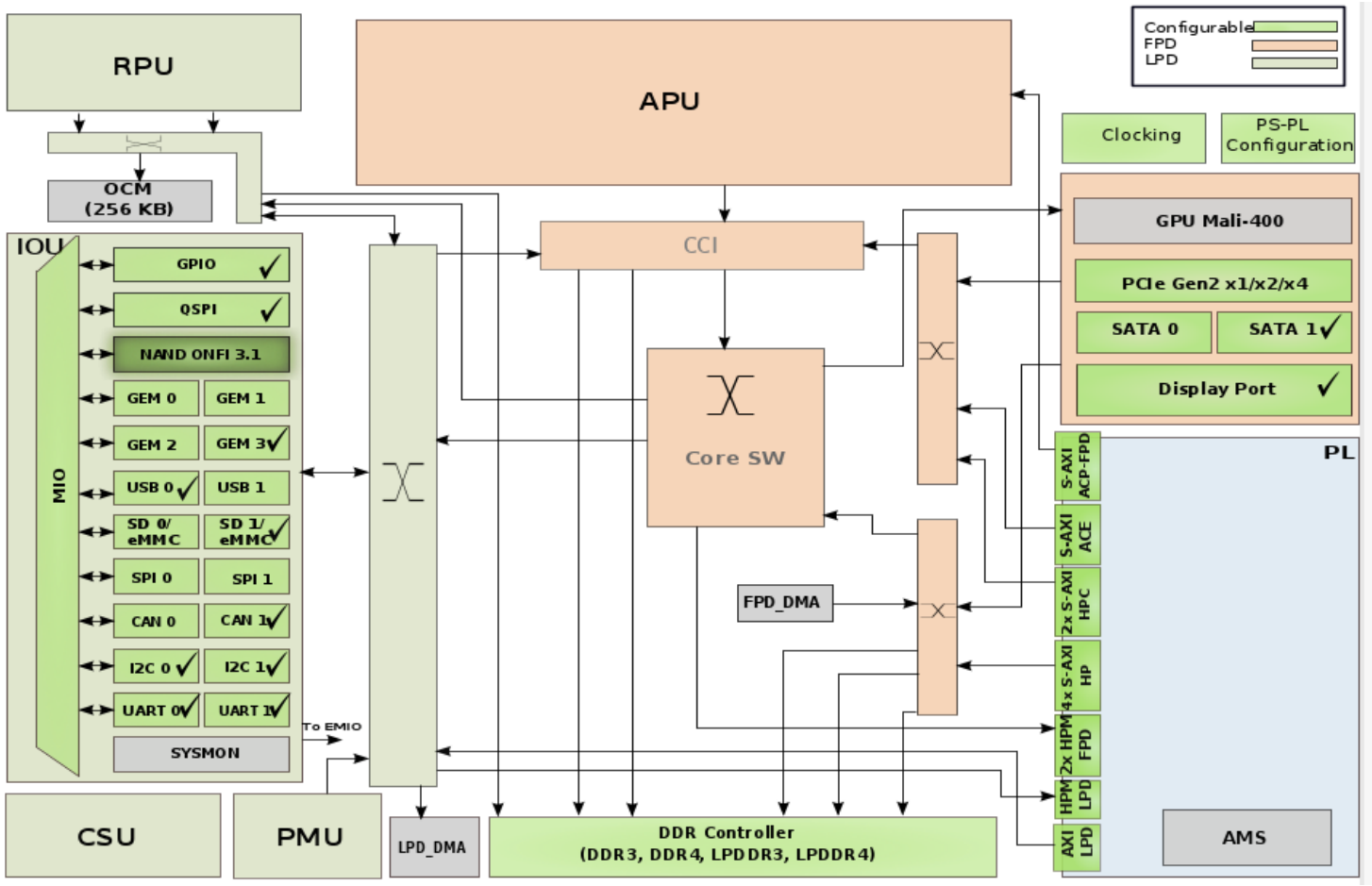


Figura 3: Diagrama de bloques del Zynq UltraScale+ MPSoC [11]

4. Método

4.1. Aceleración por Hardware.

La aceleración por hardware consiste en asignar tareas o funciones (partes de código) de computación a componentes hardware especializados dentro del propio sistema con el fin de obtener resultados en un intervalo de tiempo menor. Estas tareas de computación o algoritmos codificadas en un lenguaje software como C++, pueden ser ejecutadas en un procesador como el Cortex-R5F que se encuentra integrado en la FPGA. No obstante, al implementarlas en la sección programable de la FPGA (hardware), se consigue un aumento considerable de la eficiencia tanto por la posibilidad de realización de tareas simultáneas como por la velocidad comparada con los resultados de implementación en software. Las FPGA para este tipo de diseños son la mejor opción por su rendimiento y flexibilidad [12] [13]. Gracias a esta posibilidad, el ingeniero decide, desde el punto de vista del desarrollo de un algoritmo realizado en software, atendiendo a criterios de velocidad de procesado, qué sección del código se ejecuta en el procesador y cuál en la parte programable.

La optimización del tiempo al implementar tareas en hadrware y software se puede observar en la Figura 4 y Figura 5. La Figura 4 muestra un modelo de realización de tareas secuencual, correspondiente realizar todas las tareas en software. Mientras que, al paralelizar tareas, en la Figura 5, en un menor intervalo de tiempo se consigue ejecutar el mismo número de tareas.

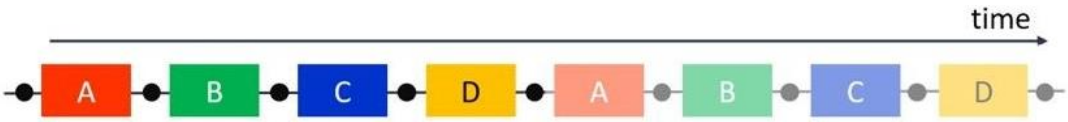


Figura 4: Tareas de forma secuencial [14]

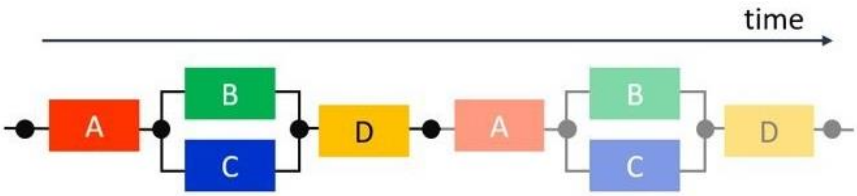


Figura 5: Tareas de forma paralela [14]

A estas ventajas que presentan este método basado en la plataforma de software unificado Vitis, hay que añadir que el ingeniero desarrolla la aplicación en un único entorno. A partir de ahí, se estudian los cuellos de botella en la aplicación y con directrices dadas por la metodología, se decide qué parte del sistema se implementa en hardware y qué partes en software.

Normalmente, y en este proyecto, la aceleración por hardware comienza haciendo un diseño de la plataforma hardware en Vivado desde cero, continúa en la creación de un sistema operativo Linux (imagen Linux) haciendo uso de las herramientas Petalinux y termina con la creación de aplicaciones software en Vitis.

Otro punto de partida para realizar la aceleración hardware es partir de una plataforma base y una imagen Linux común, disponibles por el fabricante de FPGA, Xilinx, en [15]. En la Figura 6 se muestran los dos flujos de trabajo.

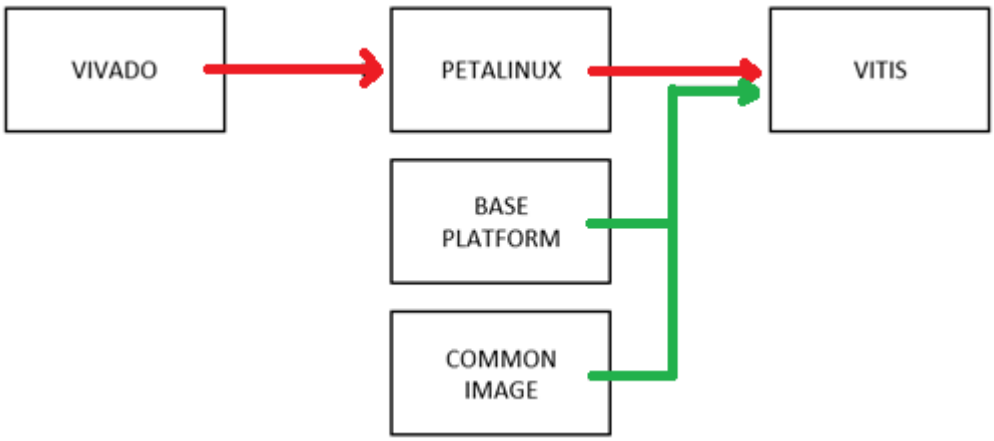


Figura 6: Flujos de trabajo

4.2. Generación del archivo XSA (XelaSoft Archive) en Vivado

Primero, hay que realizar el diseño del diagrama de bloques del sistema. En este paso se optará por una configuración de la FPGA ligada a las necesidades de nuestro proyecto pudiendo incluir diferentes módulos como temporizadores, contadores, indicadores led, señales, botones, definir puertos y conexiones entre los bloques o interfaces externas, etc. El diseño hardware que se puede observar en la Figura 7 cuenta con:

Tabla 1: Módulos del diagrama de bloques de Vivado

ELEMENTOS	UNIDADES
Procesador Zynq UltraScale+ MPSoC	1
Asistente de reloj	1
Reset del procesador de sistema	7
IP de verificación AXI[1]	2
Interconexión IP AXI	4
Registro AXI	1
Controlador de interrupción AXI	1

[1] Buses de microcontrolador [25].

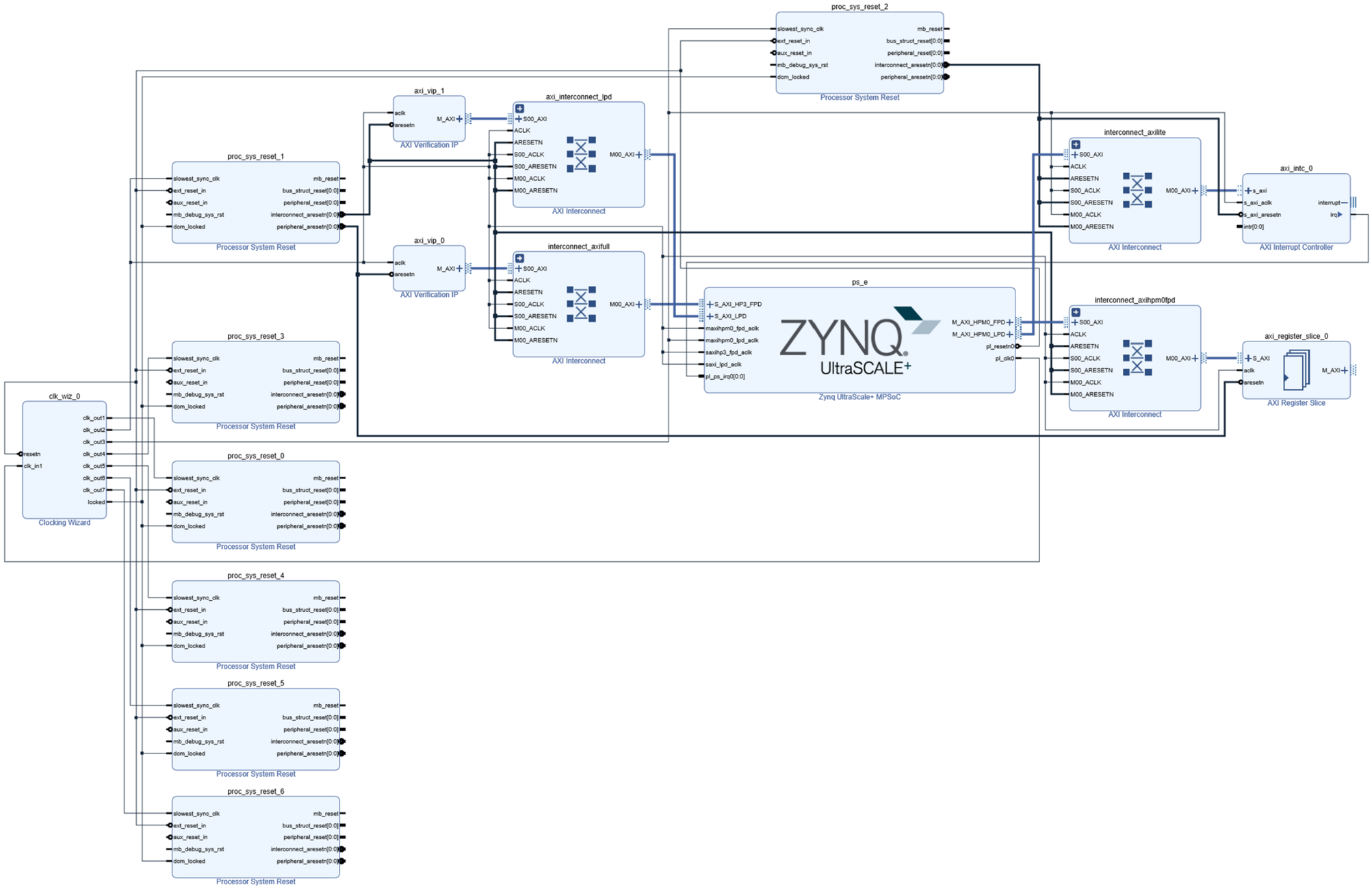


Figura 7: Diseño Hardware en Vivado

Cada uno de los módulos empleados en el diseño tienen una funcionalidad específica. El procesador Zynq UltraScale+ MPSoC corresponde con el procesador de la placa ZCU102 que se está programando. Luego se encuentran los relojes, en este caso unificados en un asistente de reloj donde se puede indicar la frecuencia a la que trabajan, además es necesario que cada uno de ellos tenga una señal de reset asociada sincronizada con ese reloj. De la misma forma, cada plataforma debe tener asignado un único reloj por defecto [16]. En cuanto a las interfaces AXI, se dividen en control de kernels y memoria, y en este diseño se definen módulos de:

- Verificación de IP. Útil para permitir al usuario verificar la depuración en un entorno virtual, así como crear estímulos para comprobar el comportamiento esperado [17].

- Interconexión. Conecta uno o más dispositivos maestros a uno o más dispositivos esclavos, ambos mapeados en memoria AXI [18].

- Registro. Misma función que la interconexión, pero conecta a través de un conjunto de registros de canalización con el objetivo de romper una ruta de temporización [19].

- Controlador de interrupción. Concentra múltiples entradas de interrupción de los periféricos en una única salida de interrupción para el procesador del sistema [20].

Una vez terminado el diseño de la plataforma base, se genera el bitstream que realizará la síntesis e implementación, así como programar y depurar el diagrama de bloques diseñado. A continuación, se exporta el diseño hardware que generará un archivo con extensión ".xsa", como se puede observar en la Figura 8. Este archivo XSA es un

formato de compresión especial diseñado para poder distribuir imágenes de disco en bruto (archivos DSK) a través de Internet y contiene toda la información que se ha incluido en el diagrama de bloques. Es crucial para la continuación del proyecto ya que servirá para crear el proyecto Petalinux y diseñar las aplicaciones en Vitis a implementar asociadas al diseño realizado y que, posteriormente, se cargarán a la placa.

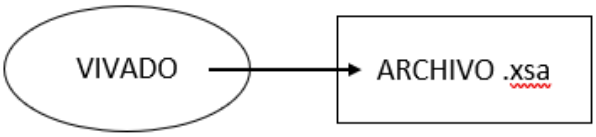


Figura 8: Resultado de Vivado

4.3. Generación de la imagen Linux (Petalinux)

Petalinux es un conjunto de herramientas que nos permiten crear un sistema operativo Linux asociado al diseño implementado en Vivado, este sistema operativo se cargará en la FPGA. Para este paso se ha hecho uso de tutoriales disponibles de Xilinx [21]. Se podrá acceder a ellos para una explicación más detallada de los pasos a seguir para la creación del proyecto.

Una vez descargadas las herramientas Petalinux es posible emplearlas para crear el proyecto en un terminal del ordenador. Este proceso de creación consta principalmente de dos pasos: importación tanto del diseño hardware (XSA) como del paquete de soporte de la placa (BSP) y selección de una serie de configuraciones del proyecto necesarias para el funcionamiento deseado.

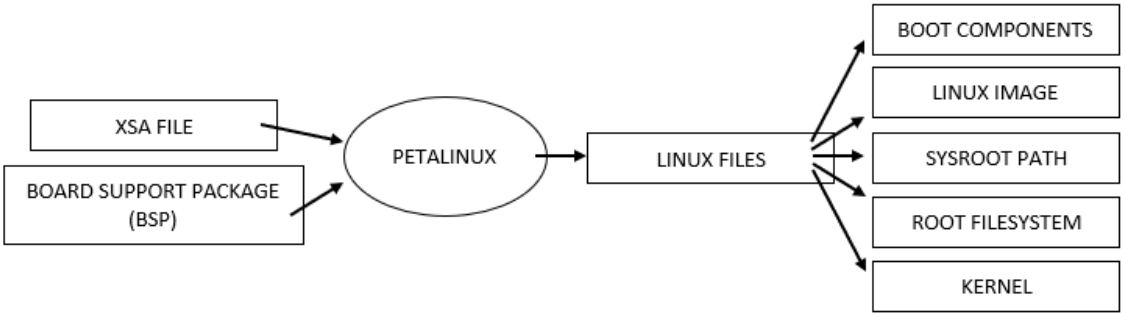


Figura 9: Diagrama de bloques del Petalinux

Como se puede observar en la Figura 9, este proceso genera una serie de archivos que nos servirán para la creación de las aplicaciones necesarias para el funcionamiento del proyecto en Vitis, que posteriormente se cargará en una tarjeta SD y se introducirá en la ZCU102 junto con la aplicación creada.

Para esto, se utiliza el comando que creará el proyecto en el directorio "zcu102_petalinux":

petalinux -create -t project –template zynqMP -n zcu102_petalinux

A continuación, lo que se debe hacer es importar la descripción de la plataforma, es decir, el archivo XSA creado anteriormente en el Vivado. Para ello, se emplea el siguiente comando de las herramientas Petalinux indicando la ruta del archivo XSA:

petalinux-config –get-hw-description=*plataform.xsa*

También, al ejecutar el comando config, aparece una ventana (Figura 10) donde se especifican todas las configuraciones necesarias para el proyecto como seleccionar si se precisan unos parámetros de arranque específicos.

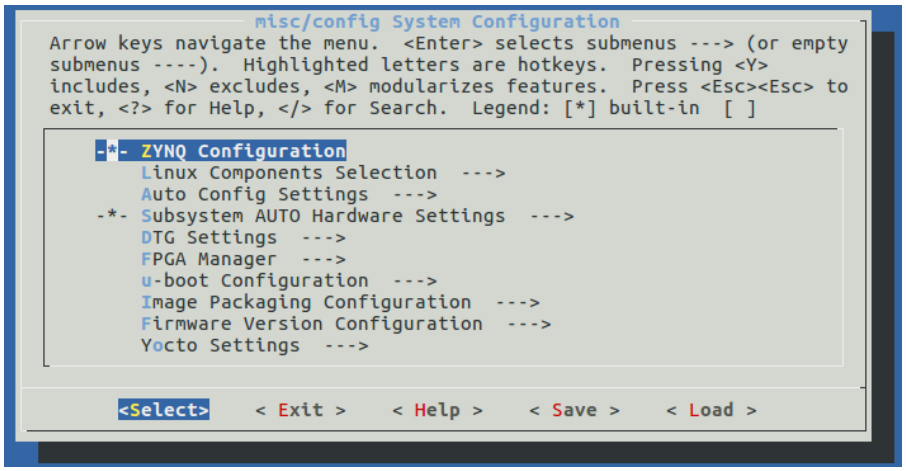


Figura 10: Ventana configuración del proyecto Petalinux

Una vez guardado los cambios realizados, comenzará el proceso de configuración del proyecto. Cuando termina este proceso, se continúa con la realización de varias modificaciones en diferentes ficheros del proyecto referentes a una configuración personalizada del "device tree"[2]. El siguiente paso consiste en las configuraciones del kernel (Figura 11).

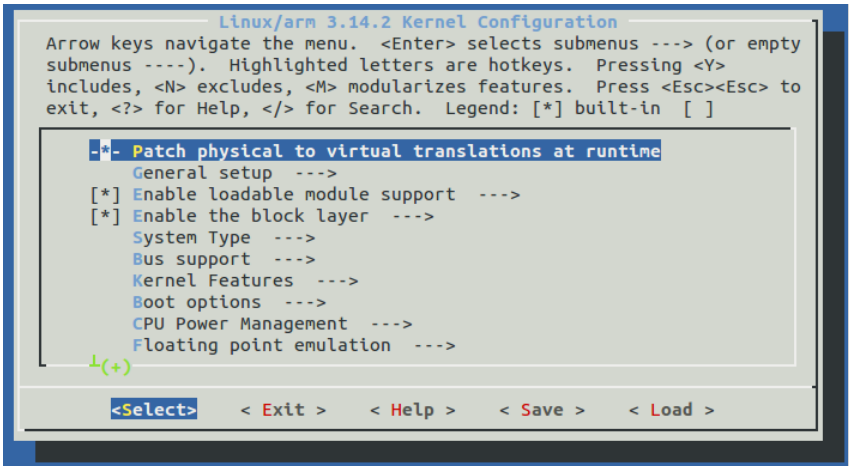


Figura 11: Ventana configuración del kernel del proyecto Petalinux

[2] Es una estructura de datos que describe los componentes de hardware para que el núcleo del sistema operativo pueda utilizar y gestionar dichos componentes [24].

Para ello, se ejecuta el comando que abrirá la ventana correspondiente para ello, donde se pueden modificar los controladores o desactivar la gestión de energía de la CPU:

petalinux-config -c kernel

Luego, se realiza la configuración del sistema de archivos de raíz (root filesystem configurations) con el comando:

petalinux-config -c rootfs

En esta ventana se seleccionan los paquetes de archivos de usuario necesarios para la aceleración del hardware de la plataforma y seleccionar librerías específicas para el diseño y funcionamiento del proyecto (Figura 12).

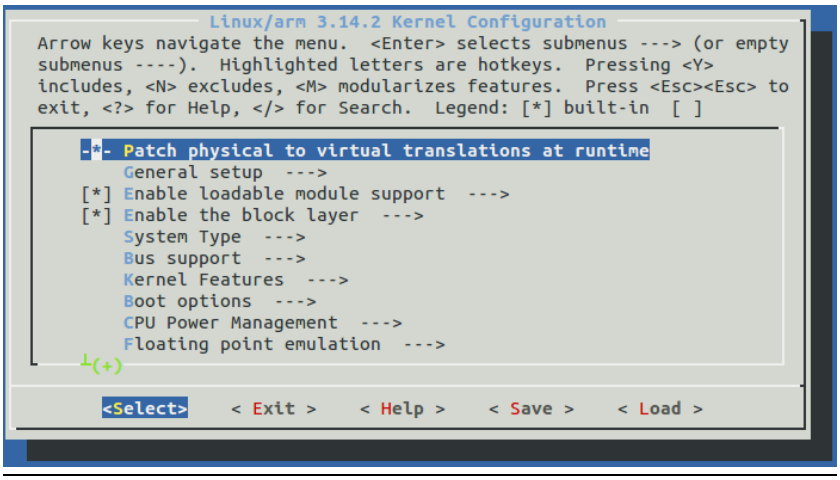


Figura 12: Ventana configuración del "root filesystem" del proyecto Petalinux

Por último, se construye el proyecto con el comando que implementará las configuraciones realizadas anteriormente:

petalinux-build

4.4. Creación de un proyecto en Vitis

Llegados a este punto, crearemos un proyecto en Vitis para desarrollar la aplicación para el funcionamiento requerido a nivel software. El flujo de diseño de este programa consta de 7 pasos, dispuestos en la Figura 13.

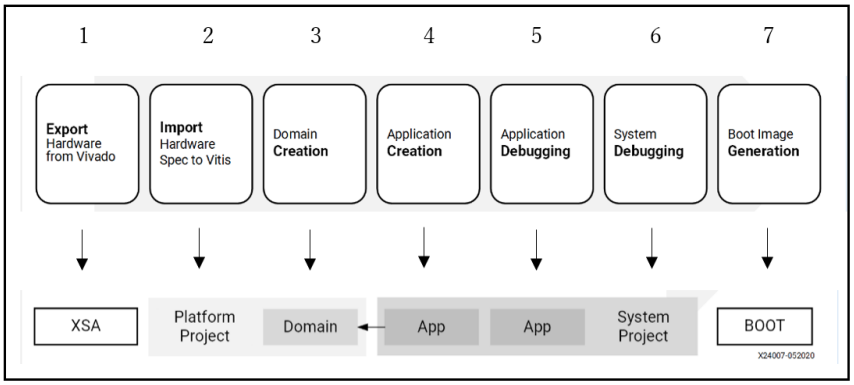


Figura 13: Vitis Design Flow

- *Creación de la aplicación (en C++ & OpenCV)*

Como se ha comentado en los apartados anteriores, la creación del diseño hardware de la placa resulta en un archivo XSA correspondiente al paso número 1 de la Figura 13. Al crear una aplicación en Vitis, esta se asocia a la configuración hardware diseñada en Vivado de la plataforma ZCU102 (archivo XSA), correspondiente al paso 2 de la Figura 13.

Los pasos 3 y 4 consisten en la creación de la "platform project" y "application project". Consisten en la importación del archivo XSA resultante de vivado, creando una plataforma equivalente, para así poder asociar la aplicación software para que se ejecute en la placa.

La depuración de esta aplicación creada está reflejada en los pasos 5 y 6. Vitis proporciona dos herramientas para comprobar rápidamente si su funcionamiento es correcto antes de continuar con los siguientes pasos y no perder más tiempo al cargarlo en la FPGA. Estas dos herramientas son la emulación software y hardware.

- *Software Emulation*

Esta herramienta consiste en la simulación puramente software de la aplicación creada de forma nativa en un procesador x86 o, en nuestro caso, en el entorno de emulación QEMU[3]. Es una emulación bastante sencilla que nos permite comprobar de forma rápida el funcionamiento de la aplicación sin aceleración, aunque posteriormente si se realice en el emulador hardware [22].

- *Hardware Emulation*

En este caso el código se compila en un modelo de comportamiento RTL que se ejecuta en el simulador Vivado o en otros simuladores de terceros compatibles. Este ciclo de construcción y ejecución requiere más tiempo, pero proporciona una vista precisa del ciclo de la lógica del kernel [22].

- *Hardware Execution*

Este último paso consiste en, una vez realizadas las emulaciones software y hardware, la generación de los archivos que posteriormente se cargarán en la tarjeta sd y

[3] QEMU son las siglas de Quick Emulator. Es un emulador de máquina genérico y de código abierto. Xilinx proporciona un modelo QEMU personalizado que imita el sistema de procesamiento basado en Arm presente en los dispositivos Versal ACAP, Zynq® UltraScale+™ MPSoC y Zynq-7000 SoC. El modelo QEMU proporciona la capacidad de ejecutar instrucciones de la CPU en tiempo casi real sin necesidad de hardware real [22].

a su vez a la placa. El kernel se compila en un modelo de hardware (RTL) y luego se implementa en la FPGA, lo que da como resultado un binario que se ejecutará en la FPGA real. De esta forma el compilador Vitis genera el .xclbin para el acelerador de hardware utilizando Vivado Design Suite para ejecutar la síntesis y la implementación. Este proceso está automatizado para generar resultados de alta calidad; sin embargo, los desarrolladores expertos en hardware pueden aprovechar al máximo las herramientas herramientas Vivado y utilizar todas las funciones disponibles para implementar los kernels en su proceso de diseño [22] como se ha mencionado en el apartado 4.2. Vivado Design Suite de este mismo documento.

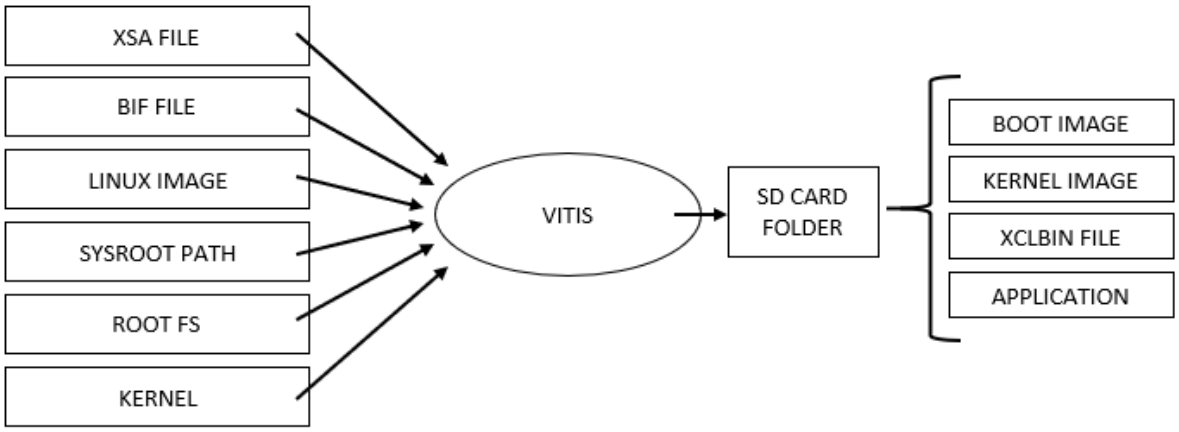


Figura 14: Ejecución Hardware en Vitis

Como resumen de los pasos detallados previamente, en la Figura 15, se puede observar un diagrama de bloques general. En esta ilustración vemos los pasos necesarios de la metodología Vitis para crear y ejecutar una aplicación en la Zynq UltraScale+ MPSoC. Lo primero que se observa es que hay dos bloques principales: “PS Application Compilation” (processing system) y “PL Kernel Compliation and Linking“ (programmable logic). En el primero de ellos se realiza la compilación del sistema

operativo para que se ejecute en el procesador Cortex-A53 utilizando el compilador GNU Arm para crear un archivo ELF. Mientras que, en el segundo se realiza la compilación del kernel para implementarlo en la región PL de la plataforma, y el enlace (en caso de que se realicen varios kernel) para generar el dispositivo ejecutable: XCLBIN.

El siguiente bloque, "System Package", reúne los archivos necesarios para configurar y arrancar el sistema (sysroot, rootfs y kernel Image), y para cargar y ejecutar la aplicación (.xclbin).

Por último, el bloque "Running the Application" es donde se ejecutan las depuraciones software y hardware, así como la ejecución hardware. Además, este flujo consta de un bloque opcional correspondiente a Versal AI Engine Core, muy útil cuando el proyecto precisa de inteligencia artificial.

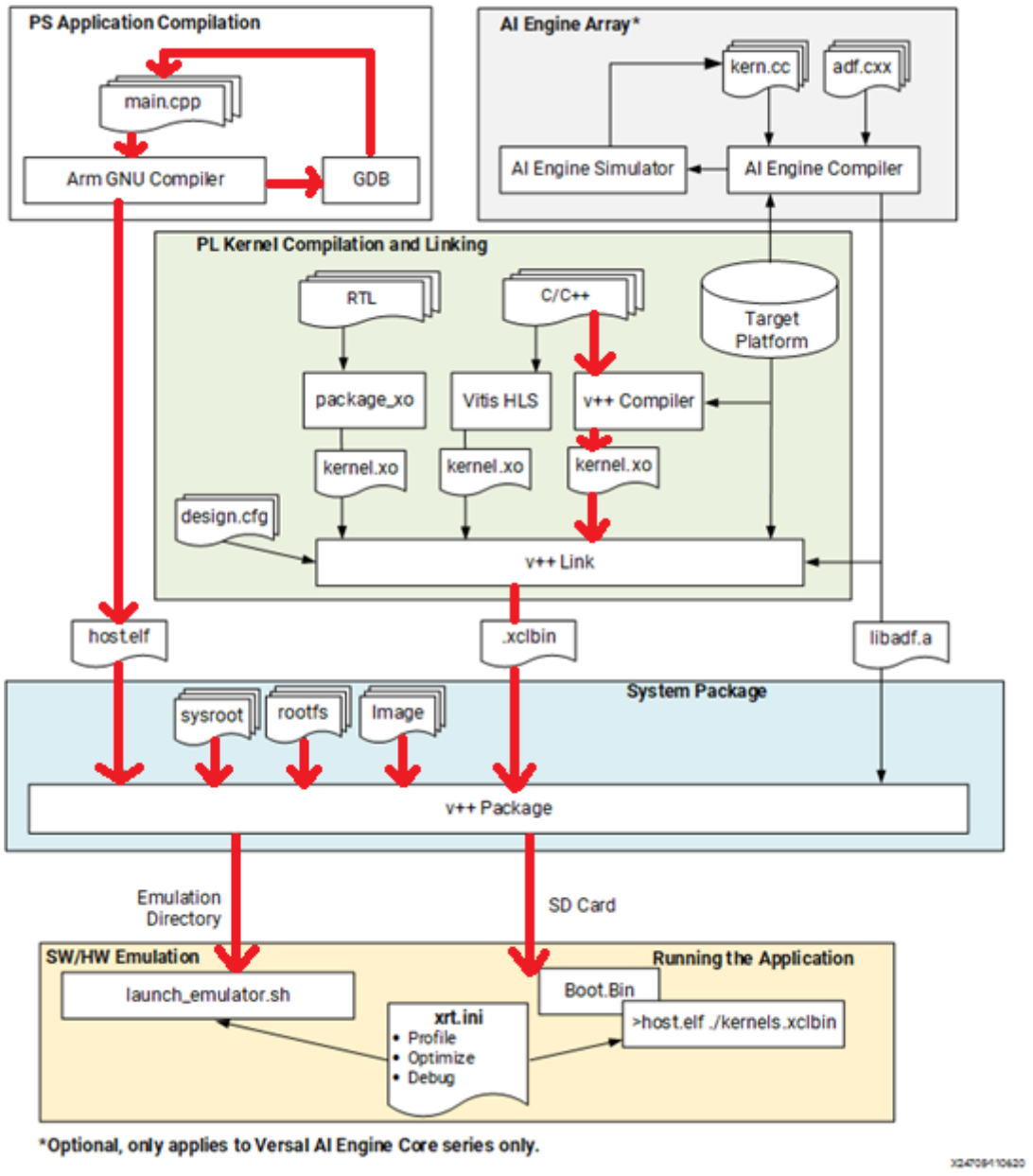


Figura 15: Flujo de desarrollo de aplicaciones para dispositivos Zynq UltraScale + MPSoC [22]

5. Descripción de flatfield

El flatfield (campo plano) es un tipo de corrección de imágenes de una fuente de iluminación uniforme captadas por una cámara CCD astronómica. La imagen de flatfield (imagen FF en el artículo de J.A. Bonet) es una imagen que se divide a la imagen original para corregirla y esta se obtiene mediante el algoritmo de Kuhn, Lin y Loranz para la calibración de la ganancia [3].

Las imágenes solares observadas (d) por la cámara CCD son el resultado de la imagen incidente (s) con unas pérdidas (g) (variaciones en la sensibilidad de píxel a píxel del detector, distorsiones en la trayectoria óptica, ruido o suciedad) [23]. Las figuras 16-18 muestran de forma visual el efecto de este error.

$$d_i(\mathrm{r}) = \mathrm{g}(\mathrm{r}) * s_i(\mathrm{r}) \qquad \text{donde:}$$

- **d** es la imagen observada
- **s** es la imagen incidente
- **g** es la ganancia del píxel (no varía respecto de las imágenes)
- **r** es la coordenada del píxel de la imagen
- **i** es iteración entre las 8 imágenes desplazadas*

*Se supone que no hay variación del sol entre las 8 imágenes.

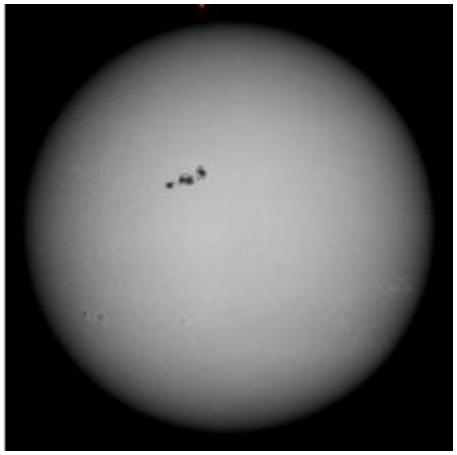


Figura 16: Imagen incidente (s)

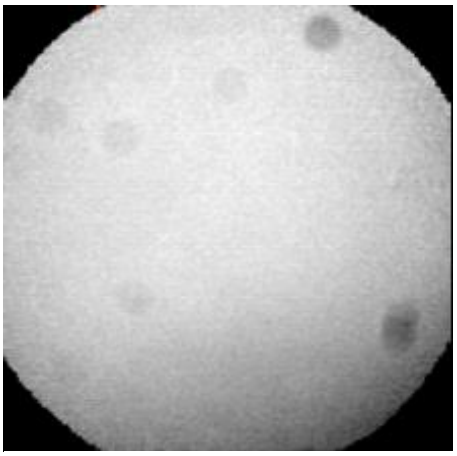


Figura 17: Ganancia (g)

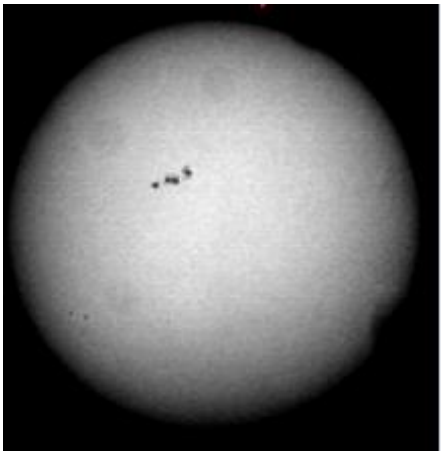
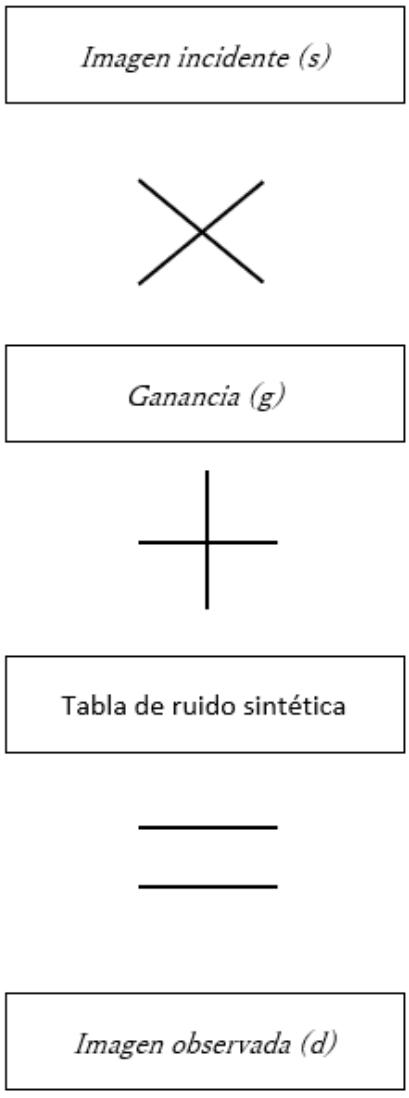


Figura 18: Imagen observada (d)



Normalmente, en laboratorio, la ganancia es un parámetro controlable pero cuando se pretende que trabaje en una estación espacial, el problema cambia. Por esto se quiere obtener el valor de la ganancia para aplicar el flatfielding en el espacio. El propósito general del flatfielding es calibrar la uniformidad de la ganancia píxel a píxel y así generar una imagen con una iluminación espacial uniforme.

Finalmente, como se puede observar en la Figura 19 y Figura 20, se obtiene la imagen FF que se divide a la imagen original para corregirla.

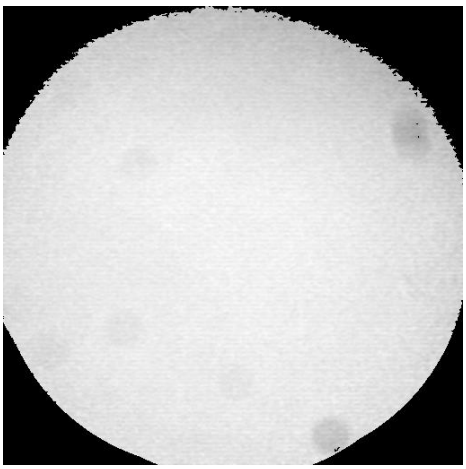


Figura 19: Imagen FF

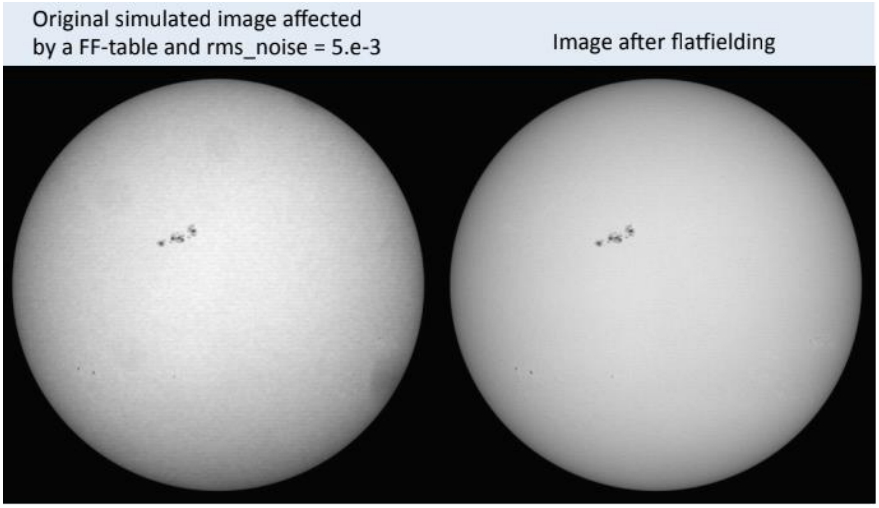


Figura 20: Imagen con ruido vs imagen corregida [23]

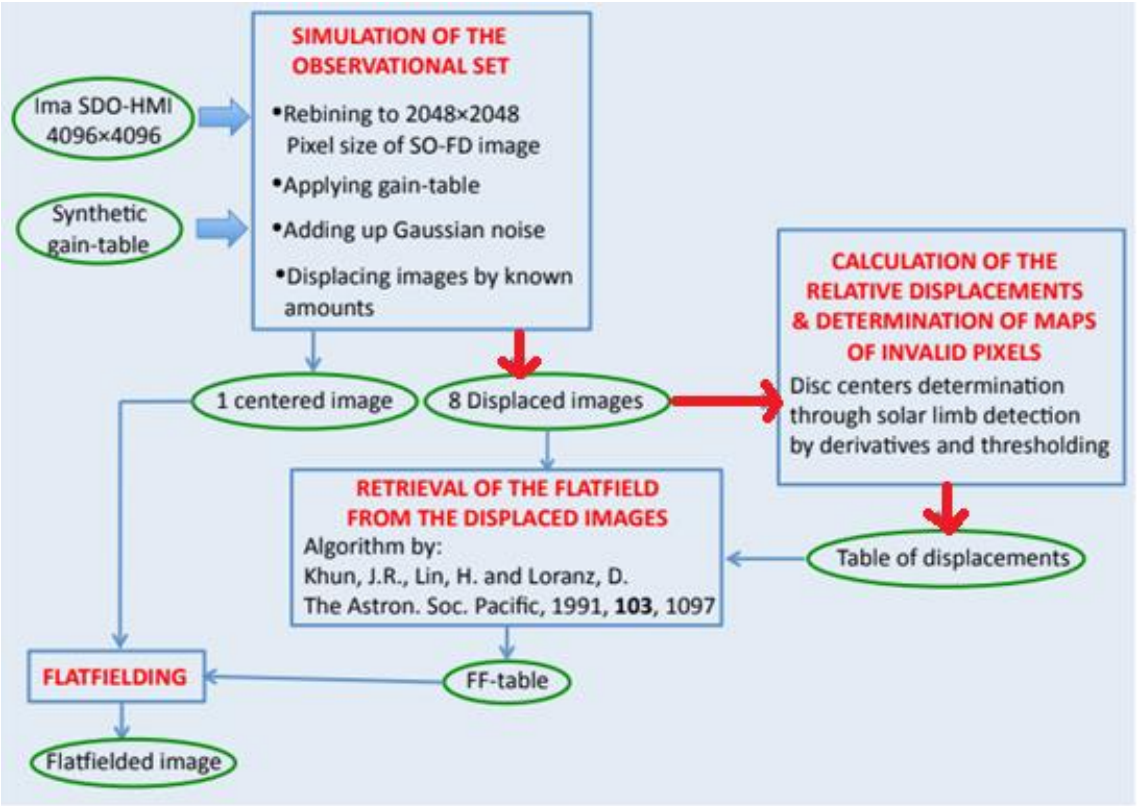


Figura 21: Diagrama de bloques de calibración de flatfield, de Kun, Lin y Loranz [23]

La Figura 21 muestra el procedimiento a seguir para la corrección de flafield, partiendo de la imagen observada. El primer paso busca generar la imagen centrada y 8 desplazadas a las que se le han añadido las pérdidas del paso descrito en la Figura 14. A continuación, a partir de esas 8 imágenes, se procede a realizar el cálculo de los desplazamientos para después poder aplicar el algoritmo descrito por Kun, Lin y Loranz y obtener la tabla con la ganancia píxel a píxel, FF-table. La imagen centrada, previamente obtenida en el primer paso, servirá para realizar el flatfielding, dividiendo la imagen observada por la imagen de flatfield (FF-table).

Como comentamos en el apartado de objetivos de este Trabajo de Fin de Grado, centramos nuestra tarea en acelerar parte del proceso de obtención de desplazamientos.

Este algoritmo tiene un elevado coste computacional, por lo que proponemos implementar un acelerador Hardware aplicando la función Resize, de la Vitis Vision Library, para reducir la imagen original. A continuación, sobre esta imagen aplicamos la transformada de Hough para la detección de los centros de los discos de la imagen. Estos centros se escalan a su tamaño original y así estimamos la posición real de los mismos.

6. Resultados y conclusiones

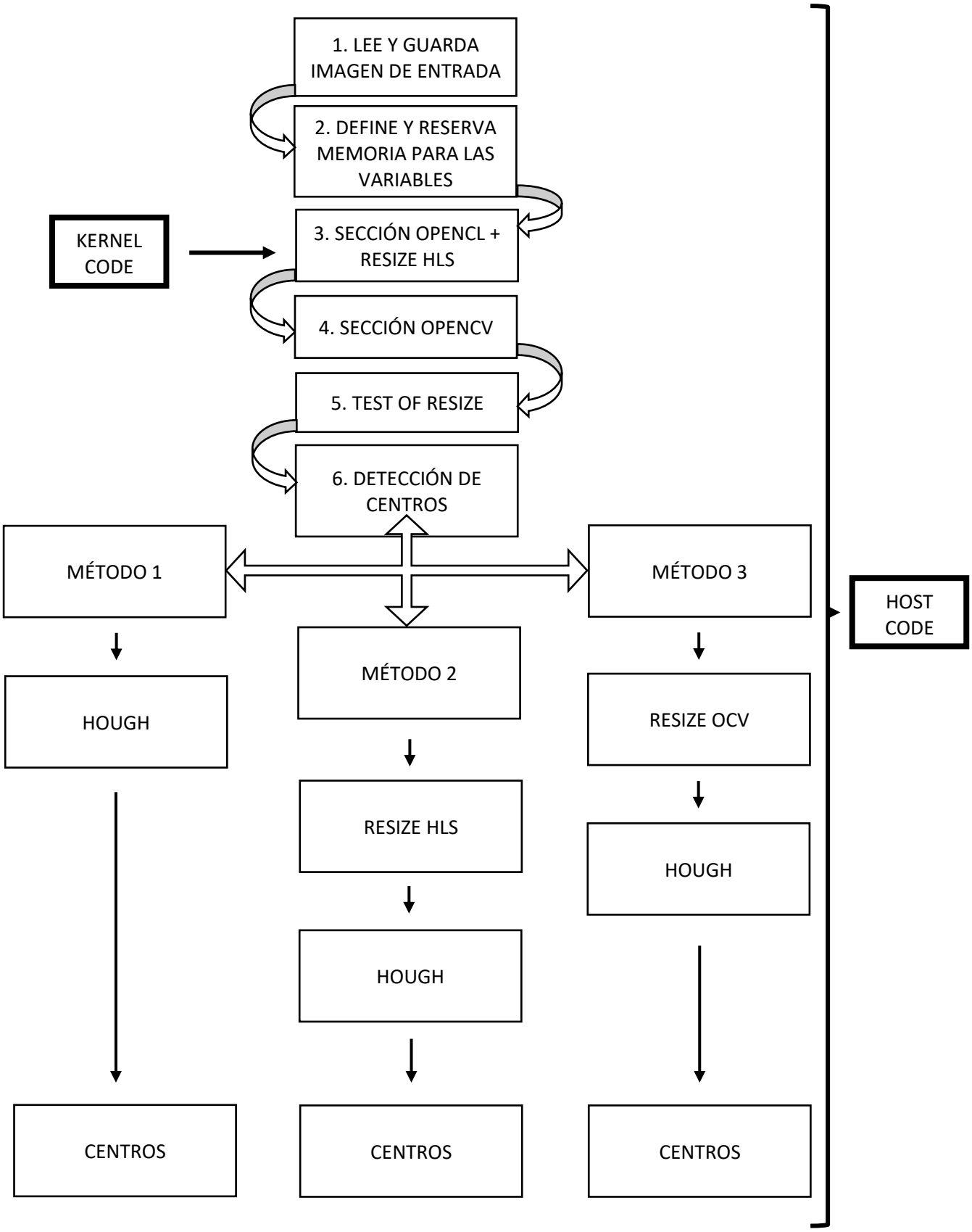
6.1. Algoritmo

Bajo esta metodología marcada por Vitis y como comentamos anteriormente, todo proceso de aceleración hadrware pasa por la creación de un algoritmo o descripción en software del diseño. Este diseño está formado por dos bloques principales: **host code** y **kernel code**.

El código de host tiene dos funciones principales: configurar el entorno de ejecución y ejecución de comandos en la FPGA así como implementar el test bench donde se comparan los resultados obtenidos con el acelerador y los obtenidos con la función software. El código del kernel es una parte del algoritmo que requiere un uso intensivo de la computación y está destinado a ser acelerado en la FPGA.

Son dos bloques diferentes, pero, durante el tiempo de ejecución, el ejecutable del kernel de C / C ++ se llama a través del ejecutable del código de host [22]. Como se mostró previamente en la Figura 15 de este documento, el host code se ejecuta en PS y el kernel code en PL generando los archivos host.elf y <kernel>.xclbin, respectivamente.

Con el fin de poder realizar una comparación de los resultados obtenidos hemos realizado el cálculo de centros por tres métodos. El código fuente diseñado se puede encontrar en el apartado de *Anexos*: "Anexo I: Código del host en C++ desarrollado en Vitis" de este documento y se estructura de la siguiente forma:


1. LEE Y GUARDA IMAGEN DE ENTRADA
2. DEFINE Y RESERVA MEMORIA PARA LAS VARIABLES
KERNEL CODE
3. SECCIÓN OPENCL + RESIZE HLS
4. SECCIÓN OPENCV
5. TEST OF RESIZE
6. DETECCIÓN DE CENTROS
MÉTODO 1
MÉTODO 2
MÉTODO 3
HOST CODE
HOUGH
RESIZE HLS
RESIZE OCV
HOUGH
HOUGH
CENTROS
CENTROS
CENTROS

1. LEE Y GUARDA IMAGEN DE ENTRADA

El objetivo de esta etapa es definir la imagen de entrada, así como sus dimensiones, ya que estas variables serán empleadas más adelante. De la misma forma, se almacena la información del redimensionamiento que se quiere realizar a la imagen (ancho y alto de la imagen redimensionada). Estos parámetros son proporcionados por el usuario como argumentos de entrada al ejecutar la aplicación.

2. DEFINE Y RESERVA MEMORIA PARA LAS VARIABLES

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 + RESIZE HLS

En esta sección se realiza el reconocimiento del dispositivo y el código de host configura la plataforma OpenCL con la FPGA de procesamiento de datos requeridos que, en nuestro caso, al emplear las librerías Vitis Vision, los datos son una imagen. Una vez detectado el dispositivo, se ejecuta el kernel y, por tanto, el archivo XCLBIN. Al ejecutar el kernel se obtendrá el resultado de la aceleración del redimensionamiento de la imagen realizado en hardware (HLS).

4. SECCIÓN OPENCV

En esta sección se realiza ejecuta la función Resize, pero esta vez de OpenCV, empleando la interpolación Bilineal. Este tipo de interpolación se realiza para matrices regulares de dos dimensiones, realizando la interpolación en una dirección y después en otra.

5. TEST OF RESIZE

Realización de la diferencia absoluta entre el redimensionamiento de la imagen realizado por HLS y por OCV (OpenCV).

6. DETECIÓN DE CENTROS

MÉTODO 1

Consiste en realizar la detección de centros de la imagen sin redimensionar de 1024x1024 píxeles. Para ello aplicamos HoughCircles de las librerías OpenCv, que se encargará de generar un vector de vectores tipo Vec3f (con tres valores tipo float) con las coordenadas del centro de la circunferencia detectada y el radio.

MÉTODO 2

Consiste en utilzar la función Resize de OpenCV (no acelerado) y redimensionar la imagen a 512x512 píxeles. Luego se aplica Hough de la misma forma que en el apartado anterior.

MÉTODO 3

Consiste en realizar el mismo proceso que en el apartado anterior, pero empleando el redimensionamiento de la imagen obtenido de HLS (acelerado).

En cada uno de los tres métodos, se genera una imagen resultante con una representación gráfica de los centros detectados y el borde del disco solar, como se puede observar en la Figura 23 de la siguiente página.

6.2. Resultados

Una vez tenemos el algoritmo definitivo y se han realizado las emulaciones software y hardware, se ejecuta el hardware y se carga la aplicación y la imagen Linux (todos los archivos del directorio sd_card) en la ZCU102 mediante la tarjeta SD. Esta aplicación muestra en el terminal los centros detectados con el siguiente formato: [coordenada x, coordenada y, radio] como se puede observar en la Figura 22. Además, genera los archivos indicados en el código y se guardan en la tarjeta SD.

```
INFO OPTION 1: Begin of center detection.
INFO OPTION 1: End of center detection.
RESULTS OPTION 1: Center and radius results for 1024x1024 image :
[510.5, 508.5, 481.8]
INFO OPTION 2: Begin of center detection.
INFO OPTION 2: End of center detection.
RESULTS OPTION 2: Center and radius results for 512x512 image :
[255.5, 254.5, 241.3]
INFO OPTION 3: Begin of center detection.
INFO OPTION 3: End of center detection.
RESULTS OPTION 3: Center and radius results for 512x512 image :
[255.5, 254.5, 241.3]
```

Figura 22: Centros detectados por los 3 métodos

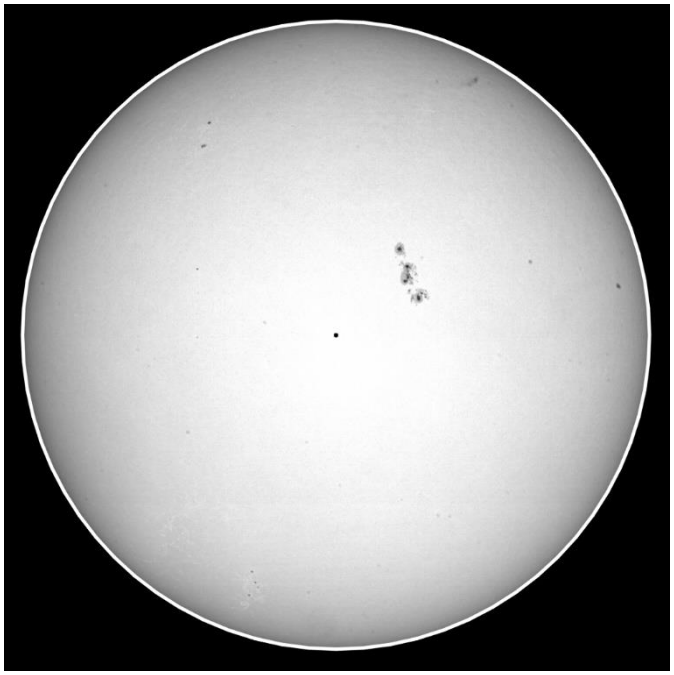


Figura 23: Representación gráfica de la detección

Ahora, con los resultados obtenidos de la detección de centros, hay que calcular el error producido al realizar el redimensionamiento de la imagen tomando como valor correcto el resultante del método 1. Para realizar el cálculo se tiene que comparar el resultado del primer método con el segundo y el tercero, respectivamente. Para ello, se emplea la siguiente fórmula:

$$error(\%) = \frac{valor\ reducido - valor\ real}{valor\ real} \times 100 \qquad donde:$$

- **valor reducido** es el valor x, y o r de la imagen redimensionada
- **valor real** es el valor x, y o r de la imagen sin redimensionar

Los resultados de este cálculo se pueden observar en la *Tabla 2*.

Tabla 2: Centros obtenidos por los diferentes métodos

	CENTRO Y RADIO OBTENIDOS [x,y,r]	CENTRO Y RADIO ESCALADOS [x,y,r]	ERROR RELATIVOS AL TAMAÑO DE LA IMAGEN RESPECTO AL MÉTODO 1
MÉTODO 1 Imagen 1024x1024	[510.5, 508,5, 481,8]		-
MÉTODO 2 Imagen 512x512	[255.5, 254,5, 241,3]	[511, 509, 482,6]	Coordenadas: 0,098% Radio: 0,166%
MÉTODO 3 Imagen 512x512	[255.5, 254,5, 241,3]	[511, 509, 482,6]	Coordenadas: 0,098% Radio: 0,166%

6.3. Conclusiones y líneas futuras

Para finalizar con este Trabajo de Fin de Grado abordaremos las conclusiones obtenidas del proyecto realizado.

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 las librerías Vitis Vision y OpenCV. Este proyecto nos deja con una percepción clara de la versatilidad del diseño en este entorno y las ventajas que ofrece la metodología de Xilinx, basada en Vitis, a la hora de diseñar y crear aplicaciones, así como la opción de implementarlas en software o hardware.

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.

Hemos diseñado un algoritmo que acelera una etapa del proceso de la calibración de imágenes solares (flatfield). 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. Por tanto, así es posible paralelizar los procesos que de otra forma se realizarían secuencialmente y ralentizarían considerablemente la ejecución de la aplicación. Para ello, hemos incluido en el algoritmo la función acelerada Resize, de la librería Vitis Vision, para reducir la imagen y reducir, en consecuencia, el tiempo de cálculo del centro del disco solar aplicando la transformada Hough.

A partir de los resultados obtenidos en este experimento, podemos concluir en que los métodos 2 y 3 son equivalentes en resultado, pero no en temporización, principalmente debido a que el segundo se ejecuta en hardware. Comparando los centros y radios resultantes con los del primer método, como se buscaba, se puede considerar que son correctos debido a que el error es del 0,098% para las coordenadas y del 0,166% para el radio, ambos relativos al tamaño de la imagen.

A la vista de los resultados que se han obtenido en este trabajo y como trabajos futuros, se plantea usar esta nueva metodología de diseño, así como combinar las diferentes herramientas y librerías de procesamiento de imágenes, con el objetivo de reducir el esfuerzo de diseño, implementación y tiempos de ejecución del algoritmo de calibración de flatfield al completo.

Referencias bibliográficas

[1] The European Space Agency, «Solar Orbiter,» [En línea]. Available: https://www.esa.int/Science_Exploration/Space_Science/Solar_Orbiter.

[2] Instituto de Astrofísica de Canarias, «SOPHI (Solar Orbiter),» [En línea]. Available: https://www.iac.es/es/proyectos/sophi-solar-orbiter.

[3] J.R. Kuhn, H. Lin and D. Loranz, «"Gain calibrating nonuniform image-array data using only the image data",» *Publications of the Astronomical Society of the Pacific,* vol. 103, nº 668, pp. 1097-1108, octubre, 1991.

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

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

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

[7] G. I. Viera-Maza, *"Procesamiento de Imágenes usando OpenCV aplicado en Raspberry Pi para la clasificación del cacao",* Trabajo de Fin de Grado, Univ. de Piura, Piura, Perú, 2017. [En línea]. Disponible en: https://pirhua.udep.edu.pe/bitstream/handle/11042/2916/IME_218.pdf.

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

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

[10] Digi-Key, «Fundamentos de los FPGA - Parte 4: Introducción de los FPGA de Xilinx,» [En línea]. Available: https://www.digikey.es/es/articles/fundamentals-of-fpgas-part-4-getting-started-with-xilinx-fpgas.

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

[12] J. L. V. Gutiérrez, *"Aceleradores de código basados en FPGA: qué son y cómo hacerlos",* Trabajo de Fin de Grado, Univ. de Cantabria, Cantabria, España, 2016. [En línea]. Disponible en: https://repositorio.unican.es/xmlui/bitstream/handle/10902/9649/Vazquez%20Gutierrez%20Jose%20luis.pdf?sequence=1&isAllowed=y.

[13] E. G. Fernández, *"Aceleración hardware con FPGA de algoritmo para estegoanálisis",* Trabajo de Fin de Grado, Univ. Carlos III, Leganés, España. [En línea]. Disponible en: https://e-archivo.uc3m.es/bitstream/handle/10016/16578/PFC_Eric_Gutierrez_Fernandez.pdf?sequence=1&isAllowed=y.

[14] Xilinx, «Task-level parallelism and pipelining in HLS,» 2019. [En línea].

Available: https://developer.xilinx.com/en/articles/task-level-parallelism-and-pipelining-in-hls.html.

[15] Xilinx, «Vitis Embedded Base Platforms,» 2020. [En línea]. Available: https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-platforms/2020-2.html.

[16] Xilinx, «Demystify Vitis Embedded Acceleration Platform Creation,» Diciembre 2020. [En línea]. Available: https://china.xilinx.com/publications/presentations/D2-05.pdf.

[17] Xilinx, «How to use AXI Verification IP to Verify and Debug your Design using Simulation,» [En línea]. Available: https://www.xilinx.com/video/hardware/how-to-use-axi-verification-ip-to-verify-debug-design-using-simulation.html.

[18] Xilinx, «AXI Interconnect,» [En línea]. Available: https://www.xilinx.com/products/intellectual-property/axi_interconnect.html.

[19] Xilinx, «AXI Registrer Slice,» 2020. [En línea]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_register_slice/v2_1/pg373-axi-register-slice.pdf.

[20] Xilinx, «AXI Interrupt Controller,» [En línea]. Available: https://www.xilinx.com/products/intellectual-property/axi_intc.html.

[21] Xilinx, «Petalinux Tools Documentation: Reference Guide,» Noviembre 2020. [En línea]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1144-petalinux-tools-reference-guide.pdf.

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

[23] J. A. Bonet, «"Numerical experiment to calibrate the Flat-Field in the full-disc solar images from SO/PHI",» 2013. [En línea].

[24] Wikipedia, «Explicación "device-tree",» [En línea]. Available: https://en.wikipedia.org/wiki/Device_tree.

[25] Xilinx, «Vivado Design Suite: AXI Reference Guide,» 2017. [En línea]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

Anexos

Anexo I: Código del host en C++ desarrollado en Vitis

```
#include "common/xf_headers.hpp"
#include "xf_resize_config.h"
#include "xcl2.hpp"
#include <vector>

int main(int argc, char** argv) {
      //In case the input arguments are not the 4 required, the following message is printed
    if (argc != 4) {
        fprintf(stderr, "Usage: <executable> <input image> <output image height> <output image width>\n");
        return -1;
    }

    //Creation of the type Mat variables
    cv::Mat img, result_hls, result_ocv, error;

      //Reading and saving the input image
      std::cout << "INFO: Reading input image." << std::endl;
#if GRAY //GRAY is a configuration specified in the "xf_resize_config.h" file
    img = cv::imread(argv[1], 0);
#else
    img = cv::imread(argv[1], 1);
#endif

    if (!img.data) {
        return -1;
    }

      //Defining the height and width of images
        //Input image values
    int in_width = img.cols;
    int in_height = img.rows;
```

```
      //Values of the resized image from the values entered as input arguments
    int out_height = atoi(argv[2]);
    int out_width = atoi(argv[3]);

      //Creation of the resulting images of the OpenCV and HLS section
#if GRAY  //GRAY is a configuration specified in the "xf_resize_config.h" file
    result_hls.create(cv::Size(out_width, out_height), CV_8UC1);
    result_ocv.create(cv::Size(out_width, out_height), CV_8UC1);
    error.create(cv::Size(out_width, out_height), CV_8UC1);
#else
    result_hls.create(cv::Size(out_width, out_height), CV_8UC3);
    result_ocv.create(cv::Size(out_width, out_height), CV_8UC3);
    error.create(cv::Size(out_width, out_height), CV_8UC3);
#endif

//Beginning of the OpenCL section:
#if GRAY //GRAY is a configuration specified in the "xf_resize_config.h" file
    size_t image_in_size_bytes = in_height * in_width * 1 * sizeof(unsigned char);
    size_t image_out_size_bytes = out_height * out_width * 1 * sizeof(unsigned char);
#else
    size_t image_in_size_bytes = in_height * in_width * 3 * sizeof(unsigned char);
    size_t image_out_size_bytes = out_height * out_width * 3 * sizeof(unsigned char);
#endif

    cl_int err;
    std::cout << "INFO: Running OpenCL section." << std::endl;

    // Get the device:
    std::vector<cl::Device> devices = xcl::get_xil_devices();
    cl::Device device = devices[0];

    // Context, command queue and device name:
    OCL_CHECK(err, cl::Context context(device, NULL, NULL, NULL, &err));
```

```
OCL_CHECK(err, cl::CommandQueue q(context, device, CL_QUEUE_PROFILING_ENABLE, &err));
OCL_CHECK(err, std::string device_name = device.getInfo<CL_DEVICE_NAME>(&err));

std::cout << "INFO: Device found - " << device_name << std::endl;

// Load binary:
std::string binaryFile = xcl::find_binary_file(device_name, "krnl_resize");
cl::Program::Binaries bins = xcl::import_binary_file(binaryFile);
devices.resize(1);
OCL_CHECK(err, cl::Program program(context, devices, bins, NULL, &err));

// Create a kernel:
OCL_CHECK(err, cl::Kernel krnl(program, "resize_accel", &err));

// Allocate the buffers:
std::vector<cl::Memory> inBufVec, outBufVec;
OCL_CHECK(err, cl::Buffer imageToDevice(context, CL_MEM_READ_ONLY, image_in_size_bytes, NULL, &err));
OCL_CHECK(err, cl::Buffer imageFromDevice(context, CL_MEM_WRITE_ONLY, image_out_size_bytes, NULL, &err));

// Set the kernel arguments
OCL_CHECK(err, err = krnl.setArg(0, imageToDevice));
OCL_CHECK(err, err = krnl.setArg(1, imageFromDevice));
OCL_CHECK(err, err = krnl.setArg(2, in_height));
OCL_CHECK(err, err = krnl.setArg(3, in_width));
OCL_CHECK(err, err = krnl.setArg(4, out_height));
OCL_CHECK(err, err = krnl.setArg(5, out_width));

/* Copy input vectors to memory */
OCL_CHECK(err, q.enqueueWriteBuffer(imageToDevice,       // buffer on the FPGA
                                    CL_TRUE,             // blocking call
                                    0,                   // buffer offset in bytes
                                    image_in_size_bytes, // Size in bytes
                                    img.data));          // Pointer to the data to copy
```

```
    // Profiling Objects
    cl_ulong start = 0;
    cl_ulong end = 0;
    double diff_prof = 0.0f;
    cl::Event event_sp;

    // Execute the kernel:
    OCL_CHECK(err, err = q.enqueueTask(krnl, NULL, &event_sp));

    clWaitForEvents(1, (const cl_event*)&event_sp);

    event_sp.getProfilingInfo(CL_PROFILING_COMMAND_START, &start);
    event_sp.getProfilingInfo(CL_PROFILING_COMMAND_END, &end);
    diff_prof = end - start;
    std::cout << (diff_prof / 1000000) << "ms" << std::endl;

    // Copying Device result data to Host memory
    OCL_CHECK(err, q.enqueueReadBuffer(imageFromDevice, // This buffers data will be read
                                       CL_TRUE,         // blocking call
                                       0,               // offset
                                       image_out_size_bytes,
                                       result_hls.data)); // Data will be stored here

    q.finish();
////////////////////////////////////// end of CL
//////////////////////////////////////////

std::cout << "INFO: Start resize function in OpenCV." << std::endl;

#if INTERPOLATION == 0 //INTERPOLATION is a configuration specified in the "xf_resize_config.h" file
    cv::resize(img, result_ocv, cv::Size(out_width, out_height), 0, 0, CV_INTER_NN);
#endif
```

```
#if INTERPOLATION == 1
    cv::resize(img, result_ocv, cv::Size(out_width, out_height), 0, 0, CV_INTER_LINEAR);
#endif
#if INTERPOLATION == 2
    cv::resize(img, result_ocv, cv::Size(out_width, out_height), 0, 0, CV_INTER_AREA);
#endif

      //Calculation of the difference between the results by hls and ocv
    cv::absdiff(result_hls, result_ocv, error);
    float err_per;
    xf::cv::analyzeDiff(error, 5, err_per);

    if (err_per > 1.0f) {
        fprintf(stderr, "ERROR: Test Failed.\n ");
        return -1;
    }

    std::cout << "Test Passed" << std::endl;

    //Saving the output resized image from the two methods
    std::cout << "INFO: Writing output image." << std::endl;
    cv::imwrite("img_out_HLS.png", result_hls);
    cv::imwrite("img_out_OCV.png", result_ocv);

    //Define variables of the minimum centre-to-centre distance of each case for Hough. As required by the function
it is float type
    float = mindist_op1;
      float = mindist_op2;
      float = mindist_op3;

//CASE 1: USING THE ORIGINAL (NOT RESIZED) IMAGE AND APPLYING HOUGH

    std::cout << "INFO OPTION 1: Begin of center detection." << std::endl;
```

```
    //Definition of a vector of vectors type Vec3f (three float type values)
    std::vector<cv::Vec3f> circles_op1(2, {0.0,0.0,0.0}); //It is important to assign a memory address, otherwise
the software emulation will fail

    //Define the value of the minimum centre-to-centre distance for this case
    mindist_op1 = float(in_width % 2);

      //Application of hough function for circles detection. Store in "circles" the values of the detected centres
and their radius
    cv::HoughCircles(img, circles_op1, cv::HOUGH_GRADIENT, 1.0,
                 mindist_op1,  // change this value to detect circles with different distances to each other
                 100.0, 30.0,
                         ((in_height % 2) * 0.9), (in_height % 2); // change the last two parameters (min_radius &
max_radius) to detect larger circles
    std::cout << "INFO OPTION 1: End of center detection." << std::endl;

      //Drawing the results of the detection on the screen
    std::cout << "RESULTS OPTION 1: Center and radius results for " << in_height << "x" << in_width << " image:" <<
std::endl;
    for (size_t i = 0; i < circles_op1.size(); i++) {
      std::cout << circles_op1[i] << std::endl;
    }

    //Drawing the results of the detection
    for (size_t i = 0; i < circles_op1.size(); i++) {
        cv::Vec3i c = circles_op1[i];
        cv::Point center = cv::Point(c[0], c[1]);
        // circle center
        cv::circle(img, center, 1, cv::Scalar(0,100,100), 3, cv::LINE_AA);
        // circle outline
        int radius = c[2];
        cv::circle(img, center, radius, cv::Scalar(255,0,255), 3, cv::LINE_AA);
```

```
    }
    cv::imwrite("HoughCircles_op1.png", img);

//CASE 2: USING THE RESIZED (HLS) IMAGE AND APPLYING HOUGH

    std::cout << "INFO OPTION 2: Begin of center detection." << std::endl;

    //Definition of a vector of vectors type Vec3f (three float type values)
    std::vector<cv::Vec3f> circles_op2(2, {0.0,0.0,0.0}); //It is important to assign a memory address, otherwise
the software emulation will fail

    //Define the value of the minimum centre-to-centre distance for this case
    mindist_op2 = float(out_width % 2);

      //Application of hough function for circles detection. Store in "circles" the values of the detected centres
and their radius
    cv::HoughCircles(result_hls, circles_op2, cv::HOUGH_GRADIENT, 1.0,
                 mindist_op2,  // change this value to detect circles with different distances to each other
                 100.0, 30.0,
                         ((out_height % 2) * 0.9), (out_height % 2); // change the last two parameters (min_radius &
max_radius) to detect larger circles
    std::cout << "INFO OPTION 2: End of center detection." << std::endl;

      //Drawing the results of the detection on the screen
    std::cout << "RESULTS OPTION 2: Center and radius results for " << out_height << "x" << out_height << " image:"
<< std::endl;
    for (size_t i = 0; i < circles_op2.size(); i++) {
      std::cout << circles_op2[i] << std::endl;
    }

    //Drawing the results of the detection
    for (size_t i = 0; i < circles_op2.size(); i++) {
        cv::Vec3i c = circles_op2[i];
```

```
        cv::Point center = cv::Point(c[0], c[1]);
        // circle center
        cv::circle(result_hls, center, 1, cv::Scalar(0,100,100), 3, cv::LINE_AA);
        // circle outline
        int radius = c[2];
        cv::circle(result_hls, center, radius, cv::Scalar(255,0,255), 3, cv::LINE_AA);
    }
    cv::imwrite("HoughCircles_op2.png", result_hls);

//CASE 3: USING THE RESIZED (OCV) IMAGE AND APPLYING HOUGH

    std::cout << "INFO OPTION 3: Begin of center detection." << std::endl;

    //Definition of a vector of vectors type Vec3f (three float type values)
    std::vector<cv::Vec3f> circles_op3(2, {0.0,0.0,0.0}); //It is important to assign a memory address, otherwise
the software emulation will fail

    //Define the value of the minimum centre-to-centre distance for this case
    mindist_op3 = float(in_width % 2);

      //Application of hough function for circles detection. Store in "circles" the values of the detected centres
and their radius
    cv::HoughCircles(result_ocv, circles_op3, cv::HOUGH_GRADIENT, 1.0,
                 mindist_op3,  // change this value to detect circles with different distances to each other
                 100.0, 30.0,
                         ((out_height % 2) * 0.9), (out_height % 2); // change the last two parameters (min_radius &
max_radius) to detect larger circles
    std::cout << "INFO OPTION 3: End of center detection." << std::endl;

      //Drawing the results of the detection on the screen
    std::cout << "RESULTS OPTION 3: Center and radius results for " << out_height << "x" << out_height << " image:"
<< std::endl;
    for (size_t i = 0; i < circles_op3.size(); i++) {
```

```
      std::cout << circles_op3[i] << std::endl;
    }

    //Drawing the results of the detection
    for (size_t i = 0; i < circles_op3.size(); i++) {
        cv::Vec3i c = circles_op2[i];
        cv::Point center = cv::Point(c[0], c[1]);
        // circle center
        cv::circle(result_ocv, center, 1, cv::Scalar(0,100,100), 3, cv::LINE_AA);
        // circle outline
        int radius = c[2];
        cv::circle(result_ocv, center, radius, cv::Scalar(255,0,255), 3, cv::LINE_AA);
    }
    cv::imwrite("HoughCircles_op3.png", result_ocv);

    return 0;
}
```