ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Máster

VERIFICACIÓN DE UN DECODIFICADOR FEC EN UN SISTEMA HETEROGÉNEO BASADO EN TECNOLOGÍA FPGA

(Verification of a FEC decoder in a heterogeneous system based on FPGA technology)

Para acceder al Título de

Máster Universitario en Ingeniería de Telecomunicación

Autor: Daniel Nicolás Suárez Plata

Septiembre - 2021

MASTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE MASTER

Realizado por: Daniel Nicolás Suárez Plata
Director del TFM: Víctor Manuel Fernández Solórzano
Título: "Verificación de un decodificador FEC en un sistema heterogéneo basado en tecnología FPGA"
Title: "Verification of a FEC decoder in a heterogeneous system based on FPGA technology**"**

Presentado a examen el día:

para acceder al Título de

MASTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

Composición del Tribunal:
Presidente: Villar Bonet, Eugenio
Secretario: Posadas Cobo, Hector
Vocal: Sánchez Espeso, Pablo Pedro

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente	Fdo.: El Secretario
Fdo.: El Vocal	Fdo.: El Director del TFM (sólo si es distinto del Secretario)
Vº Bº del Subdirector	Trabajo Fin de Máster Nº (a asignar por Secretaría)

Tabla de contenido

Índice de figuras

CAPÍTULO 1: INTRODUCCIÓN

1.1 CONTEXTO

La verificación de un decodificador de errores de canal es un proceso complicado debido a la propia naturaleza correctora de estos sistemas, que tienden a ocultar, incluso, los errores de funcionamiento. Un enfoque habitual en la verificación de este tipo de componentes se centra en la supervisión de los resultados obtenidos durante las distintas etapas del proceso de decodificación. De esta forma, mediante la comparación de los valores obtenidos con los valores esperados obtenidos a partir de un modelo de referencia, se puede verificar y garantizar que el decodificador funciona correctamente. Por otra parte, otro planteamiento típico del problema es el de verificar algunos de los parámetros de rendimiento más característicos de estos componentes, como podría ser el caso del ratio de errores de bit (BER) o el ratio de errores de palabras código (CER).

La implementación de cualquiera de estos dos enfoques se puede realizar bien mediante una serie de simulaciones o bien mediante prototipos programados sobre FPGAs.

En el primer caso se tienen una serie de simulaciones sobre un conjunto grande de datos que se utilizan como entradas del decodificador. Dependiendo del nivel de abstracción con el que haya sido descrito el decodificador puede tratarse de simulaciones *software*, cuando ha sido descrito a nivel algorítmico mediante un lenguaje de alto nivel cómo C/C++, o de simulaciones *hardware* cuando ha sido descrito a un nivel RTL mediante lenguajes de descripción *hardware* cómo VHDL o Verilog. El presente trabajo se centrará en el segundo de los casos.

Los esquemas de simulación utilizados típicamente están compuestos por una serie de fases que modelan los distintos componentes que atraviesa un dato con información en un sistema de comunicaciones. Estas fases abarcan desde una codificación que incluya una adecuada generación de números aleatorios, la

posterior modulación que prepare los datos para ser enviados a través de un canal con su correspondiente modelo de ruido y la recepción final de los datos que permita su demodulación y posterior utilización como entradas del decodificador que se pretende verificar.

En cualquier caso, dependiendo de las aplicaciones para las que haya sido diseñado el decodificador, los dos esquemas de simulación mencionados pueden resultar ser demasiado lentos. Es el caso, especialmente, de aplicaciones con una necesidad de ratios de error muy bajos.

Como alternativa a las simulaciones, en etapas de diseño pre-silicio, es habitual utilizar prototipos implementados sobre FPGAs que aprovechen el potencial para la paralelización que ofrece la lógica programable *hardware*. Sin embargo, los principales problemas de utilizar este tipo de implementación para la verificación radican en la necesidad de sintetizar y programar sobre la FPGA, no solo el componente a verificar, sino el resto de los módulos descritos previamente que modelan el sistema de comunicaciones completo. Este problema se puede solventar parcialmente con la utilización de componentes predefinidos en librerías. Sin embargo, se pueden dar casos de incompatibilidad entre estos componentes predefinidos y los diseños más novedosos de los componentes a verificar. Además, módulos tan importantes como aquellos encargados de la generación de datos aleatorios y del modelado del ruido del canal pueden requerir de una gran exactitud en las distribuciones utilizadas. Esto contrasta con algunos de estos componentes predefinidos donde las distribuciones utilizadas no son de la mejor calidad posible.

1.2 MOTIVACIÓN

Una vez se han presentado las dos implementaciones típicas (simulación HDL y prototipado sobre FPGAs) que se utilizan habitualmente para la verificación de

componentes en etapas de diseño pre-silicio es fácil comprender que cada una presenta una serie de ventajas y desventajas.

La principal ventaja de las simulaciones es la precisión con la que se puede comprobar el correcto funcionamiento de cada parte del decodificador, así como la flexibilidad para adaptar las entradas necesarias. En la parte negativa destaca como su mayor problema la gran lentitud que pueden presentar para la obtención de parámetros de rendimiento muy restrictivos en aplicaciones que así lo requieran.

En cuanto a la utilización de prototipos programados sobre FPGAs cabe destacar que aunque en principio pueda parecer una buena solución, capaz de aprovechar el potencial para la paralelización presente en la lógica programable *hardware* y así mitigar el principal problema que presentan las simulaciones, realmente es una solución que añade nuevas dificultades en lo referido al diseño y síntesis de modelos de gran calidad para el resto de los componentes que componen el esquema de simulación utilizado.

En [1] se propone una solución conjunta que aproveche las capacidades tanto *software* como *hardware* de un SoC FPGAs.

En esta solución se sintetiza e integra un decodificador LDPC descrito en VHDL, sobre la parte lógica programable (PL) de un Xilinx SoC FPGA. Al mismo tiempo, se gestiona mediante *software*, ejecutándose en el sistema de procesamiento (PS), el modelado del resto de componentes del esquema de simulación y la implementación de los dos enfoques mencionados para la verificación del decodificador (comparación contra modelo de referencia y obtención de curvas de rendimiento).

La puesta en práctica del enfoque basado en la coincidencia entre los resultados obtenidos y los resultados esperados se lleva a cabo de dos formas en función del modelo de referencia utilizado para la obtención de los valores esperados. Por un

lado, se utiliza como modelo de referencia un diseño a nivel algorítmico del decodificador descrito en un lenguaje de programación de alto nivel como C, mientras que, por otro lado, se realiza una síntesis de alto nivel del decodificador descrito en C y se integra en la parte lógica programable (PL) del SoC junto con el decodificador a verificar.

El objetivo de utilizar dos modelos de referencia distintos es el de poder medir los tiempos que tarda la verificación utilizando uno u otro y de esta forma establecer una comparación temporal que permita escoger el modelo de referencia más rápido.

Además, se implementan ambos enfoques sobre dos plataformas distintas:

- Tarjeta **Zedboard (Zynq-7000 XC7Z020 SoC)** con dos procesadores.
- Tarjeta **ZCU102 (Zynq Ultrascale+ XZCU9EG MPSoC)** con cuatro procesadores.

Se utilizan cuatro instancias del decodificador por cada hilo y un hilo por cada procesador. Esto provoca un total de ocho instancias para la plataforma con dos procesadores y de dieciséis instancias para la plataforma con cuatro procesadores.

Se puede apreciar la validez de la solución propuesta en la drástica reducción del tiempo de simulación experimentado por ambos enfoques. Se pasa del orden de días al orden de minutos en lo que se refiere a los tiempos de simulación obtenidos con una simulación *hardware* llevada a cabo con la herramienta Vivado HDL simulator de Xilinx sobre un PC con un procesador Intel i7 de 2,6 GHz con respecto a los obtenidos utilizando cualquiera de las dos tarjetas con un SoC integrado. Además, destaca la notable mejoría que se puede apreciar en la plataforma con un mayor número de procesadores. Finalmente, también se puede observar cómo son ligeramente mejores los resultados utilizando como

modelo de referencia el decodificador descrito en un nivel algorítmico mediante C.

Sin embargo, en el enfoque basado en la computación de parámetros de rendimiento como la CER, donde se llevan a cabo hasta 10^8 simulaciones de Montecarlo para Eb/No altas, el tiempo necesario para completar todo el proceso continúa siendo relativamente elevado llegando en ocasiones a durar hasta 67 minutos.

Ante estos resultados y con el éxito que ha supuesto repartir las tareas del proceso de verificación entre una parte *software* y una parte *hardware* de un mismo sistema surge la motivación de buscar nuevas soluciones que trabajando sobre esta misma línea permitan acelerar aún más el proceso de verificación de un decodificador diseñado a nivel RTL.

1.3 OBJETIVOS

Después de contextualizar y de exponer los motivos que impulsan a continuar trabajando en la investigación de nuevas soluciones para la aceleración del proceso de verificación de un componente diseñado a nivel RTL, concretamente un decodificador LDPC, es momento de describir los objetivos intermedios y finales que se fijaron para el presente proyecto.

El objetivo final de este proyecto deriva de los resultados obtenidos en las investigaciones mencionadas y se concreta en reducir la duración del proceso de verificación planteado en estas.

Para ello se plantean los siguientes objetivos parciales:

- Replicar una implementación básica del proceso de verificación planteado en la investigación que inspira este proyecto sobre una aplicación válida para un sistema heterogéneo basado en una tarjeta aceleradora tipo Alveo de Xilinx. Como solución continuista de la línea trabajo presentada puede

considerarse un primer objetivo a partir del que empezar a construir los siguientes objetivos más ambiciosos.

- Una vez se alcanza el objetivo inicial surge la necesidad de indagar en técnicas que permitan acelerar la aplicación. En este punto se plantean dos objetivos en base a la naturaleza del propio sistema heterogéneo. El primero propone acelerar la aplicación explotando los recursos disponibles para el procesado de *software*, mientras que el segundo propone realizar lo mismo, pero aprovechando los recursos *hardware* del sistema.
- Finalmente se puede optimizar una implementación con ambos escenarios mejorando otros aspectos de la aplicación como, por ejemplo, las transacciones entre memoria del sistema o la propia implementación de la paralelización.

1.4 ESTRUCTURA DEL DOCUMENTO

Este documento se estructura de la siguiente forma. Tras este capítulo de introducción, los conceptos teóricos necesarios para seguir el desarrollo del proyecto se presentan en el capítulo 2. Este capítulo tendrá cuatro secciones.

La primera sección describe las características básicas de la plataforma para el desarrollo de *software* Vitis™. Se enumeran las distintas tecnologías integradas en la herramienta y se hace énfasis en la librería Xilinx® Runtime al considerarse como la más importante para el desarrollo del proyecto.

La segunda sección se dedica por completo al análisis detallado del dispositivo Alveo de Xilinx® utilizado durante el proyecto. Se repasa la arquitectura de la tarjeta y las plataformas disponibles. La arquitectura de la plataforma instalada sobre la tarjeta también se repasa detenidamente.

La tercera sección se utiliza para incidir en la importancia de conocer el modelo de programación que definen las tecnologías utilizadas en el desarrollo de

cualquier aplicación. Además, presenta los modelos de ejecución soportados por los dispositivos Xilinx®.

La cuarta y última sección del capítulo 2 describe las características de la tecnología OpenCL utilizada para gestionar la interacción entre los distintos dispositivos que conforman el sistema heterogéneo del proyecto.

A continuación, en el capítulo 3 se muestra todo lo referido a la implementación del proceso de verificación sobre el sistema heterogéneo utilizado. Esto se distribuye entre cinco secciones. Las tres primeras enfocadas en presentar los recursos y escenario de partida y las dos últimas en explicar el desarrollo de una aplicación válida para el nuevo escenario y la puesta en marcha de la aplicación sobre el sistema.

En el cuarto capítulo se exponen las opciones que existen para acelerar la aplicación funcional que se ha implementado sobre el sistema en el capítulo previo. Básicamente se plantean dos escenarios. El primero propone la aceleración de la aplicación mediante la explotación de recursos *hardware* que permitan implementar paralelización de datos y el segundo propone la aceleración de la aplicación al implementar la paralelización de tareas sobre los diversos núcleos del multiprocesador del sistema heterogéneo.

Por último, se muestran unas conclusiones en base a los resultados obtenidos.

CAPÍTULO 2: ESTADO DEL ARTE

2.1 PLATAFORMA DE *SOFTWARE* VITIS™

Para el desarrollo del proyecto se decide utilizar la plataforma para el desarrollo de *software* unificado Vitis™ de Xilinx®. Se trata de una nueva herramienta que hereda las características y funcionalidades de otras herramientas de Xilinx® como SDAccel, SDSoC o SDK. La principal ventaja que ofrece reside en su capacidad para combinar todos los aspectos del desarrollo de *software* para distintas plataformas de Xilinx® en un único entorno. Esto quiere decir que soporta el desarrollo de aplicaciones para dispositivos de Xilinx® tan distintos como las tarjetas aceleradoras Alveo™ y aquellos basados en procesadores embebidos como lo son las tarjetas Zynq® UltraScale+™ MPSoC *y* Zynq®-7000 SoC.

Se trata de una herramienta que permite dos flujos de trabajo. Por un lado, contiene un entorno de desarrollo integrado (IDE) para el desarrollo del proyecto de forma interactiva, mientras que, por otro lado, permite el desarrollo de aplicaciones de una forma más manual al incorporar múltiples herramientas para trabajar desde la línea de comandos. Además, también incluye Vivado® Design Suite para la implementación de la lógica *hardware* sobre el dispositivo deseado o para el desarrollo de plataformas *hardware* personalizadas.

En cuanto a los flujos de desarrollo que soporta la herramienta destacan el flujo para el desarrollo de *software* embebido y el flujo para el desarrollo de aplicaciones aceleradas mediante *hardware*. En el primer caso, Vitis™ puede ser considerado como la tecnología de última generación por los usuarios habituales del kit de desarrollo *software* de Xilinx® (SDK) y en el segundo caso, es una buena solución para los usuarios que están buscando las últimas novedades en la aceleración de *software* mediante el uso de FPGAs de Xilinx®.

2.1.1 Tecnologías integradas en Vitis™

Las tecnologías utilizadas por la plataforma de *software* unificado Vitis™ pueden clasificarse en cuatro capas según su función como se muestra en la figura 2.1 [2]:

- Plataformas para los dispositivos Xilinx®. Es necesario diferenciar entre la plataforma de despliegue y la plataforma de desarrollo de un dispositivo especifico. La plataforma de despliegue proporciona el *firmware* necesario para ejecutar aplicaciones precompiladas en el dispositivo, es decir, que no puede utilizarse para compilar o crear nuevas aplicaciones, mientras que la plataforma de desarrollo permite a Vitis™/V++ compilar una aplicación para la plataforma prevista. La plataforma de despliegue ha de ser cargada he instalada en el dispositivo si se pretende ejecutar alguna aplicación sobre él. La plataforma de desarrollo se integra en el entorno de Vitis™ para permitir el desarrollo y la compilación de nuevas aplicaciones.
- La librería Xilinx Runtime (XRT). La librería es un conjunto de *software* de código abierto fácil de usar que facilita la gestión y el uso de los dispositivos FPGA/ACAP. Se implementa como una combinación de componentes, controladores o *drivers*, distribuidos entre el espacio de usuario y el núcleo del sistema operativo. Es compatible con las tarjetas basadas en PCIe de Alveo™, así como con las plataformas de sistemas embebidos basadas en *Zynq UltraScale+ MPSoC*, y proporciona una interfaz de *software* para los dispositivos lógicos programables de Xilinx®.
- El kit de desarrollo del núcleo de Vitis™. Se trata del conjunto de herramientas esenciales para el desarrollo de aplicaciones. Esto incluye herramientas como un compilador para arquitecturas x86 como es el caso del compilador del proyecto GNU/C++ g++, un compilador cruzado para arquitecturas de procesadores embebidos ARM, el compilador v++ para la construcción y el enlace a la plataforma destino, analizadores para el

análisis y el perfilado del rendimiento de la aplicación, y depuradores que ayudan a la localización y corrección de cualquier problema en esta.

- Librerías de Vitis™. Permiten optimizar el rendimiento de la parte de la aplicación acelerada mediante FPGAs sin necesidad de hacer grandes cambios en el código ni modificar algoritmos.

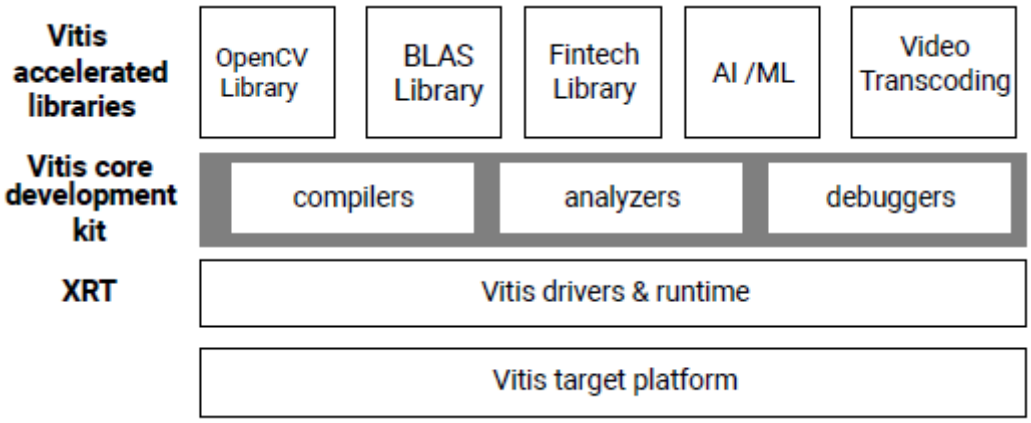

Figura 2.1. Estructura de la plataforma de software Vitis™.

2.1.2 Librería Xilinx Runtime (XRT)

En este apartado se dedica un espacio a describir algunos de los componentes más importantes del XRT y a ofrecer una visión general de la arquitectura de cualquier sistema que implementa la librería XRT.

Elementos de la líbrería XRT

En la figura 2.2 [3] se clasifican algunos de los elementos que componen la librería XRT y que son necesarios para cualquier sistema que utilice dispositivos de Xilinx®. En el nivel de aplicación destacan lenguajes de alto nivel (Python/C/C++) con soporte para la API XRT (*libxrt_core*), utilidades (*xbutil*, *xclbinutil*...) accesibles por el usuario para la gestión y administración de dispositivos Xilinx®, *drivers* (*xocl*, *xmgmt*…) con los que trabaja el sistema operativo para la gestión de dispositivos y finalmente las tecnologías (motor DMA, *shell* de la plataforma…) integradas en los propios dispositivos Xilinx® que junto a los *drivers* estandarizan de algún modo la forma de acceder al dispositivo.

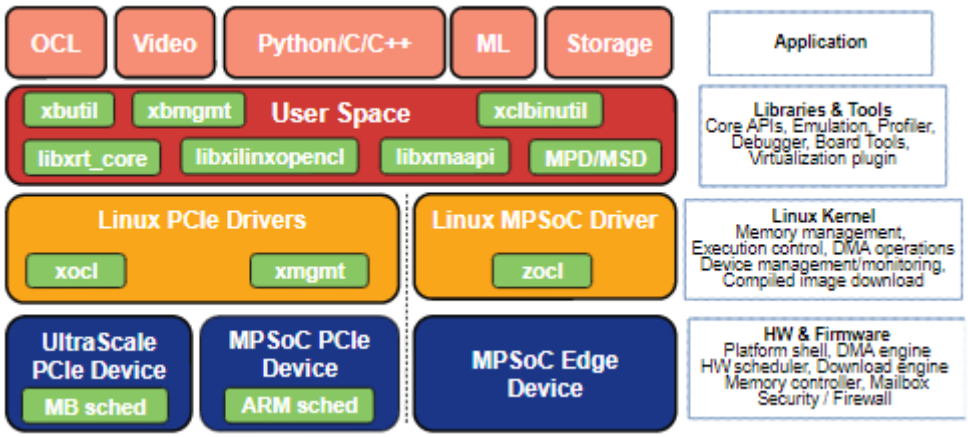

Figura 2.2. Pila de software del XRT.

Cabe destacar que la librería XRT posee una API de bajo nivel (*libxrt_core*). En el caso de modelos de uso muy avanzados o inusuales, es posible que se desee interactuar con ella directamente, pero en la mayoría de los casos se opta por utilizar una API de nivel superior como OpenCL (*libxilinxopencl*), el *framework* XMA (*libxmaapi*) u otros. En la figura 2.3 [4] se detallan las APIs envueltas durante la ejecución de una aplicación compatible con dispositivos Xilinx® PCIe. La aplicación se describe con lenguajes de alto nivel con soporte para la API de OpenCL o el *framework* de XMA. Estas APIs de alto nivel utilizan la API de XRT que a su vez utiliza el *driver xocl* para la interacción con otros dispositivos Xilinx®.

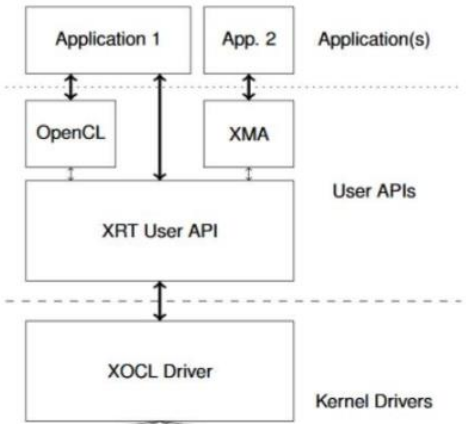

Figura 2.3. Flujo de APIs en una aplicación.

Aunque utilizar directamente la API de XRT permite una mayor personalización del diseño, también incrementa la complejidad. Por lo que para la realización de

este proyecto se utiliza la API OpenCL para subir un nivel de abstracción y facilitar el diseño.

Arquitectura de un sistema basado en XRT

Una vez se han descrito los distintos elementos que implementan la librería XRT, es momento de situarlos en el sistema y de describir la función que desempeñan en él. En este caso se explicará para un sistema heterogéneo con un dispositivo Xilinx® compatible con PCIe porque es el sistema que se utilizara durante el proyecto. Sin embargo, se ha de tener presente que esto es diferente para dispositivos con sistemas embebidos o dispositivos híbridos.

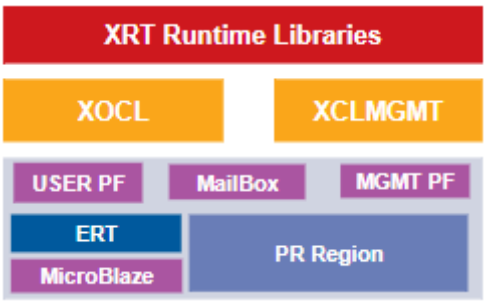

Figura 2.4. Arquitectura de un sistema heterogéneo basado en XRT.

Como se ha mencionado antes, algunas de estas tecnologías han sido diseñadas para ser implementas sobre la estación de trabajo y otras directamente sobre el dispositivo Xilinx®, representado en la figura 2.4 [5] por un rectángulo azul con más rectángulos en su interior. En el caso de un dispositivo PCIe, la plataforma estará dividida en dos regiones. La región conocida como *shell* proporciona una infraestructura básica para que los *drivers*, *xocl* y *xmgmt*, que se ejecutan en la estación de trabajo tengan acceso al dispositivo. Esto se consigue gracias a la implementación de dos funciones físicas, USER PF y MGMT PF, que permiten reconocer al dispositivo como dos dispositivos PCIe independientes. Cada driver se conecta y gestiona la comunicación con una de estas PFs. Cuál es la función del plano que implementa cada función física sobre la plataforma y a que elementos se interconecta se describe detalladamente en el subapartado dedicado a la arquitectura de una plataforma Alveo™ U50.

En términos generales, aunque se ha visto que la librería XRT está compuesta por muchos elementos, en lo referido al desarrollo de aplicaciones sus funciones se pueden clasificar en tres grandes grupos:

- Programación de los recursos lógicos del dispositivo Xilinx® y gestión del ciclo de vida del *hardware*.
- Asignación de memoria y migración de esta entre la estación de trabajo y el dispositivo Xilinx®.
- Gestionar el funcionamiento del *hardware*: secuenciar la ejecución de los *kernels*, establecer los argumentos del kernel, etc.

2.1.3 Flujo de desarrollo para la aceleración de aplicaciones mediante *hardware*

Se ha mencionado previamente que el entorno de Vitis™ ofrece dos flujos para el desarrollo de aplicaciones. En el caso de este proyecto se trabajará sobre el flujo para la aceleración de aplicaciones mediante *hardware*. Por lo tanto, se dedica este subapartado a explicar en qué consiste. En este flujo la plataforma de *software* Vitis™ proporciona un marco para desarrollar aplicaciones aceleradas vía FPGA. Esto es posible gracias al soporte de lenguajes de programación estándar utilizados en el desarrollo de los programas *software* y *hardware*.

En el caso de la aplicación *software*, o programa *host*, existe soporte para que sea desarrollado utilizando C/C++. Esto es así puesto que el kit de desarrollo del núcleo de Vitis™ integra el compilador g++ y el compilador cruzado Arm®. Además, la herramienta ofrece la posibilidad de realizar llamadas a la API OpenCL™ para gestionar las interacciones con dispositivos de Xilinx® o de utilizar directamente la API de XRT o alguna de sus adaptaciones a lenguajes de alto nivel como C++ o Python.

El programa *hardware* o *kernel*, puede desarrollarse utilizando C/C++, OpenCL C o un lenguaje de descripción *hardware* como VHDL o Verilog. Para todos ellos el kit de desarrollo del núcleo de Xilinx® integra el compilador v++ para la construcción de un archivo binario *xclbin*.

2.2 TARJETA ACELERADORA PARA CENTROS DE DATOS XILINX® ALVEO™ U50

La tarjeta Alveo™ U50 posee un mayor número de recursos lógicos programables que los disponibles en las tarjetas *Zedboard (Zynq-7000 XC7Z020 SoC)* y *ZCU102 (Zynq Ultrascale+ XZCU9EG MPSoC)* utilizadas en [1]. Para poder aprovechar estos recursos es importante conocer tanto la arquitectura como las plataformas soportadas por la tarjeta. Esto, por un lado, permite personalizar la tarjeta para que ofrezca unas funcionalidades u otras mientras que, por otro lado, permite conocer las limitaciones intrínsecas a la tarjeta, permitiendo adaptar el desarrollo de la aplicación a las funcionalidades ofertadas.

2.2.1 Arquitectura de la tarjeta Alveo™ U50

Es una tarjeta de una sola ranura PCI, con un factor de forma de bajo perfil y refrigeración pasiva que funciona hasta un límite de potencia máxima de 75 W. Es compatible con PCI Express® (PCIe®) Gen3 x16 o Gen4 x8 dual, está equipada con 8 GB de memoria de gran ancho de banda (HBM2) y capacidad de conexión a la red vía Ethernet. Está diseñada para acelerar las aplicaciones intensivas en memoria y computación en el ámbito de la informática financiera, el almacenamiento computacional, la búsqueda de datos y la analítica.

Los componentes y la forma en la que estos se interconectan se muestran gráficamente en la figura 2.5 [6].

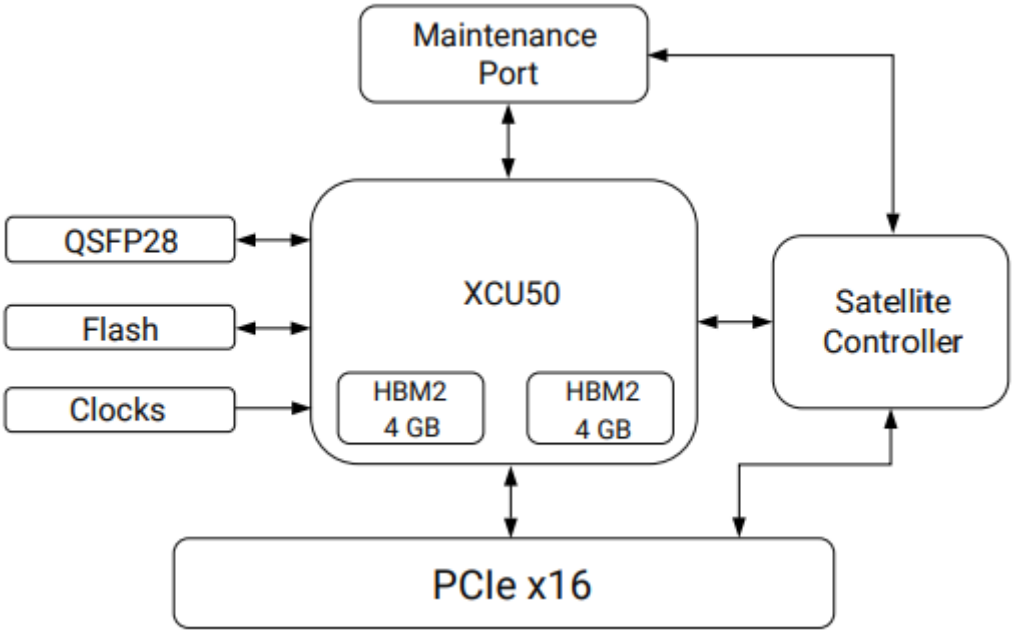

Figura 2.5. Diagrama de bloques de la Aveo™ U50.

Por un lado, están los dispositivos que han sido integrados con el fin especifico de habilitar la conexión de la tarjeta Alveo™ U50 a otros dispositivos externos.

- La conexión de la tarjeta a la placa base de la estación de trabajo se realiza mediante un bus PCI. Por ello, la tarjeta trae integrado el dispositivo PCIE4C compatible con la especificación básica PCI Express v3.1 (Gen3 x16) y con la especificación básica PCI Express v4.0 (Gen4 x8).
- La conexión de la tarjeta a otras estaciones de trabajo se puede llevar a cabo mediante el transceptor QSFP28 diseñado para la transmisión a corta distancia en la red Ethernet 100G.
- La conexión a la placa de depuración y mantenimiento (DMB) desarrollada por Xilinx® se lleva a cabo mediante el puerto de mantenimiento. Este puerto es un conector de treinta pines que permite el acceso a una variedad de características y señales incluyendo JTAG, UARTs, PMBus, resets y más.

Por otro lado, están los dispositivos integrados específicamente para la correcta operación de la tarjeta:

- El controlador satélite se encarga de ejecutar el firmware necesario para la gestión de la tarjeta, proporcionando mecanismos de comunicación dentro y fuera de banda (OOB) y protecciones térmicas y eléctricas.
- Los relojes que permiten la sincronización de los dispositivos y acotan la frecuencia de operación.
- La memoria Flash almacena la plataforma instalada en la tarjeta.
- La FPGA XCU50 cuenta con tecnología Ultrascale+ y se muestra la figura 2.6 [5] donde los componentes GTY son transceptores. Consta de dos regiones superlógicas (SLRs), con la SLR inferior (SLR0) integrando un controlador HBM2 para interactuar con la memoria HBM2 adyacente de 8 GB.

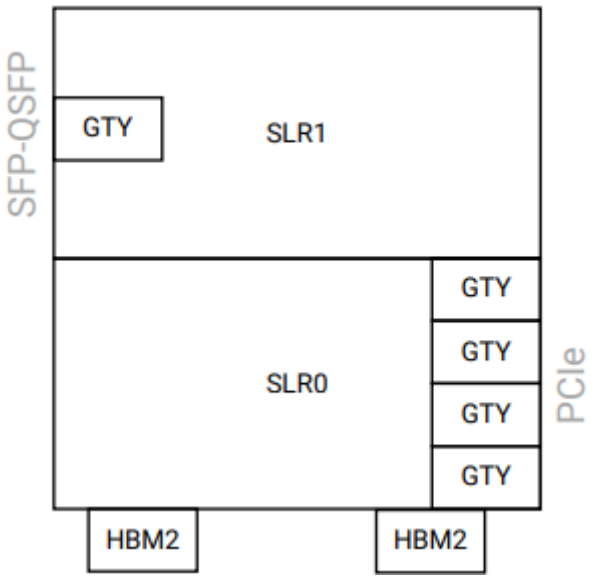

Figura 2.6. Plano de la FPGA XCU50.

2.2.2 Plataformas soportadas por la Alveo™ U50

Como se ha mencionado antes, una plataforma se instala y se almacena en la memoria *Flash* de la tarjeta con la opción de ser actualizada desde el *host* a través del bus PCIe. Sin embargo, para entender por qué es necesaria la existencia de estas, convine recordar que, a diferencia de una CPU, una GPU o un ASIC, una FPGA no tiene ninguna arquitectura preconfigurada. Tiene un conjunto de

recursos lógicos de muy bajo nivel, como flip-flops, puertas y SRAM, pero muy poca funcionalidad fija. Todas las interfaces del dispositivo, incluidos los enlaces PCIe y de memoria externa, se implementan utilizando algunos de esos recursos. Esto quiere decir que existen ciertas funcionalidades relacionadas con los enlaces PCIe, la monitorización del sistema y la seguridad de la placa que no están disponibles inicialmente en la FPGA y que de algún modo se necesita que estén siempre disponibles para el procesador anfitrión al que se encuentra conectado la Alveo™ U50 mediante el bus PCIe.

Con el fin de evitar que el usuario tenga que implementar todas esta funcionalidades cada vez que lleva a cabo cualquier diseño se conciben las plataformas. Esto explica por qué los diseños de las plataformas Alveo™ se dividen en un modelo conceptual de *shell* y de rol, que se muestra gráficamente en la figura 2.7 [7], también conocidos como región estática y región dinámica respectivamente.

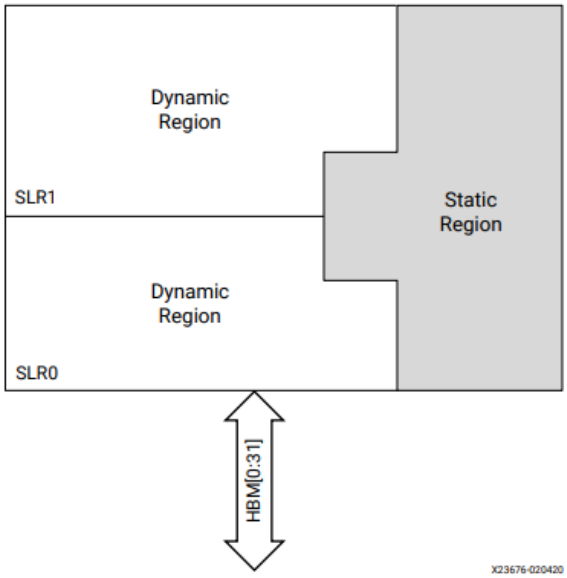

Figura 2.7. Modelo conceptual de una plataforma de Alveo™ U50.

La *shell* contiene toda la funcionalidad estática (enlaces externos, configuración, sincronización, etc) cargada desde la memoria *Flash* durante el arranque de la tarjeta y que no es reconfigurable mientras esta permanezca activa.

Por su parte la región dinámica consta de los recursos lógicos restantes de las SLRs que no han sido utilizados para la implementación de la *shell* y que, por lo tanto, quedan a disposición del usuario para la programación de una lógica personalizada.

Dependiendo de las funcionalidades requeridas por la aplicación se implementará una plataforma u otra sobre la FPGA. La principal diferencia entre plataformas diseñadas para la misma tarjeta será la configuración y funcionalidades ofrecidas por la región estática. En el caso de la tarjeta Alveo™ U50 existen tres plataformas:

- **Xilinx_u50_gen3x16_nodma_base_1.**
- **Xilinx_u50_gen3x16_xdma_201920_3.**
- **Xilinx_u50_gen3x4_xdma_base_2**

Para la realización de este proyecto se dispone, en concreto, de la plataforma **Xilinx_u50_gen3x16_xdma_201920_3.**

Para la comunicación con los 8 GB de memoria HBM2 adyacentes a la FPGA se establecen treinta y dos pseudo canales. Además, la plataforma ofrece la posibilidad de utilizar parte de los recursos lógicos como PLRAM, dos por cada SLR, completando un total de cuatro pequeñas regiones de rápido acceso con capacidad para almacenar 128 KB de datos cada una. Sin embargo, cabe destacar que esta plataforma en concreto tan solo permite el uso de veintiocho pseudo canales según (Xilinx, July 30, 2021). El uso de más pseudo canales generará errores durante la compilación de la aplicación. Por su parte Xilinx® recomienda el uso de los canales que van desde el cero al veintisiete.

2.2.3 Arquitectura de una plataforma Alveo™

Una vez se conoce la arquitectura de la tarjeta y la separación conceptual en dos regiones de la plataforma implementada sobre la FPGA se procede a describir la arquitectura de cada región

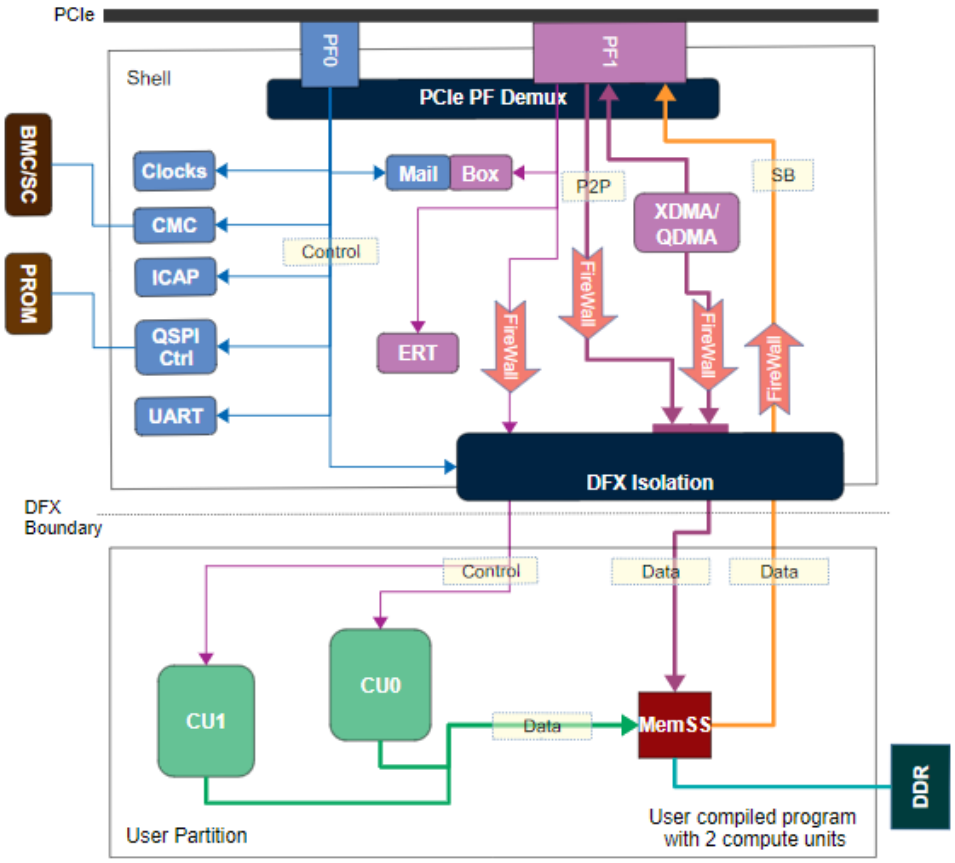

Figura 2.8. Arquitectura de la plataforma Alveo™.

Arquitectura de la región estática

Como se ha mencionado anteriormente, la región estática proporciona la infraestructura básica para garantizar servicios clave. Los componentes encargados de implementar estos servicios se pueden clasificar en dos tipos según el color, azul o morado, con el que aparecen sombreados en la figura 2.8 [8]:

- Aquellos sombreados en azul se encargan de funciones relacionadas con la administración y gestión de la tarjeta.

- Aquellos sombreados en morado se utilizan directamente para implementar la aplicación desarrollada por el usuario.

Para implementar esta separación es necesario aislar de algún modo el acceso externo a ambas funcionalidades. Esto se consigue mediante el uso de dos funciones físicas PCIe (PF0 y PF1). Estas permiten que el dispositivo PCIE4C integrado en la tarjeta, que se utiliza para la conexión al bus PCI de la placa base de la estación de trabajo, se perciba desde el exterior como dos dispositivos PCIe independientes. Con esto se consigue separar el acceso entre funcionalidades de gestión y de operación, obligando al uso de drivers diferentes para cada uno. Internamente la tarjeta utiliza un IP especializado en el enrutamiento del tráfico dirigido al exterior. Se conoce como PCIe *Demux* y se encarga de enviar el tráfico de gestión a la función física PF0 y el tráfico de operación a la función física PF1.

En el plano de gestión se incluyen componentes como un módulo ICAP para la descarga del *bitstream* generado durante la compilación de la aplicación, un controlador para la gestión térmica de la tarjeta (CMC) que además proporciona una comunicación UART/I2C con el controlador satélite, con el transceptor QSFP y con algunos sensores, un módulo QSPI para controlar el acceso a la memoria *Flash* o PROM en algunas tarjetas (actualizaciones de la *shell*) y finalmente un módulo *Clock Wizards* para el escalado del reloj.

En el plano de operación predominan dos elementos. Uno actúa como un planificador en tiempo de ejecución integrado en el sistema (ERT) y el otro es el motor para los accesos directos a memoria (DMA).

El motor DMA es el componente más importante en el plano de operación puesto que de él depende la disponibilidad o no de ciertas funcionalidades. Además, es el único elemento de la región estática o *shell* que no es esclavo desde el punto de vista del PCIe. Esto significa que puede iniciar transacciones PCIe. En este momento las plataformas diseñadas por Xilinx® pueden contar con un motor PCIe DMA XDMA o con un motor PCIe DMA QDMA.

Las características y el modelo de programación de cada motor se comparan gráficamente en la figura 2.9 [9][10].

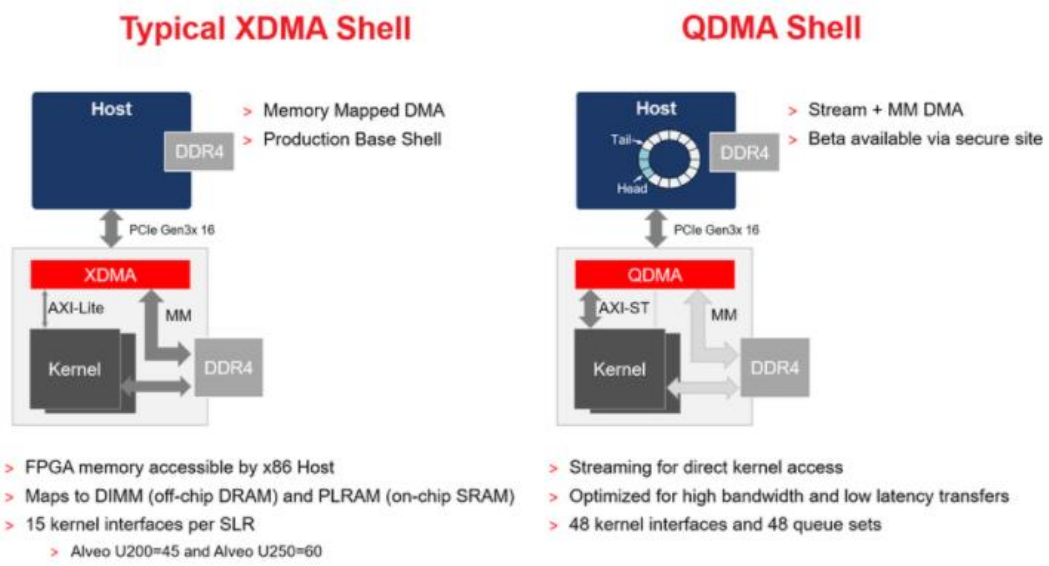

Figura 2.9. Comparación entre XDMA y QDMA.

El motor XDMA dispone de dos interfaces, una *AXI-Lite* y una *AXI-MM* o *AXI mapeada en memoria*. Estas interfaces le permiten establecer cuatro canales, dos para transferir datos desde la memoria integrada en la FPGA, tanto la que se construye directamente con los recursos de esta (PLRAM) como la que se encuentra fuera del chip (HBM2) a la memoria RAM de la estación de trabajo, y otros dos para transferir datos en sentido contrario, es decir, desde la memoria de la estación de trabajo hasta la memoria de la tarjeta.

El motor QDMA también dispone de dos interfaces, pero en esta caso se trata de una interfaz *AXI-Stream* y una interfaz *AXI-MM*. La interfaz *AXI-MM* permite al motor QDMA desempeñar el mismo cometido que al motor XDMA, pero la presencia de la interfaz *AXI- Stream* añade la posibilidad de trasferir datos entre la memoria de la estación de trabajo y de la tarjeta directamente, sin necesidad de pasar por la memoria de esta última. Sin embargo, la gran novedad introducida por el motor QDMA con respecto al XDMA es el concepto de colas, que deriva del concepto "conjunto de colas" descrito en la tecnología *Remote Direct Memory Access* (RDMA) de las interconexiones de computación de alto

rendimiento (HPC). Es esta característica la que le permite ofrecer funcionalidades como las descritas anteriormente con un ancho de banda alto y baja latencia.

En la tarjeta Alveo™ U50 utilizada para este proyecto solo han sido diseñadas plataformas en cuya región estática se implementa como motor para los accesos directos a memoria un motor XDMA. En [11] se comenta que en estos momentos se está trabajando en el diseño de una *shell* con QDMA para una plataforma válida en esta tarjeta.

El otro elemento presente en el plano de operación era el planificador en tiempo de ejecución (ERT). Este se encarga de planificar y monitorizar las unidades de cómputo (CUs) de un *kernel* durante su ejecución.

Finalmente, otro elemento muy importante en esta región y que se utiliza en ambos planos es la función dinámica de intercambio (DFX). El plano de gestión lo utiliza para almacenar el archivo *xclbin* generado durante la compilación del proyecto y que contiene toda la información necesaria para implementar sobre la región dinámica la lógica *hardware* de la aplicación. Luego es el módulo ICAP, programado durante la descarga del *xclbin*, el encargado de gestionar esta información y de llevar a cabo la implementación sobre la región dinámica. Y desde el punto de vista del plano de operación la DFX es vista como la región donde se implementan los registros e interfaces AXI de las unidades de cómputo de un *kernel*.

Arquitectura de la región dinámica

Esta región está compuesta por todos aquellos recursos de la FPGA que no han sido utilizados para la implementación de la región estática. A diferencia de esta, la región dinámica se puede modificar mientras la tarjeta permanece activa mediante la carga de archivos *xclbin* en la región DFX y la programación del ICAP, con ayuda del **driver** adecuado para el plano de gestión, para que lleve a cabo la implementación del *bitstream* empaquetado en el *xclbin*. Además, cuenta

con un pequeño controlador para las transacciones entre el DMA y la memoria HBM2 adyacente a la FPGA.

El acceso a esta región es posible principalmente a través de dos caminos definidos por el plano de operación. Estos son el camino de control y el camino de datos. El camino de control tiene lugar entre la PF1 y la interfaz *AXI-Lite*, localizada en la región DFX, correspondiente a la CU del kernel implementado. El camino de datos tiene lugar entre la PF1 y el controlador de memoria implementado sobre la región dinámica de la FPGA. En el camino atraviesa las interfaces correspondientes del DMA y la interfaz *AXI-MM*, localizada en la región DFX, correspondiente a la CU del kernel implementado. Dependiendo de la plataforma pueden existir más caminos de datos que implementan otras funcionalidades como serían la de P2P, para la comunicación entre tarjetas conectadas en ranuras PCIe de la misma estación de trabajo, o el de SB o también conocido como HM, que habilita a la FPGA de la tarjeta a escribir y leer directamente en o de la memoria de la estación de trabajo sin pasar por el DMA. Los diferentes componentes que atraviesa el camino de datos clásico que utiliza el DMA desde la estación de trabajo hasta la tarjeta se muestran de forma gráfica en la figura 2.10 [5].

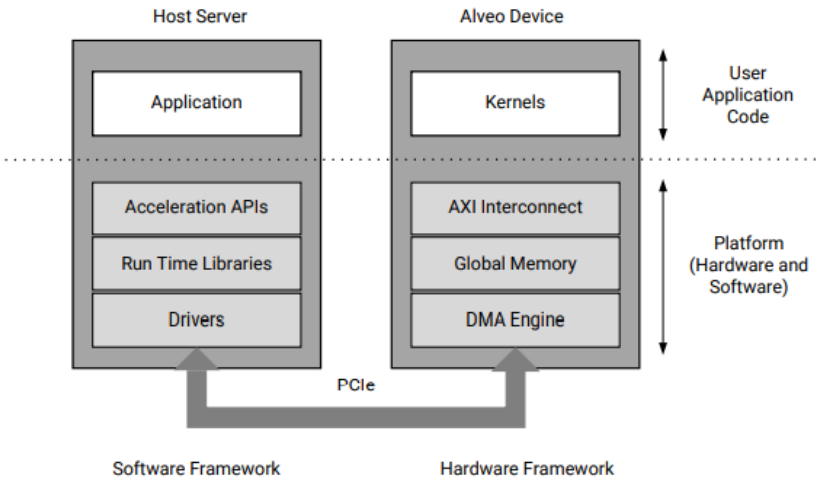

Figura 2.10.Interfaces software/hardware en el camino de datos típico.

2.3 MODELO DE PROGRAMACIÓN

Se entiende como modelo de programación de una aplicación al modelo de ejecución o conjuntos de modelos de ejecución que definen el comportamiento de esta. Normalmente un lenguaje de programación se compone de una sintaxis y de un modelo de ejecución. El modelo de ejecución describe el comportamiento de un programa o aplicación durante su ejecución y es característico del lenguaje con el que ha sido programado. Por ejemplo, en C el modelo de ejecución define el orden con el que se han de ejecutar las sentencias del programa. Normalmente lo harán de forma secuencial exceptuando casos como las llamadas a funciones o estructuras de control (bloques condicionales) donde este flujo se ve alterado. Sin embargo, es necesario distinguir entre modelo de ejecución del lenguaje y el modelo de programación de la aplicación. Aunque habitualmente estos coinciden, un programa o aplicación puede incluir llamadas a librerías que alteren el modelo de ejecución del programa. En el caso de que estas llamadas no alteren el flujo de ejecución propio del lenguaje con el que ha sido escrito el programa no es necesario hablar de modelo de programación. Sin embargo, cuando estas llamadas alteran el flujo de ejecución impuesto por el lenguaje es necesario definir un modelo de programación que explique el cambio en el modelo de ejecución.

En el caso de este proyecto, la aplicación se desarrollará en C/C++ y se utilizará la librería de OpenCL. Por lo tanto, el modelo de ejecución de la aplicación, que en principio se corresponderá al definido por C/C++, puede verse alterado. Además, hay que tener en cuenta que, en este caso, el modelo de ejecución de OpenCL no puede entenderse en términos de C/C++. Esto se observa claramente en aquellas llamadas donde el programa cede el control a un entorno de ejecución como puede ser el XRT para que este realice distintas operaciones sobre el dispositivo Xilinx®. Estas llamadas no pueden ser entendidas por la aplicación como simples llamadas de C a una función puesto que no siguen el flujo de

ejecución propio de una función en C e implementan su propio modelo de ejecución. Todo esto obliga a hablar de una aplicación con un modelo de programación definido por el modelo del ejecución del lenguaje con el que se ha desarrollado, en este caso C/C++, y el modelo de ejecución implementado sobre XRT para la interacción con la lógica programable del dispositivo.

2.3.1 Flujos de diseño de un *kernel*

La presencia de otro modelo de ejecución invocado durante la ejecución del programa implica nuevas consideraciones en el diseño de la lógica *hardware*. A esta lógica *hardware* se la conoce como *kernel* en el entorno de Vitis™ y antes de entrar en detalle con los nuevos requisitos introducidos es necesario conocer las opciones que existen a la hora de diseñar *kernels*.

El *kernel* puede ser escrito en C, C++, OpenCL™ C o en un lenguaje de descripción *hardware* para los diseños a nivel RTL. La construcción del kernel se divide en dos pasos, la compilación y el enlace o *linkado*, y se esquematiza en la figura 2.11 [12].

La compilación en todos los casos debe devolver como resultado un archivo *Xilinx® object* (XO). Las herramientas utilizadas y el proceso seguido es la diferencia entre utilizar un lenguaje u otro.

- En el caso de *kernels* diseñados en OpenCL™ C el compilador v++ con la opción de compilación -c retornara como resultado el archivo *Xilinx® object*.
- Para los *kernels* diseñados en C/C++ el entorno de Vitis™ ofrece dos posibilidades. Por un lado, permite utilizar el compilador v++ con la opción -c del mismo modo que para los *kernels* escritos en OpenCL™ C y, por otro lado, permite utilizar Vivado HLS para sintetizar el *kernel* con la personalización adecuada mediante pragmas HLS.
- Los *kernels* diseñados a nivel RTL deben ser empaquetados como un bloque IP con ayuda de la herramienta *Vivado IP Packager* de *Vivado®*

Design Suite. Después el bloque IP junto con un fichero XML de configuración han de ser compilados mediante *xo_package*.

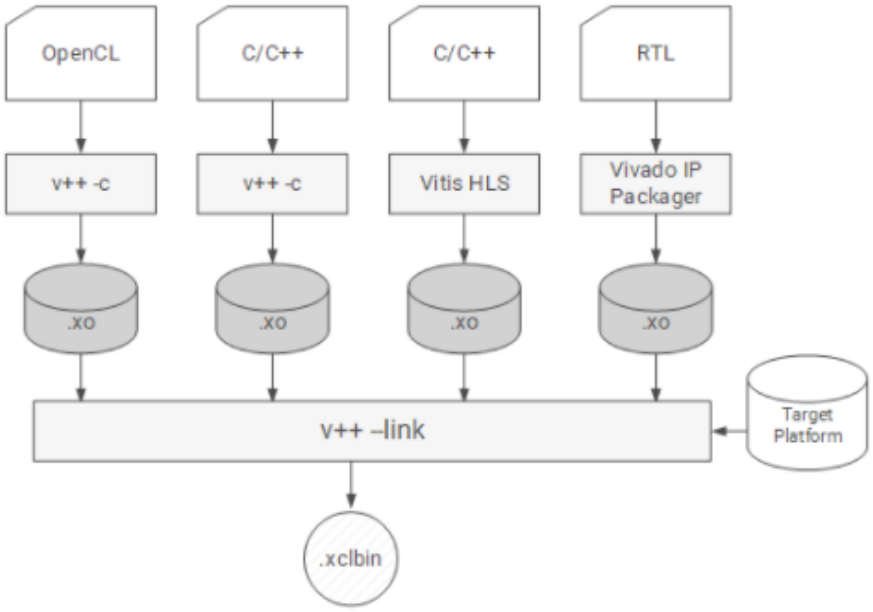

Figura 2.11. Proceso de construcción de un kernel en el entorno Vitis™.

El enlace o *linkado* se lleva a cabo sobre el o los *Xilinx® objects* compilados y la plataforma destino mediante la opción -l del compilador v++.

2.3.2 *Kernels* controlados por *software*

El modelo de ejecución invocado por la aplicación depende del protocolo de control que se implemente entre el *host* y el *kernel*. Por lo tanto, depende en gran medida del *driver* utilizado en el *host*. Al tratarse de dispositivos Xilinx® los *drivers* utilizados se encuentran incluidos en la librería XRT y la operación con estos es transparente para el usuario gracias a la existencia de la API nativa. Sin embargo, a pesar de que se trata de una API de bajo nivel, esta ofrece dos niveles de abstracción al usuario.

- Por un lado, está la clase *xrt::kernel* para identificar un *kernel* con una interfaz de control estándar en el código del *host*. En esta caso la comunicación entre ambos, *host* y *kernel*, se podrá realizar mediante otras

clases predefinidas de la API. Este uso de la API ofrece un mayor nivel de abstracción al usuario.

- Por otro lado, está la clase *xrt::ip* para referenciar un *kernel* con una interfaz de control personalizable en el código de *host*. En esta caso la comunicación entre ambos, *host* y *kernel*, es completamente personalizable mediante escrituras y lecturas sobre los registros definidos en el *kernel*.

Cuando el control sobre las operaciones que tienen lugar en el *kernel* depende de las llamadas realizadas a la API nativa de XRT se habla de *kernels* gestionados o administrados por XRT. Al contrario, cuando el control sobre lo que ocurre en el *kernel* está muy controlado por el propio usuario mediante escrituras y lecturas directas en registros del *kernel* se habla de *kernels* gestionados o administrados por el usuario. En ambos casos se habla de *kernels* controlados mediante *software*. Además, en ambos casos los *kernels* necesitan cumplir una serie de requisitos para ser compatibles con la implementación de XRT. Para conseguir esta compatibilidad se han de añadir una serie de interfaces en los *kernels*. Además de la presencia de estas interfaces, es obligatoria la presencia de al menos un reloj y de forma opcional de un *reset* y un puerto de interrupciones:

- **Interfaz de registros programables:** se trata de una interfaz de carácter obligatorio para el acceso a los diferentes registros de un *kernel*. Se debe implementar como una única interfaz esclava usando el protocolo *AXI-Lite* y debe nombrarse *s_axi_control*.
- **Interfaz de datos:** se trata de una o varias interfaces a través de las cuales el *kernel* recibirá los datos con los que operar. Se debe implementar mediante cualquier combinación de interfaces mapeadas en memoria actuando como maestro utilizando el protocolo *AXI4* o mediante interfaces para el *stream* de datos utilizando el protocolo *AXI-Stream*.

La presencia de estas interfaces en un *kernel* permite al compilador de Vitis™ v++ enlazar o *linkar* el *kernel* a la plataforma destino instalada en el dispositivo Xilinx®. La principal diferencia entre ambos tipos de *kernels* es que los *kernels* gestionados por XRT serán controlados mediante llamadas a su API implementando uno de los dos modelos de ejecución predefinidos por XRT mientras que los *kernels* gestionados por el usuario serán controlados mediante el uso de drivers UIO e implementarán cualquier protocolo de control o modelo de ejecución.

Sin embargo, mientras que en los *kernels* gestionados por el usuario no hace falta cumplir ningún requisito más, en el caso de los *kernels* gestionados por XRT no es así. En este último caso los *kernels* deben cumplir unos requisitos de control además de los requisitos relativos a interfaces. Para cumplir estos requisitos se deben implementar una serie de registros con sus correspondientes señales y mapeo de direcciones. Los registros y las direcciones a las que deben ser mapeados se muestran a continuación en la tabla 2.1.

Offset	**Nombre**	**Descripción**
0x0	*Control*	Controla y proporciona el estado del *kernel*.
0x04	*Global Interrupt Enable (GIE)*	Usado para habilitar la interrupción al *host*.
0x08	*IP Interrupt Enable (IIE)*	Se utiliza para controlar qué señales generadas por IP se utilizan para generar una interrupción.
0x0C	*IP Interrupt Status (IIS)*	Proporciona el estado de la interrupción.
0x10	Argumentos del *kernel*	Esto incluiría, por ejemplo, escalares y argumentos de memoria global.

Tabla 2.1. Registros y mapeo de direcciones.

Además, el protocolo para el control del *kernel* se debe implementar mediante el registro de control. Por ello, los bits de este se utilizan como señales de control que permitan al XRT y al *kernel* establecer una comunicación donde se indique el inicio, final, y otros aspectos que permitan al XRT conocer en todo momento el estado de la operación que está teniendo lugar en el *kernel*. Las señales del registro de control y su utilidad se describen en la tabla 2.2.

Bit	Nombre	Descripción
0	*ap_start*	Se activa cuando el *kernel* puede empezar a procesar datos. Se borra en el *handshake* cuando se activa *ap_done*.
1	*ap_done*	Se activa cuando el *kernel* ha completado la operación. Se borra cuando se lee.
2	*ap_idle*	Se activa cuando el *kernel* está inactivo.
3	*ap_ready*	Se activa desde el *kernel* cuando esté listo para aceptar los nuevos datos.
4	*ap_continue*	Se activa desde XRT para permitir que el *kernel* siga funcionando.
7	*auto_restart*	Se utiliza para activar el reinicio automático del *kernel*.
31:5	*reserved*	Bits reservados por el XRT.

Tabla 2.2. Señales del registro de control.

2.3.3 Modelos de ejecución soportados por *kernels* gestionados por XRT

Se trata de los dos posibles modelos de ejecución presentes en la aplicación cuando el usuario ejecuta el *kernel* mediante una API de alto nivel como OpenCL (*clEnquqeTask*) o la API nativa de XRT (objeto de la clase *xrt::run*). En ambos casos se consigue ocultar los detalles de la implementación al usuario, evitando

que este tenga que manejar el registro de control explícitamente dentro del código del *host*.

Por otra parte, se trata de un tipo de *kernel* pensado para el flujo de diseño basado en el uso de HLS. Esto es así debido a la existencia de una serie de pragmas que automatizan la generación de lógica RTL compatible con XRT. Sin embargo, es importante que el diseñador RTL conozca el funcionamiento de este tipo de *kernels*, es decir, los detalles en la implementación de los dos modelos de ejecución soportados. Con esto el diseñador RTL, que en principio tendría la flexibilidad para crear un modelo de ejecución totalmente personalizado, tendrá la capacidad de adaptar sus diseños RTL para aprovechar las ventajas de la automatización que ofrece XRT en estos dos modelos de ejecución.

Los dos principales modelos de ejecución soportados son y se representan en las figuras 2.12 y 2.13 [13] respectivamente:

- Modelo de ejecución secuencial
- Modelo de ejecución en línea

Modelo de ejecución secuencial

La idea del modelo de ejecución secuencial es el de establecer un esquema de sincronización de un punto entre el *host* y el *kernel* utilizando dos señales: *ap_start* y *ap_done*. Este modo de ejecución permite que el *kernel* reinicie su ejecución sólo cuando haya completado la ejecución en la que se encuentre. Esto hace que cuando existan múltiples solicitudes para la ejecución del *kernel* desde el *host*, este se ejecute en orden secuencial.

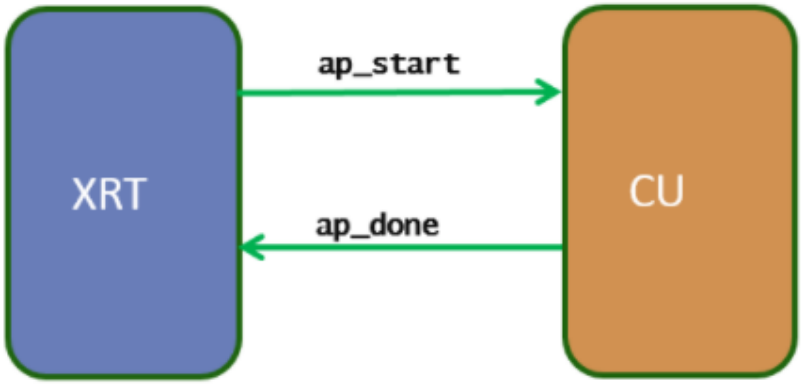

Figura 2.12. Modelo de ejecución secuencial.

El flujo de operación entre el *driver* de XRT y una instancia de un *kernel* se describe a continuación:

- El *driver* de XRT activa la señal *ap_start* para iniciar el *kernel*.
- El *driver* de XRT espera a que *ap_done* sea activado por el *kernel* (esto garantiza que la ejecución del *kernel* haya finalizado).

Este flujo de operación se repite con todas las peticiones de ejecución del *kernel* que el *driver* del XRT vaya acumulando. La implementación de este protocolo se lleva a cabo mediante la especificación del pragma *AP_CTRL_HS* en la interfaz del *kernel* cuando este se ha diseñado mediante un flujo basado en HLS. De otra forma, en un diseño RTL, es el usuario el que ha de implementar los puertos y la lógica que garanticen una interfaz operando de esta manera.

Modelo de ejecución en línea

El *kernel* está implementado de tal manera que puede permitir que múltiples ejecuciones del *kernel* se superpongan. Para lograr esto se divide la sincronización entre el *host* y el *kernel* en dos puntos: la sincronización de entrada (dictada por las señales *ap_start* y *ap_ready*) y la sincronización de salida (*ap_done* y *ap_continue*). Este modo de ejecución permite que el *kernel* se reinicie incluso si se encuentra todavía trabajando en la ejecución actual. Por lo tanto, cuando existen múltiples solicitudes de ejecución del *kernel* desde el *host*, este superpone las múltiples ejecuciones a la vez.

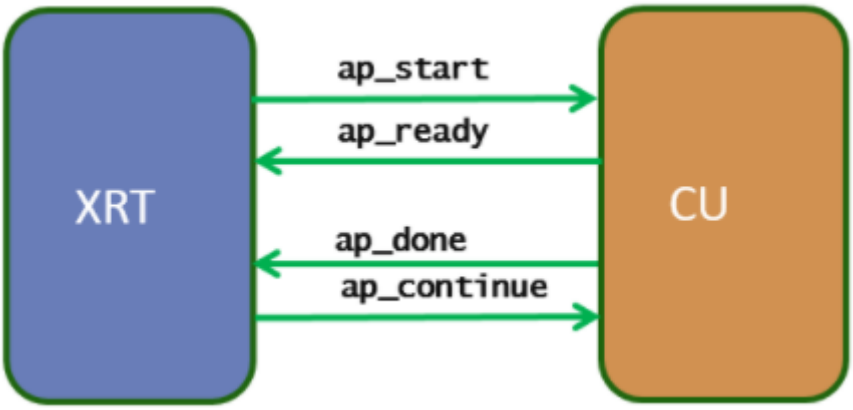

Figura 2.13. Modelo de ejecución en línea.

El flujo de operación entre el *driver* de XRT y una instancia del *kernel* se describe a continuación distinguiendo entre la sincronización de entrada y la sincronización de salida.

En el caso de la sincronización de entrada se tienen tres fases:

- El *driver* de XRT activa la señal *ap_start* para iniciar el *kernel*.
- El *driver* de XRT espera a que *ap_ready* sea activado por el *kernel* (con esto se garantiza que el *kernel* está listo para aceptar nuevos datos para la siguiente ejecución, incluso si todavía está trabajando en la solicitud de ejecución anterior).
- El *driver* de XRT activa de nuevo la señal *ap_start* para iniciar de nuevo la operación del *kernel*

Para la sincronización de la salida se tienen únicamente dos fases:

- El *driver* de XRT espera a que *ap_done* sea activado por el *kernel* (garantizando que los datos de salida son producidos completamente por el *kernel*).
- El *driver* de XRT activa la señal *ap_continue* para que el *kernel* siga funcionando.

La sincronización de entrada y salida se produce de forma asíncrona, permitiendo que el *kernel* realice múltiples ejecuciones de forma solapada. La

implementación de este protocolo se lleva a cabo mediante la especificación del pragma *AP_CTRL_CHAIN* en la interfaz del *kernel* cuando este se ha diseñado mediante un flujo basado en HLS

2.3.4 *Kernels* no controlados por *software*

En este caso el modelo de ejecución invocado por la aplicación no necesita que se implemente un protocolo de control entre el *host* y el *kernel*. Por ello, en general, el diseño de este tipo de *kernel* se realiza sin una interfaz de registro programable. El único requisito impuesto por el entorno de Vitis™ es que posean al menos una interfaz *AXI4-Stream*. La sincronización entre el *kernel* y el resto del sistema tiene lugar a través de estas interfaces de *streaming*.

Los *kernels* no controlados por *software* se consideran una característica avanzada y sólo deben utilizarse cuando no se pueda utilizar un *kernel* controlable por *software*. No requieren una API de *software*, ya que la aplicación no interactúa directamente con el *kernel*. El *kernel* puede desarrollarse como *kernel* RTL o como *kernel* C/C++, en este último caso especificando el pragma *AP_CTRL_NONE* para la herramienta de HLS.

El modelo de ejecución consiste en que el *kernel* sólo funcione cuando los datos estén disponibles en su entrada, y se detenga cuando no haya datos para procesar, esperando que lleguen nuevos datos para empezar a trabajar de nuevo.

2.4 OpenCL

OpenCL es un estándar abierto de la industria utilizado para la programación de una colección heterogénea de CPUs, GPUs y otros dispositivos de computación discreta organizados en una única plataforma. No se trata únicamente de un lenguaje. OpenCL es un marco de trabajo para la programación paralela e incluye un lenguaje, una API, librerías y un sistema de ejecución para apoyar el desarrollo de *software*.

Para describir completamente las ideas centrales de OpenCL, se utiliza una jerarquía de modelos:

- Modelo de plataforma
- Modelo de memoria
- Modelo de ejecución
- Modelo de programación

2.4.1 Modelo de plataforma

El modelo consiste en un *host* conectado a uno o más dispositivos OpenCL. Un dispositivo OpenCL está dividido en una o más unidades de cálculo (CUs) que a su vez están divididas en uno o más elementos de procesamiento (PEs). Los cálculos en un dispositivo ocurren dentro de los elementos de procesamiento. Este modelo se esquematiza en la figura 2.14 [14].

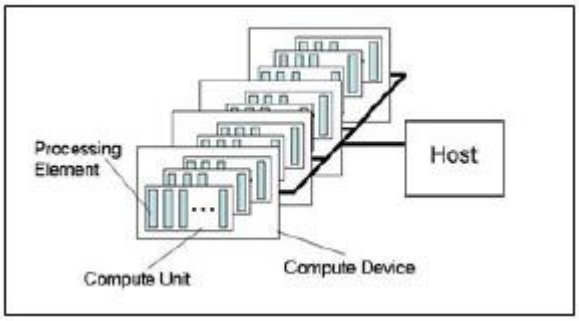

Figura 2.14. Modelo de plataforma OpenCL.

2.4.2 Modelo de ejecución

El modelo de ejecución de OpenCL se define en términos de dos unidades de ejecución distintas: los *kernels* que se ejecutan en uno o más dispositivos OpenCL y un programa anfitrión o *host* que se ejecuta en el *host*. En lo que respecta a OpenCL, los *kernels* son el lugar donde se produce el "trabajo" asociado a un

cálculo. Este trabajo se produce a través de elementos de trabajo que se ejecutan en grupos (grupos de trabajo).

Un kernel se ejecuta dentro de un contexto bien definido gestionado por el *host*. El contexto define el entorno en el que se ejecutan los *kernels*. Incluye los siguientes recursos

- Dispositivos: Uno o más dispositivos expuestos por la plataforma OpenCL.
- Objetos del *kernel*: Las funciones OpenCL con sus valores argumentales asociados que se ejecutan en los dispositivos OpenCL.
- Objetos del programa: El código fuente del programa y el ejecutable que implementan los *kernels*.
- Objetos de memoria: Las variables visibles para el *host* y los dispositivos OpenCL. Las instancias de los *kernels* operan en estos objetos mientras se ejecutan.

El programa anfitrión utiliza la API OpenCL para crear y gestionar el contexto. Las funciones de la API OpenCL permiten al *host* interactuar con un dispositivo a través de una cola de comandos. Cada cola de comandos está asociada a un único dispositivo. Los comandos colocados en la cola de comandos pueden ser de tres tipos:

- Comandos *Kernel-enqueue*: Poner en cola un *kernel* para su ejecución en un dispositivo.
- Comandos de memoria: Transfieren datos entre la memoria del *host* y del dispositivo, entre objetos de memoria, o mapear y desmapear objetos de memoria del espacio de direcciones del *host*.
- Comandos de sincronización: Puntos de sincronización explícitos que definen restricciones de orden entre comandos.

Cada comando pasa por seis estados como se resume en la figura 2.15 [14]. El conjunto de estados y de transiciones entre estados define el modelo de ejecución en términos del *host*.

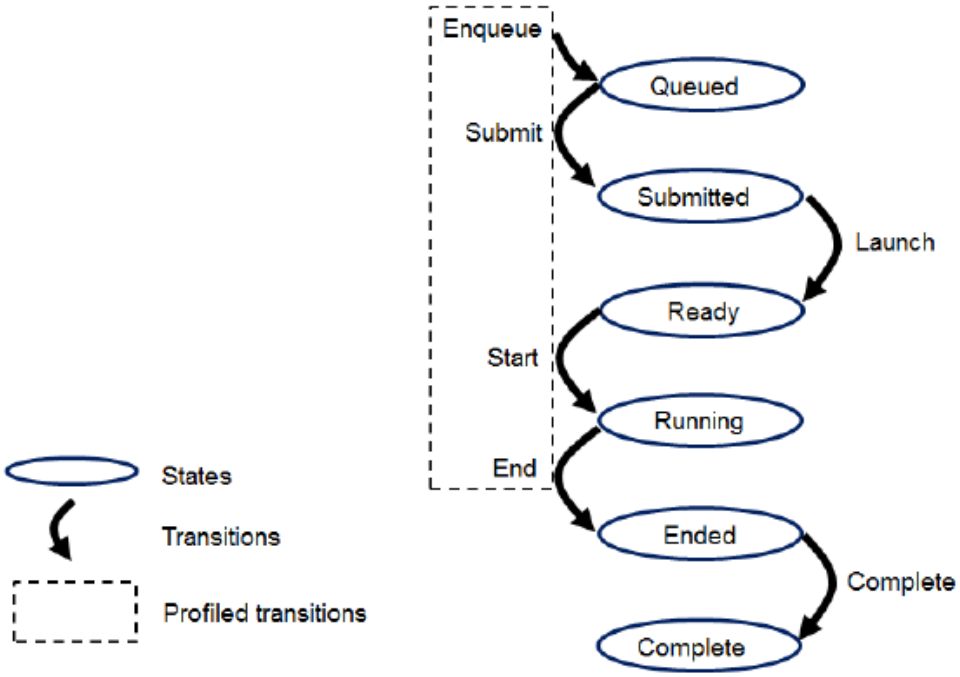

Figura 2.15. Modelo de ejecución OpenCL referido al host.

El núcleo del modelo de ejecución de OpenCL está definido por la forma en que se ejecutan los *kernels*. Cuando un comando *kernel-enqueue* envía un *kernel* para su ejecución, se define un espacio de índice. El *kernel*, los valores de los argumentos asociados al *kernel* y los parámetros que definen el espacio de índice definen una instancia del *kernel*. Cuando una instancia se ejecuta en un dispositivo, la función del *kernel* se ejecuta para cada punto del espacio de índice definido. Cada una de estas funciones que se ejecutan se denominan elemento de trabajo. Los elementos de trabajo asociados a una determinada instancia del kernel son gestionados por el dispositivo en grupos denominados grupos de trabajo.

2.4.3 Modelo de memoria

La memoria en OpenCL se divide en dos partes.

- Memoria del *host*: La memoria directamente disponible para el *host*. El comportamiento detallado de la memoria del *host* se define fuera de OpenCL. Los objetos de memoria se mueven entre el *host* y los dispositivos a través de funciones dentro de la API de OpenCL o a través de una interfaz de memoria virtual compartida.
- Memoria de dispositivo: Memoria directamente disponible para los *kernels* que se ejecutan en los dispositivos OpenCL.

La memoria de dispositivo consta de cuatro espacios de direcciones o regiones de memoria como se muestra en la figura 2.16 [14]:

- Memoria global: Esta región de memoria permite el acceso de lectura/escritura a todos los elementos de trabajo en todos los grupos de trabajo que se ejecutan en cualquier dispositivo dentro de un contexto. Los elementos de trabajo pueden leer o escribir en cualquier elemento de un objeto de memoria. Las lecturas y escrituras en la memoria global pueden ser almacenadas en caché dependiendo de las capacidades del dispositivo.
- Memoria constante: Una región de memoria global que permanece constante durante la ejecución de una instancia del *kernel*. El *host* asigna e inicializa los objetos de memoria colocados en la memoria constante.
- Memoria local: Una región de memoria local a un grupo de trabajo. Esta región de memoria se puede utilizar para asignar variables que son compartidas por todos los elementos de trabajo en ese grupo de trabajo.
- Memoria privada: Una región de memoria privada para un elemento de trabajo. Las variables definidas en la memoria privada de un elemento de trabajo no son visibles para otro elemento de trabajo.

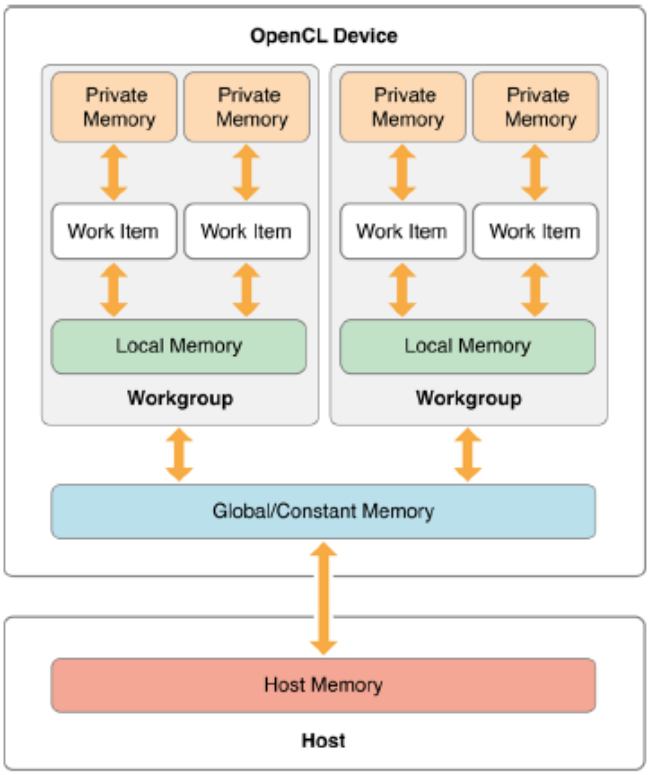

Figura 2.16. Modelo de memoria de OpenCL.

2.4.4 Modelo de programación

El marco de trabajo OpenCL permite a las aplicaciones utilizar un *host* y uno o más dispositivos OpenCL como un único sistema informático paralelo heterogéneo. El marco contiene los siguientes componentes:

- Capa de plataforma OpenCL: La capa de plataforma permite al programa *host* descubrir los dispositivos OpenCL y sus capacidades y crear contextos.
- Tiempo de ejecución de OpenCL: El tiempo de ejecución permite al programa *host* manipular los contextos una vez creados.
- Compilador OpenCL: El compilador OpenCL crea ejecutables de programa que contienen *kernels* OpenCL. El compilador de OpenCL puede crear ejecutables de programa a partir de cadenas de código fuente de OpenCL C, del lenguaje intermedio SPIR-V o de objetos binarios de programa específicos del dispositivo, en función de las capacidades de

éste. Algunas implementaciones pueden admitir otros lenguajes de *kernel* o intermedios.

CAPÍTULO 3: IMPLEMENTACIÓN SOBRE UN SISTEMA HETEROGÉNEO

3.1 TRABAJO PREVIO DE PARTIDA

En este caso el punto de partida son los resultados y el trabajo realizado en [1], es decir, que para este proyecto se cuenta con el código *software* y *hardware* detallado en la figura 3.1.

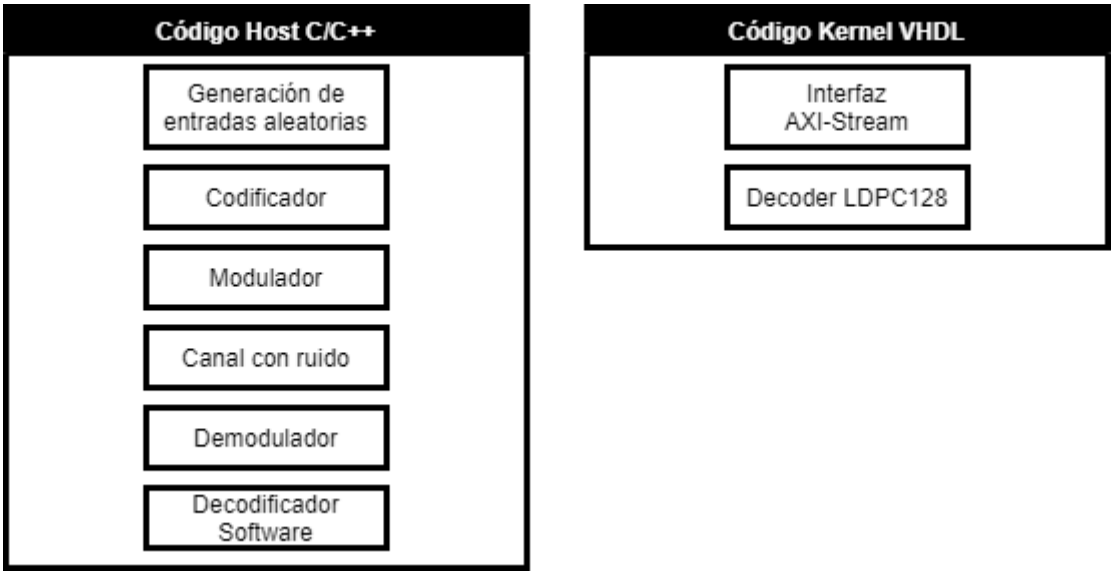

Figura 3.1. Código software y hardware disponible.

En versión *software* se dispone de funciones para la generación de datos aleatorios, codificación, modulación, la simulación de su transmisión a través de un canal con ruido, la demodulación y de una versión *software* del decodificador LDPC. En cuanto al código *hardware* se dispone del decodificador LDPC descrito mediante VHDL junto a una interfaz *AXI-Stream*. El decodificador *hardware* es el elemento para verificar.

Además, el trabajo también deja una línea de investigación bastante clara en base al éxito de los resultados obtenidos. Aprovechar las capacidades para el procesamiento *software* y la lógica programable *hardware* de un sistema embebido en un chip ha supuesto una reducción en la dificultad y un aumento en la

velocidad con respecto a las técnicas de verificación tradicionales basadas en simulaciones y prototipos sobre FPGAs.

3.2 ARQUITECTURA DEL SISTEMA HETEROGENEO

El sistema propuesto como solución en este proyecto continua con esta línea de investigación. Se trata de un sistema capaz de exportar los beneficios que aportaba un sistema embebido a un sistema heterogéneo que bajo la misma idea aporta mayor capacidad y un mayor número de recursos al proceso de verificación permitiendo una mayor aceleración.

En la figura 3.2 se representa de forma gráfica el sistema utilizado. Este consta de dos partes.

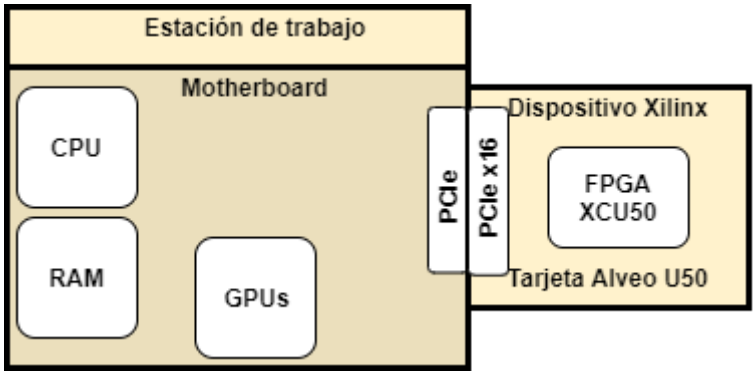

Figura 3.2. Arquitectura del sistema heterogéneo propuesto.

Una parte para el procesamiento (*software*) situado en lo que será la estación de trabajo o host. El host está compuesto entre otras cosas por un placa base con varios chips interconectados. Entre estos chips se encuentran un procesador, una memoria RAM y varias GPUs que en este proyecto no serán utilizadas. Además, consta de un bus PCIe de alta velocidad para la conexión directa de varios dispositivos a la placa base mediante una serie de ranuras PCIe, de tal forma que estos dispositivos puedan comunicarse con el procesador.

La parte del sistema que cuenta con recursos lógicos programables para la implementación *hardware* del decodificador LDPC se conecta al sistema de procesamiento mediante una de estas ranuras PCIe. Se trata de un dispositivo

Xilinx® Alveo U50 para la aceleración en el procesado de datos. Esta consta de una conexión PCIe, una serie de bancos de memoria HBM2 y una FPGA XCU50. Más detalles en la sección 2.2 de este documento.

3.3 VERIFICACIÓN SOBRE EL SISTEMA EMEBEBIDO

Una vez se ha clarificado el punto de partida, es decir, la herencia de la investigación realizada en [1] y el sistema propuesto como solución para continuar la búsqueda de nuevas soluciones se puede establecer como primer objetivo replicar sobre este sistema los dos enfoques de verificación implementados en [1] sin aplicar ninguna de las técnicas de paralelización utilizadas para la aceleración del proceso.

Antes de ver cómo se puede llevar a cabo esta replica es necesario repasar dos aspectos relacionados con la implementación sobre el sistema embebido en un chip. Por un lado, conviene repasar en qué consisten y como se llevan a cabo los dos enfoques de verificación implementados y, por otro lado, detallar cuál es el flujo o movimiento al que se ven expuestos los datos (generados aleatoriamente) durante todo el proceso de verificación.

En cuanto a los dos enfoques, por un lado, se tiene la verificación mediante la evaluación de un parámetro de rendimiento muy importante en los decodificadores como es la CER y, por otro lado, se tiene la verificación por comparación de los resultados de la decodificación *hardware* a los de un modelo de referencia como puede ser la versión *software* del decodificador.

3.3.1 Verificación mediante el cálculo de la CER

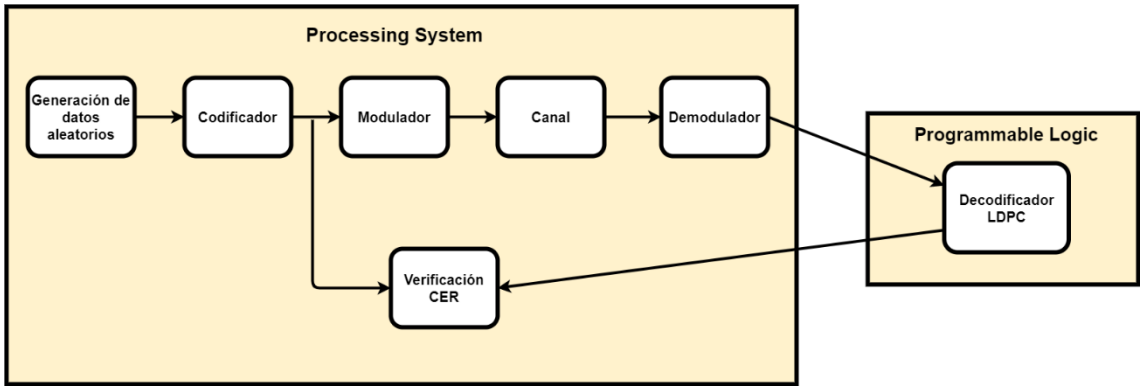

Figura 3.3. Esquema de la verificación basada en el cálculo de la CER.

En el primer enfoque, representado en la figura 3.3, se realiza el cálculo para un rango de valores de Eb/No comprendido entre cero y seis. En cada valor de Eb/No se evalúan aproximadamente tantas *codewords* como se necesiten para alcanzar cien *codewords* erróneas. Este número de *codewords* es conocido a priori y se tomará como constante con el valor que se muestra en la siguiente tabla.

Eb/No	0	1	2	3	4	5	6
Codewords	1000	1000	1000	10000	100000	1000000	100000000

Tabla 3.1. Número de codewords para cada Eb/No.

El cálculo de la CER se realiza por comparación entre la *codeword* generada por el codificador y la *codeword* devuelta por el decodificador LDPC *hardware*. Para que la corrección de errores aplicada por el decodificador LDPC tenga éxito deben coincidir las *codewords* comparadas. En caso de que no coincidan ambas *codewords* se cuenta como error. Llevando la cuenta del número de *codewords* erróneas y dividendo por el número de *codewords* evaluadas se obtiene la CER para cada iteración de Eb/No.

3.3.2 Verificación mediante modelo de referencia

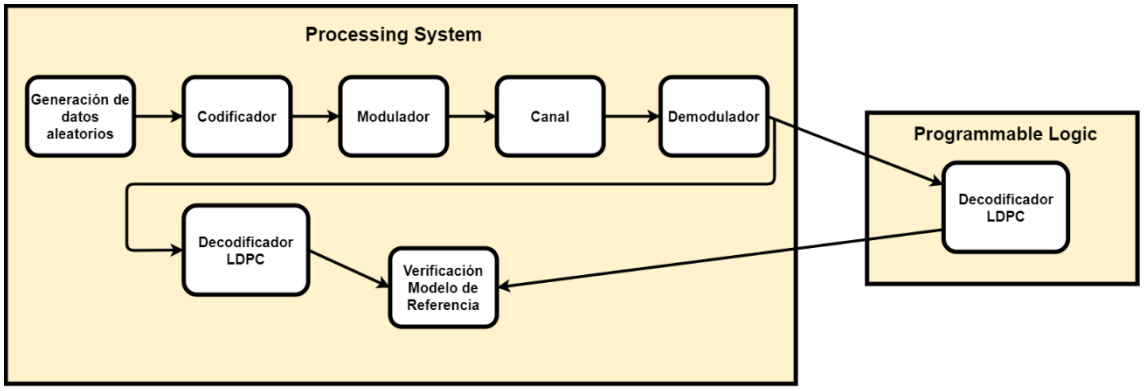

Figura 3.4. Esquema de la verificación basada en la comparación con un modelo de referencia.

En el segundo enfoque, representado en la figura 3.4, de nuevo se realiza el cálculo para un rango de valores de Eb/No comprendido entre cero y seis. En cada valor de Eb/No se evalúan 10^5 *codewords* reduciendo considerablemente el número de *codewords* a tratar con respecto al primer enfoque. Cada *codeword* es evaluada y corregida tanto por la versión *software* como por la versión *hardware* del decodificador LDPC. En cada iteración la salida de ambos codificadores debe ser idéntica para verificar que el diseño RTL del decodificador LDPC es correcto. Las salidas a evaluar pueden ser las primarias y/o también otros valores que se consideren de interés.

3.3.3 Modelo de ejecución (trayecto de una *codeword*)

En ambos enfoques el proceso que engloba desde la generación de datos hasta la corrección de errores realizada por la versión *hardware* del decodificador LDPC es el mismo. Todo comienza por la generación de 64 bits de información. Estos se introducen en el codificador que añade otros 64 bits de redundancia. El resultado de codificar 64 bits de información es una *codeword* de 128 bits. El modulador BPSK mapea cada bit de la *codeword* al símbolo correspondiente. La función de transmisión modela el canal añadiendo ruido sobre cada símbolo. Para demodular los símbolos con ruido se realiza una cuantificación de ocho niveles (3 bits). Esto quiere decir que el decodificador LDPC hardware recibe una

codeword con ruido, la corrige y genera a su salida una *codeword* de 128 bits. La implementación *software* codifica los bits como enteros de 32 bits. Para un solo bit usa sólo el menos significativo del entero. Para tres bits, los tres menos significativos. En la implementación *hardware/software*, el decodificador LDPC recibe 128 enteros de 32 bits, seleccionando los 3 bits menos significativos de cada uno. Tras operar con ellos devuelve a su salida una *codeword* corregida de 128 bits. De nuevo se almacena cada bit de la *codeword* corregida como un entero de 32 bits donde solo contiene información el bit menos significativo. Este proceso se describe gráficamente en la figura 3.5.

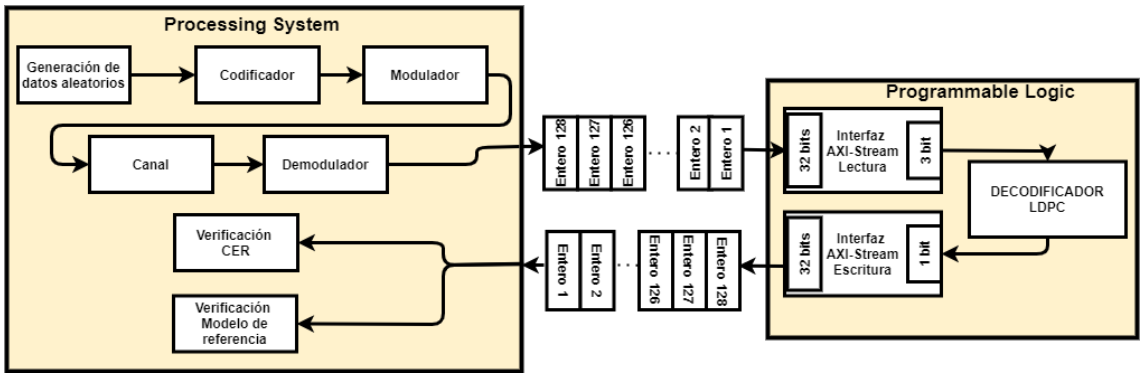

Figura 3.5. Modelo de ejecución del SoC.

3.4 VERIFICACIÓN SOBRE EL SISTEMA HETEROGENEO

Cualquier aplicación desarrollada para su ejecución sobre un sistema heterogéneo como el utilizado en este proyecto consta de dos partes muy claras y que, atendiendo a la terminología de la tecnología utilizada, se conocen como *host* y *kernel* en referencia al código *software* y *hardware* respectivamente.

La aplicación se desarrolla con ayuda del entorno Vitis™ de Xilinx® y más concretamente utilizando su entorno de desarrollo integrado (IDE) que ofrece facilidades para la generación y construcción tanto del *host* como del *kernel*. El código *software* y *hardware* heredado se encuentra escrito en C y VHDL respectivamente y se decide utilizar la API de OpenCL para gestionar la interacción entre *host* y el *kernel*.

3.4.1 Justificación del modelo de programación

El modelo de programación que describe por completo el comportamiento de la aplicación queda definido en torno a dos modelos de ejecución: el modelo de ejecución impuesto por el propio lenguaje en el que se desarrolla el host que en este caso es C/C++ y el modelo de ejecución invocado al hacer uso de la API de OpenCL. Este último se define en el estándar de OpenCL diferenciando en términos de dos unidades de ejecución como son el *host* y el *kernel*.

Modelo de ejecución del host (C++/OpenCL)

El hecho de que OpenCL sea un estándar muy enfocado en la interacción con dispositivos con una arquitectura y estructura fija como pueden ser las CPUs y las GPUs trae asociado una serie de desventajas para su uso con dispositivos sin arquitectura preconfigurada como es el caso de las FPGAs. Estas se pueden ilustrar de forma clara con un ejemplo.

Ejemplos de dispositivos de una plataforma OpenCL pueden ser una GPU NVidia o una GPU AMD. Estos poseen al menos una unidad de cómputo que puede ser un multiprocesador de flujo en una GPU NVidia o un motor SIMD en una GPU AMD. Del mismo modo cada unidad de cómputo está compuesta por varios elementos de procesamiento (ALU/procesador de flujo). Sin embargo, para una plataforma OpenCL un dispositivo con una FPGA como es el caso de la tarjeta Alveo™ U50 solo posee una unidad de cómputo con un único elemento de procesamiento.

Este problema se refleja directamente sobre el modelo de ejecución definido para los *kernels* por el estándar de OpenCL. Se trata de un modelo de ejecución muy pensado para su aplicación sobre GPUs. En él la ejecución de los *kernels* OpenCL se realiza mediante el concepto de *work ítems* agrupados en *work groups* que son mapeados sobre un *index space* previamente definido por un *NDRange*. El concepto de *work item* se puede asemejar de algún modo al concepto de hilo mientras que el concepto *NDRange* define el tamaño de los bloques de memoria

de los que se hará cargo cada *work group*. Este modelo es válido gracias a la estructura y arquitectura fijas que presentan este tipo de dispositivos. En el caso de una FPGA no existe una estructura fija sobre la que el sistema de ejecución de OpenCL pueda asignar la ejecución de un determinado *work item*. Sobre una FPGA, OpenCL construye el programa implementando sobre la FPGA el binario que incluye el o los *kernels* diseñados y entiende que dicha implementación constituye en sí misma una unida de cómputo con un único elemento de procesamiento. Por ello, independientemente del número de *kernels* o instancias de un mismo *kernel* incluidos en el *xclbin* con el que se realizó la implementación sobre la FPGA, OpenCL solo es capaz de asignar un único *work item* sobre el que mapea el *index space* al completo independientemente de cualquier definición previa que se haya realizado del *NDRange*. El modelo de ejecución que define el estándar de OpenCL para los *kernels* carece de validez en este tipo de dispositivos, así como muchas ideas expuestas en los distintos modelos que definen por completo el funcionamiento de una plataforma OpenCL.

Analizar cuál es la principal función para la que fue diseñado cada dispositivo ayuda a entender por qué el modelo de ejecución de OpenCL, diseñado para explotar la principal ventaja de las GPUs, no es de mucha utilidad al ser aplicado sobre una FPGA. La principal ventaja que aportan las GPUs reside en el hecho de que poseen una arquitectura muy específica buscando aprovechar las ventajas del paradigma para la paralelización de datos (SIMD). En cambio, las FPGAs facilitan la implementación de arquitecturas muy concretas para la operación de un determinado algoritmo que permita obtener una eficiencia muy superior a la que tendría ese mismo algoritmo ejecutándose sobre una GPU o una CPU de propósito general.

Cualquier característica para la paralelización SIMD que ofrezca OpenCL no se puede aprovechar sobre dispositivos basados en FPGAs. Esto provoca que cualquier tipo de paralelización que se pretenda implementar sobre una FPGA

tenga que ser explotada manualmente. Al final el uso de OpenCL para este tipo de dispositivos se restringe únicamente al de una herramienta para la interacción con el dispositivo.

La conclusión que se puede obtener de todo esto se resume en que desde la perspectiva del host no es posible controlar la ejecución de los *kernels* bajo el modelo de ejecución de OpenCL.

Modelo de ejecución del kernel (XRT/AP_CTRL_HS)

En cualquier caso, para entender el comportamiento del *kernel* basta con eliminar el nivel de abstracción dispuesto por OpenCL y recordar el modelo de programación típico de una aplicación soportada por el entorno de Vitis™. Del mismo modo que el del estándar de OpenCL, se trata de un modelo de programación que describe por completo el comportamiento de la aplicación diferenciando en términos de dos unidades de ejecución como son el *host* y el *kernel*. La diferencia es que en este caso existen diferentes modelos de ejecución dependiendo del grado de control que se pretenda atesorar desde el *host* hacia el kernel. Esto lleva a diferenciar entre *kernels* controlados por *software* y *kernels* no controlados por *software*.

El hecho de que el *kernel* heredado sea un decodificador con una interfaz *AXI-Stream* invita a intentar aprovechar el modelo de ejecución propuesto para los *kernels* no controlados por *software* donde el único requisito impuesto por el entorno de Vitis™ es la presencia de una interfaz *AXI-Stream* para la recepción y transmisión de ráfagas de datos. Se trata de un modelo de ejecución donde el poco control o sincronización entre *host* y *kernel* se lleva a cabo a través de señales propias del protocolo *AXI-Stream* en la misma interfaz dedicada para el intercambio de datos. Por ello, se dice que son *kernel* sin control *software* en los que no es necesario ninguna API en el *host*. Sin embargo, pese a lo idóneo que resulta este modelo de ejecución para el *kernel* heredado no es posible su implementación directa, debido a la tecnología utilizada para los accesos directos

a memoria en la plataforma **Xilinx_u50_gen3x16_xdma_201920_3** que se utiliza en este proyecto. Esta plataforma utiliza un motor XDMA para llevar a cabo el intercambio de datos entre *host* y *kernel*. Lo hace mediante el traslado de estos desde la memoria de un dispositivo a la del otro, es decir, no posee las funcionalidad necesaria para el *stream* de datos. La tecnología habilitada con esta capacidad se conoce como QDMA y se encuentra disponible en algunas plataformas de Xilinx®. Lamentablemente, es una tecnología que no está disponible en ninguna de las tres plataformas diseñada hasta el momento para la tarjeta Alveo™ U50.

Este contratiempo obliga a explorar la opción de modificar el *kernel* heredado para que se adecue a uno de los dos tipos de *kernels* controlados por *software* válidos. Es conveniente recordar que al estar trabajando con dispositivos de Xilinx® el control se ejerce con ayuda de XRT desde la perspectiva del host. Dependiendo del modelo de ejecución o protocolo de control que el usuario pretenda implementar puede utilizar la propia API de XRT o puede hacerlo mediante escrituras y lecturas de registros. En el primer caso existen dos modelos de ejecución bien definidos y en el segundo caso es posible implementar cualquier modelo de ejecución que se le ocurra al usuario.

La segunda opción es muy útil para diseños RTL como es el caso del kernel de este proyecto. Sin embargo, requiere de un mayor conocimiento y supone mayor grado de dificultad, además de que términos prácticos no supone un beneficio muy superior al de utilizar alguno de los modelos de ejecución predefinidos por la API de XRT.

Por ese motivo se decide utilizar uno de los dos modelos de ejecución que se encuentran automatizados al hacer uso de la API de XRT. Además, utilizar estos modelo de ejecución habilita el uso de APIs de mayor nivel como OpenCL a las que XRT da soporte. Los modelos de ejecución en cuestión son conocidos como *AP_CTRL_HS* y *AP_CTRL_CHAIN* y aunque el segundo quizás permite una

mayor velocidad se decide implementar el primero para tener un control total sobre cada ejecución del *kernel*.

3.4.2 Adaptación del *kernel*

La implementación del modelo de ejecución *AP_CTRL_HS* necesita de la adaptación del *kernel* heredado a uno con las interfaces y el protocolo de control requeridos por el entorno de Vitis™. Esto se puede llevar a cabo de forma manual implementando la lógica y las interfaces mediante un lenguaje de descripción *hardware* o con ayuda de la herramienta *RTL Kernel Wizard* integrada en el IDE de Vitis™. Utilizar esta herramienta facilita los dos primeros pasos del flujo de trabajo descrito en la figura 3.6.

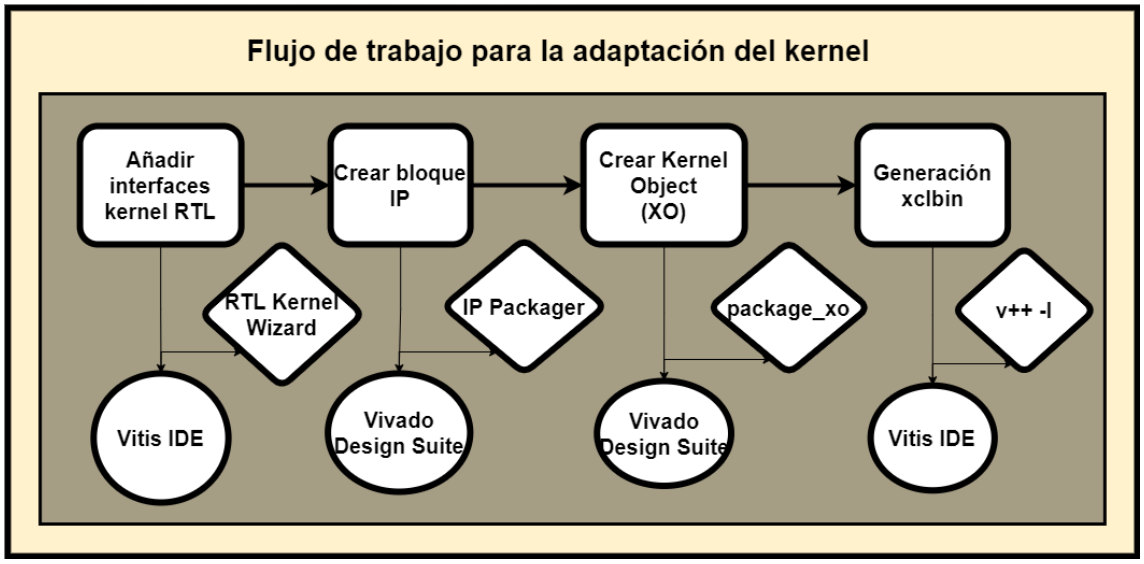

Figura 3.6. Flujo de trabajo para la adaptación del kernel.

Se trata de una herramienta que asiste al usuario en el proceso de generación de un bloque IP que pueda ser empaquetado como un fichero *kernel object* (XO) válido para el compilador de Vitis™. La herramienta obliga al usuario a especificar entre otras cosas el tipo de *kernel* que se pretende generar, un modelo de ejecución y otras cuestiones relacionadas con los argumentos del *kernel*.

La opción para especificar el tipo de *kernel* deseado ofrece dos posibilidades: un *kernel* RTL o un diseño de bloques. La elección de un *kernel* RTL resulta en la

generación de un *kernel* descrito a nivel de transferencia de registros mediante la mezcla de dos lenguajes de descripción *hardware* como son Verilog y SystemVerilog. El *kernel* consiste en un módulo de nivel superior descrito en Verilog que funciona como envoltorio de dos módulos descritos en SystemVerilog: un módulo que implementa la interfaz de control junto con una variedad de registros y un módulo que funciona como envoltorio de un *kernel* ejemplo junto con la implementación de una interfaz para el intercambio de datos. La estructura descrita del módulo de nivel superior generado se muestra gráficamente en la figura 3.7.

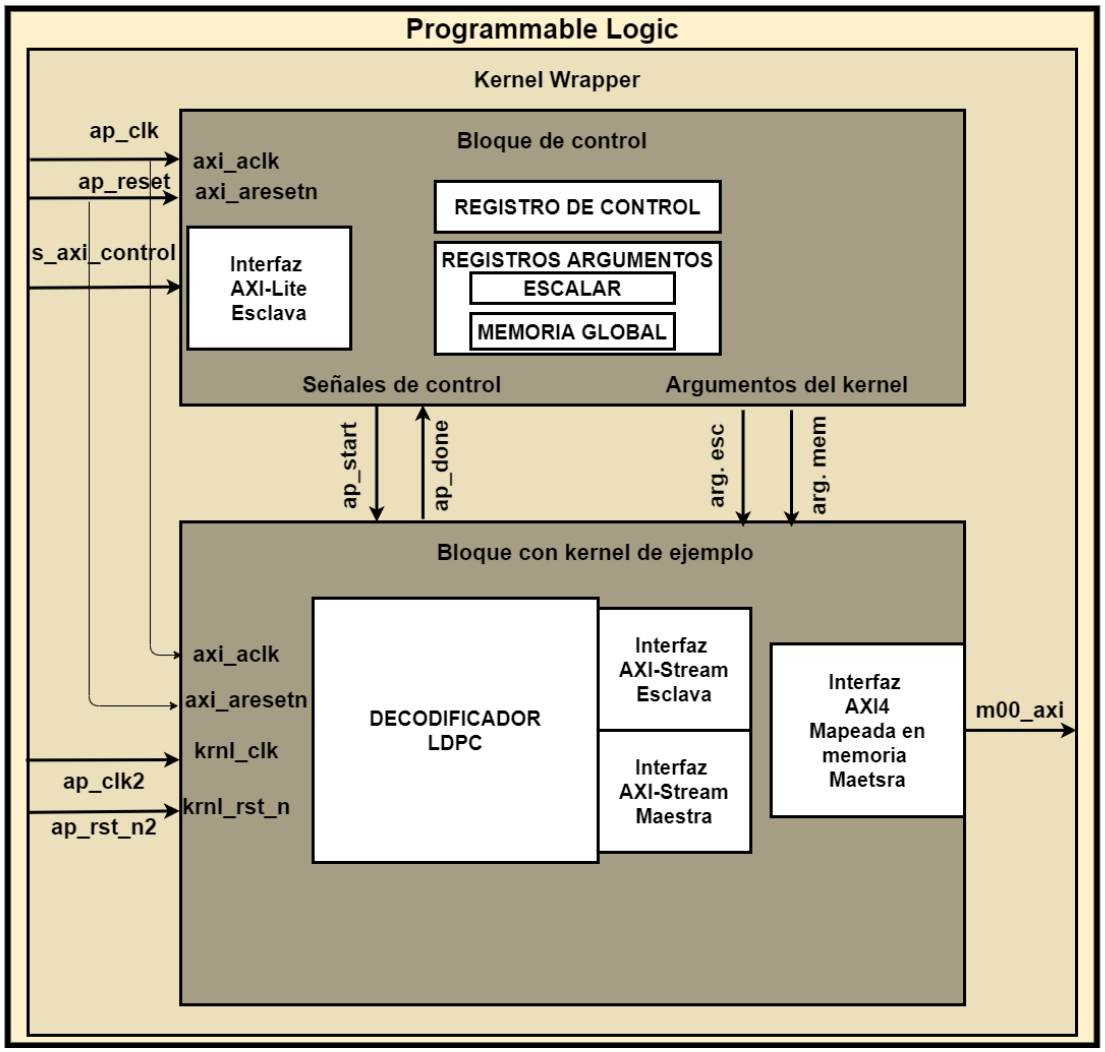

Figura 3.7. Diagrama de bloques de la adaptación del kernel.

El modelo de ejecución seleccionado permite a la herramienta generar el registro de control con las señales y la lógica necesaria para implementar dicho modelo.

La elección de un modelo de ejecución *AP_CTRL_HS* supone que la lógica generada selecciona del registro de control un par de bits que serán utilizados como señales de comienzo y final de la ejecución del *kernel*. El bloque con el *kernel* de ejemplo comenzará su ejecución cuando reciba la señal *ap_start* transmitida desde el bloque de control y este activará y transmitirá la señal *ap_stop* en dirección al bloque de control cuando haya finalizado su ejecución. Además, al tratarse de un modelo de ejecución que posee un registro de control la herramienta incluye una interfaz para permitir el acceso a este que implementa como una interfaz *AXI-Lite*.

La herramienta permite seleccionar el número de argumentos que tendrá el *kernel*. Cada argumento se implementa como un registro en el bloque de control y se utiliza de una forma u otra en el bloque con el *kernel* de ejemplo. En primer lugar, ofrece la posibilidad de seleccionar el número de argumentos escalares que se desean implementar. Al menos se ha de implementar uno, aunque luego no sea utilizado. En segundo lugar, la herramienta permite seleccionar el número de interfaces de datos que se desea implementar junto al número de argumentos que se desea asociar a cada una de estas interfaces. Cada interfaz se implementa como una interfaz maestra *AXI4* mapeada en memoria, esto quiere decir que sirve como conexión directa entre el *kernel* y la memoria del dispositivo. Los argumentos asociados a cada interfaz son implementados como registros en el bloque de control y son utilizados por sus respectivas interfaces de datos en el otro bloque como dirección base de memoria para el acceso a los datos sobre los que debe operar el *kernel*. Esto quiere decir que este tipo de registros almacenan punteros a objetos de memoria y en este caso concreto punteros a *buffers* en el espacio de direcciones del dispositivo. Además, el acceso a los registros de argumentos se realiza a través de la interfaz *AXI-Lite* del bloque de control del mismo modo que se hace para el registro de control.

Por último, la herramienta muestra un resumen de la información que utiliza para empaquetar el módulo de nivel superior del *kernel* como un bloque IP. Entre esta información se incluye el VLNV del IP del *kernel* RTL, los registros y el mapa de direcciones de los registros con información detallada como el ID del *software* anfitrión, el nombre del argumento, el *offset* respecto a la dirección base de los registros, el tipo y la interfaz asociada para cada registro.

Una vez se finaliza todo este proceso se abre un proyecto en el IDE de *Vivado Design Suite* con el bloque IP generado. El proyecto contiene entre otras cosa el código RTL del *kernel* completo incluyendo el módulo que funciona como envoltorio y los dos bloques descritos. El bloque con el *kernel* de ejemplo ha de ser modificado para sustituir el *kernel* de ejemplo por el *kernel* del decodificador LDPC. El hecho de que este esté descrito en VHDL y el bloque que lo engloba lo esté en SystemVerilog no supone ningún problema puesto que SystemVerilog permite instanciar entidades de VHDL. La implementación de la interfaz de datos mapeada en memoria incluye lógica para soportar la conversión de los datos recibidos en ráfagas de datos válidas para la instancia VHDL del decodificador y viceversa. Esta ´lógica se basa en la presencia de una fifo con capacidad para almacenar 256 enteros de 32 bits. Sin embargo, su comportamiento se rige por las señales de control *tready*, *tvalid* y *tlast* del protocolo *AXI-Stream* que desde la instancia del decodificador indican el tamaño de las ráfagas que han de ser almacenadas en la fifo antes de ser transmitidas entre interfaces.

Una vez se finaliza la modificación del proyecto se posee un bloque IP del módulo de nivel superior del *kernel* RTL y un archivo *kernel.xml* que recoge información necesaria para poder empaquetar el bloque IP como un *kernel object* (XO). Para este paso se puede utilizar la herramienta *package_xo* integrada en el IDE de Vivado de forma manual o se puede utilizar el comando *Generate RTL Kernel* disponible en uno de los menús del IDE de *Vivado Design Suite*. Este

comando utiliza la herramienta *package_xo* pero automatiza los argumentos que se le pasan a esta, escogiendo directamente el IP, el fichero *kernel.xml* y la ruta de salida del fichero *kernel object* (XO) para que pueda ser accedido desde el IDE de Vitis™ por el compilador v++ en el proceso de construcción del fichero *xclbin*.

3.4.3 Adaptación del *host*

Desde la perspectiva del *host* el modelo de ejecución descrito por el estándar de OpenCL sí es válido, por lo que todas la funciones *software* referidas a la generación y acondicionamiento de *codewords* para su utilización como entradas del decodificador han de ser complementadas con el *framework* de OpenCL.

Este proceso de adaptación se puede clasificar en el seis pasos divididos en tres bloques.

El primer bloque esquematizado en la figura 3.8 engloba el descubrimiento de dispositivos con soporte para OpenCL y la construcción del programa que será ejecutado sobre dichos dispositivos. Para el descubrimiento y selección de algún dispositivo previamente se ha de descubrir y seleccionar la plataforma OpenCL. La creación de un contexto permite al sistema de ejecución de OpenCL manipular objetos de uno o varios dispositivos dentro de un mismo entorno. En este caso solo se trabajará con un dispositivo OpenCL por lo que simplemente da paso a la creación y manipulación de objetos. En primer lugar, se crea un objeto programa.

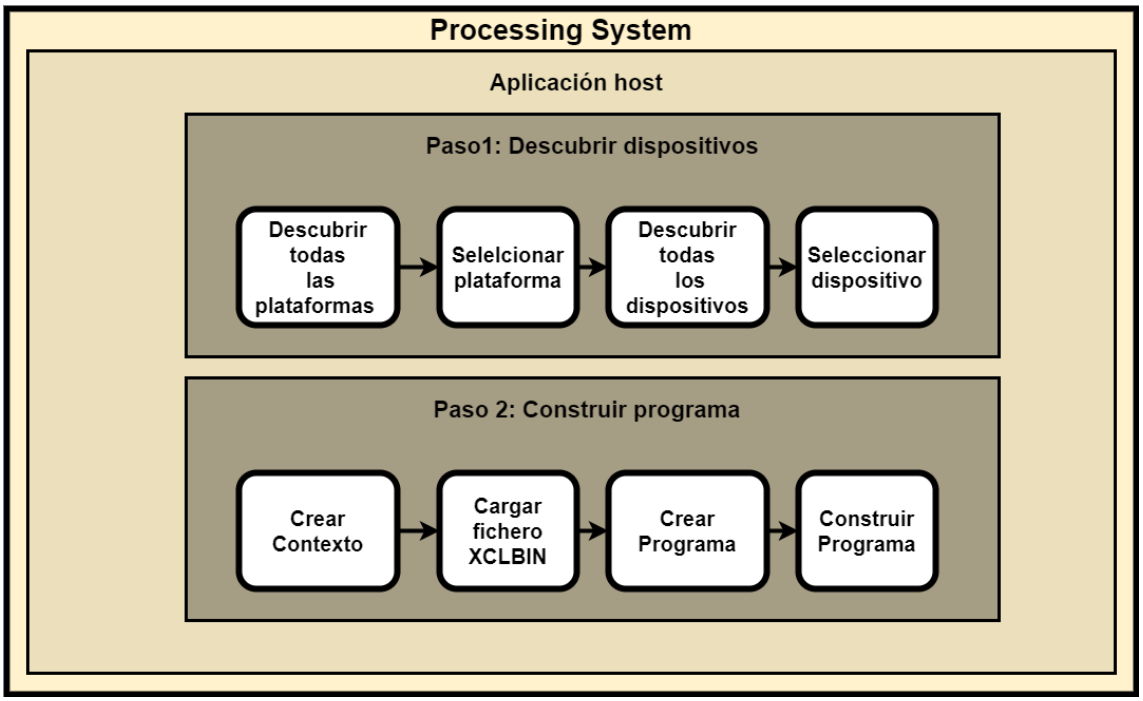

Figura 3.8. Diagrama de bloques 1 de la adaptación del host.

Para la creación de un programa se necesita el *kernel*, que puede haber sido compilado previamente o hallarse en su condigo fuente para su compilación *online*. En nuestro caso el *kernel* ha sido compilado *offline* y se incluye en el fichero *xclbin* almacenado en el disco. Por lo que antes de crear el programa se carga el fichero *xclbin* a la memoria del *host*. Una vez ha sido cargado se crea y se construye el programa dentro del contexto especificado.

El segundo bloque esquematizado en la figura 3.9 abarca lo referido a la creación y a la manipulación de objetos de memoria. Primero se crean y se rellenan los objetos de memoria en la memoria del *host*. También se generan y se adecuan todas los datos que se utilizarán como entradas del decodificador.

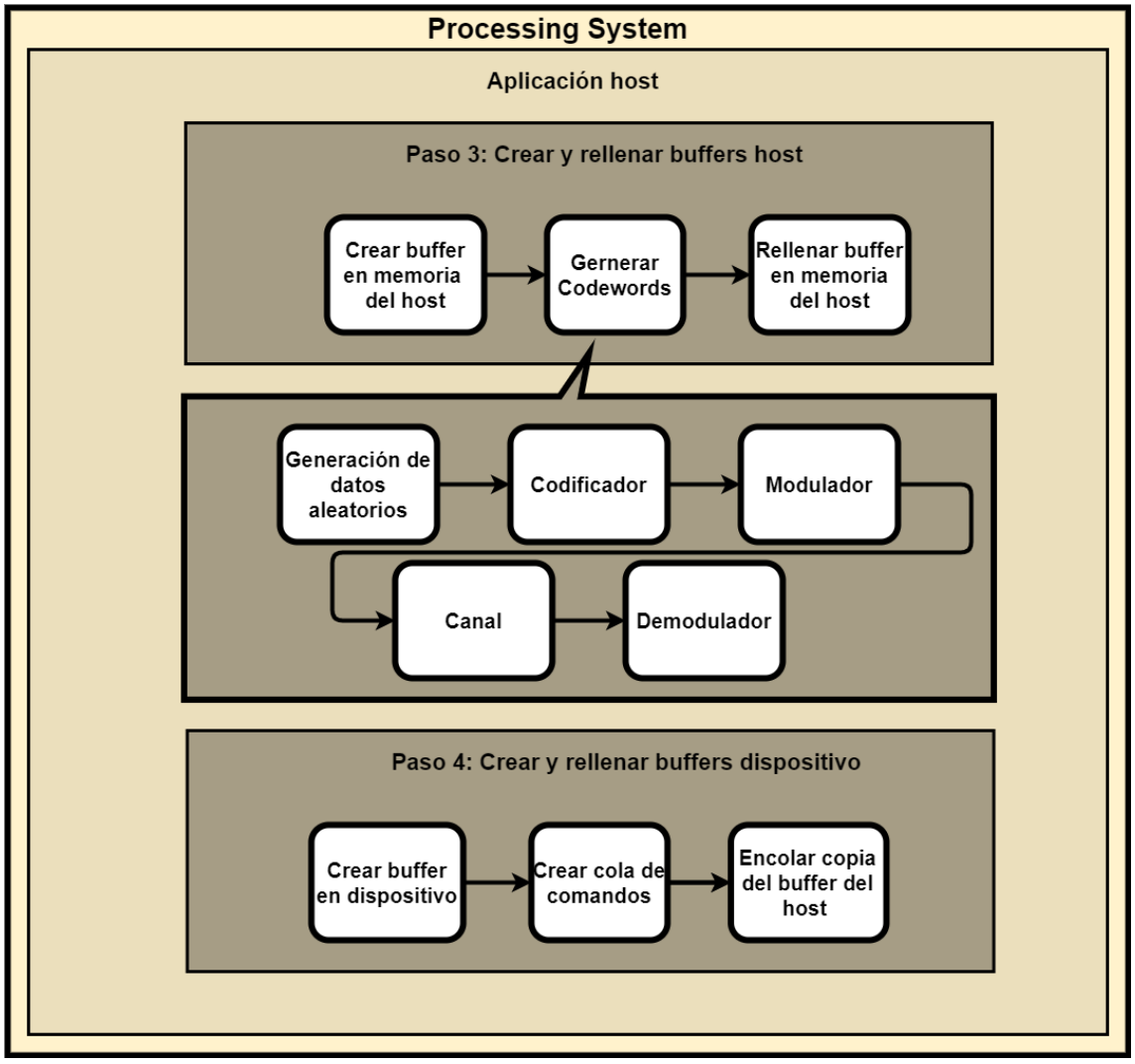

Figura 3.9. Diagrama de bloques 2 de la adaptación del host.

La creación de un *buffer* en el *host* se lleva a cabo mediante la reserva dinámica de memoria. Se debe reservar memoria suficiente para almacenar el número de *codewords* que se pretende enviar a la memoria del dispositivo. En este caso se reserva memoria para 512 *bytes* que es el espacio suficiente para almacenar una *codeword* completa. La creación de un *buffer* en la memoria del dispositivo se realiza mediante una llamada específica a la API de OpenCL. Como resultado de esta llamada el **host** obtiene un puntero a una dirección de memoria del espacio de direcciones del dispositivo. Este puntero se necesita para manipular o realizar cualquier modificación sobre el objeto localizado en el dispositivo. Sin embargo, no es suficiente con dicho puntero puesto que para realizar cualquier operación

en el dispositivo es necesaria la presencia de otro objeto sobre el que se debe encolar cualquier comando o instrucción dirigido al dispositivo. Una vez se dispone de los tres objetos, el puntero al *buffer* del *host*, el puntero al *buffer* del dispositivo y la cola de comandos, se utiliza una OpenCL para encolar un comando que se encargue de copiar el contenido del *buffer* del *host* al *buffer* del dispositivo.

El último bloque esquematizado en la figura 3.10 hace referencia a la ejecución del *kernel* y a la verificación del decodificador en base a los resultados obtenidos.

La creación de un objeto *kernel* permite identificar la función que se pretende ejecutar dentro del objeto programa. Normalmente el programa que se construye en pasos previos contiene múltiples *kernels* por lo que para ejecutar uno en concreto es necesario identificarlo de algún modo en el *host*. En nuestro caso el programa que se ha construido está compuesto por un único *kernel* por lo que solo hace falta crear un objeto *kernel* que lo identifique. Se debe recordar que el *kernel* necesita de una serie de argumentos que le permiten acceder a los datos u objetos cargados en la memoria del dispositivo. Estos argumentos deben asociarse al *kernel* antes de comenzar su ejecución mediante una función de OpenCL. Para el *kernel* utilizado basta con asociar el puntero al *buffer* del dispositivo con el objeto kernel recién creado. Del mismo modo que para realizar cualquier operación sobre un objeto en la memoria del dispositivo se necesita encolar un comando, también lo es necesario para realizar cualquier operación sobre el *kernel* construido en el dispositivo. Por ello se utiliza la misma cola de comandos creada para copiar el contenido de un *buffer* a otro, ahora para encolar un comando que le indique al *kernel* cuando comenzar su ejecución. Es en este punto donde el modelo de ejecución previsto en el diseño del *kernel* es aplicado internamente por la API utilizada para la abstracción. En este caso, la API de forma totalmente transparente para el usuario envía los argumentos a través de la interfaz *AXI-Lite*. Estos son almacenados en los registros correspondientes por

el *kernel* para que justo después la API, de nuevo de forma totalmente transparente para el usuario, escriba sobre el registro de control las señales adecuadas al protocolo de control implementado como modelo de ejecución. En este caso, la API activa la señal *ap_start* del registro de control.

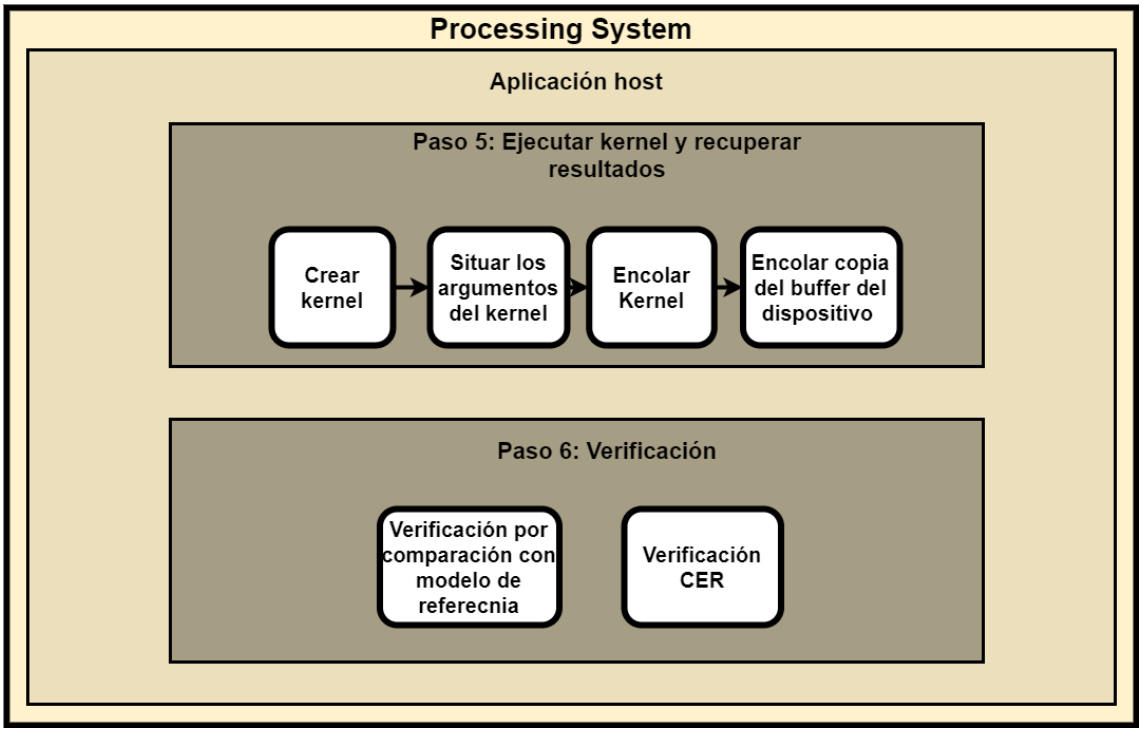

Figura 3.10. Diagrama de bloques 3 de la adaptación del host.

Durante su ejecución el *kernel* lee secuencialmente 128 enteros de 32 bits del buffer cargado en la memoria del dispositivo, opera con ellas y las escribe de vuelta al *buffer*. Desde el punto de vista del *host* para recuperar los resultados basta con copiar el contenido del *buffer* del dispositivo al *buffer* del *host*. De nuevo, al tratarse de una operación sobre un objeto alojado en el dispositivo es necesario utilizar la cola de comandos previamente creada que junto con ambos objetos de memoria permite encolar el comando adecuado para realizar esta copia. Tras encolar el comando, el *host* continúa con su ejecución por lo que cualquier operación en el *host* que implique el uso de este *buffer* puede acarrear problemas de sincronización. Para evitar estos problemas basta con introducir el uso de un objeto que capture el evento de fin de escritura de una memoria a otra. De este

modo se asegura que el *host* espere a terminar el traspaso de los resultados de un *buffer* a otro antes de comenzar con el proceso de verificación.

El último paso consiste en verificar si el funcionamiento del decodificador es o no el esperado mediante la implementación de los dos enfoques explicados en los subapartados previos.

3.5 EJECUCION DE LA APLICACIÓN

Vitis™ cuenta con un asistente que automatiza el proceso de construcción de la aplicación en tres fases: la construcción del enlace *hardware-software* del sistema, la construcción del *kernel* y la construcción del *host*. Además, el IDE de Vitis™ ofrece una variedad de opciones para la configuración de cada una de estas tres fases de construcción. La opción más importante en términos de desarrollo es la posibilidad de escoger el objetivo de compilación. Este define la naturaleza y el contenido del binario FPGA (*xclbin*) creado durante la compilación y el enlace del *kernel*. Existen tres objetivos de compilación diferentes: dos objetivos de emulación utilizados para fines de validación y depuración: emulación de *software* y emulación de *hardware*, y el objetivo de *hardware* del sistema por defecto utilizado para generar el binario FPGA (*xclbin*) que se carga en el dispositivo Xilinx®. Es importante destacar que la compilación para un objetivo de emulación es significativamente más rápida que la compilación para el *hardware* real. La ejecución de la emulación se realiza en un entorno de simulación, que ofrece una mayor visibilidad de depuración y no requiere una tarjeta aceleradora real.

3.5.1 Emulación *hardware*

Con el fin de validar y depurar la implementación del sistema se construye la aplicación inicialmente para el objetivo de emulación *hardware*. Durante la emulación *hardware*, el *kernel* se compila a un modelo RTL que se ejecuta en el simulador lógico de Vivado. Al tratarse de una emulación *hardware* sobre un

conjunto muy grande de datos la duración puede extenderse durante días haciendo que sea inviable completar en un tiempo razonable los dos enfoques de verificación previstos. Por ello, se realiza una emulación sobre un conjunto de datos muy reducido con el objetivo de corroborar el correcto funcionamiento de la aplicación desarrollada. Esto quiere decir que sin llegar a verificar de forma completa el decodificador, la emulación *hardware* permite validar la aplicación que se plantea para la verificación del decodificador.

Configuración para la depuración, perfilado y estimación de tiempos

Gran parte de las características de validación y depuración atribuidas a la emulación *hardware* se deben a su capacidad para recopilar datos de la actividad que tiene lugar en el dispositivo de aceleración. La recopilación de transacciones entre la memoria del dispositivo y el *kernel* permite validar que la interacciones entre el *host* y el *kernel* siguen el itinerario esperado, mientras que la recopilación de la actividad interna del kernel puede ser utilizada con fines depurativos del propio funcionamiento del decodificador. En nuestro caso al estar utilizando un decodificador cuyo funcionamiento ha sido comprobado con anterioridad, no es necesario hacer uso de esta característica y basta con asegurar que los datos que llegan y salen del decodificador se corresponden con los esperados.

La recopilación de este tipo de datos se activa durante la construcción del kernel mediante la opción -g del compilador v++. Los datos capturados se representan en la ventana de formas de onda del simulador *hardware* de Vivado. La representación puede ser durante la emulación o tras su finalización. En el primer caso se despliega la ventana de formas de onda mientras se produce la emulación de la aplicación. En el segundo caso los datos son almacenados en un fichero con formato de formas de onda para ser posteriormente abiertos y analizados desde Vivado.

Cada transacción entre la memoria del dispositivo y el *kernel* se compone por 256 transferencias. El tamaño de cada transferencia viene delimitado por la anchura

de canal de datos que en este caso se corresponde a 32 bits. Esto quiere decir que en cada transacción se transmiten dos *codewords* completas. Sin embargo, la aplicación ha sido diseñada para que desde el host se genere, se copie entre memorias y se ejecute cada *codeword* de forma individual lo que se traduce en que cada transacción solo transporta una *codeword* de interés real. En la figura 3.11 se muestran las primera transferencias de la primera transacción entre memoria y *kernel*.

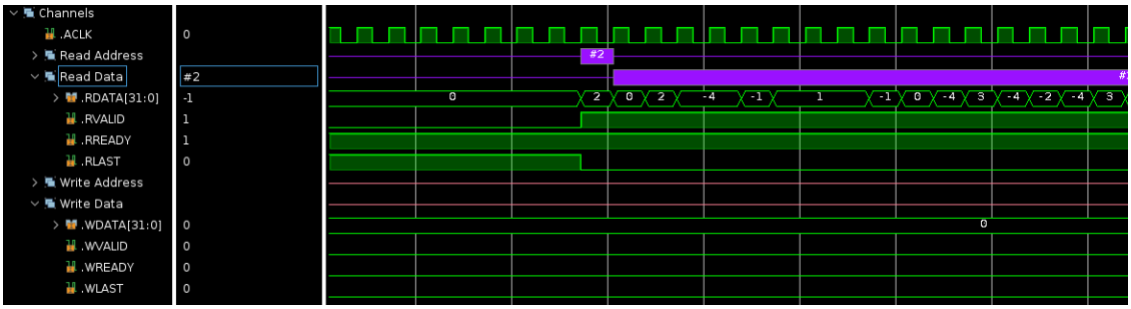

Figura 3.11. Codeword leída por el kernel.

La salida correspondiente a esta *codeword* se muestra en la figura 3.12 y se produce escaso tiempo después de recibirla.

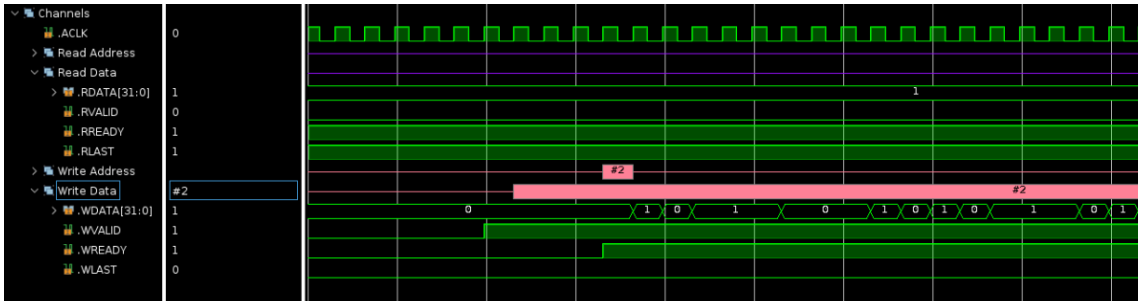

Figura 3.12. Codeword corregida escrita por el kernel.

Del mismo modo que se puede atribuir a la emulación *hardware* características de depuración también se le puede atribuir la capacidad para el perfilado y la estimación de tiempos de la aplicación.

El proceso de perfilar requiere instrumentar la aplicación de forma adecuada mediante una o varias de las siguientes modificaciones:

- Modificar la aplicación *host* para capturar datos personalizados.

- Modificar el *kernel XO* durante la compilación y el *xclbin* durante el enlace para capturar diferentes perfiles desde el dispositivo.
- Configurar el tiempo de ejecución de Xilinx® (XRT) desde el archivo *xrt.ini* para capturar datos durante el tiempo de ejecución de la aplicación.

Los datos recopilados se ponen a disposición del usuario en una variedad de informes y herramientas gráficas para su análisis.

Al igual que antes se necesita añadir la opción *-g* al compilador v++ antes de construir la aplicación, pero además se ha de configurar el XRT como se muestra en la figura 3.13 donde se han activado las opciones de *Profile, Timeline Trace y Data Transfer Trace*. Esta configuración es utilizada por el IDE de Vitis™ para generar el archivo *xrt.ini* que utiliza la aplicación al comenzar su ejecución.

XRT Configuration

Name	Key	Value
Profile	profile	true
Timeline trace	timeline_trace	true
Data transfer trace	data_transfer_trace	fine
Stall trace	stall_trace	off
Trace buffer size	trace_buffer_size	1M
Low-overhead profiling (LOP) trace	lop_trace	false
Continuous trace offload	continuous_trace	false
Continuous trace offload sample interval (ms)	continuous_trace_interval_ms	10
Power profile	power_profile	false
Launch waveform	launch_waveform	batch
Verbosity	verbosity	4

Figura 3.3.133. Configuración de la librería XRT.

Validación de la aplicación

La forma más sencilla de validar que la aplicación está trabajando de la forma esperada es visualizar sobre una misma línea temporal el comportamiento de ésta. En la figura 3.14 se representa la secuencia de tareas ejecutadas por el *host*. Se puede apreciar como la primera tarea se corresponde a la generación del programa a partir del binario cargado en memoria mediante la función

clCretеProgramWithBinary de la API de OpenCL. Después se repite una secuencia que engloba tres tareas: la copia entre memorias, el encolado del comando que indica el comienzo de la ejecución del *kernel* y la espera a que finalice dicha ejecución mediante la función ***ClFinish***.

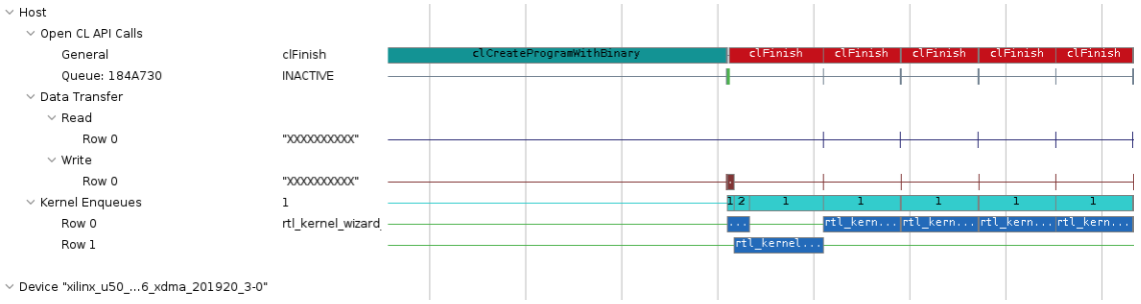

Figura 3.14. Línea temporal del host.

Mientras tanto la actividad en el dispositivo Alveo™ puede visualizarse en la figura 3.15. De forma secuencial se suceden las transacciones entre memoria de la tarjeta y el *kernel* y la ejecución de cada *codeword*.

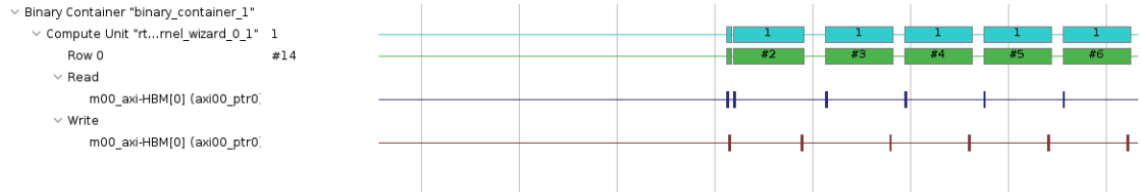

Figura 3.15. Línea temporal del kernel.

Otra forma de validar la aplicación es comprobar directamente cada enfoque de verificación implementado.

Para validar la verificación por comparación con un modelo de referencia se debe asegurar que en la aplicación una misma *codeword* aplicada como entrada tanto de la versión *software* como de la versión *hardware* del decodificador produce la misma *codeword* corregida. Este planteamiento se representa en la figura 3.16.

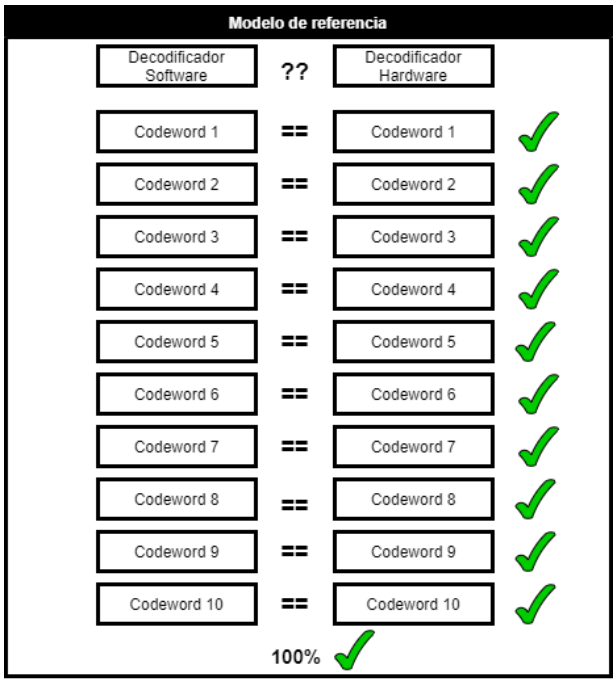

Figura 3.16. Verificación por modelo de referencia.

Para ejemplificar este proceso se muestra la comparación gráfica de la primera de las setenta *codewords* utilizadas por la aplicación que se ha desarrollado. En la figura 3.17 se muestra la *codeword* corregida por la versión *software* del decodificador correspondiente a primera *codeword* sin corregir que se muestra en la figura 3.11.

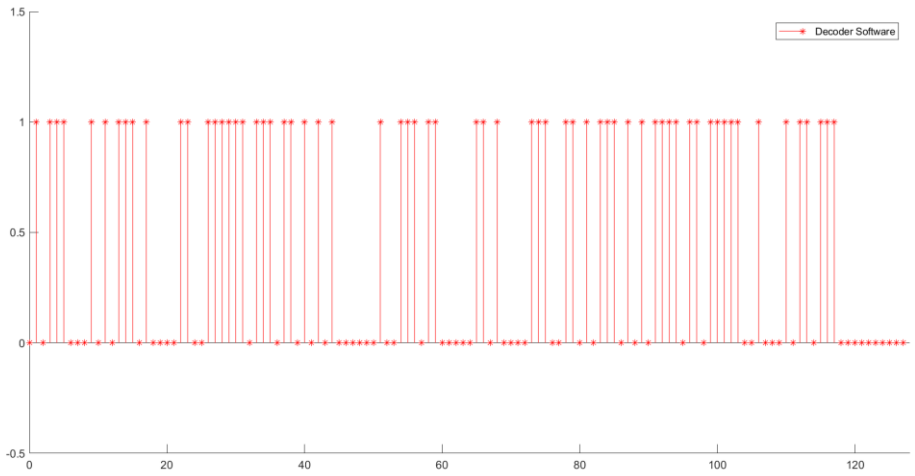

Figura 3.17. Codeword corregida por la versión software del decodificador.

Utilizando la *codeword* corregida por la versión *hardware* del decodificador que se muestra en la figura 3.12 y que corresponde también a la primera *codeword* sin

corregir del proceso se establece una comparación gráfica en la figura 3.18. En ella se puede apreciar claramente como ambas *codewords* corregidas coinciden.

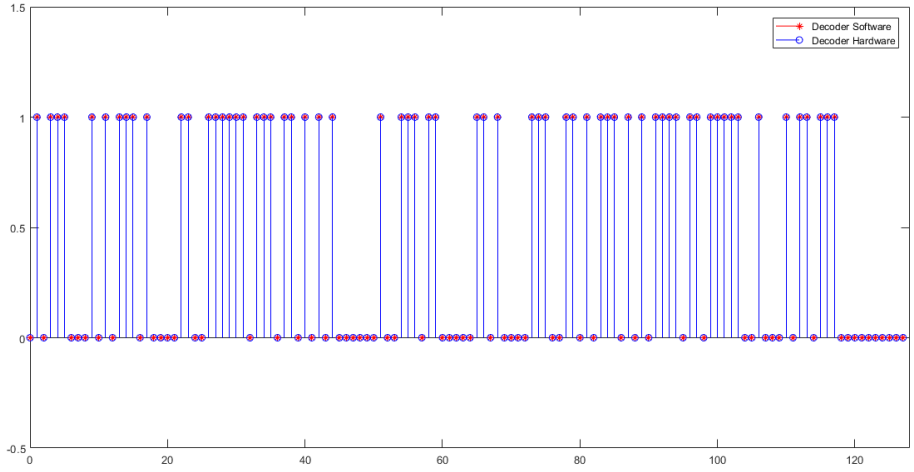

Figura 3.183.14. Comparación de la codeword corregida por la versión software y hardware.

Se trata de un resultado que sin ser una certeza absoluta en lo que se refiere al éxito de este enfoque de verificación sí que sirve como primera referencia a la hora de validar la correcta implementación de este.

El proceso es repetido para diez *codewords* en cada una las siete iteraciones de Eb/No. Esto arroja unos resultados optimistas con el cien por ciento de coincidencias en las setenta *codewords* corregidas.

En el caso de la verificación por cálculo de la CER cada *codeword* corregida se compara con la *codeword* generada por el codificador sobre la que todavía no se han introducido errores. Si no coinciden significa que el decodificador no ha sido capaz de corregir los errores introducidos y que por lo tanto no se puede recuperar la información transmitida. Un ejemplo gráfico del cálculo de la CER se representa en la figura 3.19.

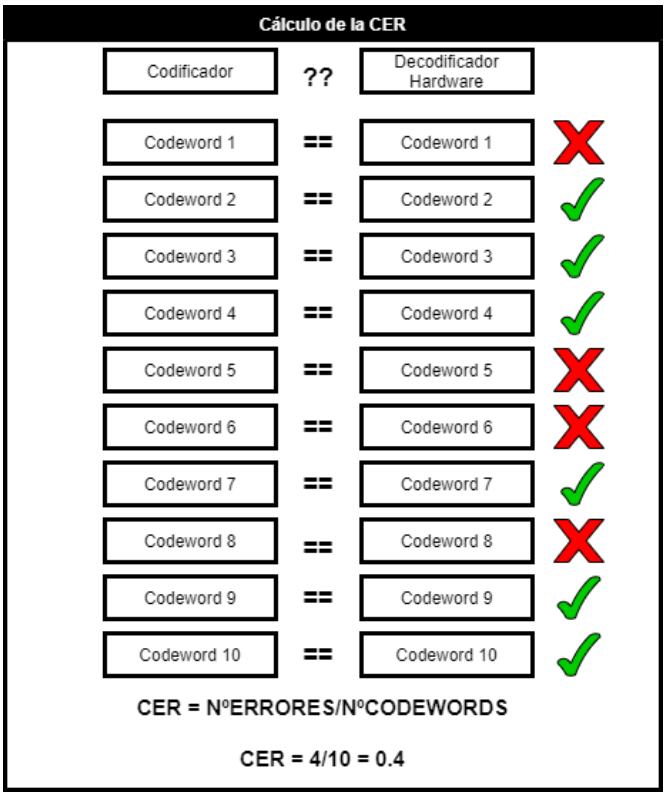

Figura 3.19. Verificación por cálculo de la CER.

El hecho de que la totalidad de las *codeword* corregidas coincidan provoca automáticamente que la CER del decodificador *software* y *hardware* también lo haga puesto que la comparación se realiza directamente sobre la misma *codeword* generada por el codificador. Sin embargo, es de utilidad calcularla para comprobar si el resultado de la CER se asemeja al valor esperado en cada Eb/No. Es de esperar que para Eb/No pequeñas la CER sea muy grande acercándose a uno, mientras que a medida que aumenta el valor de Eb/No la CER disminuya acercándose a cero. Los resultados de la CER para una emulación sobre diez *codewords* en cada Eb/No se muestran en la siguiente tabla 3.1.

Eb/No	0	1	2	3	4	5	6
Codewords	10	10	10	10	10	10	10
Errores	10	9	4	3	1	0	0
CER	1.000000	0.90000	0.40000	0.30000	0.10000	0.00000	0.00000

Tabla 3.1. Resultados de la verificación por cálculo de la CER en emulación hardware.

No son unos resultados con los que se pueda afirmar rotundamente que la verificación del decodificador es un éxito. Sin embargo, al igual que los resultados del enfoque previo sirven como una primera referencia de que la implementación ha sido adecuada. En ellos se puede observar cómo se cumple esta tendencia de CER elevadas para Eb/No pequeñas y viceversa. Sin embargo, se necesitan muchas más *codewords* para poder afirmar que el decodificador ha sido verificado, pero como se ha mencionado con anterioridad la emulación es un proceso demasiado lento para conjuntos de datos tan grandes.

3.5.2 Ejecución *hardware* sobre la tarjeta Alveo™.

Después de compilar la aplicación para un objetivo de emulación *hardware* y de validar que la implementación realizada funciona de la forma esperada es momento de compilarlo para un objetivo *hardware* que permita ejecutarlo directamente sobre la tarjeta aceleradora Alveo™.

La ejecución sobre la tarjeta aceleradora reduce de forma considerable la duración de la aplicación con respecto a la de la emulación *hardware*. Se pasa del orden de días al orden de horas. Sin embargo, son resultados que como era de esperar se encuentran muy alejados de los obtenidos en [1] donde se implementaban técnicas para la paralelización de datos y de tareas que resultaban en tiempos de verificación del orden de minutos.

En cualquier caso, a pesar de necesitar más tiempo para la verificación completa del decodificador se terminan obteniendo los resultados esperados en ambos

enfoques de verificación. Los resultados obtenidos mediante el enfoque basado en el cálculo de la CER se representan numéricamente en la tabla 3.2.

Eb/No	0	1	2	3	4	5	6
Codewords	1000	1000	1000	10000	100000	1000000	100000000
Errores	958	811	405	930	556	62	-
CER	0.958000	0.811000	0.405000	0.093000	0.005560	0.000062	-

Tabla 3.2.Resultados de la verificación por cálculo de la CER en ejecución hardware.

Estos mismos resultados se representan de forma gráfica en la figura 3.20.

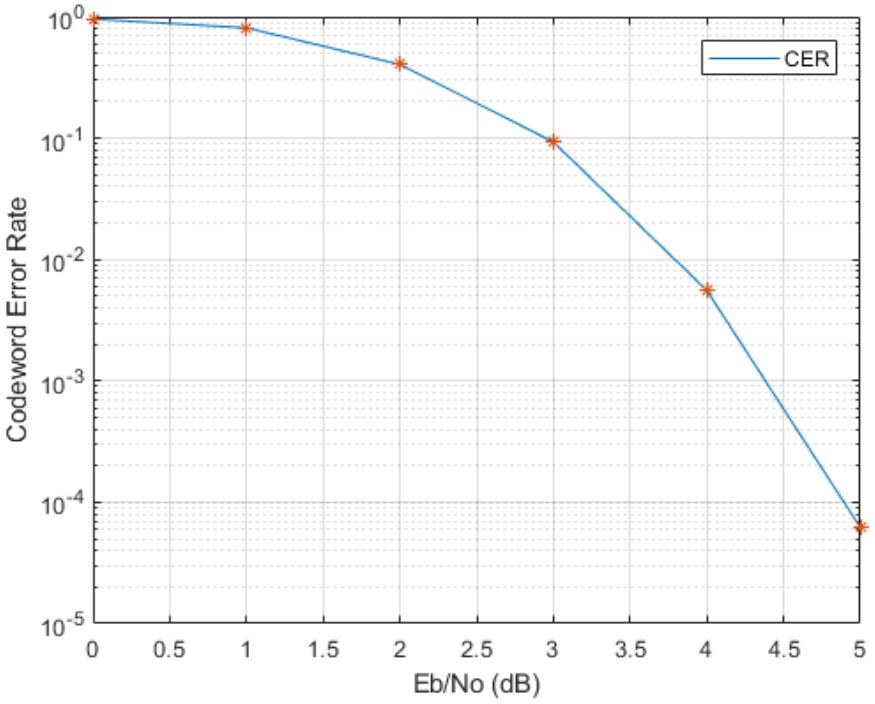

Figura 3.20. Curva de la CER vs Eb/No de la ejecución hardware.

Debido a los inconvenientes experimentados con la integridad física de la tarjeta Alveo™ la verificación del decodificador para Eb/No = 6 dB no se pudo finalizar y no se poseen los resultados correspondientes.

CAPÍTULO 4: ACELERACIÓN DEL PROCESO DE VERIFICACIÓN

Con el desarrollo de una aplicación funcional que implementa los dos enfoques de verificación sobre el sistema heterogéneo se ha alcanzado uno de los objetivos parciales del proyecto. Sin embargo, para alcanzar el objetivo final se ha de acelerar la aplicación. Esto se puede llevar a cabo desde dos perspectivas. La primera consiste en aprovechar los recursos del sistema dedicados al procesamiento de la parte *software* de la aplicación, mientras que la segunda plantea aprovechar los recursos lógicos programables dedicados a la implementación del decodificador *hardware*.

4.1 ACELERACIÓN DE LA APLICACIÓN

4.1.1 Aceleración *hardware*

Desde esta perspectiva, la aceleración de la aplicación puede alcanzarse implementando el paradigma de la paralelización de datos. En él se tiene un conjunto de datos que se subdivide en bloques más pequeños para ser manipulados por múltiples unidades de cómputo de forma simultánea.

Adaptación del *kernel*

La aplicación funcional desarrollada en el capítulo 3 de este documento consta de una única unidad de cómputo, por lo que el primer paso para acelerar la aplicación consiste en implementar múltiples unidades de cómputo sobre la FPGA de la tarjeta Alveo™. Esto se consigue al indicar el número de instancias del *kernel* y sus nombres mediante la opción *–connectivity.nk* del compilador v++ en la construcción de la aplicación. El segundo paso es conseguir que cada unidad de cómputo tenga acceso a una *codeword* distinta de forma simultánea. Para conseguir esto basta con tener en cuenta que solo se puede acceder una vez por unidad de tiempo a un banco de memoria. Esto quiere decir que cada unidad de

cómputo debe acceder a un banco de memoria distinto para que exista paralelismo. Con la opción *–connectivity.sp* del compilador v++ es posible asociar cada instancia a un banco de memoria especifico.

Adaptación del *host*

Del mismo modo que se ha modificado la implementación *hardware* del sistema también es necesario modificar el *software* para alcanzar la paralelización de datos. La parte *software* de la aplicación se puede dividir en tres bloques.

- En el primer bloque se suceden los siguientes eventos de forma secuencial: se genera una *codeword*, se almacena en un *buffer* del *host*, se genera un *buffer* asociado a un banco de memoria especifico y se copia la *codeword* del *buffer* del *host* al del dispositivo Alveo™. Esta secuencia se repite tantas veces como instancias del decodificador hayan sido generadas.
- El segundo bloque se encarga de realizar dos tareas. Asocia una instancia con sus argumentos y encola un comando para comenzar la ejecución de la instancia. Al igual que en el primer bloque, estas dos acciones se suceden de forma secuencial tantas veces como instancias del decodificador hayan sido generadas.
- El tercer y último bloque se encarga de copiar cada *codeword* desde el *buffer* de cada banco de memoria de la tarjeta Alveo™ al *buffer* correspondiente del *host*.

El comportamiento del *software* de la aplicación se describe de forma gráfica en la figura 4.1 para un ejemplo con diez instancias del decodificador LDPC implementadas.

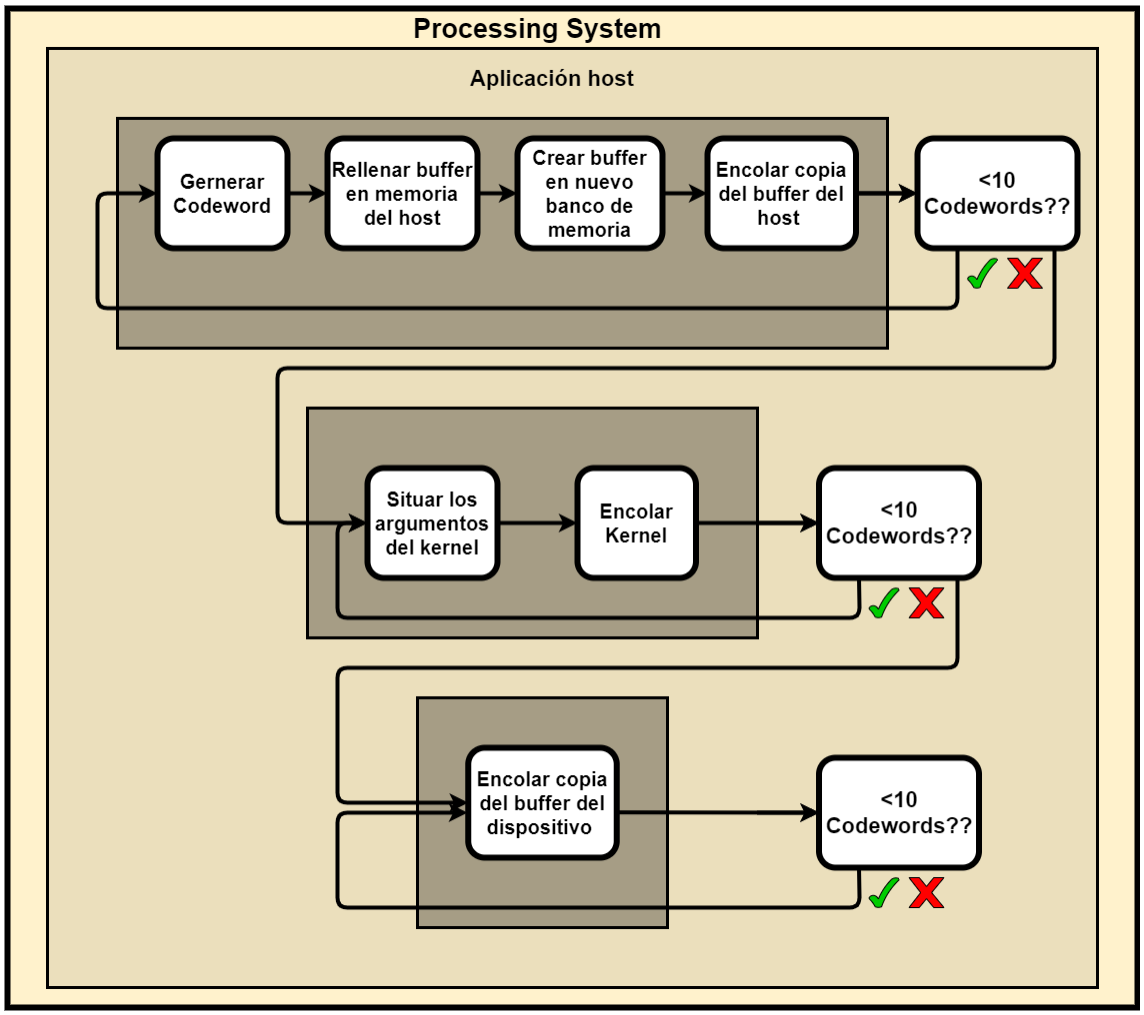

Figura 4.1. Diagrama de bloques de la adaptación del host a la paralelización hardware.

4.1.2 Aceleración *software*

Desde la perspectiva *software*, la aceleración de la aplicación puede alcanzarse implementando el paradigma de la paralelización de tareas. Este paradigma de la programación concurrente consiste en aprovechar los recursos o procesadores disponibles en el sistema asignando a cada uno una tarea diferente.

Adaptación del host

Dentro de las múltiples posibilidades que existen a la hora implementar una paralelización de tareas en la parte *software* de la aplicación, se busca una que respete y complemente en la mayor medida posible a la aceleración *hardware* propuesta en el subapartado anterior. Por ello, se propone una implementación que mediante OpenMP paralelice la generación de *codewords* del primer bloque

descrito en la figura 4.1. Se propone que el grado de paralelización dependa directamente del número de instancias del decodificador LDPC implementadas. El comportamiento del *software* de la aplicación descrito se representa de forma gráfica en la figura 4.2 para un sistema donde la aceleración *hardware* se ha implementado con diez instancias del decodificador LDPC. En el caso del enfoque de verificación por comparación con un modelo de referencia se incluye dentro del bloque paralelizado la decodificación *software* de la respectiva *codeword*.

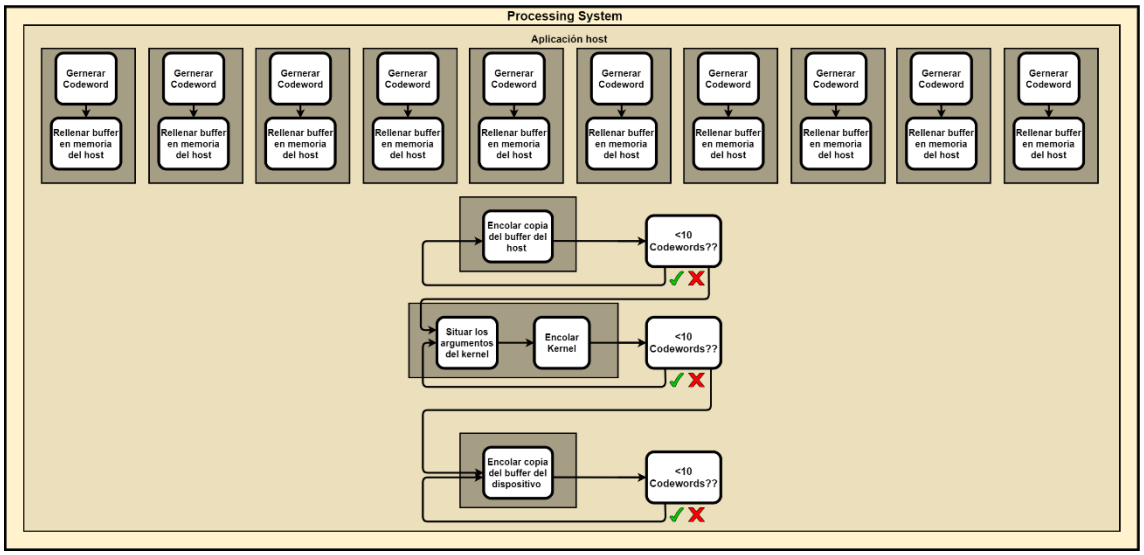

Figura 4.2. Diagrama de bloques de la adaptación del host a la paralelización software.

4.2 EJECUCIÓN DE LA APLICACIÓN

Se sigue la misma metodología que para la implementación realizada en el capítulo 3. Cada planteamiento para acelerar el proceso de verificación se validará inicialmente sobre un objetivo de emulación *hardware* con el fin de comprobar que funciona y posteriormente se implementará para un objetivo *hardware* (tarjeta aceleradora Alveo™) que permita verificar por completo el decodificador LDPC. Es importante destacar que la emulación *hardware* no soporta el uso de OpenMP, por lo que el planteamiento propuesto en la aceleración *software* será directamente implementado sobre la tarjeta Alveo™.

4.2.1 Emulación *hardware*

Se implementan dos aplicaciones:

- Una aplicación acorde a los ejemplos descritos en los subapartados dedicados a la adaptación del *host* y del *kernel* incluidos en el apartado de aceleración *hardware* de este mismo capítulo.
- Una aplicación acorde a la adaptación del *host* propuesta en el apartado dedicado a la aceleración *software* también incluido en este mismo capítulo.

La paralelización de datos queda implementada mediante diez instancias del decodificador LDPC.

Validación de la aplicación acelerada mediante *hardware*

La validación del planteamiento propuesto para la aceleración de la aplicación mediante *hardware* se puede llevar a cabo de varias formas. Una de las posibilidades es visualizar el comportamiento de la aplicación completa sobre una misma línea temporal, que ayude a corroborar que las tareas se suceden en el orden esperado. En la figura 4.3 se muestran todas las tareas ejecutadas por el *host*. Como se puede apreciar, la secuencia de cada bloque de tareas se corresponde con la secuencia descrita en el subapartado dedicado a la adaptación del *host* incluido en el aparatado de aceleración *hardware* de este mismo capítulo. Primero se suceden las transferencias de *codewords* entre memorias y una vez estas finalizan comienza el encolado de comandos para el inicio de cada instancia.

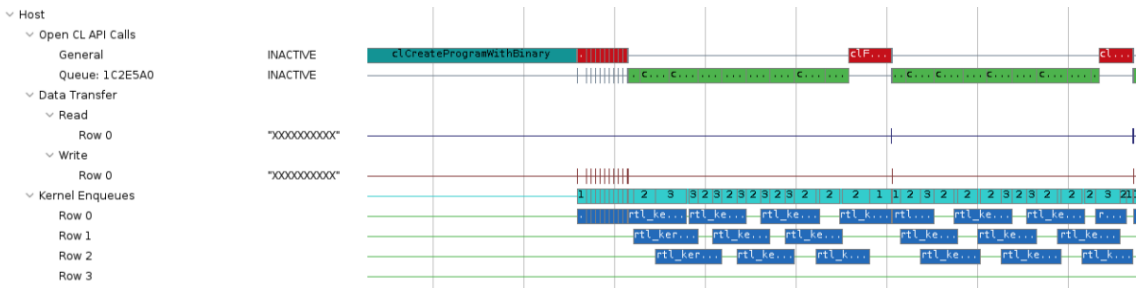

Figura 4.3. Línea temporal del host.

La paralelización de datos se puede apreciar claramente en la actividad que tiene lugar en la tarjeta Alveo™ emulada. Esta actividad se muestra en la figura 4.4. Se aprecia la ejecución paralela de cada instancia sobre una misma línea temporal y las transacciones entre instancias y memoria de la tarjeta emulada.

Figura 4.4. Línea temporal del kernel.

Estimaciones temporales de la aceleración *hardware*

Además de servir para validar el funcionamiento de la aplicación, la emulación *hardware* también se utiliza para obtener una estimación del tiempo total de uso del dispositivo y del tiempo de ejecución del *kernel*. El tiempo de ejecución del *kernel* entendido en términos del modelo de ejecución de OpenCL como el tiempo transcurrido entre el estado de *start* y de *end* del comando encolado para comenzar la ejecución del *kernel*.

Una comparación entre el tiempo de uso del dispositivo estimado para la aplicación secuencial y para la aplicación paralela se establece en la tabla 4.1. La emulación de ambas aplicaciones se realizó para el mismo número de *codewords*. Fueron un total de setenta *codewords*, diez para cada Eb/No.

	Estimación del tiempo de uso del dispositivo (ms)
Aplicación secuencial	0.647
Aplicación paralela	0.4

Tabla 4.1. Comparación del tiempo estimado de uso del dispositivo.

Con este resultado se aprecia la efectividad del planteamiento propuesto para la aceleración de la aplicación. Se reducen 0.247 milisegundos del tiempo total estimado en el que la aplicación tiene en uso la tarjeta Alveo™.

A partir de la estimación del tiempo de uso del dispositivo se puede extrapolar un tiempo aproximado de uso del dispositivo para el total de *codewords* correspondiente a cada enfoque de verificación. Se trata de una extrapolación imprecisa pero que ofrece una primera referencia de los tiempos alcanzables.

	Estimación tiempo uso dispositivo para 70 *codewords*	Tiempo aproximado para 10^5 *codewords*	Tiempo aproximado para 10^8 *codewords*
Aplicación paralela con diez instancias	0.4 ms	0.571 seg	9.52 min

Tabla 4.2. Tiempo aproximado de uso del dispositivo para ambos enfoques de verificación

El tiempo de ejecución de cada instancia del *kernel* se refleja en la tabla 4.3 y puede utilizarse para medir lo eficiente que ha sido la paralelización.

Instancia	Nº Ejecuciones	Tiempo total (ms)	Tiempo medio (ms)	Utilización (%)
1	8	0.084	0.011	21
2	8	0.090	0.011	22.5
3	8	0.089	0.011	22.25
4	8	0.080	0.010	20
5	8	0.068	0.009	17
6	8	0.068	0.008	17
7	8	0.070	0.009	17.5
8	8	0.084	0.011	21
9	8	0.079	0.010	19.75
10	8	0.085	0.011	21.25

Tabla 4.3. Tiempo de ejecución de las instancias del kernel.

Se puede apreciar como el tiempo total de ejecución de cada instancia es ligeramente inferior a 0.1 milisegundos. El tiempo promedio de cada instancia se obtiene de dividir su tiempo total de ejecución entre el número de ejecuciones que ha efectuado. La utilización es el resultado de dividir el tiempo total entre el tiempo de uso del dispositivo en la aplicación paralela.

El hecho de que el tiempo de uso del dispositivo (0.4 ms) no coincida con el tiempo de ejecución total de cada instancia (~0.1 ms) en la aplicación paralela como si lo hacía en la aplicación secuencial (0.647 ms) se debe a que la ejecución de cada instancia no se produce de forma completamente paralela. Esto se debe a que los comandos que inician la ejecución de cada instancia son encolados secuencialmente sobre una misma cola de comandos. Además, la ejecución de cada instancia se limita a una sola *codeword* por cada comando que se encola desde el *host*. Esto hace que el tiempo en el que las instancias realmente se encuentran ejecutando la *codeword* en paralelo sea muy reducido. El resultado es una utilización muy baja del dispositivo por parte de cada instancia. Ejecutar un

mayor número de *codewords* en cada ejecución de una instancia repercutiría positivamente en la utilización del dispositivo. Sin embargo, para que los resultados fuesen perceptibles se necesitaría un número elevado de *codewords*. Esto en emulación *hardware* no es posible debido al elevado tiempo que necesita la emulación sobre grandes conjuntos de datos.

Validación de la aplicación acelerada mediante *software*

Una vez ha sido validado el planteamiento *hardware* propuesto para la aceleración de la aplicación es necesario validar el planteamiento *software*. Sin embargo, el uso de OpenMP no es compatible con la emulación *hardware*. De nuevo, problemas relacionados con la integridad física de la tarjeta impiden validar directamente este planteamiento sobre la tarjeta Alveo™. Se opta por sustituir la tarjeta por la versión *software* del decodificador y validar fuera del entorno de Vitis™ que la implementación del planteamiento es correcta comparando los valores de CER que se obtienen con los obtenidos en la aplicación secuencial del capítulo 3.

Estimaciones temporales de la aceleración *software*

Las estimaciones temporales varían en función del enfoque de verificación implementado. Para un enfoque de verificación basado en el cálculo de la CER se obtienen resultados que reducen la duración del bloque paralelizado del orden de las decenas de minutos a los minutos como se muestra en la tabla 4.4.

	Tiempo promedio (ms)	Tiempo aproximado para 10^8 iteraciones	Tiempo aproximado para 10^8 iteraciones con paralelización
Generar codeword y almacenar en el buffer del host	0.0395	66 min	6.6 min

Tabla 4.4.Tiempos estimados y aproximados para enfoque basado en cálculo de la CER.

Para un enfoque por comparación con un modelo de referencia se obtiene unos tiempos promedios para Eb/No = 0 dB. Por ello, se trata de una estimación de tiempos pesimista puesto que la latencia del decodificador es mayor cuanto menor es el valor de Eb/No. Los resultados estiman una reducción en la duración del bloque paralelizado del orden de minutos a los segundos como se muestra en la tabla 4.5.

	Tiempo promedio	Tiempo aproximado para 10^5 iteraciones	Tiempo aproximado para 10^5 iteraciones con paralelización
Generar codeword y almacenar en el buffer del *host*	0.0395 ms	3.95 seg	0.395 seg
Decodificación *software* de la *codeword*	5.162 ms	8.6 min	51.62 seg
Tiempo total	5.202 ms	8.6 min	52.015 seg

Tabla 4.5. Tiempos estimados y aproximados para enfoque basado en modelo de referencia.

4.2.2 Ejecución *hardware* sobre la tarjeta Alveo™

Las primeras pruebas sobre la tarjeta aceleradora Alveo™ implementaban el planteamiento para la aceleración mediante *hardware* del enfoque de verificación basado en el cálculo de la CER.

La implementación buscaba conseguir unos tiempos de verificación similares a los extrapolados a partir de los tiempos estimados por la emulación. Sin aceleración *software*, la generación de *codewords*, por sí sola, tenía una duración superior a la hora. Esto ligado a las múltiples instancias ejecutándose sobre la FPGA de la tarjeta y a las inadecuadas condiciones de ventilación de las que disponía la estación de trabajo provocó un sobrecalentamiento de la tarjeta antes de finalizar la prueba. Sin embargo, la verificación sobre valores de Eb/No entre

cero y cinco dBs arrojaba resultados de la CER acordes con lo esperado y en tiempos que incitaban al optimismo con cerca de cincuenta millones de *codewords* procesadas en aproximadamente media hora, lo que lleva a pensar que la duración completa de la prueba estaría en torno a una hora.

CAPÍTULO 5: CONCLUSIONES Y LÍNEAS FUTURAS

5.1 CONCLUSIONES

Las dos implementaciones típicas pre-silicio de la verificación de un decodificador presentan una serie de desventajas que pueden ser solventadas al aprovechar las características de un sistema en un chip. El éxito de los resultados obtenidos en este tipo de sistemas incita a continuar investigando sobre esta línea de trabajo en busca de nuevas soluciones que mejoren aún más los resultados obtenidos.

En este proyecto se propone como solución un sistema heterogéneo compuesto por un multiprocesador x86 y una tarjeta aceleradora basada en tecnología FPGA. Se desarrolla una aplicación válida para este sistema que implemente los dos enfoques de verificación propuestos en el trabajo de investigación que inspira la realización de éste. Conseguir una aplicación funcional, que implemente ambos enfoques de verificación de una forma elemental, sin importar la duración de esta, garantiza una base sobre la que comenzar a añadir técnicas de aceleración.

Esta aplicación elemental fue validada mediante emulación *hardware* antes de ser utilizada para verificar el decodificador LDPC sobre el propio sistema heterogéneo. La curva de la CER frente a la Eb/No extraída de los resultados de la ejecución sugiere que la implementación se ha realizado de manera adecuada.

Las técnicas de aceleración propuestas se basan en dos paradigmas de la programación concurrente como son la paralelización de datos y la paralelización de tareas. Se desarrolla una aplicación para cada paradigma con el objetivo de poder validar cada planteamiento de forma individual. Se utiliza la emulación *hardware* para llevar a cabo esta validación y para obtener una primera estimación de la mejora temporal introducida por estas técnicas. Los tiempos estimados para cada planteamiento de forma individual llevan a pensar que una combinación

adecuada de ambas paralelizaciones puede producir tiempos cercanos o incluso mejores que los obtenidos en [1], donde el enfoque de verificación por comparación con un modelo de referencia duraba en torno a veinte minutos y el enfoque de verificación por cálculo de la CER duraba unos sesenta y siete minutos.

Los resultados de la emulación *hardware* invitan a pensar que el enfoque de verificación por comparación con un modelo de referencia se puede llegar a reducir a unos pocos minutos, mientras que el enfoque basado en el cálculo de la CER se puede reducir a una duración inferior a los veinte minutos. Sin embargo, problemas relacionados con la integridad física de la tarjeta aceleradora Alveo™ disponible en este proyecto impiden alcanzar resultados definitivos que sustenten este pronóstico.

5.2 LÍNEAS FUTURAS

Aunque los tiempos estimados invitan al optimismo con respecto a superar los objetivos temporales fijados al inicio de este proyecto, hay otros puntos relevantes de la aplicación que deben ser analizados y gestionados para asegurar que la duración total de la aplicación es inferior a la de los objetivos planteados.

Este es el caso de las transacciones entre la memoria del host y de la tarjeta Alveo™. En el caso de hacer transacciones de poco tamaño el *throughput* del bus PCIe puede verse muy reducido y llegar a introducir latencias que penalicen gravemente el rendimiento de la aplicación.

Además de este tipo de detalles, existen muchas opciones para optimizar aún más el rendimiento de la aplicación y que por culpa de los problemas experimentados con la tarjeta Alveo™ no han podido ser abordados durante este proyecto. Sin embargo, a continuación, se plantean algunos de los aspectos a mejorar en la aplicación para que puedan ser investigados en el futuro.

La utilización del dispositivo estimada en la paralelización mediante *hardware* era muy inferior a la utilización en la aplicación secuencial. La solución a esta baja eficiencia puede plantearse desde distintos puntos de vista, desde encolar los comandos de inicio de cada instancia en paralelo hasta maximizar el tiempo de ejecución de cada instancia por cada comando que se encole forzando de esta manera un mayor tiempo de ejecución en paralelo por parte de cada instancia.

Otro aspecto para optimizar es el de la cantidad de bits innecesarios que se almacenan y se transmiten para la corrección de cada *codeword*. Al final, cada *codeword* con ruido es demodulada y almacenada como 128 enteros en los cuales solo son de interés los tres bits menos significativos. El resto de los bits de cada entero carece de información y son innecesarios para el decodificador LDPC.

Otro aspecto para optimizar que se puede sondear en un futuro consiste en maximizar el tamaño de cada transferencia entre la memoria de la tarjeta Alveo™ y el propio *kernel.* El tamaño utilizado durante este proyecto es de 32 bits. Sin embargo, el protocolo AXI permite interfaces con canales de datos de hasta 512 bits. Con las modificaciones adecuadas al decodificador se puede aprovechar esta característica para reducir el número de transferencias.

Una línea de investigación alternativa que también puede ser interesante es la de intentar replicar este proyecto sobre una tarjeta Alveo™ U200 o U250 que disponga de una plataforma con la tecnología QDMA gestionando los accesos directos a memoria. Esta tecnología permitiría implementar soluciones mucho más eficientes que hiciesen uso de la característica para el *streaming* en el intercambio de datos entre dispositivos sin tener que pasar por la memoria de la tarjeta Alveo™.

CAPÍTULO 6: REFERENCIAS

[1]. Fernandez, V., Abad, C., Alvarez, A., Ugarte, I., & Sanchez, P. (2021). Pre-Silicon FEC Decoding Verification on SoC FPGAs. *IEEE Communications Letters, 25*(1), 127-131. doi:10.1109/lcomm.2020.3025223

[2]. *Vitis Unified Software Platform Documentation.* (21 de Marzo de 2021). Obtenido de Application Acceleration Development: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1393-vitis-application-acceleration.pdf

[3]. *Xilinx® Runtime (XRT) Architecture*. (01 de Julio de 2021). Obtenido de https://xilinx.github.io/XRT/master/html/index.html

[4]. *Get Moving with Alveo: Acceleration Basics*. (01 de Noviembre de 2019). Obtenido de https://www.xilinx.com/developer/articles/acceleration-basics.html

[5]. Xilinx. (21 de Septiembre de 2021). *XRT and Vitis™ Platform Overview*. Obtenido de https://xilinx.github.io/XRT/master/html/platforms.html

[6]. *Alveo U50 Data Center Accelerator Data Sheet.* (27 de Agosto de 2020). Obtenido de https://www.xilinx.com/content/dam/xilinx/support/documentation/data_sheets/ds965-u50.pdf

[7]. Xilinx. (July 30, 2021). *Alveo Data Center Accelerator Card Platforms User Guide.* Xilinx Alveo™. Obtenido de https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/ug1120-alveo-platforms.pdf

[8]. *Security of Alveo Platform*. (01 de Julio de 2021). Obtenido de https://xilinx.github.io/XRT/master/html/security.html

[9]. *Improve Your Data Center Performance With The New QDMA Shell*. (07 de Febrero de 2019). Obtenido de https://forums.xilinx.com/t5/Adaptable-Advantage-Blog/Improve-Your-Data-Center-Performance-With-The-New-QDMA-Shell/ba-p/990371

[10]. *QDMA vs XDMA*. (26 de Febrero de 2019). Obtenido de https://forums.xilinx.com/t5/PCIe-and-CPM/QDMA-vs-XDMA/td-p/944098

[11]. *Solved: QDMA shell Alveo U50 and U280 - Community Forums*. (14 de Marzo de 2021). Obtenido de https://forums.xilinx.com/t5/Alveo-Accelerator-Cards/QDMA-shell-Alveo-U50-and-U280/td-p/1098665

[12]. Xilinx. (February 28, 2020). *Vitis Unified Software Application Acceleration Development.* Xilinx Vitis™. Obtenido de https://www.xilinx.com/cgi-bin/docs/rdoc?t=vitis+doc;v=latest;d=kme1569523964461.html

[13]. Xilinx. (21 de Septiembre de 2021). *XRT Controlled Kernel Execution Models*. Obtenido de https://xilinx.github.io/XRT/master/html/xrt_kernel_executions.html

[14]. Khronos Group. (30 de Junio de 2021). *OPEN STANDARD FOR PARALLEL PROGRAMMING OF HETEROGENEOUS SYSTEMS.* Obtenido de Khronos OpenCL Registry: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf