Universidad del País Vasco Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

MÁSTER UNIVERSITARIO EN

INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

DESPLIEGUE Y ANÁLISIS DEL ENTORNO DE ACELERACIÓN DE APLICACIONES PARA PLATAFORMAS DE HARDWARE ACELERADO XILINX (VITIS)

Estudiante *Santiago Santo-Tomás, Adrián*
Director/Directora *Lázaro Arrotegui, Jesús*
Departamento *Tecnología Electrónica*
Curso académico *2020-2021*

Bilbao, 19, septiembre, 2021

Resumen

En los últimos 75 años, se ha disparado el crecimiento tecnológico gracias al desarrollo de numerosas soluciones Software y Hardware. Ambos conceptos y su desarrollo se encuentran intrínsecamente relacionados, de forma que, el Software, cada vez más exigente, requiere de Hardware de mayor rendimiento, y el propio Hardware, también en desarrollo, permite soluciones Software con requerimientos más elevados. Esto ha permitido no solo extender el ámbito de la computación a prácticamente la totalidad de sectores a nivel global, sino que ha permitido la aparición de nuevas tecnologías que requieren procesamientos muy exigentes, en cuanto a complejidad de computación, así como en cuanto a volumen de información a procesar. Esto se traduce en servicios que requieren procesar gran cantidad de información en tiempos reducidos, o en servicios que requieren simplemente minimizar este tiempo de procesamiento por su propia naturaleza. Algunos ejemplos de ello son Big Data Analysis, Machine Learning, Time Sensitive Networking o Internet of Things, entre muchos otros.

Estas nuevas tecnologías y sus requerimientos, han permitido la aparición de dispositivos electrónicos que permiten transformar aplicaciones tradicionalmente más lentas y menos eficientes, en aplicaciones que se adaptan a este nuevo entorno de alto rendimiento. Además, permiten establecer el entorno para la creación de nuevas aplicaciones, directamente sobre este tipo de dispositivos. Estos dispositivos permiten lo que se conoce como Aceleración Hardware, introduciendo un nuevo paradigma para el desarrollo de aplicaciones de alto rendimiento. A pesar de su gran versatilidad y posibilidades, se encuentra en un punto de baja maduración y es necesario realizar un estudio y análisis profundo de las tecnologías, para establecer los primeros pasos hacia este entorno de desarrollo y despliegue de aplicaciones de alto rendimiento, de forma que pueda extenderse a los diferentes sectores tecnológicos que podrían beneficiarse de ello.

Es por ello, que surge este proyecto, para estudiar y desplegar el entorno de desarrollo de este tipo de soluciones, así como para analizar las diferentes vías de desarrollo y metodologías, visualizando sus resultados, para determinar las posibilidades de este tipo de tecnología tan prometedora. Para ello, se utiliza el entorno de desarrollo Xilinx Vitis, implementando las soluciones sobre las tarjetas de aceleración Xilinx Alveo.

Palabras clave: Aceleración Hardware, Aplicaciones Aceleradas, Informática de Alto Rendimiento, Hardware Reconfigurable

Abstract.

Over the last 75 years, technological growth has exploded thanks to the development of numerous software and hardware solutions. Both concepts and their development are intrinsically related, in such a way that the increasingly demanding software requires higher performance hardware, and the hardware itself, also in development, allows software solutions with higher requirements. This has not only extended the scope of computing to practically all sectors globally, but has also allowed the emergence of new technologies that require very demanding processing, in terms of computing complexity, as well as in terms of the volume of information to be processed. This translates into services that require processing large amounts of information in reduced times, or services that simply require minimising this processing time by their very nature. Some examples are Big Data Analysis, Machine Learning, Time Sensitive Networking or Internet of Things, among many others.

These new technologies and their requirements have enabled the emergence of electronic devices that transform traditionally slower and less efficient applications into applications that adapt to this new high-performance environment. Moreover, they make it possible to establish the environment for the creation of new applications directly on this type of device. These devices enable what is known as Hardware Acceleration, introducing a new paradigm for the development of high performance applications. Despite its great versatility and possibilities, it is at a point of low maturity and it is necessary to carry out an in-depth study and analysis of the technologies in order to establish the first steps towards this environment for the development and deployment of high-performance applications, so that it can be extended to the different technological sectors that could benefit from it.

This is why this project was created to study and deploy the development environment for this type of solution, as well as to analyse the different development paths and methodologies, visualising their results, in order to determine the possibilities of this type of promising technology. For this purpose, the Xilinx Vitis development environment is used, implementing the solutions on Xilinx Alveo acceleration cards.

Keywords: Hardware Acceleration, Accelerated Applications, High-Performance Computing, Reconfigurable Hardware

Laburpena.

Azken 75 urteetan, teknologia-hazkundeak gora egin du, Software eta Hardware soluzio ugariren garapenari esker. Bi kontzeptu horiek eta horien garapena berez lotuta daude; horrela, gero eta zorrotzagoa den Softwareak errendimendu handiagoko Hardwarea behar du, eta Hardwareak berak ere, garatzen ari denak, eskakizun handiagoko Software-soluzioak ahalbidetzen ditu. Horri esker, konputazioaren esparrua ia sektore guztietara zabaldu da maila globalean, eta, horrez gain, prozesamendu oso zorrotzak eskatzen dituzten teknologia berriak agertu dira, bai konputazioaren konplexutasunari dagokionez, bai prozesatu beharreko informazioaren bolumenari dagokionez. Horren ondorioz, denbora laburrean informazio asko prozesatzea eskatzen duten zerbitzuak ematen dira, edo berez prozesamendu-denbora hori minimizatzea besterik eskatzen ez duten zerbitzuak. Horren adibide dira Big Data Analysis, Machine Learning, Time Sensitive Networking edo Internet of Things, besteak beste. Teknologia berri horiei eta haien eskakizunei esker, tradizionalki motelagoak eta eraginkortasun txikiagokoak izan diren aplikazioak eraldatzeko aukera ematen duten gailu elektronikoak sortu dira, errendimendu handiko ingurune berri horretara egokitzen direnak. Gainera, aplikazio berriak sortzeko ingurunea ezartzea ahalbidetzen dute, zuzenean horrelako gailuetan. Gailu hauek Hardware azelerazioa deritzona ahalbidetzen dute, errendimendu altuko aplikazioak garatzeko paradigma berri bat sartuz. Moldakortasun eta aukera handiak dituen arren, heldutasun txikiko puntu batean dago, eta beharrezkoa da teknologien azterketa eta analisi sakona egitea, errendimendu handiko aplikazioak garatzeko eta hedatzeko ingurune horretara lehen urratsak ezartzeko, horren onurak jaso ditzaketen sektore teknologikoetara zabaldu ahal izateko.

Hori dela eta, proiektu hau sortu da horrelako soluzioen garapen-ingurunea aztertzeko eta hedatzeko, baita garapen-bideak eta metodologiak aztertzeko ere, emaitzak bistaratuz, etorkizun handiko teknologia mota horren aukerak zehazteko.Horretarako, Xilinx Vitis garapen-ingurunea erabiltzen da, Xilinx Alveo azelerazio-txartelen gaineko soluzioak inplementatuz.

Hitz gakoak: Hardware Azelerazioa, Aplikazio Bizkortuak, Errendimendu Handiko Informatika, Hardware Birkonfiguragarria

Acrónimos

- 100G: 100 Gigabit Ethernet
- AMD: Advanced Micro Devices Inc.
- API: Application Programming Interface (Interfaz de Programación de Aplicaciones)
- ASIC: Application-Specific Integrated Circuit (Circuito Integrado de Aplicación Específica)
- AWS: Amazon Web Services
- AXI: Advances eXtensible Interface (Interfaz Extensible Avanzada)
- b: bit
- B: byte
- CPU: Central Processing Unit (Unidad de Procesamiento Central)
- DDR: Doble Data Rate (Doble Ratio de Datos)
- DMA: Direct Memory Access (Acceso Directo a Memoria)
- DSP: Digital Signal Processor (Procesador de Señales Digitales)
- FLOPS: Floating Point Operations per Second (Operaciones de Coma Flotante por Segundo)
- FPGA: Field-Programmable Gate Array (Matriz de Puertas Lógicas Programable en Campo
- GPU: Graphics Processing Unit (Unidad de Procesamiento Gráfico)
- GT/s: GigaTransactions per Second (múltiplo de transacciones por segundo)
- Hz: Hercios
- HBM, HBM2: High Bandwidth Memory, High Bandwidth Memory 2ª Generación (Memoria de Alto Rendimiento)
- HLS: High Level Synthesis (Síntesis de Alto Nivel)
- HPC: High Performance Computing (Informática de Alto Rendimiento)

- HW: Hardware
- I/O: Input/Output (Entrada/Salida)
- ID: Identifier (Identificador)
- IDE: Integrated Development Environment (Entorno de Desarrollo Integrado)
- IP: Intelectual Property (Propiedad Intelectual)
- JTAG: Joint Test Action Group (Grupo de acción de Prueba Conjunta)
- LTS: Long Term Support (Soporte a Largo Plazo)
- MCTP: Management Component Transport Protocol (Protocolo de Transporte de Componentes de Gestión)
- PCIe: Peripheral Component Interconnect Express (Interconexión Rápida de Componentes Periféricos)
- PL: Programmable Logic (Lógica Programable)
- PLDM: Platform Level Data Model (Modelo de Datos a Nivel de Plataforma)
- PMBus: Power Management Bus (Bus de Gestión de Alimentación)
- PS: Processing Subsystem (Subsistema de Procesamiento)
- QSFP28: Quad Small Form-Factor Pluggable 28
- RAM: Random Memory Access (Memoria de Acceso Aleatorio
- RD: Read (Lectura)
- RTL: Register Transfer Language (Lenguaje de Transferencia de Registros)
- SLR: Super Logic Region (Super Región Lógica)
- SMBus: System Management Bus (Bus de Administración del Sistema)
- SO: Sistema Operativo (Operating System)
- SPI: Serial Peripheral Interface (Interfaz de Periféricos Serie)
- SW: Software

- Tx: Transmission (Transmisión)
- UART: Universal Asynchronous Receiver-Transmitter (Transmisor-Receptor Asíncrono Universal)
- WR: Write (Escritura)
- XRT: Xilinx Runtime Library

Índice

Índice de figuras

Índice de tablas

Índice de códigos

1. Introducción.

En este documento se presenta el trabajo realizado para el estudio y análisis del estado del arte y de las diferentes alternativas y soluciones del entorno tecnológico de la aceleración de aplicaciones, mediante Hardware de aceleración especializado.

Para ello, en primer lugar, se realiza una contextualización, para comprender el entorno tecnológico y las diferentes necesidades que surgen del mismo. También se muestra, no solo la tendencia hasta la actualidad, sino lo que se espera en un futuro inmediato o próximo.

A continuación, se tratan los aspectos tecnológicos y teóricos relacionados directamente con el trabajo propuesto, para entender el trasfondo de cada uno de los elementos implicados. De esta forma, se define el estado del arte de las herramientas y tecnologías utilizadas, para permitir determinar el porqué de las alternativas seleccionadas y las diferentes posibilidades.

Una vez se han definido los conceptos teóricos de interés, se pasa a definir los objetivos y el alcance del proyecto, para asentar las bases del mismo y acotar y ubicar mejor el trabajo realizado.

En este punto, se define y explica la metodología seguida para el correcto desempeño del proyecto, definiendo cada una de las etapas del proceso y mostrando en qué consiste cada una de dichas etapas. Este trabajo permite estudiar en profundidad el problema planteado, analizar las herramientas y tecnologías implicadas y obtener un resultado para dicho proceso de estudio y análisis.

Este resultado es analizado en los siguientes puntos, para obtener los resultados finales del estudio, para la solución planteada, y extraer las conclusiones pertinentes.

Para simplificar la memoria, se introducen una serie de Anexos que permiten conocer en mayor detalle algunos puntos de interés adicionales a la memoria, relacionados con el trabajo realizado. De esta forma, se divide el documento en dos partes, una primera, correspondiente a la memoria, tratando de amenizar y facilitar la lectura, y una segunda, compuesta por los anexos, para permitir a los lectores más exigentes profundizar en el estudio.

2. Contexto.

Para ubicar el proyecto es necesario explicar previamente una serie de conceptos. Al tratarse de una solución que combina tanto Software como Hardware, es importante comprender estos conceptos, conociendo la evolución de los mismos hasta la actualidad, así como entender la tendencia actual para etapas futuras. Aunque se encuentran intrínsecamente relacionados, se puede analizar su evolución de manera separada, para posteriormente obtener la síntesis de ambos conceptos. (K. Blum & V. Aho, 2011)

Por una parte, el Software ha crecido conceptualmente y se ha extendido a numerosos ámbitos de aplicación. Desde los inicios, partiendo de un lenguaje máquina difícil de comprender y utilizar, con instrucciones y funcionalidades limitadas, hasta las diferentes abstracciones actuales de alto nivel, dando lugar a lenguajes más fáciles de utilizar y permitiendo operaciones más complejas. Inicialmente permitía solventar computación simple y se concentraba en ámbitos científicos y de investigación. Posteriormente, se han obtenido soluciones con mayor resolución matemática, permitiendo computación compleja y de alto rendimiento, viables incluso en entornos domésticos. (Wirth, 2008) (Society, 2008)

Por otra parte, el Hardware ha experimentado también una fuerte evolución, gracias a un crecimiento constante de las posibilidades de la electrónica. Este crecimiento se ve regido por la Ley de Moore, que supone una cierta limitación temporal, no permitiendo superar dicho crecimiento entre generaciones de Hardware. El gran desarrollo surge a partir de la introducción de la tecnología digital, dejando en segundo lugar la tecnología analógica, al menos en cuanto a computación y procesamiento. A partir de dicho momento, comienzan a desarrollarse diferentes dispositivos electrónicos, que se interconectan para dar lugar a otros más complejos, creando circuitos integrados, microcontroladores, microprocesadores, microcomputadores, System on Chip, FPGAs, CPUs o GPUs, entre muchas otras posibilidades. (Thompson, 1998)

Como se puede observar, el crecimiento ha sido exponencial en los últimos años y se espera que la tendencia siga siendo la misma. Es por ello que seguirán apareciendo aplicaciones más exigentes en cuanto a rendimiento y, por tanto, será necesario aportar un Hardware capaz de soportarlo. Además, si se desea desarrollar soluciones Software-Hardware eficientes y de calidad, es importante que ambas partes se coordinen de manera adecuada. En este punto, aparece el tema de interés del proyecto, el Hardware de aceleración, como

solución ante un entorno en constante cambio y crecimiento, con aplicaciones Software cada vez más exigentes y una evolución Hardware frenada por las limitaciones de la física. (Bode & Dal Cin, 1993) (Cope, Y.K. Cheung, Luk, & Howes, 2010) (Alastruey, Briz, Ibáñez, & Viñals, 2006)

En este punto, aparecen diferentes fabricantes, tecnologías y plataformas, pero, para el proyecto actual, se utiliza el fabricante Xilinx, con su conjunto de tarjetas de Hardware de aceleración, la plataforma de desarrollo Vitis y Vivado, y sus interfaces y drivers de comunicación entre equipamiento de uso general y el equipamiento de aceleración Hardware.

La elección de esta alternativa se debe a su madurez y trayectoria como fabricante de este tipo de equipamiento y a su repercusión a nivel global.

A pesar de esta importancia y participación, este tipo de tecnología no se encuentra extendida a un gran número de sectores debido a su poca madurez. En este contexto surge la naturaleza del proyecto, para desplegar este tipo de entornos de desarrollo y analizar en detalle diferentes posibilidades y alternativas de aplicaciones aceleradas mediante este Hardware específico, con la posibilidad de extraer conclusiones sobre su viabilidad y las posibles mejoras de rendimiento frente a las alternativas utilizadas actualmente para el desarrollo de aplicaciones Software tradicionales. También permite extraer posibles riesgos e inconvenientes frente a dichos casos más tradicionales y extendidos.

3. Objetivos y alcance del proyecto.

Una vez contextualizado el proyecto, se pueden definir los objetivos principales y secundarios del mismo.

El objetivo principal es el estudio de las tecnologías, tanto Software como Hardware implicadas en la Aceleración Hardware, para extraer conclusiones fundadas sobre sus posibles ventajas e inconvenientes frente al desarrollo de soluciones Software mediante metodologías de uso general o tradicionales, extendidas actualmente.

Para lograr este objetivo principal, es necesario abordar una serie de objetivos y procesos secundarios.

El primero de ellos es estudiar, comprender y desplegar el entorno de desarrollo completo de este tipo de tecnologías. Para ello, es necesario instalar y desplegar el entorno de desarrollo de Xilinx para Aceleración Hardware, basado en diferentes elementos: programas Vitis y Vivado (para desarrollo), interfaz XRT (para comunicación con Hardware de aceleración) y drivers de despliegue y desarrollo (para ejecución de aplicaciones en Hardware acelerado). Este proceso requiere analizar la documentación oficial de Xilinx, por lo que, en este documento, se sintetiza y amplía parte de dicha información para completarla y permitir una mayor comprensión del proceso.

En segundo lugar, es necesario comprender el funcionamiento completo del concepto de aceleración Hardware y analizar cómo funciona en el entorno de Xilinx. Para ello, es necesario importar y simular una serie de proyectos de ejemplo proporcionados por Xilinx para la familiarización con estos conceptos. Además, es necesario realizar una síntesis de la documentación oficial, debido a su amplia extensión y ambigüedad (gran cantidad de conceptos tratados de manera cruzada entre diversos documentos y con la propia documentación, todavía en proceso de desarrollo). En este documento se trata también de introducir dicha síntesis y facilitar el proceso para futuros estudios.

El último, consiste en realizar un proceso completo de aceleración Hardware para una aplicación concreta, de forma que se pueda estudiar y analizar, siendo comparado con sus versiones en entornos no acelerados o de uso tradicional.

Por tanto, se puede definir el alcance del proyecto en las siguientes líneas:

- Instalación y despliegue del entorno de desarrollo de aplicaciones en Hardware de aceleración de Xilinx. Ampliación de la documentación oficial y desarrollo de un manual que facilite el proceso.
- Importación, análisis y simulación de proyectos pre-desarrollados por Xilinx para la familiarización y comprensión del entorno y de las tecnologías implicadas, así como de sus conceptos teóricos intrínsecos.
- Desarrollo de un caso de optimización de aplicación Software, mediante una tarjeta de Aceleración Hardware de Xilinx, la tarjeta Alveo U50, mediante el uso del Software Vitis-Vivado y los lenguajes de programación C y OpenCL. Para esta aplicación, se desarrolla el algoritmo de Cholesky y se compara su rendimiento en el entorno de Aceleración Hardware, frente a su ejecución en entornos tradicionales, mediante ejecución en CPU, del mismo algoritmo desarrollado para otros lenguajes de uso general y mayor abstracción, en concreto para C y Python.
- Finalmente, síntesis y análisis profundo de este tipo de alternativas para el desarrollo de soluciones Software-Hardware de alto rendimiento.

4. Beneficios que aporta el trabajo.

Gracias al desarrollo de este proyecto, se establecen las bases para comprender de manera más detallada, el entorno de desarrollo y las diferentes herramientas proporcionadas por Xilinx, para la aceleración de aplicaciones Software, mediante uso de Hardware de aceleración, aplicable a numerosos ámbitos. Esto proporciona una serie de beneficios de diferente naturaleza. De esta forma, se proporciona la información necesaria para que todo aquel interesado, pueda comprender esta tecnología y tenga las herramientas suficientes para iniciarse en el proceso de aprendizaje, teniendo una guía detallada para la instalación, configuración y primeros pasos en el entorno de desarrollo de aplicaciones aceleradas.

Los principales beneficios del proyecto y la tecnología estudiada y analizada, teniendo en cuenta la base para futuros proyectos que establecen, se pueden agrupar en tres grandes bloques:

- Beneficios técnicos:

    1. Desarrollo tecnológico de un Hardware con un gran potencial, gracias a su mayor conocimiento y extensión del ámbito de aplicación. Esto permite dar a conocer la tecnología y promoverla en un mercado más competente, promoviendo el desarrollo, más intenso y rápido, de los dispositivos electrónicos.
    2. Búsqueda de soluciones óptimas de desarrollo Software sobre implementaciones Hardware adaptadas. Permite lograr mejores optimizaciones de los programas desarrollados, optimizando los recursos disponibles.
    3. Mayor innovación en Ingeniería del Software, mediante la introducción de nuevos paradigmas de desarrollo que buscan programas optimizados y personalizados, para arquitecturas Hardware configurables. Reaparecen términos como la paralelización, optimización de recursos o diseño de arquitecturas lógicas Hardware, entre otros, dejadas de lado actualmente, para desarrollar aplicaciones y dispositivos más funcionales, con menor optimización, rendimiento y flexibilidad. Trata de solucionar la tendencia degenerativa del desarrollo de soluciones tanto Software, como Hardware,

hacia elementos estáticos y que buscan simplemente acceder lo antes posible al mercado, sin buscar un verdadero desarrollo tecnológico de calidad.

- Beneficios económicos:

    1. Se logra una mayor amortización del Hardware adquirido, permitiendo la reutilización en diferentes proyectos y etapas de los mismos, sin requerir el cambio de los dispositivos para mejorar el rendimiento por cambio generacional. Esto se consigue gracias a que se utilizan dispositivos reconfigurables, cuya arquitectura puede ser definida de manera flexible, para adaptarse de manera personalizada y óptima a las aplicaciones, sea cual sea su estado o versión de desarrollo. Disminuye también costes operativos, gracias a un menor consumo energético que sus competidoras para desempeñar las mismas funciones, al mismo rendimiento.
    2. Abaratamiento de la tecnología si se consigue generalizar su uso, al generar un mercado con mayor competencia y oferta, permitiendo la participación de nuevos fabricantes. Actualmente, es un mercado bastante cerrado, en que grandes fabricantes concentran desarrollo, fabricación, venta y distribución de la tecnología.

- Beneficios sociales:

    1. Mejora en numerosos ámbitos sociales, como la salud, la ciencia o la economía, entre muchos otros, gracias al Hardware de alto rendimiento, que permite una optimización y adaptación a cada aplicación particular, proporcionando mejores resultados en algoritmos y programas aplicados a la investigación clínica, farmacéutica, universitaria, científica, analítica de datos, financiera y de fraude, y muchos otros. Esta mejora en los procesos de dichos estudios, permite mejorar la calidad de vida de las personas en diferentes grados, junto con una mejora del funcionamiento global de la sociedad, servicios y mercados.
    2. Mayor respeto por el Medio Ambiente, mediante el uso de Hardware reutilizable gracias a la reconfiguración de sus componentes, reduciendo considerablemente la generación de residuos electrónicos entre generaciones

de dispositivos, fomentados por la tendencia actual hacia una electrónica con arquitecturas y componentes estáticos, abocados a la sustitución tras cierto tiempo de uso. También favorece un menor consumo energético con un mejor aprovechamiento de recursos.

5. Análisis del estado del arte.

En esta sección, se muestra una síntesis de los conceptos tratados en los diferentes apartados del Anexo A, para permitir una lectura más breve y amena. Se proporciona mayor información, para los lectores más exigentes, en sus respectivos subapartados del anexo correspondiente, indicados al inicio de cada uno de los puntos de esta sección.

5.1. Tecnologías implicadas en el estudio: CPU, GPU, FPGA y ASIC

Más detalles en Anexo A.1 en la página 125.

En las primeras etapas de los sistemas de tecnologías de la información y la comunicación, las arquitecturas de sistema no estaban pensados para las aplicaciones actuales, en cuanto a requerimiento computacional, ni en cuanto a capacidad de procesamiento de volumen de datos. Estos sistemas han tenido que adaptarse con la evolución de estas tecnologías, las cuales se han desarrollado de manera exponencial en los últimos años. Se espera que dicha tendencia siga creciendo considerablemente.

Por tanto, para alcanzar las demandas actuales, el Hardware debe evolucionar en términos de latencia, confiabilidad, movilidad, seguridad, eficiencia energética y costes de transmisión de datos. Para ello, se debe abandonar en cierta medida el desarrollo estático basado en arquitecturas y reglas estáticamente definidas, para tomar un rumbo hacia arquitecturas y entornos basados en datos y funcionalidades dinámicas.

En función del caso de uso, existen diferentes posibilidades en cuanto a núcleos de procesamiento, pero, en cuanto a lo que a esta memoria se refiere, es importante destacar tres elementos principales, de los cuales se comparan dos, CPU y FPGA, durante la solución propuesta. Estos tres núcleos de procesamiento son:

- CPU: “Central Processing Unit” o “Unidad Central de Procesamiento”. Unidad de procesamiento de uso general que cuenta con una serie de núcleos, capaces de realizar ciertas tareas, simples o complejas, de manera secuencial. Versatilidad con baja optimización de recursos. Los núcleos, se encargan de ejecutar conjuntos predefinidos de instrucciones, con rutas de datos con anchos fijos. Las capacidades de ejecución están definidas desde fabricación y las funcionalidades son exclusivas de los elementos internos que conformen la CPU (núcleos, periféricos, etc.).

- GPU: “Graphics Processing Unit” o “Unidad de Procesamiento Gráfico”. Se trata de una unidad de procesamiento capaz de realizar numerosas tareas en paralelo, contando para ello con un número de núcleos mucho más elevado que en el caso de una CPU. Presenta más núcleos, pero menos eficientes individualmente que los correspondientes a una CPU. Requiere mayor tamaño y consumo energético. De uso general, óptimo para tareas susceptibles de paralelización. La arquitectura está definida desde el proceso de fabricación, con las limitaciones que conlleva.
- FPGA: “Field-Programmable Gate Array” o “Matriz de Puertas Lógicas Programable en Campo”. Es un sistema que cuenta con una serie de puertas lógicas reconfigurables, tras su fabricación, para diseñar arquitecturas de procesamiento personalizadas, adaptadas a la aplicación concreta deseada. Presenta mejor uso de recursos con un menor consumo energético respecto a las alternativas CPU y GPU. Es Hardware configurado en campo (tras fabricación, antes de implementación y ejecución) específicamente para la aplicación para la cual se desarrolla, permitiendo un mejor rendimiento y potencial computacional, gracias a la posibilidad de diseñar arquitecturas flexibles y adaptables, de diferente naturaleza, sobre el mismo dispositivo electrónico.

La arquitectura y las principales diferencias de estas alternativas se muestran respectivamente en la Figura 1 y en la Tabla 1

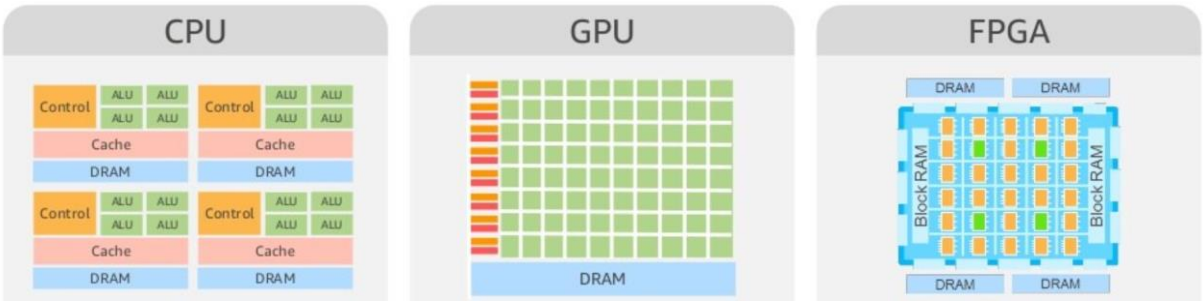


Figura 1:Componentes arquitecturas CPU, GPU y FPGA (Resumen) (Gómez, 2020)

Tipo	Consumo	Descripción	Ventajas	Inconvenientes
CPU	Alto	Flexible, uso general	Tareas e instrucciones complejas.	Lento acceso a memoria. Pocos núcleos de procesamiento
GPU	Alto	Numerosos núcleos con gran capacidad de paralelización	Alto rendimiento con gran capacidad de paralelización	Elevado consumo energético. Gran tamaño
FPGA	Medio	Puertas lógicas reconfigurables	Flexibilidad. Programabilidad in-field	Elevado consumo energético. Complejidad de programación
ASIC	Bajo	Lógica personalizada estática	Rapidez y bajo consumo. Pequeño tamaño	Funcionalidad fija. Coste económico elevado

Tabla 1: Características principales CPU, GPU, FPGA y ASIC (Resumen) (ADLINK, 2021)

La necesidad de analizar estos elementos se debe a la diferenciación no solo en las arquitecturas internas y a las posibilidades que permiten en cuanto a potencial computacional, latencias y flexibilidad de adaptación, sino también, al modo en que realizan las funciones de computación. De esta forma, se pueden diferenciar dos modos de operación computacional:

- Temporal: funcionamiento secuencial, operación tras operación, función tras función, cada una por unidad de tiempo.
- Espacial: funcionamiento paralelo, mediante definición de caminos de datos que permiten ejecutar simultáneamente varias tareas, en la misma unidad de tiempo, sobre el conjunto de datos a procesar.

En la Figura 2 se muestra un ejemplo de esta diferenciación, donde se observan los beneficios que aporta el uso de una FPGA, sobre una CPU, logrando una computación espacial, de forma que, simultáneamente, se procesan y ejecutan diferentes instrucciones sobre el flujo de datos de entrada, mejorando la capacidad de rendimiento y disminuyendo latencias. Esto es fundamental y el punto de partida de este proyecto, estableciendo la base de la aceleración Software mediante Hardware.

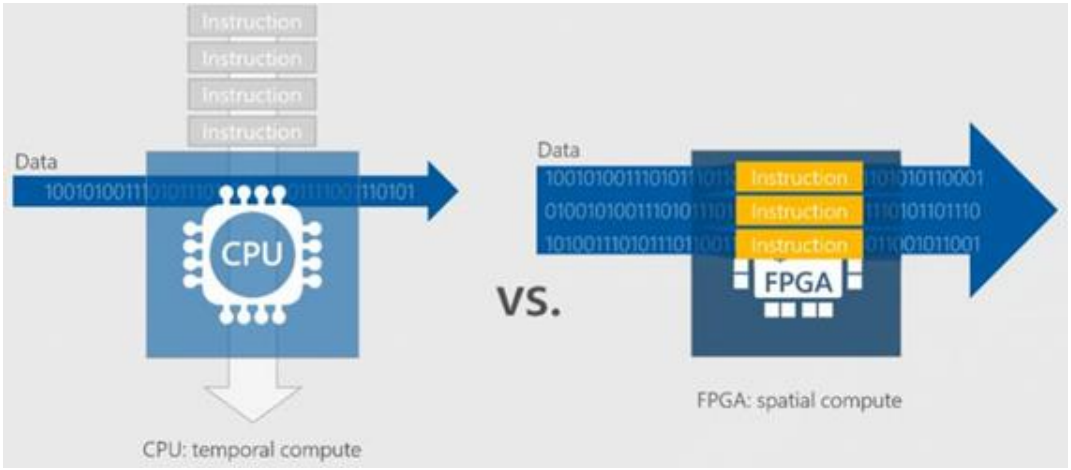


Figura 2:Procesamiento y secuencia de instrucciones para CPU y FPGA (Resumen) (Gómez, 2020)

5.2. Software Xilinx: Vitis Unified Software Platform

Más detalles en Anexo A.2 en la página 130.

Para permitir el desarrollo de aplicaciones aceleradas sobre Hardware de aceleración, Xilinx proporciona un entorno completo de desarrollo que denomina “Vitis Unified Software Platform” o “Plataforma Unificada de Desarrollo Vitis”. Junto al entorno de desarrollo, proporciona, además, una serie de abstracciones, metodologías y pasos para el correcto desarrollo de aplicaciones aceleradas.

El esquema general se divide en dos grandes componentes, comunicados mediante tecnología PCIe:

- Programa Host: Se trata de un programa escrito en C/C++, desarrollado y compilado mediante APIs, como OpenCL, ejecutado sobre un procesador del host o una CPU (procesador x86, en el caso de las tarjetas de aceleración). Se construye mediante compiladores GNU C++ y se enlazan las librerías correspondientes con la librería compartida “Xilinx Runtime Library” (XRT).
- Kernel Plataforma de Aceleración: Es un programa generado y compilado para ser ejecutado sobre el Hardware de aceleración, es decir, sobre la lógica programable de los dispositivos Xilinx. Utiliza lenguajes C, C++ o RTL y se construye con un compilador propio de Vitis (v++), para generar los archivos binarios correspondientes que se ejecutan sobre la FPGA, en formato “.xclbin”.

En la Figura 3, se muestra un esquema de estos componentes y la secuencia de pasos para el intercambio de datos entre ambos. Básicamente consiste en que, el programa host, ejecutado sobre la CPU del mismo, interactúa con los dispositivos FPGA mediante APIs OpenCL. Mediante la gestión de la librería XRT, se consiguen realizar intercambios de información entre host y kernels a través de PCIe. Aparecen entonces dos zonas de memoria separada, una relacionada con las señales de control correspondientes tanto para ejecución de instrucciones como para configuración de componentes, y otra, destinada exclusivamente al intercambio de información útil para su procesamiento. Esta segunda recibe el nombre de memoria global, y ambos elementos, host y kernel, cuentan con acceso a la misma.

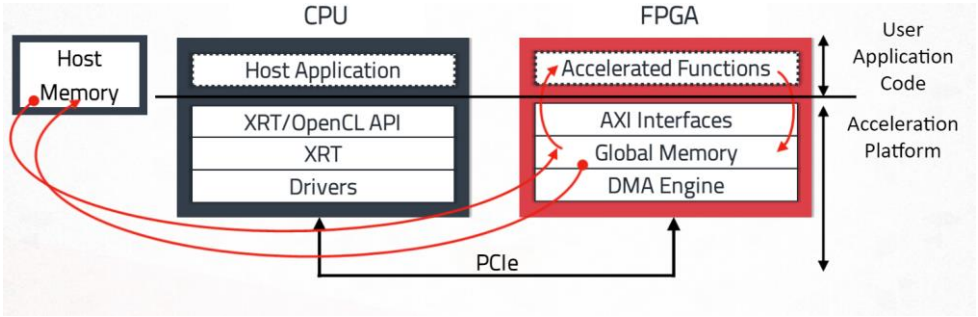


Figura 3: Secuencia de pasos a ejecutar en modelo de aceleración propuesto por Xilinx (Resumen) (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

Teniendo esto en cuenta, el funcionamiento general de cualquier aplicación acelerada se puede simplificar al siguiente patrón: el host prepara los datos para ser procesados y realiza las tareas menos exigentes para, a continuación, transmitir los datos al dispositivo de aceleración, que ejecuta las tareas más complejas y exigentes, tras lo cual, devuelve los datos procesados al host.

Para lograr este funcionamiento e interacción entre ambos dispositivos, es necesario generar los programas ejecutables del host y del kernel. El proceso seguido se muestra en la Figura 4, donde se aprecia que son dos procesos claramente diferenciados, mediante el uso de sus respectivas herramientas.

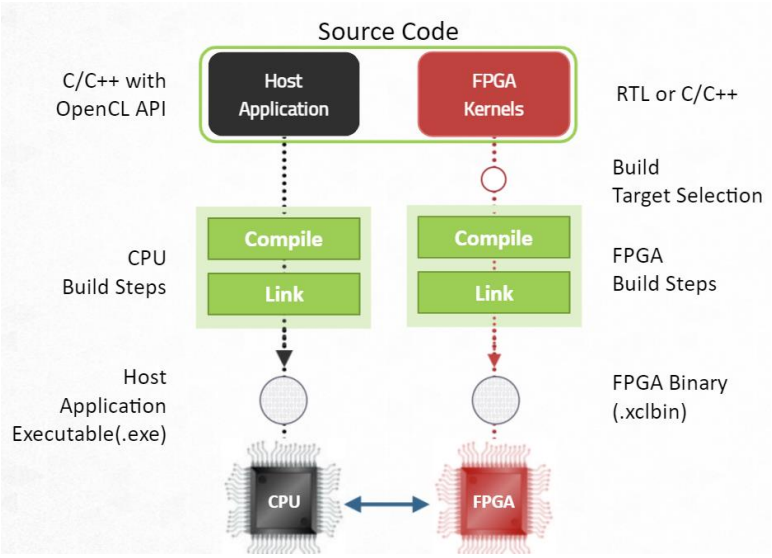


Figura 4: Proceso de generación de ejecutables host y kernel (Resumen) (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

El gran inconveniente de esta metodología es la latencia introducida por las transferencias, a través de PCIe, introducidas en los intercambios de información entre host y kernels. Es por ello que hay que tratar de minimizar estos intercambios.

La aceleración de cada aplicación consiste en un proceso particular, en función de la naturaleza de la misma y de los datos a procesar, de forma que Xilinx solo proporciona una serie de recomendaciones y metodologías, aportando ejemplos y casos de uso para la inicialización en este ámbito, pero sin dar soluciones globales a los problemas particulares de cada programa o algoritmo concreto.

Para realizar el proceso completo, Xilinx proporciona la plataforma unificada comentada, contando con las soluciones Software, librerías, drivers e interfaces necesarias. Una representación de los diferentes niveles de abstracción en los que permite actuar la plataforma, se observa en la Figura 5, donde se aprecian los diferentes niveles:

- Desarrollo de aplicaciones: permite desarrollar programas en diferentes lenguajes de uso general como Python, C o C++, para los hosts, así como para las plataformas de aceleración, proporcionando librerías de abstracción de alto nivel para facilitar el proceso. Además, en la parte de aceleración, soporta desarrollos mediante OpenCL y/o RTL. También existen diferentes soluciones ya desarrolladas que pueden ser importadas y se encuentran preparadas para su ejecución.
- Herramientas de desarrollo Vitis: se proporcionan compiladores para construir los ejecutables y las librerías necesarias, depuradores para verificar el correcto funcionamiento de las aplicaciones y optimizar el desarrollo, y analizadores, para elaborar perfiles de la ejecución de las aplicaciones y determinar limitaciones y posibles vías de mejora o puntos de no complimiento de requisitos. Para la correcta comunicación entre los elementos compilados tanto del host como de la plataforma de destino, se hace uso de XRT.
- Despliegue de aplicaciones: en la parte del host, XRT se encarga de dar las directrices necesarias mediante PCIe a los dispositivos de aceleración, que cuentan con los binarios de la aplicación que contienen los kernels preparados para su ejecución. Consiste en el entorno de ejecución final y Xilinx proporciona los dispositivos adecuados para esta labor. Existen soluciones tanto On-Premise como Cloud.

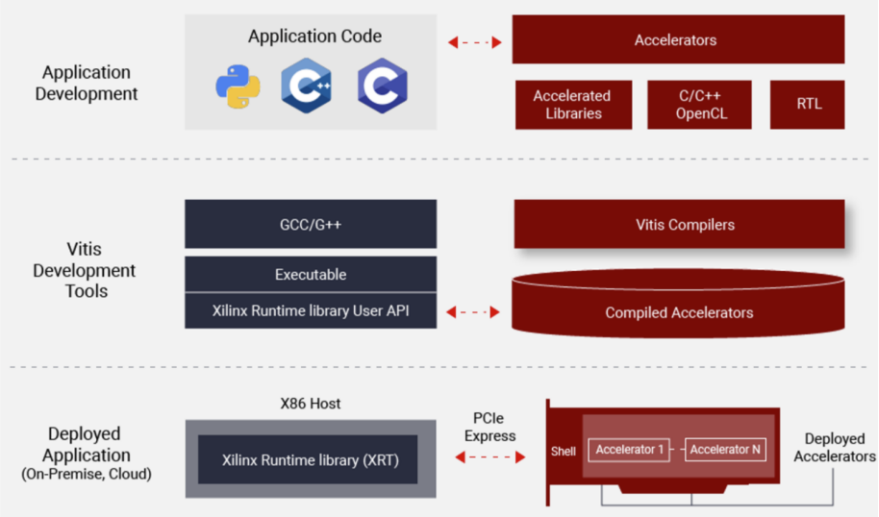


Figura 5: Niveles de abstracción de entorno de desarrollo unificado Xilinx Vitis (Resumen) (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

Un aspecto importante es la capacidad que proporcionan las herramientas de desarrollo para ejecutar las aplicaciones desarrolladas. Existen dos modos de emulación, que no requieren siquiera con contar con el dispositivo de aceleración físico, y uno de ejecución sobre Hardware, que si lo requiere. Esto permite una mayor flexibilidad para organizar las diferentes etapas de los procesos de diseño, pudiendo dividir las primeras etapas de desarrollo y depuración iniciales, ejecutadas sobre entornos controlados de emulación, y una etapa final de despliegue o ejecución final en entorno de interés sobre el equipo final, ya adquirido solamente cuando sea necesario. Estos tres modos son:

- Emulación Software: todo el código, de host y de kernel, es compilado para su ejecución en el procesador del host, de forma que permite procesos de compilación-ejecución más rápidos, para depuración y refinamiento de algoritmos. Permite detectar errores de sintaxis y depurar a nivel de código el programa kernel, junto al del host. No permite conocer cómo se comportará finalmente en el dispositivo de destino.
- Emulación Hardware: el código del kernel es compilado en un modelo de Hardware (RTL) y ejecutado sobre un simulador dedicado, de forma que, aunque el proceso de compilación-ejecución sea más lento, proporciona una visión más detallada y precisa de la actividad del kernel. Permite una primera aproximación del rendimiento real de la aplicación final, probando el funcionamiento de la lógica que se implementa en etapas posteriores sobre la FPGA.

- Ejecución Hardware: el código del kernel es compilado en un modelo Hardware (RTL) implementado directamente sobre la FPGA. Es el entorno de despliegue o ejecución final de la aplicación, en la cual debe ejecutarse de manera óptima sobre el dispositivo de aceleración físico. En este punto se ejecuta el programa host sobre el procesador del host y los kernels directamente sobre la FPGA, para realizar las comunicaciones mediante las interfaces PCIe cuando sea necesario.

Como resumen de la documentación, se pueden extraer unas conclusiones de cara a entender de manera general la filosofía de Xilinx para el entorno de desarrollo unificado de aceleración Hardware Vitis:

1. Mediante APIs, se trata de abstraer al desarrollador de la complejidad del propio Hardware y de las funcionalidades de bajo nivel de la aceleración de aplicaciones mediante dispositivos FPGA. Mediante llamadas a funciones definidas en la API OpenCL, el programa host puede descubrir dispositivos, enviar datos, solicitar ejecuciones de kernels y recuperar los datos procesados.
2. El paradigma se divide en dos grandes elementos: el programa host (que se ejecuta sobre procesadores) y el kernel (ejecutado sobre el Hardware de aceleración). Ambos elementos se comunican mediante una interfaz, en el caso de la tecnología utilizada, una interfaz PCIe.
3. No todos los algoritmos son susceptibles de ser acelerados, ya que es necesario entender el funcionamiento y comportamiento de los algoritmos implicados, para conocer si pueden existir beneficios de su aceleración y estableciendo un objetivo acorde a las plataformas e interfaces utilizadas.

5.3. Hardware Xilinx: tarjetas de aceleración Alveo

Más detalles en Anexo A.3 en la página 147.

Para solucionar la necesidad de Hardware a la altura de los requerimientos computacionales actuales y de las tendencias futuras esperadas, Xilinx ha desarrollado un conjunto de tarjetas, tanto de evaluación como de producción, para proporcionar soluciones configurables de manera personalizada a las demandas particulares de cada programa y de cada ámbito de aplicación. De esta forma, pretende solucionar los problemas de arquitecturas estáticas, como es el caso de las CPU y GPU, para permitir una mejor adaptación al gran crecimiento y la fuerte variación de las aplicaciones en cortos periodos de tiempo. Esto permite desarrollar aplicaciones y programas adaptables al propio crecimiento computacional y de los propios algoritmos, respetando los dispositivos Hardware adquiridos, permitiendo una mejor amortización mediante su reutilización en sucesivas etapas de los proyectos.

Junto a las propias tarjetas, Xilinx ha evolucionado hacia un ecosistema completo, como se observa en la Figura 6. De esta forma, pretende proporcionar todos los elementos que componen el entorno, tanto de desarrollo como de producción, así como los dispositivos físicos, para permitir una mejor cooperación entre todos los componentes, sin necesidad de recurrir a soluciones de terceros. Mediante este ecosistema completo, trata de permitir el desarrollo de aplicaciones de alto rendimiento, desarrolladas mediante un entorno que permite optimizar el proceso para el Hardware específico de alto rendimiento reconfigurable comentado.

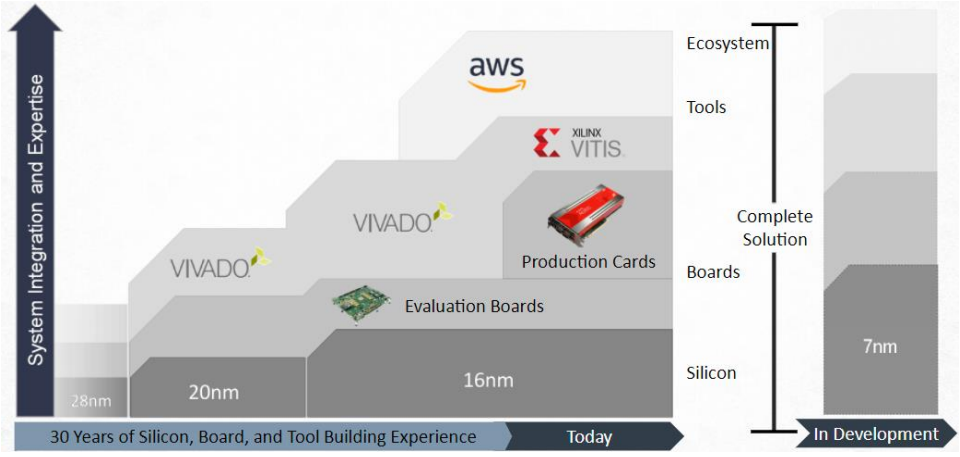


Figura 6: Evolución del silicio, tarjetas y herramientas Xilinx (Resumen) (Xilinx, Developer Program Secure Site: Using Xilinx Alveo Cards to Accelerate Dynamic Workloads, 2021)

En cuanto a lo que al proyecto concierne, se centra el foco en las tarjetas de aceleración Alveo, ya que permiten estudiar y analizar la aceleración de aplicaciones. Además, se cuenta con una tarjeta física en el propio grupo de investigación. Por ello, de entre todos los modelos, se selecciona la tarjeta Alveo U50.

La filosofía que sigue Xilinx en cuanto a las tecnologías que envuelven a la colección de tarjetas de aceleración se basa en tres conceptos principales, los cuales son de especial interés para comprender el planteamiento del fabricante en cuanto a las tarjetas y al entorno completo unificado. Estos puntos clave son:

- Rapidez y rendimiento: Se busca un mayor rendimiento y menores latencias respecto a los casos CPU y GPU, de forma que se logre optimizar la ejecución de las aplicaciones ya desarrolladas o la creación de nuevas aplicaciones ya optimizadas para la aceleración. La clave de la aceleración frente al resto de tecnologías, se basa en la capacidad de adaptación de la propia arquitectura a los requerimientos de memoria y de caminos de datos, que permitan la optimización de la aplicación. De esta forma, se consigue entregar de manera directa los datos al bloque físico de procesamiento, sin requerir etapas intermedias que añadan latencias y cuellos de botella. Un ejemplo de ello se muestra en la Figura 7, donde se observa la diferencia entre las entregas de datos para cada bloque de ejecución funcional.

  Varios estudios, mostrados en el Anexo correspondiente, indican índices de mejora de rendimiento frente al caso tradicional de las CPUs de hasta en 90 veces, para casos como gestión de bases de datos, o de hasta en 20 veces, para el caso de computación en Machine Learning. Además, se muestra también cómo se puede mejorar el rendimiento global de ciertas aplicaciones mediante la interacción CPU-Alveo (según plantea Xilinx). Un punto importante adicional es el correspondiente al consumo energético, ya que el uso de las tarjetas Alveo, permite optimizar este parámetro, frente a los casos CPU y GPU analizados.

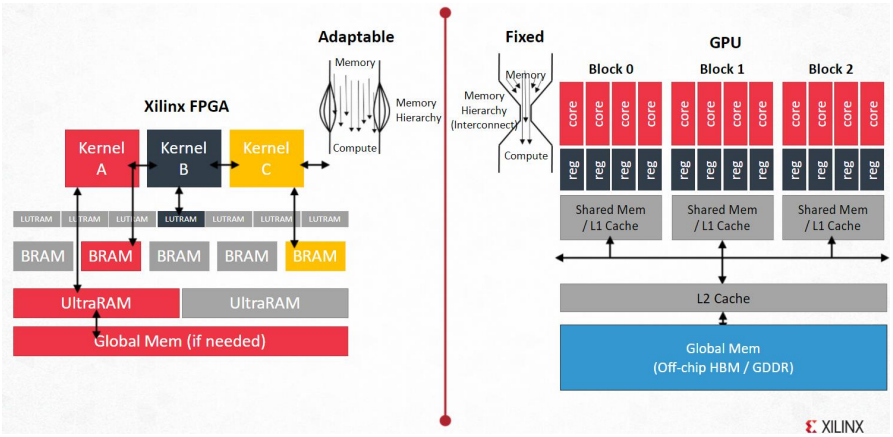


Figura 7: Diferencias en acceso a datos y flujo de los mismos a través del flujo de ejecución, para arquitecturas FPGA y GPU (Resumen) (Xilinx, Developer Program Secure Site: Using Xilinx Alveo Cards to Accelerate Dynamic Workloads, 2021)

- Adaptabilidad para acelerar cualquier flujo: El principal objetivo es permitir la optimización de cualquier flujo o aplicación, tanto existentes, como por desarrollar, para permitir una aceleración adecuada, optimizando rendimiento y latencias, sin necesidad de sustituir el Hardware sobre el que se implementan las soluciones. Gracias a la reprogramación de este tipo de dispositivos, se consiguen adaptar las aplicaciones, para lograr un alto rendimiento computacional, conforme evoluciona la propia aplicación en sí, junto a sus algoritmos y funcionalidades internas. Trata de evitar retardos en desarrollo por cambio generacional del Hardware estático, proporcionando flexibilidad y versatilidad del Hardware programable, para conseguir mejores soluciones, con un mejor rendimiento, independientemente del ámbito de aplicación. Esta adaptabilidad de los dispositivos Alveo se ve en Figura 8.

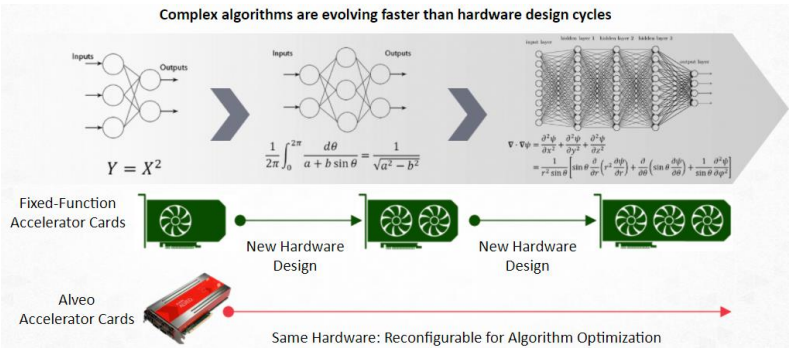


Figura 8: Evolución del hardware con el desarrollo de los requerimientos de los algoritmos implicados (Resumen) (Xilinx, Developer Program Secure Site: Using Xilinx Alveo Cards to Accelerate Dynamic Workloads, 2021)

- Accesibilidad: Xilinx trata de proporcionar soluciones que permitan acceder a todos los procesos que engloba la aceleración de aplicaciones, de forma que, tanto aquellos usuarios que deseen implementar directamente soluciones de terceros, puedan acceder a las soluciones Hardware y Software necesarias, como aquellos usuarios más exigentes, que deseen aplicaciones personalizadas, puedan ser capaces de elaborarlas con un entorno completo de desarrollo que simplifique en la mayor medida posible, el proceso y los tiempos entre el inicio del desarrollo y la puesta en marcha en entornos de producción. Para ello, Xilinx propone soluciones Cloud operativas desde tiempo cero, como soluciones On-Premise, mediante la adquisición y el montaje de las tarjetas físicas en equipos propietarios del usuario en cuestión.

Resumiendo, los principales beneficios que aporta este entorno junto con las tarjetas Alveo son:

- Posibilidad de implementación en ámbitos y aplicaciones de naturaleza diversa.
- Mejora de rendimiento frente a CPU/GPU, con una optimización y aceleración adecuada de las aplicaciones.
- Menor coste total de adquisición, permitiendo conservar el Hardware a lo largo de la evolución de las aplicaciones y funcionalidades.
- Visión de futuro, permitiendo adaptar tanto el Software, como el Hardware, a la evolución de los flujos de trabajo y nuevas tendencias comerciales y tecnológicas.

Para la instalación del equipamiento, Xilinx divide el entorno en dos modos diferenciados de operación, cada uno con unos requerimientos de instalación y configuración determinados:

- Despliegue (Deployment): Es el entorno de producción en que la aplicación se espera ejecutar, proporcionando el servicio o la funcionalidad esperada. Para poder ejecutar las aplicaciones en este modo, las herramientas necesarias son la librería XRT, junto con la plataforma de destino de despliegue (ya sea junto con la tarjeta física montada, o mediante arquitectura Cloud, con la tarjeta montada en un servidor remoto). Requiere los pasos 1 y 2 del proceso de instalación comentado posteriormente.
- Desarrollo (Development): Se corresponde con el entorno de desarrollo o codificación y depuración de las propias aplicaciones. Permite desarrollar, compilar, depurar y

analizar las aplicaciones aceleradas, para posteriormente ser desplegadas sobre el Hardware de aceleración. Requiere de un mayor número de herramientas software y librerías, teniendo que seguir la totalidad de los pasos del proceso de instalación comentado posteriormente.

De esta forma, ambos modos de operación dividen el flujo para permitir diferentes instalaciones, siendo los pasos de la instalación completa los siguientes:

1. Instalación de librería Xilinx Runtime o XRT. Incluye las API y drivers necesarios para la comunicación entre host y tarjeta Alveo.
2. Instalación de plataforma de despliegue o deployment destino. Incluye el nivel físico de comunicación implementado e instalado en la propia tarjeta de aceleración.
3. Instalación de plataforma de desarrollo o development destino. Incluye el interfaz para el desarrollo de aplicaciones que generan nuevos niveles físicos implementados sobre la propia tarjeta de aceleración.
4. Instalación del entorno de desarrollo unificado Vitis. Conjunto de herramientas para el desarrollo de Software acelerado, para su posterior implementación sobre Hardware de aceleración. Proporciona las herramientas, tanto para el desarrollo, como para la generación de los ficheros ejecutables. También permite la carga de los programas en las tarjetas, así como la ejecución de los mismos una vez completado el proceso de desarrollo y configuración.

El proceso de instalación se explica en un apartado diferenciado tanto en la descripción de la solución, como en el Anexo correspondiente.

También es importante comentar que para poder ejecutar y desarrollar aplicaciones aceleradas, es necesario contar con equipos suficientemente potentes y que cuenten con los interfaces PCIe correspondientes. En la Figura 9 se muestran los requerimientos mínimos recomendados por el fabricante, para este tipo de tareas, que deben cumplir los ordenadores empleados para ellas. Para facilitar la compatibilidad, Xilinx proporciona una lista de servidores certificados (lista accesible en las referencias de este documento).

Component	Requirement
Motherboard	PCI Express 3.0 compatible with one dual-width x16 slot
System Power Supply	75 W(U25, U50), 225 W(U200, U250, U280)
Operating System	Linux, 64-bit • Ubuntu 16.04, 18.04, 20.04 • CentOS 7.4, 7.5, 7.6, 7,7, 7.8, 8,1, 8.2 • RHEL 7.4, 7.5, 7.6, 7.7, 7.8, 8.1, 8.2
System Memory	Minimum of 64 GB, but 80 GB is recommended
Hard Disk	Satisfy the minimum system requirements for your operating systems
Internet Connection	Required for downloading drivers and utilities

Figura 9: Requerimientos mínimos recomendados por Xilinx para equipos destinados a desarrollo y/o despliegue de aplicaciones aceleradas en tarjetas Alveo (Resumen) (Xilinx, Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393), 2021)

5.3.1. Xilinx Alveo U50

Más detalles en Anexo A.3.1 en la página 161.

Para el desarrollo de este proyecto, se selecciona la tarjeta de aceleración Alveo U50, ya que se cuenta con un dispositivo físico de este modelo en el grupo de investigación.

Se trata de una tarjeta de un único slot, con factor de forma de pequeño tamaño, con una potencia máxima de operación de 75W. Soporta comunicaciones PCI Express Gen3x16 o dual Gen4x8, conectividad Ethernet y cuenta con una memoria de 8GB de tipo HBM2. Con estas características, se recomienda su uso para aplicaciones ligadas a capacidades de memoria y a una computación intensiva. Cuenta, además, con un conector óptico QSFP28, para comunicaciones de hasta 4 canales individuales a 25 Gb, cada uno.

Este dispositivo está preparado para su integración con el software unificado de Xilinx, de forma que, mediante lenguajes de alto nivel como C, C++ y OpenCL, se pueda desarrollar aplicaciones aceleradas y, mediante la plataforma de destino correspondiente y las librerías y drivers necesarios, cargar los kernels y configurar la lógica programable en las FPGA, para su ejecución sobre el Hardware de aceleración. Además de las propias aplicaciones, mediante el Software Vivado Desing Suite, es posible modificar y desarrollar la propia plataforma de destino que define la configuración de la lógica de la tarjeta de aceleración, permitiendo un mayor número de posibilidades y aportando un mayor potencial.

Un diagrama de bloques de la tarjeta Alveo U50 se muestra en la Figura 10, donde se aprecian los siguientes elementos principales:

- Conector PCIe: la FGPA cuenta con un bloque PCIE4C que permite comunicaciones de hasta 8 GT/s (Gen3x16) o de hasta 16GT/s (Gen4x8).
- Interfaces de comunicaciones y red: gracias al componente QSFP28, de cuatro líneas separadas, que aceptan módulos de hasta 5W, se consigue conectar interfaces de hasta 100G. Cuenta con un reloj dedicado, permitiendo habilitar diferentes núcleos IP Ethernet. Además, cuenta con una memoria flash no volátil, de 1 Gb, para configuración, mediante comunicación Quad SPI.
- Controlador satélite: es un TI MSP432 que permite controlar y monitorizar tensiones, corrientes y temperaturas de la tarjeta, de forma que el controlador del host, pueda interactuar con el controlador satélite interno de la tarjeta para controlar y monitorizar

la misma, mediante comunicaciones fuera de banda. Soporta protocolo PLDM sobre MCTP, sobre SMBUS. Permite detectar cualquier mal funcionamiento.

- Puerto de mantenimiento: puerto con un conector de 30 pines que permite acceder a diferentes funcionalidades y señales como JTAG; UARTs, PMBus y resets. Xilinx proporciona una tarjeta independiente para la conexión a este puerto.
- FPGA: Se trata de una FPGA Ultrascale+ personalizada para funcionar de manera óptima en arquitecturas Alveo. Es una FPGA XCU50, que utiliza tecnología "Stacked Silicon Interconnect" para mejorar sus prestaciones, permitiendo incrementar la densidad, combinando múltiples "Super Logic Regions" (SLR). Cuenta con dos SLR, siendo la inferior la encargada de la comunicación hacia la memoria de 8GB mediante dos interfaces HBM2.

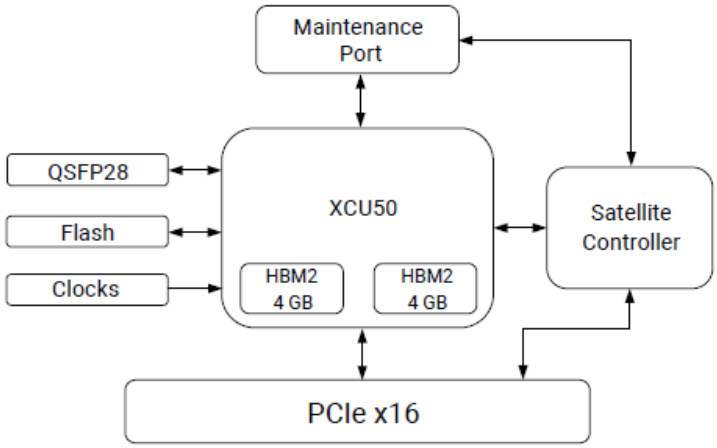


Figura 10 Diagrama de bloques tarjeta Alveo U50 (Resumen) (Xilinx, Alveo U50 Data Center Accelerator Card Data Sheet (DS956), 2020)

Cuando se hace uso del entorno de desarrollo Vitis, se crea una plataforma específica para gestionar todas las interfaces PCIe, las transferencias de datos y el intercambio de información de estado de la tarjeta, permitiendo, además, cargar los kernels sobre el propio Hardware para su posterior ejecución. Esta plataforma se encuentra sobre la región estática de la FPGA, no reconfigurable, que consume ciertos recursos de la tarjeta. Además de esta región estática, existe una dinámica, en la cual se cargan los kernels propiamente dichos. El esquema de estas regiones de la tarjeta Alveo U50 se muestra en la Figura 11.

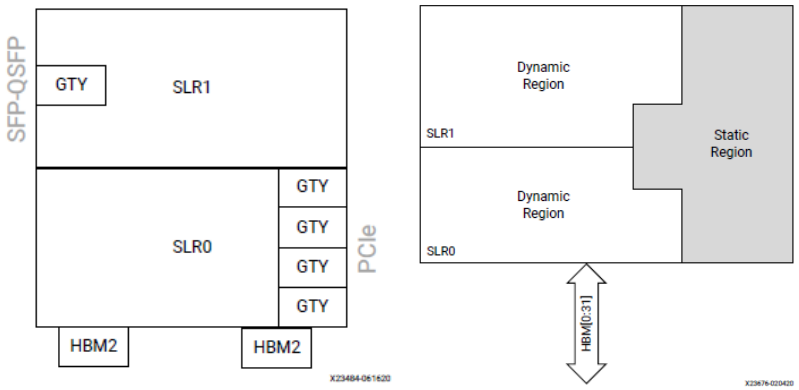


Figura 11: Esquema de distribución de regiones lógicas de FPGA XCU50 (Resumen) (Xilinx, Alveo U50 Data Center Accelerator Card Data Sheet (DS956), 2020)

De esta forma, la región estática proporciona la infraestructura básica para la comunicaciones host-Alveo, así como el Hardware necesario para soportar el kernel. Esta región estática cuenta con una serie de bloques definidos, mostrados la Figura 12, que permiten una correcta gestión y ejecución del kernel sobre la región dinámica.

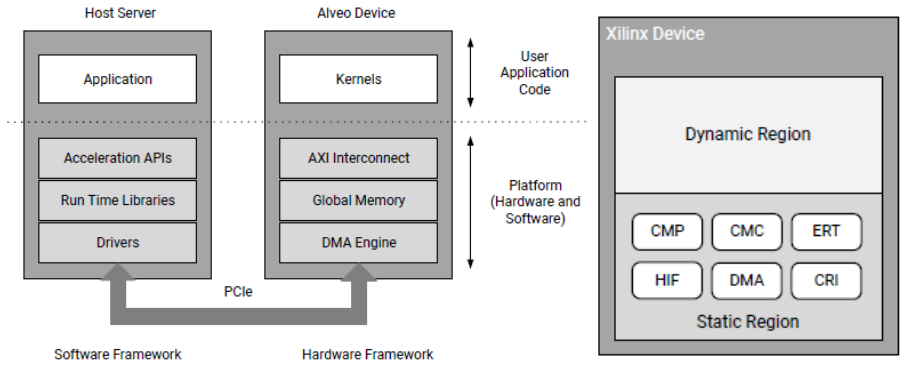


Figura 12: Esquema de distribución de componentes en regiones lógicas de plataforma destino Xilinx (Resumen) (Xilinx, Alveo Data Center Accelerator Card Platforms: User Guide (UG1120), 2021)

5.4. Informática de Alto Rendimiento.

También conocida por su nombre en inglés “High Performance Computing”, es una práctica que consiste en agregar potencia de computación, por encima de lo que cualquier ordenador o estación de trabajo típicos puedan proporcionar, de forma que se consiga un mayor rendimiento a la hora de resolver problemas complejos relacionados con la ciencia, ingeniería o negocios, entre otros. Busca un procesamiento más eficiente, confiable y rápido que el de métodos tradicionales. (insideHPC, 2021)

El fundamento de esta tecnología se basa en la paralelización de las unidades de procesamiento, de forma que se consiga un mayor rendimiento por unidad de tiempo. Esto se puede conseguir con Hardware de alto rendimiento o mediante la combinación de Hardware más sencillo, dando lugar a clústeres o superordenadores, según la forma de agrupación.

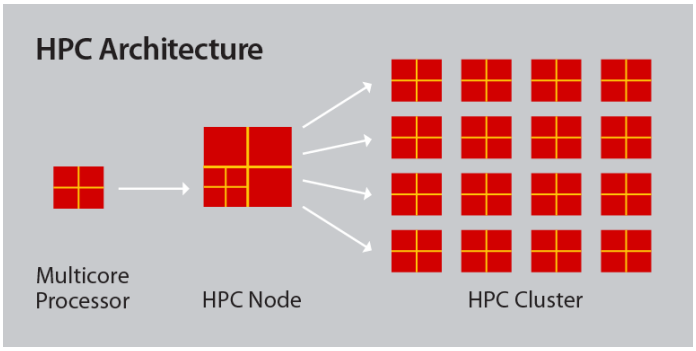


Figura 13: Escalado de arquitectura HPC (Morgan, 2020)

Son muchos los ámbitos que se pueden beneficiar de esta práctica, por lo que se destacan solamente unos pocos ejemplos para caracterizar el potencial de dicha tecnología, tanto en plataformas cloud como on-premise: (NetApp, 2021)

- Laboratorios de investigación: búsqueda de soluciones para investigaciones de energía, comprensión del universo, predicción de desastres naturales o descubrimiento de nuevos materiales.
- Multimedia, social y entretenimiento: procesamiento de vídeo, audio e imagen, transmisión en directo de eventos, redes sociales e ingeniería social o comunicaciones globales.

- Excavación y recursos naturales: detección de lugares de extracción de recursos de manera eficiente y adecuada, determinación de disponibilidad de recursos o predicción de producción energética en base a recursos disponibles.
- Inteligencia Artificial y Machine Learning: detección de fraude mediante tarjetas de crédito, soporte técnico autónomo, vehículo de conducción autónoma o detección de objetos en tiempo real.
- Servicios Financieros: análisis de mercado en tiempo real o automatización de compraventa de activos.
- Producción y manufactura: simulación de diseño de productos y de escenarios prácticos o estimación de requerimientos de producción en tiempo real.
- Salud, medicina, farmacéutica y biología: desarrollo de curas y vacunas para numerosas enfermedades, mejora de diagnósticos de cáncer o análisis del genoma humano.

Algunos casos reales, implementados sobre tecnología cloud AWS son los siguientes: (AWS, Informática de alto rendimiento, 2021)

- Dinámica de fluidos computacional: FLYING WHALES desarrolla un estudio para transporte en aeronaves más respetuosas con el medio ambiente, mejorando el rendimiento de sus simulaciones en un factor de x15. (AWS, FLYING WHALES Runs CFD on AWS to Quickly Launch Environmentally Friendly Cargo Transport Airships, 2021)
- Genómica: La Escuela de Medicina de Baylor trabaja para identificar genes que contribuyan al envejecimiento y enfermedades del corazón, almacenando y procesando más de 1 PB de datos genómicos. (AWS, Caso práctico de Baylor, 2014)
- Servicios financieros: Capital One reduce la presencia del centro de datos y reinventa su banca, migrando sus servicios a nube AWS, incorporando microservicios para mejora de las operaciones e introduciendo Machine Learning para mejor detección de fraudes logrando, además, una mejor escalabilidad del servicio. (AWS, Capital One en AWS, 2021)

- Conducción autónoma: Toyota Research Institute hace uso de tecnologías AWS para mejorar el rendimiento y procesamiento de grandes cantidades de datos recolectados para acelerar el proceso de desarrollo de su sistema de conducción autónoma, mediante Deep Learning, a nivel global. Esto le ha permitido reducir los tiempos de entrenamiento de sus modelos en un 75%, incrementando considerablemente la capacidad de investigación y desarrollo. (AWS, Toyota Research Institute accelerates safe automated driving with deep learning at a global scale on AWS, 2018)
- Salud y farmacología: AstraZeneca ejecuta 51 000 millones de pruebas estadísticas al día en torno a su línea de producción genómica para acelerar y mejorar la precisión de la medicina y trasladarlo a un descubrimiento de nuevos fármacos. (AWS, AstraZeneca’s Genomics Data Processing Solution Runs 51 Billion Tests in 1 Day on AWS, 2021)

6. Análisis de alternativas.

El principal objetivo del proyecto se centra en torno a la aceleración de aplicaciones y, para la decisión del entorno de desarrollo y despliegue de aplicaciones, se opta por utilizar las herramientas y dispositivos del fabricante Xilinx. Esta primera selección se basa en el gran potencial teórico de la tecnología, junto con la extendida repercusión mundial de las soluciones y los equipos electrónicos desarrollados por este fabricante. Esta extensión global tan consolidada, sumada a que el grupo de investigación con el que se realiza la colaboración, cuenta con una tarjeta Alveo física, habiendo trabajado sobre esta tecnología, suponen los factores condicionantes para optar por esta alternativa, la cual condiciona considerablemente el resto del desarrollo del proyecto.

Por tanto, se define como entorno de desarrollo, el entorno unificado de desarrollo Software de aceleración denominado Xilinx Vitis. Partiendo de este punto, se selecciona el modelo de dispositivo Hardware de aceleración, una tarjeta Alveo, en concreto el modelo U50, disponible en el laboratorio del grupo de investigación.

El siguiente punto a tener en cuenta, es el proyecto práctico a desarrollar para poner a prueba las tecnologías implicadas, estudiando en mayor detalle, los conceptos teóricos y prácticos relacionados con la aceleración de aplicaciones Software, mediante el uso y la configuración de Hardware de aceleración.

Para este análisis, se plantean dos ideas: desarrollar un proyecto completo desde cero, analizando al completo la metodología seguida, o importar un proyecto proporcionado por Xilinx, siguiendo un proceso guiado de aceleración de la aplicación, probado y estudiado, para analizar la metodología en paralelo a dicho proceso. Por alcance y extensión del proyecto, se opta por la segunda opción, acudiendo al repertorio de proyectos y tutoriales de Xilinx, disponible en GitHub. (Xilinx, Vitis In-Depth Tutorials (Repositorio GitHub), 2021)

En dicho repertorio, existen diferentes categorías de aplicaciones y es necesaria una nueva selección, para adecuar el proyecto al alcance y objetivos del trabajo expuesto en esta memoria. Para ello, se realizan dos análisis. El primero, para la selección de la naturaleza de las aplicaciones relacionadas con los proyectos de prueba, y el segundo, para la selección de la aplicación particular implementada, dentro del grupo seleccionado en el primer análisis.

Para realizar una ponderación adecuada, se exponen los puntos de interés, junto con las alternativas del primer análisis, en la Tabla 2, y del segundo análisis, en la Tabla 3. En ellas se muestran los puntos cubiertos mediante un punto verde, los cubiertos parcialmente con un punto naranja y aquellos no tratados, mediante una cruz roja.

Temática Tutoriales	Metodología de aceleración HW	Entorno de desarrollo Vitis	Plataforma y tarjetas Alveo	Sencillez, ajuste a alcance y objetivos	Requerimientos ajustados, sin herramientas adicionales
Introducción: Vitis	●	●	✖	●	●
Machine Learning	✖	✖	●	✖	✖
Aceleración	●	●	●	●	●
Motor IA	●	●	✖	✖	✖
Sistemas Embebidos	●	●	✖	●	✖
Creación de plataformas	✖	●	●	✖	✖
XRT y optimización de sistemas	●	●	✖	✖	✖

Tabla 2: Comparativa de puntos de interés tratados por diferentes tutoriales de Xilinx, según temática

Como se observa, el punto que cumple con todos los requisitos es el correspondiente a “Aceleración”, de forma que los mejores candidatos son sus diferentes implementaciones.

Implementación Tutorial	Visión global de la tecnología	Proceso completo de desarrollo	Comparable con casos tradicionales CPU	Sencillez, concepto introductorio	Implementación sobre Alveo U50
Introducción: Vitis Hardware Accelerators (Cholesky algorithm)	🟡	🟡	🟢	🟢	🟢
Filtro Bloom	🟢	🟢	✖	🟢	🟢
Filtro de Convolución de Vídeo	🟢	🟡	✖	✖	✖
Integración de sistemas RTL	🟡	🟡	✖	✖	🟢
Iniciación a Kernels RTL	✖	✖	✖	✖	✖
Mezclando C y RTL	🟡	✖	✖	✖	✖
Flujo de depuración y optimización	🟡	✖	✖	✖	✖

Tabla 3: Comparativa de puntos de interés tratados por diferentes tutoriales de Xilinx, para la temática de "Aceleración"

Finalmente, se opta por el proyecto de introducción de aceleración, denominado “Acceleration Tutorial for Alveo U50”, o “Introducción: Vitis Hardware Accelerators” en la tabla, que permite trabajar todos los aspectos deseados, cumpliendo con los objetivos y alcances del proyecto. Como los dos primeros puntos se tratan de manera genérica, se opta por realizar primero el “Filtro Bloom”, de forma que se desarrolla conjuntamente con el desarrollo de la metodología completa. Teniendo en cuenta que la implementación seleccionada se basa en el algoritmo de Cholesky, se seleccionan para su comparación de desarrollo, ejecución y rendimiento, sus implementaciones homólogas en los lenguajes de programación, de uso general, C, C++, Python, Java y Octave.

7. Descripción de tareas, fases y procedimientos.

EDT	Subtarea	Duración (dias)	Duración (horas)	Predecesoras
PT.1	**Investigacion Previa y Documentación**	**26**	**208**	**-**
T.1.1	Analizar herramientas y entornos Xilinx: Búsqueda de informacion general sobre herramientas Xilinx Vitis, metodología, documentación, implementaciones...	7	56	-
T.1.2	Selección de herramientas y entorno a utilizar	2	16	T.1.1
T.1.3	Investigar sobre despliegue de entorno de desarrollo: Herramientas específicas, modos de instalación, procedimientos, validaciones...	2	16	T.1.2
T.1.4	Búsqueda de proyectos guiados sobre aceleración de aplicaciones: Búsqueda de proyectos para desarrollo de metodología y comparación de resultados	2	16	T.1.3
T.1.5	Estudio completo del estado del arte: estudio detallado de conceptos relacionados con proyecto	3	24	T.1.4
T.1.6	Lectura y síntesis detallada de documentación oficial de Xilinx para desarrollo, metodología e implementación de procesos de aceleración de aplicaciones	9	72	T.1.5
T.1.7	Selección de proyecto guiado a desarrollar: con conocimientos base y mayor conocimiento del alcance, seleccion de las alternativas adecuadas	1	8	T.1.6
H.1	**Asimiliación de conceptos teóricos y primeros pasos**	**0**	**0**	**T.1.7**
PT.2	**Despliegue de entorno**	**7**	**56**	-
T.2.1	Deteccion de dependencias y herramientas necesarias	2	16	H.1
T.2.2	Pruebas de Instalacion/desinstalacion completa de herramientas	3	24	T.2.1
T.2.3	Instalación y despliegue del entorno de desarrollo Xilinx Vitis	2	16	T.2.2
H.2	**Entorno de desarrollo desplegado y operativo**	**0**	**0**	**T.2.3**
PT.3	**Desarrollo solucion**	**26**	**208**	-
T.3.1	Desarrollo proyecto guiado I (metodología + filtro Bloom)	9	72	H.2
T.3.2	Desarrollo proyecto guiado II (aceleración+comparación + Cholesky)	9	72	T.3.1
T.3.3	Implementaciones lenguajes C/C++, Python, Java y Octave de Cholesky	4	32	T.3.2
T.3.4	Análisis de resultados y extracción de conclusiones	4	32	T.3.3
H.3	**Solución propuesta completa y obtención de resultados**	**0**	**0**	**T.3.4**
PT.4	**Redacción Memoria**	**25**	**200**	-
T.4.1	Redactar anexos	5	40	H.3
T.4.2	Redactar apartados generales memoria	9	72	T.4.1
T.4.3	Redactar apartados específicos solución memoria	9	72	T.4.2
T.4.4	Redactar resultados y conclusiones	2	16	T.4.3
H.4	**Memoria completa**	**0**	**0**	**T.4.4**
H.5	**Proyecto completado**	**0**	**0**	**H.4**

Tabla 4: Tareas, fases y procedimientos de la planificación del proyecto

Se obtiene una duración total del proyecto de 84 días, con fecha de inicio el 12 de abril de 2021 y finalización el 6 de agosto de 2021. El trabajo es realizado por una única persona, con una jornada de 8 horas diarias, suponiendo un total de 672 horas.

8. Diagrama de Gantt/Cronograma.

EDT	Nombre	Duración	Fecha de ini...	Fecha de ...
PT.0	Proyecto Fin de Máster	84	12/4/21	5/8/21
PT.1	Investigacion Previa y Documentación	26	12/4/21	17/5/21
T.1.1	Analizar herramientas y entornos Xilinx: Búsqueda de informacion gen...	7	12/4/21	20/4/21
T.1.2	Selección de herramientas y entorno a utilizar	2	21/4/21	22/4/21
T.1.3	Investigar sobre despliegue de entorno de desarrollo: Herramientas esp...	2	23/4/21	26/4/21
T.1.4	Búsqueda de proyectos guiados sobre aceleración de aplicaciones: Bús...	2	27/4/21	28/4/21
T.1.5	Estudio completo del estado del arte: estudio detallado de conceptos r...	3	29/4/21	3/5/21
T.1.6	Lectura y síntesis detallada de documentación oficial de Xilinx para des...	9	4/5/21	14/5/21
T.1.7	Selección de proyecto guiado a desarrollar: con conocimientos base y ...	1	17/5/21	17/5/21
H.1	Asimiliación de conceptos teóricos y primeros pasos	0	18/5/21	18/5/21
PT.2	Despliegue de entorno	7	18/5/21	26/5/21
T.2.1	Deteccion de dependencias y herramientas necesarias	2	18/5/21	19/5/21
T.2.2	Pruebas de Instalacion/desinstalacion completa de herramientas	3	20/5/21	24/5/21
T.2.3	Instalación y despliegue del entorno de desarrollo Xilinx Vitis	2	25/5/21	26/5/21
H.2	Entorno de desarrollo desplegado y operativo	0	27/5/21	27/5/21
PT.3	Desarrollo solución	26	27/5/21	1/7/21
T.3.1	Desarrollo proyecto guiado I (metodología + filtro Bloom)	9	27/5/21	8/6/21
T.3.2	Desarrollo proyecto guiado II (aceleración+comparación + Cholesky)	9	9/6/21	21/6/21
T.3.3	Implementaciones lenguajes C/C++, Python, Java y Octave de Cholesky	4	22/6/21	25/6/21
T.3.4	Análisis de resultados y extracción de conclusiones	4	28/6/21	1/7/21
H.3	Solución propuesta completa y obtención de resultados	0	2/7/21	2/7/21
PT.4	Redacción Memoria	25	2/7/21	5/8/21
T.4.1	Redactar anexos	5	2/7/21	8/7/21
T.4.2	Redactar apartados generales memoria	9	9/7/21	21/7/21
T.4.3	Redactar apartados específicos solución memoria	9	22/7/21	3/8/21
T.4.4	Redactar resultados y conclusiones	2	4/8/21	5/8/21
H.4	Memoria completa	0	6/8/21	6/8/21
H6	Proyecto completado	0	6/8/21	6/8/21

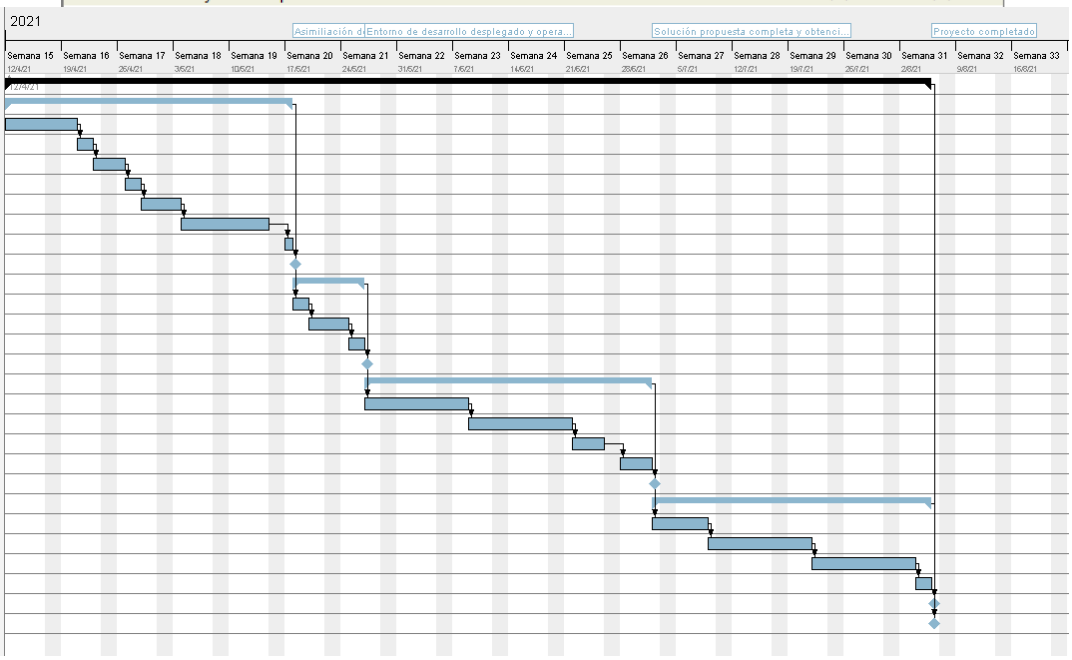


Figura 14: Tareas y diagrama de Gantt de la planificación del proyecto

9. Análisis de costes.

Debido a la naturaleza sin ánimo de lucro de la investigación, con un objetivo puramente educativo y de estudio tecnológico, este análisis no es fundamental, pero, si se desea extrapolar a una situación particular de investigación, en una cierta empresa española, es interesante analizar el impacto económico del proyecto. Para el análisis se establecen las siguientes premisas, según la partida correspondiente:

- Horas Internas: La labor es realizada por un único trabajador, suponiéndose graduado en Ingeniería Técnica de Telecomunicaciones, con un salario horario estimado de 11€/h (valor estimado para recién graduado de 22000€/año, repartidos entre los 250 días laborables de 2021, suponiendo jornadas de 8 horas).
- Gastos: solamente se tiene en cuenta el material de oficina consumido (fotocopias, bolígrafos, papel…)
- Amortizaciones: se calculan a partir de los costes de adquisición y la vida útil en horas, para obtener el coste total, según el uso en horas dado a cada recurso.
    - Licencia de Microsoft Office: su adquisición supone 126.24€ y presenta un periodo de validez de 1 año, o lo que es lo mismo, de 8760 horas.
    - Ordenador y componentes: se adquieren los componentes por separado y se monta el equipo, suponiendo un coste total de 1500€. Se supone una vida útil de unos 4 años (a máximo uso), considerando su uso las 24 horas del día, es decir, una vida útil de 35040 horas.
    - Equipamiento de refrigeración: se adquiere por un coste total de 60€. Presenta la misma naturaleza que los componentes del ordenador, por lo que se supone una vida útil de 35040 horas.
    - Tarjeta de aceleración Alveo U50: se adquiere por un coste de 2100€ y se supone como vida útil el periodo de garantía del fabricante, 3 años, suponiendo un total de 26280 horas.

Las diferentes partidas y el coste total se pueden observar en las siguientes tablas:

Horas internas				
Concepto	Nº de recursos (uds.)	Coste horario (€)	Nº de horas	Coste total (€)
Ingeniero Técnico en Telecomunicaciones	1	11€	672	7.452€
			SUBTOTAL	7.452€

Tabla 5: Presupuesto. Horas internas

Gastos			
Concepto	Nº de recursos (uds.)	Coste unitario (€)	Coste total (€)
Material de oficina	-	-	75€
		SUBTOTAL	75€

Tabla 6: Presupuesto. Gastos

Amortizaciones				
Concepto	Coste adquisición (€)	Vida útil (h)	Uso (h)	Coste total (€)
Licencia Microsoft Office	126,24	8760	200	2,88
Ordenador	1500	35040	672	28,77
Equipamiento refrigeración	60	35040	264	0,45
Alveo U50	2100	26280	264	21,10
			SUBTOTAL	53,20

Tabla 7: Presupuesto. Amortizaciones

Coste Total	
Horas internas	7.452€
Amortizaciones	53,20
Gastos	75€
Total	**7.580€**

Tabla 8: Presupuesto. Costes totales

10. Selección, descripción y diseño de la solución propuesta.

Una vez definida la alternativa adecuada, junto con las tareas y procesos necesarios para el cumplimiento de los objetivos, se procede a desarrollar la solución propuesta.

En ella se trata, en primer lugar, de estudiar e investigar sobre el propio entorno de desarrollo de la plataforma unificada Xilinx Vitis. Para ello, es necesario acudir a la documentación oficial, para establecer el punto de partida, identificando las herramientas Software necesarias para el correcto funcionamiento del sistema. Una vez identificadas, se realiza una primera instalación, siguiendo los pasos proporcionados para intentar desplegar el sistema completo de desarrollo en una máquina virtual. Debido a la escasa y escueta documentación referente a este tema, es necesario realizar un proceso iterativo de adaptación de la instalación para, finalmente, lograr instalar todas las herramientas necesarias. Para realizar una verificación de las mismas, se importa y ejecuta un proyecto de ejemplo, accesible desde el propio IDE de Xilinx.

Tras la verificación de la instalación, ya es posible comenzar con el desarrollo de aplicaciones aceleradas. En este punto, se divide el flujo de trabajo en dos fases.

La primera fase consiste en estudiar la metodología propuesta por Xilinx para el desarrollo de aplicaciones aceleradas sobre Hardware de aceleración, mediante la plataforma Xilinx Vitis. Para una mejor comprensión de los conceptos teóricos, se realiza en paralelo junto a este estudio, un proceso guiado de aceleración de aplicaciones. Este proyecto o tutorial guiado implementa un filtro de Bloom para la detección del grado de coincidencia entre contenido de documentos y patrones de búsqueda. Esto permite asentar en mayor medida las bases teóricas, implementándolas sobre procesos prácticos.

En cuanto a la segunda fase, se trata de un proceso similar al anterior, aunque a un mayor nivel. Se realiza una aceleración de un algoritmo matemático conocido como “Descomposición de Matrices de Cholesky”, que permite descomponer ciertas matrices en otras más simples, según se explica en el apartado correspondiente. El objetivo de esta fase reside en la comprensión del proceso, obteniendo un resultado cuantificable de la aceleración de aplicaciones, para su posterior comparación con los resultados obtenidos mediante implementaciones con lenguajes de programación de uso general, ejecutados puramente sobre CPU.

Esto permite comprender y analizar el entorno completo de aceleración de aplicaciones, junto con un estudio de la metodología y de los procesos necesarios, para obtener una serie de conclusiones relativas a esta tecnología, identificando sus ventajas y desventajas.

10.1. Instalación y verificación de Software y herramientas Xilinx Vitis.

Los componentes Xilinx a instalar, se pueden agrupar en 4 grandes elementos: entorno unificado de desarrollo Vitis, librería XRT y plataformas de destino de despliegue y de desarrollo. Según el esquema general de la plataforma unificada Vitis, mostrada en la Figura 15, son claramente identificables, correspondiéndose en sentido vertical al nivel de abstracción correspondiente (cuanto más arriba, mayor nivel de abstracción sobre el Hardware). Existen, además, una serie de librerías de alto nivel que facilitan el desarrollo en ciertos ámbitos, siendo opcional su instalación, pero formando parte del entorno de desarrollo Vitis.

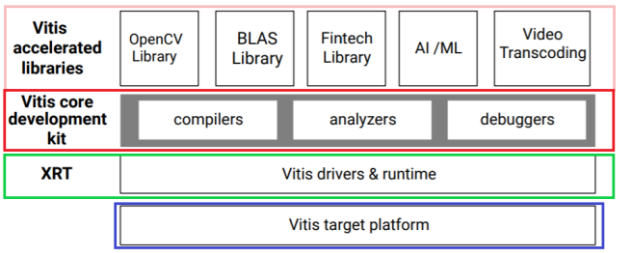


Figura 15: Componentes Xilinx a instalar ordenador por nivel de abstracción (Xilinx, Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393), 2021)

Siguiendo la documentación oficial y ampliándola en aquellos puntos donde es necesario, se pueden identificar los siguientes pasos, para conseguir desplegar el entorno de desarrollo completo, estableciendo el punto de partida para el resto de tareas detalladas en esta memoria:

1. Instalación y configuración del PC y de un Sistema Operativo compatible.

   En cuanto a este punto, se instala un SO Ubuntu 20.04 LTS (última versión compatible), sobre una máquina virtual con las máximas especificaciones posibles. El equipo host tiene un procesador AMD Ryzen 7 3700X con 8 núcleos a 3.59 GHz, de los cuales se utilizan 8 hilos de los 16 disponibles para el entorno virtualizado. En cuanto a la memoria RAM instalada, se cuenta con 32 GB, siendo el mínimo recomendado, aunque estando disponibles en el entorno virtualizado unos 25 GB. Esto

supone un menor rendimiento durante el proceso de desarrollo, aunque siendo suficiente para la implementación. Es recomendable la instalación en un SO nativo, con un mínimo de 64 GB de RAM.

2. Descarga y verificación de software Xilinx, Vitis Unified Software Platform.

   Para ello, simplemente acudir a la página web del fabricante, en la que es necesario descargar la versión adecuada del software Vitis, de la librería XRT y de las plataformas de destino de la tarjeta Alveo utilizadas. Además, es interesante descargar los ficheros para verificación de archivos, como son clave pública, la firma y los códigos, para garantizar la autenticidad e integridad de los instaladores y herramientas obtenidas. (Xilinx, Downloads: Vitis Core Development Kit, 2021)

   Para lo referente a esta memoria, se descargan los elementos generales y aquellos relacionados con la tarjeta Alveo U50:

   - Instalador unificado Vitis (Xilinx_Unified_2020.2_1118_1232_Lin64.bin)
   - XRT (xrt_202020.2.8.743_20.04-amd64-xrt.deb).
   - Plataforma de despliegue destino para Alveo U50 (xilinx-u50-gen3x16-xdma-201920_3-3_all.deb.tar.gz)
   - Plataforma de destino para desarrollo en Alveo U50 (xilinx-u50-gen3x16-xdma-dev-201920.3-2784799_20.04.deb).

3. Instalación de librerías adicionales requeridas.

   Completando la documentación y requiriendo varias iteraciones de instalación, se obtiene la siguiente lista de software necesario, fácilmente instalable mediante el gestor de paquetes de Ubuntu, con un simple comando “sudo apt-get install <paquete>”, siendo “<paquete>” las diferentes herramientas a instalar. Estos paquetes son:

   - NCurses: “libtinfo5 libncurses5 libncursesw5”
   - Git: “git”
   - Curl: “curl libcurl4-openssl-dev”
   - OpenCL: “ocl-icd-libopencl1 opencl-headers ocl-icd-opencl-dev”

4. Instalación de software Xilinx, Vitis Unified Software Platform.

   Ejecutando el instalador unificado y siguiendo el asistente, es posible instalar de manera sencilla todos los componentes referentes al IDE y sus herramientas. En caso de necesitar el Software Vivado para modificación de plataformas de destino, es necesario adquirir previamente una licencia (no tratado en esta memoria).

   En caso de ocurrir algún error, el propio instalador almacena información en unos archivos log, ubicados en "<directorio_instalacion>/.xinstall/xinstall_<cadena>.log". Acudiendo a estos, es posible identificar el error y detectar si se debe a falta de alguna herramienta para su instalación y posterior reanudación del instalador Vitis.

5. Instalación de librerías mediante script Xilinx.

   Xilinx proporciona unos scripts automatizados para la instalación de las herramientas adicionales tras la instalación del entorno Vitis, basadas en el SO y el Hardware utilizados. Estos están ubicados en "<directorio_instalacion>/Vitis/<version>/scripts". El script a ejecutar se denomina "installLibs.sh".

6. Inicialización y arranque de Vitis.

   Acudiendo directamente al dock de aplicaciones y ejecutando el programa o acudiendo a un terminal y ejecutando el comando "vitis", se lanza el entorno gráfico del programa de desarrollo. Si se ejecuta sin problemas, la primera fase de instalación es satisfactoria y es posible proceder. En caso negativo, revisar errores y reinstalar todo aquello que sea necesario.

7. Instalación de librería Xilinx Runtime (XRT) y plataformas destino.

   Instalación de los paquetes descargados mediante simples comandos "sudo apt-get install <paquete_deb>". En cuanto a la plataforma de destino, descomprimir los ficheros en un directorio seguro, para su posterior exportación al entorno de desarrollo, que denominaremos "ruta_plataformas".

8. Configuración del entorno de trabajo, terminal y variables de Vitis.

   Este proceso es necesario cada vez que se inicie el equipo y/o un terminal de trabajo. Es necesario ejecutar una secuencia de comandos en el terminal a utilizar:

   - Configuración de variables para Xilinx Vitis y Vidado:

"source <directorio_instalacion>/Vitis/<versión>/settings64.sh"

- Configuración de librería XRT para plataformas Alveo:

  "source /opt/xilinx/xrt/setup.sh"

- Definición de directorio de plataformas de destino:

  "export PLATFORM_REPO_PATHS=<ruta_plataformas>"

9. Selección de plataforma en Vitis.

   Al crear o modificar un proyecto de aplicaciones aceleradas en el IDE de Vitis, es posible seleccionar la plataforma mediante una serie de ventanas y exploradores, localizando simplemente la carpeta en que se descomprime la plataforma de destino descargada. En caso de haber instalado correctamente la plataforma, debe visualizarse sin necesidad de ningún proceso adicional.

10. Comprobación de instalación mediante importación de proyecto ejemplo.

    Con todas las herramientas instaladas, un paso para la verificación de que todo se encuentra correctamente configurado, consiste en importar un proyecto preconfigurado por Xilinx, cuyo funcionamiento está comprobado. Para ello, simplemente es necesario crear un nuevo proyecto de aplicación acelerada, seleccionar y descartar los ejemplos del IDE Vitis y seleccionar el deseado. Para la prueba de esta memoria, se seleccionan los proyectos "Burst Read/Write (OpenCL Kenel)" y "Hello World (HLS C/C++ Kernel)", para comprobar todas las herramientas implicadas.

    Mediante la compilación y ejecución, tanto en emulación Software como Hardware, se verifica que el proceso de instalación es correcto y que se puede comenzar con un desarrollo personalizado.

En el Anexo B se indica con mayor detalle y paso por paso, con imágenes del proceso, todas las indicaciones necesarias para instalar correctamente el Software necesario. Además, incluye información sobre actualización y desinstalación de los componentes y programas, ausentes en la documentación oficial.

10.2. Metodología de aceleración de aplicaciones

A lo largo de la documentación, Xilinx proporciona una serie de aspectos técnicos y procedimientos para la aceleración de aplicaciones mediante su entorno de desarrollo Vitis. Dicha documentación es extensa y los conceptos se encuentran presentados y detallados en diferentes documentos y para soluciones diferentes. En este apartado, se pretende realizar una síntesis de esta información, para presentar un esquema general de introducción que permita comprender, a grandes rasgos, el funcionamiento general de la aceleración de aplicaciones. De esta forma, se establece una base para que, mediante la realización de un ejercicio práctico, realizando una aceleración sobre un proyecto con una implementación concreta, puedan materializarse dichos conceptos y demostrar su aplicabilidad y sus posibilidades y potencial.

Un esquema general de dicha metodología es el mostrado en la Figura 16.

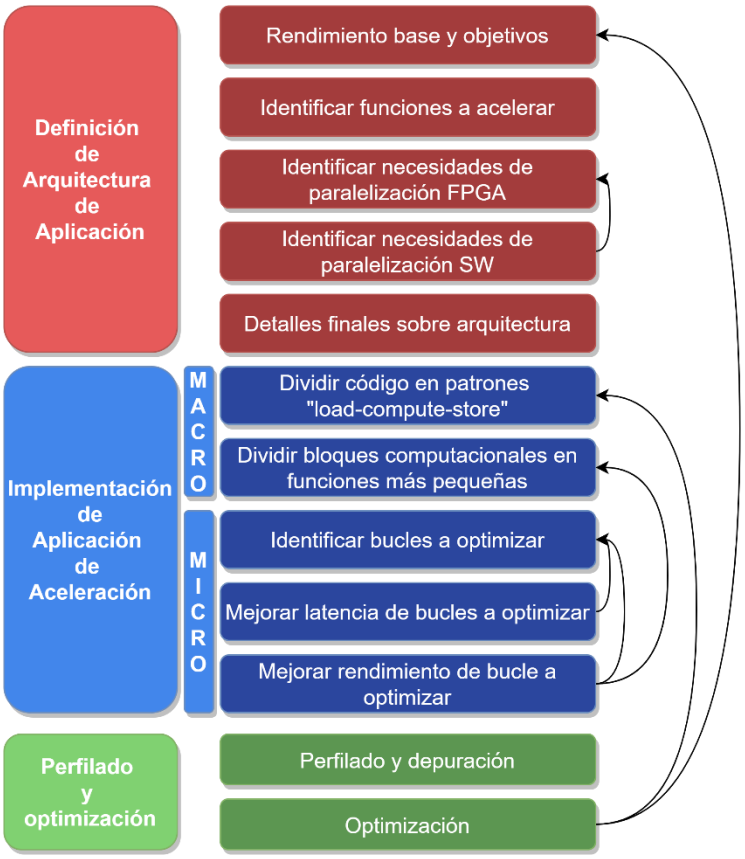


Figura 16:Esquema metodología de aceleración de aplicaciones mediante Xilinx Vitis (Solución)

Según dicho esquema, se puede agrupar el proceso completo de aceleración en tres fases generales, cada una de ellas compuesta por una serie de procesos internos, que deben realizarse de manera secuencial, existiendo la posibilidad de retroceder en el flujo para lograr una mejor optimización de la aplicación:

1. Definición de arquitectura de la aplicación: Decisiones clave sobre arquitectura, definiendo distribución funcional entre host y kernel, e indicando grados de paralelización y forma de llevarlo a cabo, a grandes rasgos. Los aspectos a definir en esta etapa son los siguientes:

    a. Rendimiento base y objetivos:

        Es necesario realizar una estimación del tiempo de ejecución total y parcial de la aplicación completa y de los bloques funcionales que la componen, analizando la capacidad o throughput de procesamiento de datos, basados en el volumen de datos procesados por unidad de tiempo. Además, es necesario definir la máxima capacidad o throughput alcanzable, teniendo en cuenta las funciones, los algoritmos y las tecnologías utilizadas. Por norma general, la comunicación PCIe supone el límite superior de aceleración.

        Conocidas estas primeras estimaciones, se definen los objetivos generales, estableciendo de manera realista un objetivo de aceleración de rendimiento.

    b. Identificación de funciones a acelerar:

        En primer lugar, se identifican los cuellos de botella de rendimiento, para detectar puntos de interés de cara a la aceleración. Para localizar bloques críticos, es necesario hacer un análisis particular, pero existen unos factores que condicionan a los bloques funcionales a ser buenos candidatos para ello:

            1. Elevada complejidad computacional (gran número de operaciones básicas requeridas).
            2. Gran intensidad computacional (gran número de operaciones en relación al volumen de datos de entrada y salida).

3. Baja localización espacial (distancia entre accesos a memoria consecutivos) y temporal (número de operaciones para acceder a direcciones de memoria durante la ejecución)
4. Gran impacto de ejecución de la funcionalidad particular, frente al rendimiento global

c. Identificación de necesidades de paralelización en FPGA:

Primero, se requiere estimar el rendimiento Hardware sin paralelización, simplemente portando la aplicación a la FPGA. A continuación, se debe determinar el grado de paralelización necesario para alcanzar el objetivo, partiendo de la primera aproximación. Tras el análisis, deben definirse tanto el número de muestras que deben ser procesadas en paralelo, como el número de unidades de cómputo o kernels instanciados en paralelo. En la Figura 17 se muestra un ejemplo en el que se definen tres instancias de kernels en paralelo, cada una de las cuales pudiendo procesar varias muestras por ciclo.

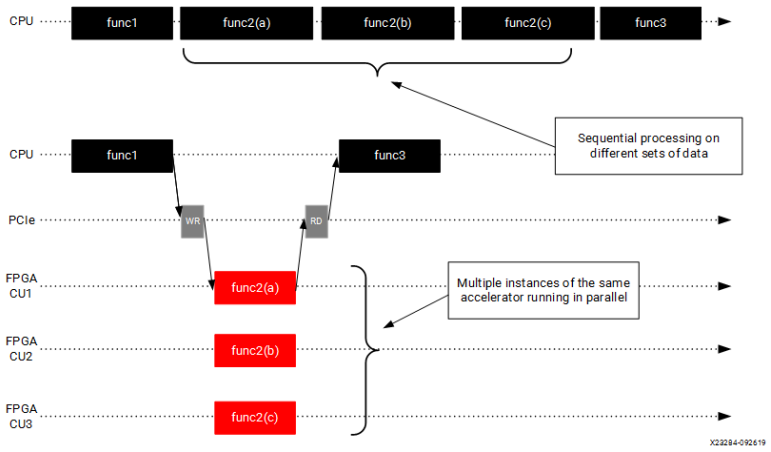


Figura 17: Ejemplo de aceleración mediante múltiples kernels en paralelo (Solución) (Xilinx, Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393), 2021)

d. Identificación de necesidades de paralelización en Software:

Mientras los kernels se diseñan para explotar el potencial de la paralelización, el programa host debe desarrollarse para hacerlo de la misma manera. Para ello, se diseña en torno a la minimización de los tiempos de espera en CPU,

la maximización del uso del acelerador FPGA y la optimización de las transferencias de datos entre CPU y FPGA.

e. Finalización de detalles sobre arquitectura:

En caso de definir varias instancias de kernels, es necesario tener en cuenta el número máximo de puertos disponibles en la plataforma utilizada. Por tanto, es necesario definir la frontera del kernel, indicando si se desean varias instancias de kernels, o si es más eficiente desarrollar motores internos para la paralelización de las tareas.

Una vez decidida la frontera del kernel, es necesario definir la localización y conectividad de los mismos en el acelerador. Es necesario definir las características de I/O del kernel para adecuar a un flujo o a otro.

2. Implementación de la aplicación acelerada: Desarrollo e implementación de la aplicación en sí, estableciendo como base los resultados del proceso anterior. Para un correcto desarrollo, se divide en dos fases:

a. Macroestructura:

Funcionalidades de más alto nivel siguiendo el patrón “load-compute-store” (carga-cómputo-guardado). Se establece un bloque de alto nivel que define la interfaz global del kernel. Mediante las funciones “load” y “store” se realizan los intercambios de información hacia fuera del kernel, mientras que la función “compute” realiza el procesamiento interno de los datos. Es importante adaptar todas las funciones que se comuniquen mediante interfaces, a las características de cada tipo de comunicación. Se deben conectar las funciones según el estilo de flujo de datos o dataflow, que define una serie de pautas: solo datos hacia adelante, con transacciones de productor único para consumidor único y solamente permitiendo acceder a la interfaz primaria del kernal a las funciones “load” y “store”. Un esquema de este patrón de diseño se observa en la Figura 18.

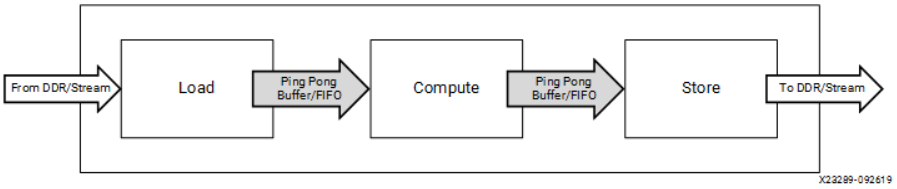


Figura 18:Esquema de bloques siguiendo patrón "load-compute-store" (Solución)

Tras definir los bloques generales, se realiza la división boques computacionales más pequeños, identificando objetivos particulares de rendimiento, definiendo funciones con bucles de anidación única, e introduciendo los bucles secuenciales en funciones secuenciales para facilitar al compilador la optimización. Las funciones de más bajo nivel deben tener un bucle único perfectamente anidado. Las funciones internas también se conectan según dataflow.

b. Microestructura:

Funcionalidades particulares de bajo nivel que componen la macroestructura. Para su desarrollo, es necesario identificar los bucles a optimizar, definiendo tres valores fundamentales:

- "Steps": duración de una iteración simple del bucle, medida en número de ciclos de reloj.
- "TripCount": número de iteraciones en el bucle
- "II" o "Initiation Interval": número de ciclos de reloj entre el inicio de dos iteraciones consecutivas. Cuando un bucle no usa pipeline, "II" es igual al valor de "Steps".

Una vez identificados los bucles a optimizar y habiendo definido sus parámetros, se puede proceder a su optimización. Según el parámetro a optimizar, se pueden aplicar diversas técnicas:

- Mejora de latencia de bucle a optimizar:
  - "Unroll": Técnica que permite "desenrollar" un bucle para ejecutar de manera simultánea varias iteraciones del mismo disminuyendo el "TripCount". Esto permite procesar varias

muestras por iteración, aunque aumentando la cantidad de datos requeridos para la salida del bloque funcional. al aumentar el ancho de la ruta de datos. Modifica requerimientos I/O.

- Partición de arrays: Mediante directivas de compilador, se logra definir cómo se mapean los arrays (secuencias o cadenas de elementos del mismo tipo, como vectores o matrices) a lo largo de los recursos de memoria, adaptando esta tarea a los requerimientos.

- Mejora de rendimiento de bucle a optimizar: Se centra en la optimización del "II", buscando una mejora de rendimiento mediante la eliminación de contenciones I/O por conflictos en acceso a memorias, o mediante la eliminación de dependencias entre datos de diferentes iteraciones de bucles. También existen otras técnicas avanzadas basadas en pragmas de compilación a las que se puede recurrir en caso necesario.

3. Perfilado y optimización:

   a. Perfilado y depuración: Consiste en la elaboración de informes, antes, durante y después del proceso, para diseñar, validar y establecer futuras optimizaciones, buscando el máximo potencial de la aplicación acelerada. Para ello existen numerosas herramientas de perfilado, además del IDE proporcionado por Xilinx, para depurar la ejecución de los kernels tanto en emulación como en ejecución. Es importante establecer un proceso secuencial en el que se comience con la emulación Software, para validar la funcionalidad, se siga con una emulación Hardware, para validar la implementación teórica sobre el Hardware de aceleración y, finalmente, con una ejecución sobre el propio dispositivo Alveo para determinar el funcionamiento real.

   b. Optimización: Proceso iterativo, de diseño e implementación, basado en todo el flujo, para tomar decisiones y elaborar procedimientos para obtener el máximo potencial de las aplicaciones aceleradas. Implica volver a diferentes puntos de la metodología para, iterativamente, introducir mejoras, hasta alcanzar el objetivo deseado.

10.2. Proyecto guiado “Filtro Bloom”

El código relacionado con este apartado se indica en el Anexo D.1 en la página 199, y se adjunta en el archivo comprimido “TFM_source_files”.

Habiendo establecido la base teórica de la metodología, es posible comenzar a acelerar una aplicación particular. Para profundizar en los conceptos y fundamentos de este proceso, en este apartado se explica el desarrollo de un proceso completo de aceleración, partiendo de una implementación puramente en CPU, para pasar al paradigma de Vitis, con interacción CPU y FPGA. Durante el proceso, se realizan una serie de modificaciones al código fuente, analizando el porqué de las mismas y determinando el impacto sobre el rendimiento parcial de las funcionalidades y global de la aplicación.

10.2.1. Preparación y experimentación de la aceleración.

La aplicación se basa en la búsqueda a través de un flujo de entrada de documentos, para encontrar aquellos que mejor encajan con el interés de un usuario, basado en un perfil de búsqueda. El algoritmo implementado es el filtro Bloom, usado en ámbitos como el análisis de datos, o como búsquedas en emails no estructurados y ficheros de datos de texto.

El filtrado de documentos consiste en que un sistema monitoriza un flujo entrante de documentos, los clasifica en función de su contenido, y selecciona los documentos relevantes para un usuario o tema específicos. Se trata de un proceso extensamente utilizado actualmente, en escenarios en los que, por norma general, presentan un volumen de documentos muy elevado. Además, están relacionados con aplicaciones que deben ejecutarse en tiempo real, requiriendo tiempos de respuesta muy cortos.

En este tutorial, se genera una puntuación para cada documento, indicando la relevancia del mismo. Esta puntuación se calcula en base al interés de un usuario, representado mediante un array de palabras, cada una con una ponderación de relevancia. El filtro Bloom utiliza estructuras de datos basadas en tablas de hashes para determinar qué elementos están presentes en el conjunto de datos.

En esta implementación, cada documento consiste en un array de palabras, cada una representada por un entero sin signo de 32 bits, compuesto por un ID de palabra de 24 bits y un entero de 8 bits (indicando frecuencia de aparición). El array de búsqueda se compone de las palabras de interés, que representa un conjunto más pequeño de ID de palabras de 24

bits, cada una con un peso, que indica su importancia. La puntuación se calcula como el producto acumulativo del peso de la palabra referente al ID, por la frecuencia de aparición.

El tutorial proporcionado está preparado para la tarjeta Alveo U200, por lo que es necesario portar el proyecto a la plataforma de destino utilizada en la Alveo U50 disponible. Esto permite trabajar en mayor detalle el funcionamiento de los tutoriales y conocer cómo funcionan globalmente los proyectos. Debido a que la generación de los binarios ejecutables finalmente sobre una FPGA requieren de horas de compilación (6-7 horas por proceso según el fabricante, para este proyecto), para este proceso se centra el estudio en los modos de operación de emulación, tanto SW como HW.

En primer lugar, es necesario descargar el repositorio de tutoriales Vitis desde GitHub. Mediante un comando directamente desde el terminal, es posible descargar el contenido para posteriormente, acceder al proyecto concreto “/Hardware_Accelerators/Design_Tutorials/02-Bloom” y visualizar el árbol de ficheros.

```
user@ubuntu-pc:~/Xilinx/Development$ git clone http://github.com/Xilinx/Vitis-Tutorials
/home/user/Xilinx/Development/Vitis-Tutorials-2021.1
Clonando en '/home/user/Xilinx/Development/Vitis-Tutorials-2021.1'...
warning: redirigiendo a https://github.com/Xilinx/Vitis-Tutorials/
remote: Enumerating objects: 22131, done.
remote: Counting objects: 100% (547/547), done.
remote: Compressing objects: 100% (469/469), done.
remote: Total 22131 (delta 88), reused 365 (delta 53), pack-reused 21584
Recibiendo objetos: 100% (22131/22131), 1.61 GiB | 9.31 MiB/s, listo.
Resolviendo deltas: 100% (11083/11083), listo.
user@ubuntu-pc:~/Xilinx/Development$ cd /home/user/Xilinx/Development/Vitis-Tutorials-20
21.1/Hardware_Accelerators/Design_Tutorials/02-bloom
```

```
user@ubuntu-pc:~/Xilinx/Development/Vitis-Tutorials-2021.1/Hardware_Accelerators/Design_
Tutorials/02-bloom$ ls -al
total 128
drwxrwxr-x 6 user user  4096 ago 10 20:50 .
drwxrwxr-x 6 user user  4096 ago 10 20:50 ..
-rw-rw-r-- 1 user user  4344 ago 10 20:50 1_overview.md
-rw-rw-r-- 1 user user  3740 ago 10 20:50 2_experience-acceleration.md
-rw-rw-r-- 1 user user 14431 ago 10 20:50 3_architect-the-application.md
-rw-rw-r-- 1 user user 25205 ago 10 20:50 4_implement-kernel.md
-rw-rw-r-- 1 user user 25900 ago 10 20:50 5_data-movement.md
-rw-rw-r-- 1 user user  9998 ago 10 20:50 6_using-multiple-ddr.md
drwxrwxr-x 2 user user  4096 ago 10 20:50 cpu_src
drwxrwxr-x 2 user user  4096 ago 10 20:50 images
drwxrwxr-x 2 user user  4096 ago 10 20:50 makefile
-rw-rw-r-- 1 user user  6708 ago 10 20:50 README.md
drwxrwxr-x 2 user user  4096 ago 10 20:50 reference_files
```

Figura 19: Descarga y listado de archivos de proyecto Xilinx: Bloom Filter

En este directorio aparecen cuatro carpetas de interés:

- cpu_src: contiene todo el código fuente original, antes de la modificación
- images: contienen las figuras del tutorial
- makefile: el fichero Makefile, define las reglas y comandos a ejecutar para configurar y construir los ficheros necesarios, tanto para host como para la FPGA. Mediante “PLATFORM”, se puede modificar la plataforma destino, permitiendo portar el proyecto. Mediante las variables “STEP”, “PF” e “ITER”, se compilan los diferentes proyectos referentes al tutorial.
- reference_files: contiene los ficheros de kernel y host modificados, para lograr el mejor rendimiento

Una vez se cuenta con los ficheros del proyecto, es posible realizar la primera ejecución, tanto en CPU como en FPGA, para tomar contacto con el modelo de aceleración. Para ello, es necesario portar el proyecto a la plataforma Alveo U50, mediante la modificación de los siguientes ficheros:

- "makefile/Makefile": Modificar "TARGET" (hw_emu) y "PLATFORM" (xilinx_u50_gen3x4_xdma_2_202010_1) para utilizar por defecto emulación Hardware y la plataforma Alveo U50 instalada.
- "makefile/common.mk": Modificar valores "DDR" por valores "HBM" en las tres opciones del documento.
- "makefile/connectivity.cfg": Sustituir valores "DDR" a "HBM", especificando los bancos de memoria para cada parámetro del kernel ("input_words:HBM[0:14]", "output_flags:HBM[15:29]" y "Bloom_filter:HBM[30]"). Esto es necesario ya que los bancos de memoria HBM cuentan con un tamaño menor que los casos DDR de la tarjeta U200 (HBM cuenta con 256MB por banco, frente a los 16 GB de DDR).

Una vez portado el proyecto, es posible ejecutar sobre CPU y emular sobre Hardware para comprobar las diferencias, preparando el terminal sobre el que se ejecuta previamente, con los comandos "source" y "export" correspondientes. Para diferentes valores:

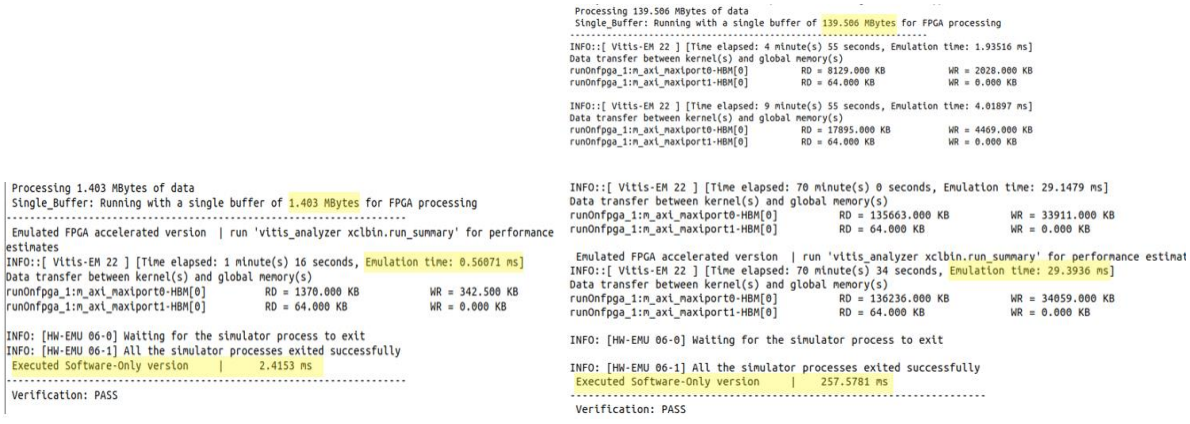


Figura 20: Resultados de emulación Hardware con flujos de 100 y 10000 documentos

Estos primeros resultados permiten detectar una mejora considerable de la capacidad de la aplicación al realizar la aceleración. Mientras en CPU requiere de 2.415 ms y 257.578 ms, para los casos de 100 y 10000 documentos respectivamente, la FPGA requiere solamente de 0.560 ms y 29.393 ms. Esto se traduce en que la aplicación final acelerada, pasa de un

rendimiento de 0.58 GB/s y 0.54 GB/s en CPU, a 2.50 GB/s y 4.76 GB/s, o lo que es lo mismo, se multiplica el rendimiento por un factor de 4.31 y de 8.81 (para 100 y 10000 documentos).

10.2.2. Definición de arquitectura de aplicación.

Una vez visualizado el potencial del proceso de aceleración, se procede a analizar la forma de lograr esta mejora de rendimiento. El primer paso indicado en la metodología consiste en la elaboración de la arquitectura de la aplicación, definiendo qué bloques deben ser acelerados y mediante qué grado de paralelización.

Para esta etapa, es importante dividir el programa en bloques funcionales para determinar el impacto de cada uno de ellos. El proyecto cuenta con una serie de marcas de tiempo que permiten ver cuánto tiempo de ejecución requiere cada bloque funcional, tras la ejecución de la aplicación, como se observa en la Figura 21.

```
user@ubuntu-pc:~/Xilinx/Development/Vitis-Tutorials-2021.1/Hardware_Accelerators/Desig
n_Tutorials/02-bloom/cpu_src$ make run
rm -rf temp_dir log_dir report_dir *log host runOnfpga* *.csv *summary .run .Xil vitis
* *jou xilinx* gpofresult.txt gmon.out
g++ -D__USE_XOPEN2K8 -D__USE_XOPEN2K8 \
        -I. \
        -O3 -Wall -fmessage-length=0 -std=c++11\
        ./compute_score_host.cpp \
        ./MurmurHash2.c \
        ./main.cpp \
        -o ./host
./host 100000
Initializing data
Creating documents - total size : 1398.903 MBytes (349725824 words)
Creating profile weights

 Total execution time of CPU          |  2916.3556 ms
 Compute Hash processing time         |  2512.4995 ms
 Compute Score processing time        |   403.8560 ms
------------------------------------------------------------------------
 Execution COMPLETE
```

Figura 21: Resultados ejecución Software puramente sobre CPU de la aplicación original

De esta forma, se pueden observar dos funcionalidades diferenciadas: el cálculo del hash de los documentos, y el cálculo de la puntuación final de los mismos. La diferencia de tiempos de ejecución es considerable, teniendo aproximadamente un 86% del total para el hash, y un 14%, para la puntuación final.

10.2.2.1. Identificación de funciones a acelerar.

Una vez definido una primera estimación del impacto de cada bloque funcional, es necesario analizar en detalle, de manera recursiva, el código de dichos bloques. Teniendo en cuenta los dos bloques comentados, se puede realizar el siguiente análisis:

- Cómputo de flags de documento:

Se analiza la función “MurmurHash2” encargada de calcular los flags de cada documento. Esta función requiere de varias operaciones lógicas XOR, desplazamientos aritméticos y operaciones de multiplicación. Los desplazamientos necesitan un ciclo de reloj completo por operación, por lo que supone que, en el peor de los casos, sean necesarios 44 ciclos completos de reloj. Además, las operaciones lógicas y multiplicaciones se pueden realizar de manera óptima sobre la FPGA con arquitecturas personalizadas y componentes dedicados. Al poder desarrollarse la arquitectura de manera personalizada y teniendo en cuenta que, por cada documento, la función “MurmurHash2” debe ser ejecutada dos veces y que ambas ejecuciones son independientes la una de la otra, es posible lograr una paralelización que mejore el rendimiento.

Un punto adicional a tener en cuenta es que en cada iteración se accede a un array de manera secuencial para la totalidad de los documentos. Adecuando el código, se puede lograr un mayor rendimiento utilizando bancos de memoria de alto rendimiento disponibles en Alveo, como DDR o HBM, así como dividiendo el bucle de acceso en varios bucles ejecutados en paralelo de manera simultánea.

Gracias a estos puntos y teniendo en cuenta que el tiempo de ejecución de este bloque es superior al 80% del global de la aplicación, esta función es una gran candidata para la aceleración, mediante una paralelización adecuada, utilizando memorias más eficientes y rápidas, y ejecutando la función sobre una arquitectura personalizada con mejor rendimiento. Por tanto, esta función se destina a ser ejecutada sobre la FPGA.

- Cómputo de puntuación de documento:

  En este caso, se trata de una operación con menor impacto en el tiempo de ejecución global, requiriendo de operaciones más simples. Como solamente se ejecuta una vez por fichero, con accesos aleatorios a memoria (este acceso no es tan eficiente sobre DDR o HBM), se rechaza, al menos temporalmente, para la aceleración, dedicando mayores esfuerzos a la función hash. Por tanto, es ejecutada sobre CPU.

10.2.2.2. Establecimiento de objetivos globales.

Una vez definidas las funciones a acelerar, es necesario analizar cuál es el objetivo realista y posible de aceleración de la aplicación global. Para ello, se realiza un análisis tanto de la capacidad alcanzable debido al bus PCIe, como por las funciones de computación.

En cuanto a las funciones, la función de cálculo de puntuación se realiza en CPU. Teniendo en cuenta las pruebas anteriores, toma unos 403 ms. Dada una CPU, no se puede acelerar la función más allá del rendimiento definido por la arquitectura estática. Incluso en el caso ideal en que la FPGA ejecutará las funciones de hash en tiempo cero, la aplicación global requeriría esos 403 ms. Además, las ejecuciones en FPGA con tiempo cero tampoco son viables, por lo que es necesario tener en cuenta el tiempo de ejecución en la misma. Junto a estos tiempos de ejecución en host y en FPGA, hay que sumar las latencias introducidas por las comunicaciones entre ambas partes.

Se establece como objetivo para la aplicación el punto de aceleración en que la función de cómputo del hash en FPGA se ejecute tan rápido como la de cómputo de puntuación en la CPU, definiendo un escenario en que la función hash no supone el cuello de botella. Usando como entrada unos 100000 documentos para los cálculos, siendo un equivalente de unos 350 millones de palabras, con aproximadamente 3500 palabras por documento, manteniendo solamente la función de hash en FGPA, supone un total de ejecución de unos 2512.5 ms.

El objetivo es, por tanto, lograr una aceleración de forma que se ejecute la función hash y se obtenga el resultado final en un tiempo en torno a los 403 ms, en vez de los 2916 ms actuales, para los 10000 documentos generados.

El siguiente paso es analizar la capacidad de las comunicaciones PCIe. Para ello, se ejecuta el comando “xbutil validate” que, teniendo una tarjeta Alveo instalada, realiza un proceso de prueba para verificar el funcionamiento. Como en esta etapa no se cuenta con una tarjeta física, se añade una captura como ejemplo, para el caso de la Alveo U50, proporcionada en la propia documentación de Xilinx. En la Figura 22 se aprecia un valor de unos 11.9GB/s tanto para escritura como para lectura.

```
INFO: Found 1 cards
INFO: Validating card[0]: xilinx_u50_gen3x16_xdma_201920_3
INFO: == Starting AUX power connector check:
AUX power connector not available. Skipping validation
INFO: == AUX power connector check SKIPPED
INFO: == Starting PCIE link check:
INFO: == PCIE link check PASSED
INFO: == Starting SC firmware version check:
INFO: == SC firmware version check PASSED
INFO: == Starting verify kernel test:
INFO: == verify kernel test PASSED
INFO: == Starting DMA test:
Host -> PCIe -> FPGA write bandwidth = 11933.1 MB/s
Host <- PCIe <- FPGA read bandwidth = 11966.5 MB/s
INFO: == DMA test PASSED
INFO: == Starting device memory bandwidth test:
..........
Maximum throughput: 52428 MB/s
INFO: == device memory bandwidth test PASSED
INFO: == Starting PCIE peer-to-peer test:
P2P BAR is not enabled. Skipping validation
INFO: == PCIE peer-to-peer test SKIPPED
INFO: == Starting memory-to-memory DMA test:
M2M is not available. Skipping validation
INFO: == memory-to-memory DMA test SKIPPED
INFO: Card[0] validated successfully.
INFO: All cards validated successfully.
```

Figura 22: Resultado de test de validación de tarjeta Alveo mediante herramienta "xbutil" (Xilinx, Alveo U50 Data Center Accelerator Card Installation Guide (UG1370), 2020)

Comparando con el objetivo de aceleración que supone procesar 1.398GB en 403ms (3.469GB/s requeridos), se observa que este se encuentra por debajo del límite establecido por la tecnología PCIe, por lo que es un objetivo alcanzable.

Una vez se define qué partes hay que acelerar y el grado en que debe hacerse, se puede pasar a identificar la paralelización para la aplicación en la FPGA.

10.2.2.3. Identificación de paralelización en FPGA

En cuanto a la solución Software anterior, el flujo sigue un diagrama como el de la Figura 23. En este flujo se aprecia que la función hash, ejecutada dos veces para cada palabra de cada documento, de manera secuencial, supone el mayor impacto al rendimiento. En cuanto al cálculo de la puntuación final, no se realiza hasta que los valores hash de todos los documentos han sido calculados. Además, si se ejecuta sobre la FPGA, hay que tener en cuenta un retardo adicional introducido por la comunicación PCIe.

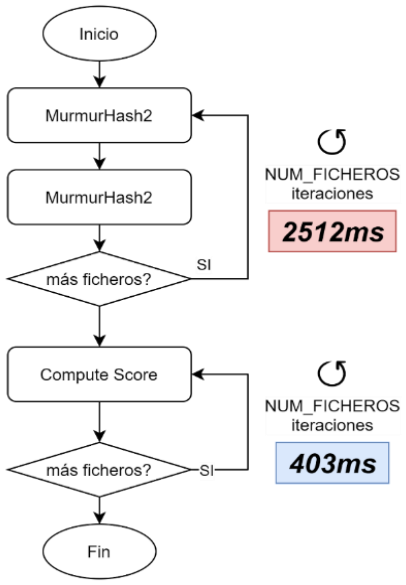


Figura 23: Diagrama de flujo de aplicación original Filtro Bloom

La aplicación completa, teniendo en cuenta la interacción host-FPGA, se puede dividir temporalmente en las siguientes funciones, de forma que se acelere la función hash en FPGA:

- A: Transferencia de datos de un tamaño de 1.398 GB desde el host hacia la memoria HBM de la tarjeta Alveo, mediante PCIe. Teniendo en cuenta la velocidad de escritura/lectura obtenida del test, de unos 11.9 GB/s, el tiempo de transferencia se estima de unos 117,5 ms.
- B: Cálculo de los valores hash de los flags de salida en la FPGA
- C: Transferencia de los datos de flags obtenidos en la FPGA, hacia el host mediante PCIe. Teniendo en cuenta las pruebas realizadas, tanto para 100 palabras (1.3 MB), como para 10000 (139MB), se obtiene un valor de hash estimado de ¼ del valor de datos de entrada (342/1370 = 34059/136236 = ¼). Realizando el mismo cálculo del tiempo de transferencia, ahora para un total de 349.5 MB de datos (1.398GB*¼), la estimación es de 29.4 ms.
- D: Cálculo de puntuación final a partir de los flags disponibles tras su procesamiento y envío por parte de la FPGA. Este punto toma aproximadamente 403 ms, como se ha visto anteriormente.

Teniendo en cuenta el objetivo de la aplicación de 403 ms, obteniendo una suma total de retardo estimada de 550 ms (A+C+D), en ejecución secuencial puramente en FPGA, para los 100000 documentos, sin tener en cuenta el tiempo necesario para calcular los hashes, se ve claramente que no es suficiente. Sin embargo, gracias a la FPGA, es posible paralelizar la ejecución de las funciones, además de desarrollar una arquitectura de aceleración adecuada y con el mayor rendimiento posible para la función hash, evitando que la FPGA suponga un cuello de botella.

10.2.2.4. Identificación de paralelización en Software.

Si los cuatro pasos o funciones anteriores se adaptan a un flujo de procesamiento concurrente y solapado de procesamientos CPU-FPGA, FPGA-FPGA, CPU-PCIe y FPGA-PCIe, se puede lograr un rendimiento acorde al objetivo. En la Tabla 9 se muestra cómo se definen cada uno de los estados de ejecución, en función de la disponibilidad de los datos de entrada para las funciones A, B, C y D comentadas. En ella se muestra una estimación de los bloques funcionales en que se puede dividir la aplicación, de cara a la paralelización de tareas.

Words para TX. a FPGA	Datos para Hash en FPGA	Flags para Tx. a Host	Datos para Puntuación en Host	Funciones ejecutadas en paralelo	Número de iteraciones del caso para N bloques de procesamiento
SI	NO	NO	NO	A	1 (primer ciclo)
SI	SI	NO	NO	A,B	1 (segundo ciclo)
SI	SI	SI	NO	A,B,C	1 (tercer ciclo)
SI	SI	SI	SI	A,B,C,D	N - 6
NO	SI	SI	SI	B,C,D	1 (antepenúltimo ciclo)
NO	NO	SI	SI	C,D	1 (penúltimo ciclo)
NO	NO	NO	SI	D	1 (último ciclo)

Tabla 9: Ejemplo de estimación para la división de tareas del paradigma Vitis en aplicación "Bloom Filter"

Para valores de N mayores que 100 supone que, teóricamente, durante más de un 94% de los ciclos de ejecución de estos bloques funcionales estimados, se están ejecutando simultáneamente las 4 tareas. Este esquema se ve más claramente en la Figura 24, donde se aprecia que los únicos puntos en que el proceso es menos óptimo, son al inicio y al final, suponiendo una parte ínfima respecto al cómputo global, sobre todo en implementaciones con grandes conjuntos de datos a procesar.

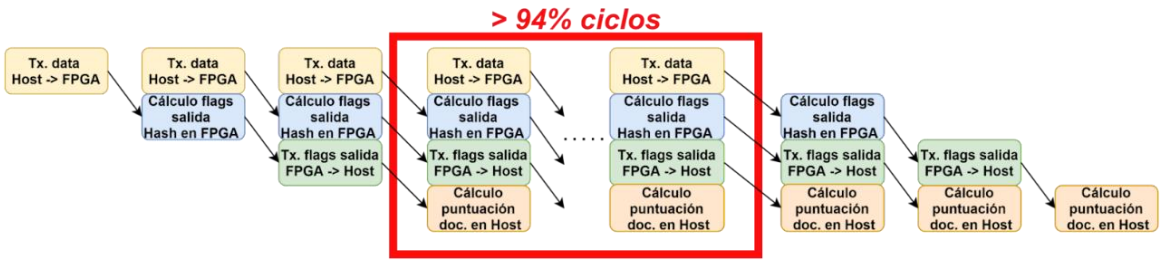


Figura 24: Diagrama de flujo global estimado para lograr aceleración en aplicación "Bloom Filter"

Para lograr esto, es necesario dividir el conjunto de datos de entrada en bloques que puedan ser enviados y procesados de manera independiente, de forma que el kernel no tenga que esperar a tener datos disponibles, disponiendo de ellos en el instante inmediato en que libera sus recursos para comenzar a procesar nueva información. Es crucial la paralelización de las ejecuciones en CPU y FPGA, además de definir una arquitectura en la FPGA que sea capaz de procesar un número suficiente de palabras en paralelo, de forma que se ejecute la función hash tantas veces por ciclo de reloj como sea necesario. Adicionalmente, si la CPU lo permite, es posible añadir un mayor grado de rendimiento, paralelizando ligeramente el cálculo de las puntuaciones de los documentos, mediante el uso de varios núcleos o utilizando varios bancos de memoria, por ejemplo.

10.2.3. Implementación de aplicación acelerada

Una vez definida la arquitectura de la aplicación, definiendo las funciones a ejecutar tanto en CPU como en FPGA, indicando cómo y cuándo se deben realizar las transferencias de datos y estableciendo el nivel de paralelismo requerido, se puede pasar a desarrollar e implementar el kernel que permite ejecutar la aplicación acelerada sobre la tarjeta Alveo. Como se ve en la metodología, es necesario definir dos elementos principales, macroestructura y microestructura, definiendo las funciones de alto y bajo nivel, respectivamente.

En esta sección, se desarrolla un kernel optimizado que permite procesar 4, 8 o 16 palabras en paralelo, mediante la transmisión de la totalidad de los datos, desde la CPU hacia la FPGA, utilizando un único buffer y mediante una transferencia única.

Para la creación del kernel, se definen una serie de requerimientos sobre la interfaz, definidos por las características de los puertos I/O de la tarjeta, con tamaños de 512 bits:

- Lectura de múltiples palabras almacenadas en DDR como accesos de 512 bits (equivalente a leer 16 palabras de 32 bits por acceso)
- Escritura de múltiples flags en DDR como accesos de 512 bits (equivalente a escribir 32 flags de 16 bits por acceso)
- Procesamiento de “PF” palabras en paralelo, con 2 ejecuciones de función hash por palabra, simultáneamente

10.2.3.1. Macroestructura

Las características de la interfaz permiten definir la macroestructura, dividiendo la aplicación completa en un patrón “load-compute-store”. Para definir estas funcionalidades, se utilizan los códigos fuente contenidos en la carpeta “reference_files”, donde se destaca lo siguiente:

- Los arrays de entrada y salida se adaptan para tener un ancho adecuado al del puerto I/O, de 512 bits.
- Se define el patrón “load-compute-store” con las subfunciones necesarias. Además, se añaden arrays locales de tipo “hls::stream” para optimizar las transferencias entre bloques funcionales.
- Se establecen las directivas “INTERFACE” para los arrays de entrada y salida de la interfaz definiendo los parámetros “m_axi” (indicando puerto AXI maestro), “port” (especificando nombre del argumento a mapear en interfaz AXI), “offset=slave” (indicando que las direcciones están disponibles mediante esclavo AXI del kernel) y “bundle” (definiendo el nombre de la interfaz “m_axi” a la que se mapea).
- Se crea y configura el array y la forma de lectura, en modo ráfaga, de los datos de entrada como bloques de 512 bits
- Se crea el array de palabras en paralelo a procesar, en función del parámetro de paralelización indicado (“PARALLELIZATION”)
- Se define como función de alto nivel “compute_hash_flags_dataflow” que llama de manera sucesiva a la función “compute_hash_flags”, encargada del procesamiento de las palabras en paralelo.

- Con el valor de “PARALLELIZATION”, se define el flujo de datos de salida, conteniendo los flags de hash calculados, para adecuarlo al puerto de salida de 512 bits
- Se define la escritura de los datos de salida en la memoria global, mediante el interfaz correspondiente
- Mediante la directiva “DATAFLOW”, se habilita el segmentado de las tareas, de forma que se indica al compilador Vitis High-Level Synthesis (HLS) que se deben ejecutar en paralelo todas las funciones de manera simultánea, creando un pipeline de tareas, con ejecución concurrente.

La macroestructura se define siguiendo el diagrama de la Figura 25, cumpliendo con el patrón “load-compute-store”:

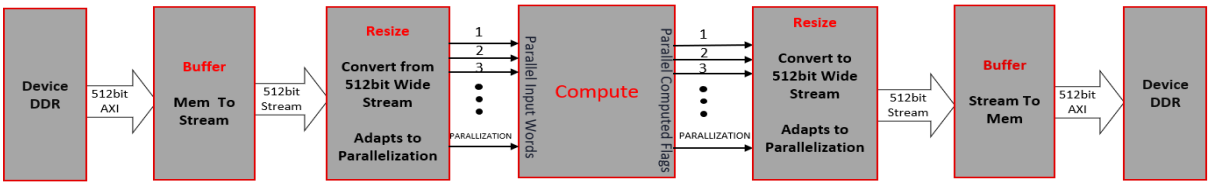


Figura 25: Diagrama de bloques con patrón "load-compute-store" (Xilinx, Vitis In-Depth Tutorials (Repositorio GitHub), 2021)

De esta forma, se puede simplificar el proceso según el diagrama de la Figura 26, únicamente mediante ejecución secuencial.

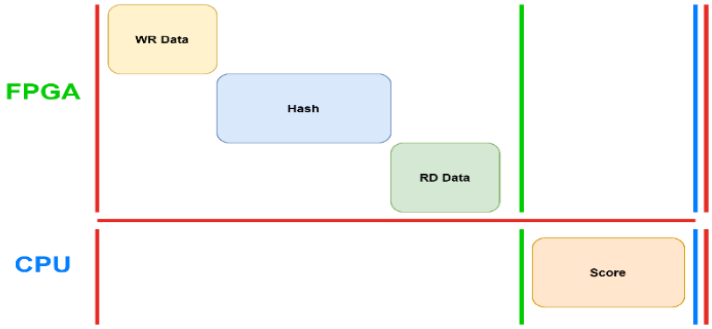


Figura 26: Diagrama de flujo secuencial para primera iteración del proceso de aceleración de "Bloom Filter”

10.2.3.2. Microestructura

Una vez definidas las funciones de alto nivel, es necesario identificar los bucles a optimizar para mejorar el rendimiento y la latencia:

- Función principal de ejecución en FPGA “runOnfpga”:
    - Se hace uso de la directiva “HLS PIPELINE II=1”, permitiendo los accesos a memorias en ráfagas, pudiendo leer los coeficientes del filtro cada ciclo. Esta opción permite establecer el tiempo entre iteraciones consecutivas al valor definido.
- Función de cómputo de hash “compute_hash_flags”:
    - Mediante la directiva “HLS UNROLL” se despliegan los bucles internos, de forma que se consigue ejecutar tantas iteraciones en paralelo como se definan en el parámetro “PARALLELIZATION”. Se consigue ejecutar en paralelo tantas funcionalidades hash como se indiquen.
    - Vitis HLS intenta, por defecto, establecer un “II=1” para el bucle externo. Con el bucle interno “unrolled” se puede iniciar el bucle externo en cada ciclo de reloj, pudiendo procesar “PARALLELIZATION” palabras en paralelo, por cada iteración.
    - Con la directiva “HLS LOOP_TRIPCOUNT min=1 max=t_sixe/pf” se informa sobre la latencia de la función tras la síntesis HLS.

10.2.3.3. Construcción y ejecución de kernels

Tras la fase de desarrollo, es posible compilar y construir los programas ejecutables, tanto en CPU, como en FPGA, pudiendo pasar a su emulación, ya sea de tipo Software o Hardware, o a su ejecución, directamente sobre la tarjeta Alveo.

Como en este apartado se centra el foco en el uso de la paralelización para procesar un mayor número de muestras por ciclo, se compilan las soluciones para valores de “PARALLELIZATION” de 4, 8 y 16 (que indican el número de muestras procesadas por ciclo). De esta forma, se generan tres kernels diferentes, con arquitecturas diferentes.

Para el proceso se compilan todas las opciones, tanto para emulaciones como ejecución, aunque para este apartado sólo es de interés la emulación Hardware. Mediante la emulación Hardware de cada kernel, se pueden extraer una serie de conceptos, comparando el impacto del rendimiento y los recursos utilizados. Para los tres casos, para un valor de 1000 documentos, se obtiene:

- Tiempos de ejecución:

Núm. Documentos		1000		
Total MB.		13,971		
Kernel	Bloom 4 x	Bloom 8 x	Bloom 16 x	CPU
runOnfpga (ms)	2,972	1,518	0,787	x
Tiempo Total (ms)	3,176	1,714	0,989	25
fpga/total (%)	93,6	88,6	79,6	x
CU utilización (%)	99,8	99,5	99,6	x
Throuhput (GB/s)	4,399	8,151	14,126	0,559
Índice mejora	7,87	14,59	25,28	1,00

Figura 27: Tiempos de ejecución para las diferentes implementaciones de "Bloom Filter", según número de muestras procesadas en paralelo

- Recursos utilizados:

Kernel	Bloom 4 x	Bloom 8 x	Bloom 16 x
BRAM	124 (4%)	188 (6%)	316 (11%)
DSP	8 (~0%)	16 (~0%)	32 (~0%)
FF	15365 (~0%)	20001 (1%)	28200 (1%)
LUT	24306 (2%)	26165 (3%)	24435 (2%)
URAM	0 (0%)	0 (0%)	0 (0%)

Figura 28: Cantidad de recursos utilizados para las diferentes implementaciones de "Bloom Filter", según número de muestras procesadas en paralelo

La mejora de rendimiento al procesar un mayor número de palabras en paralelo es posible gracias a que los puertos I/O, con un ancho de 512 bits, pueden ser óptimamente utilizados si se leen de manera simultánea 16 palabras (utilizando los 512 bits), frente a los casos previos de 4 y 8 palabras (que solamente hacen uso de 128 y 256 bits, respectivamente), desperdiciando parte de los recursos disponibles reservados. Este ancho supone también un límite para la aceleración, ya que no es posible leer más de esos 512 bits por ciclo.

También hay que destacar que estos tiempos, al tratarse de emulación Hardware, no incluyen los tiempos reales de transferencia entre Host y FPGA, ya que se idealizan estas comunicaciones. Sin embargo, sirven para hacerse una idea del funcionamiento particular del kernel.

10.2.4. Optimización.

10.2.4.1. Optimización de transferencias CPU-FPGA mediante solapamiento de funciones.

A pesar de que los kernel vistos hasta el momento operan prácticamente al máximo rendimiento, existen latencias debidas a la espera entre la disponibilidad de los datos y procesamiento de los mismos. Xilinx recomienda utilizar buffers de gran tamaño, lo cual puede suponer tiempos largos de espera si no se realiza el proceso adecuadamente. Para solucionar

estas limitaciones debidas a la ejecución secuencial, es posible solapar ejecuciones y transferencias, evitando tiempos de espera no eficaces, tanto en CPU como en FPGA. De esta forma, se divide el buffer total en varios segmentos, que se envían de manera secuencial, para permitir al kernel procesar los segmentos recibidos, mientras la CPU continúa enviando el resto de datos. Este nuevo proceso se representa en la Figura 29, donde se aprecia que al reducir el tiempo de ejecución en FPGA gracias a la paralelización (VERDE), se consigue reducir el tiempo total de ejecución de la aplicación (ROJO), manteniendo constante el tiempo de ejecución en CPU (AZUL).

Para lograrlo, es necesario modificar el código del Host:

- Se crean tantos sub buffers como se desee, para enviar de manera secuencial dichos segmentos del total de documentos. Para definir el número de segmentos se hace uso de la variable “ITER”.
- Se crea un vector de eventos para coordinar las operaciones de lectura, computación y escritura, de forma que cada iteración sea independiente del resto, permitiendo el solapamiento.
- Se preparan los argumentos y se ponen en cola la ejecución del kernel y la lectura de los datos procesados. Durante las iteraciones sucesivas, se pone en cola el nuevo kernel, con los nuevos parámetros, junto con la lectura de sus datos correspondientes cuando estén procesados.
- Finalmente, el host espera hasta que los datos de salida son enviados al completo por la FPGA para procesar los resultados finales.

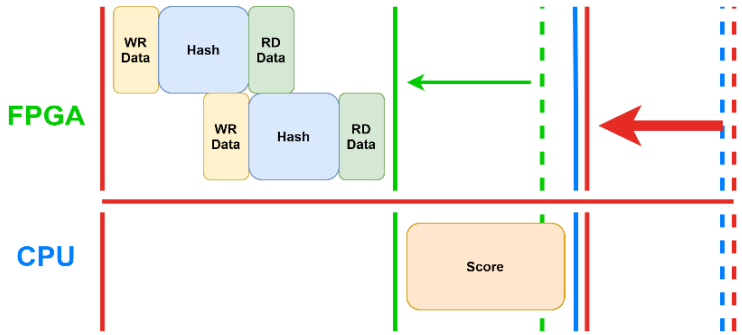


Figura 29: Diagrama de flujo secuencial para segunda iteración del proceso de aceleración de "Bloom Filter”, con solapamiento de funciones en FPGA

En este modelo, se aprecia que la CPU presenta tiempos de espera no deseados, tras recibir los primeros datos de la FPGA. Otra práctica recomendada es la de solapar la transferencia desde la FGPA con el procesamiento de los propios datos en la CPU. Es el comportamiento análogo visto en el caso anterior, aplicado a la comunicación FPGA-CPU, en vez de a la comunicación CPU-FPGA. El resultado es el observado en la Figura 30, donde se aprecia que, aunque requiera prácticamente el mismo tiempo de ejecución en CPU, al eliminar los tiempos de espera, el rendimiento global de la aplicación aumenta (ROJO).

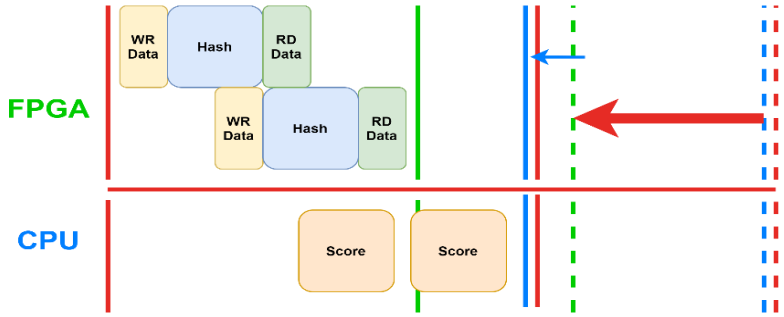


Figura 30:Diagrama de flujo secuencial para tercera iteración del proceso de aceleración de "Bloom Filter", con solapamiento de funciones en FPGA y CPU

De esta forma, se consigue solapamiento de computación CPU-FPGA y FPGA-FPGA, así como de computación y transferencias, CPU-PCIe y FPGA-PCIe, reduciendo aún más los tiempos ineficientes de espera. Para ello, en el proyecto se realizan las siguientes modificaciones en el programa host:

- Se crean variables particulares para el seguimiento de las palabras procesadas.
- Define como única condición de bloqueo del Host aquella situación en la que no se encuentra ningún hash procesado disponible para el procesamiento.

10.2.4.2. Optimización de transferencias CPU-FPGA eliminando contenciones I/O.

Para finalizar el proceso y lograr un rendimiento óptimo, se realiza una nueva mejora, mediante la eliminación de las contenciones debidas a accesos a memoria, de manera simultánea por parte de la CPU y de la FPGA.

Para lograr esto, se realiza una modificación que permite utilizar varios bancos de memoria HBM de la tarjeta Alveo, de manera que mientras un banco de memoria está ocupado, se utiliza otro disponible.

Además, es necesario modificar el programa del Host para permitir este comportamiento:

- Mediante una extensión de la API OpenCL, se establece el envío de las palabras a los diferentes bancos de memoria, alternativamente. Mediante una serie de flags, por cada banco de memoria, se indica al kernel la disponibilidad de los datos.
- Se crean los buffers para la transferencia de palabra, en cada banco de memoria a utilizar.
- Mediante un array de buffers de cada banco de memoria, se ejecutan los kernels de manera alterna, utilizando el banco de memoria disponible en cada momento.

Se modifica el valor de asignación de puertos y se recompila la solución.

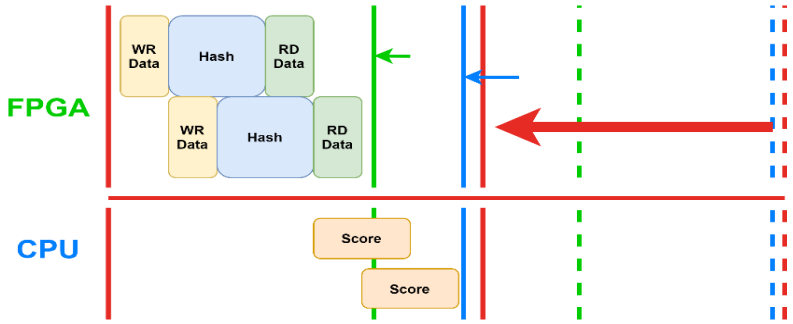


Figura 31: Diagrama de flujo secuencial para cuarta iteración del proceso de aceleración de "Bloom Filter", con solapamiento de funciones en FPGA y CPU, eliminando contingencias I/O

10.2.5. Resultados.

Tras el desarrollo del proyecto, se procede a la compilación y ejecución de la aplicación acelerada sobre la tarjeta Alveo, dejando de lado la emulación. De esta forma, se obtienen los siguientes resultados, para cada caso de interés:

1. PARALLELIZATION=8, ITER=8, sin solapamiento global:

Kernel Execution

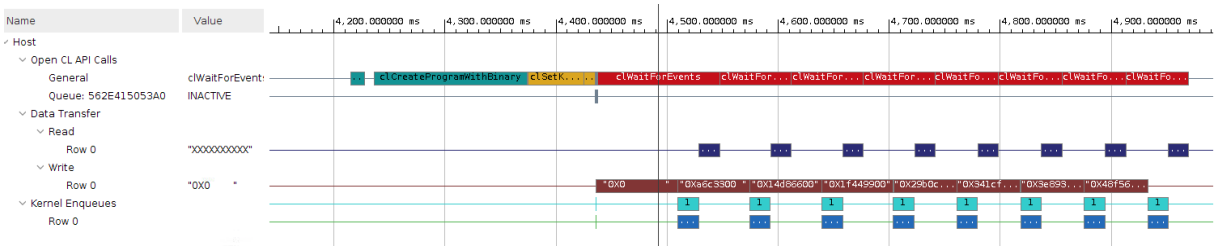
Kernel	Enqueues	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)
run0nfpga	9	150.128	0.258	16.681	18.852

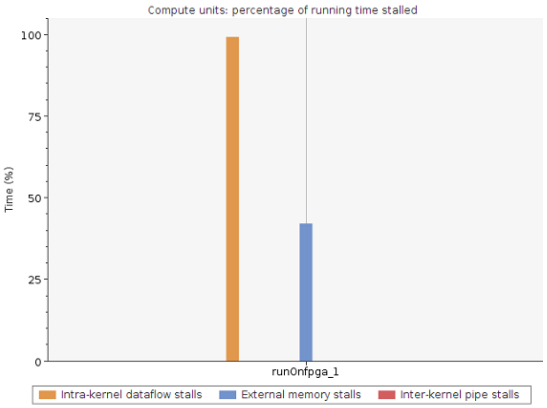
Top Kernel Transfer

Compute Unit	Device	Number of Transfers	Avg Bytes per Transfer	Transfer Efficiency (%)	Total Data Transfer (MB)	Total Write (MB)	Total Read (MB)	Total Transfer Rate (MB/s)
run0nfpga_1	xilinx_u50_gen3x4_xdma_base_2-0	1707725	1023.000	25.000	1748.700	349.726	1398.970	8785.840

Host Transfer

Context:Number of Devices	Transfer Type	Number of Buffer Transfers	Transfer Rate (MB/s)	Avg Bandwidth Utilization (%)	Avg Size (KB)	Total Time (ms)	Avg Time (ms)
context0:1	READ	8	2372.909	24.718	43715.800	147.383	18.423
context0:1	WRITE	9	2824.876	29.426	155441.000	495.232	55.026






```
Processing 1398.905 MBytes of data
Generic_Buffer : Splitting data in 8 sub-buffers of 174.863 MBytes for FPGA processing
----------------------------------------------------------------
Executed FPGA accelerated version  |  1023.5772 ms   ( FPGA 551.430 ms )
Executed Software-Only version     |   2214.1181 ms
----------------------------------------------------------------
Verification: PASS
```

Figura 32: Resultados de ejecución en Alveo U50 del proyecto "Bloom Filter", mediante procesamiento de 8 muestras en paralelo, con una transferencia del buffer global dividida en 8 iteraciones, sin solapamientos

2. PARALLELIZATION=8, ITER=8, con solapamiento global:

Kernel Execution

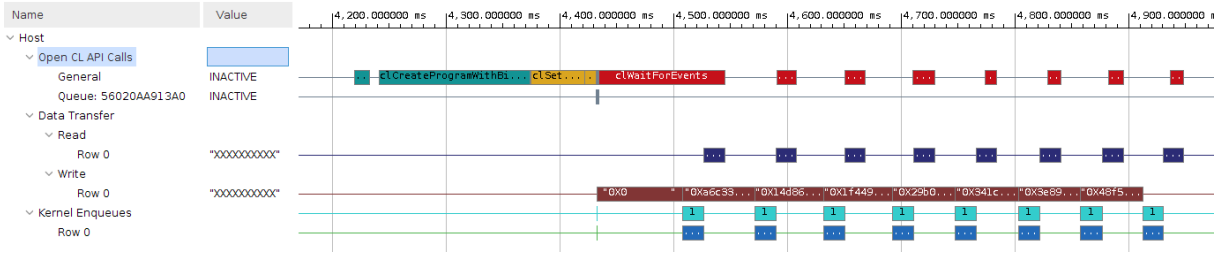
Kernel	Enqueues	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)
run0nfpga	9	149.102	0.408	16.567	18.799

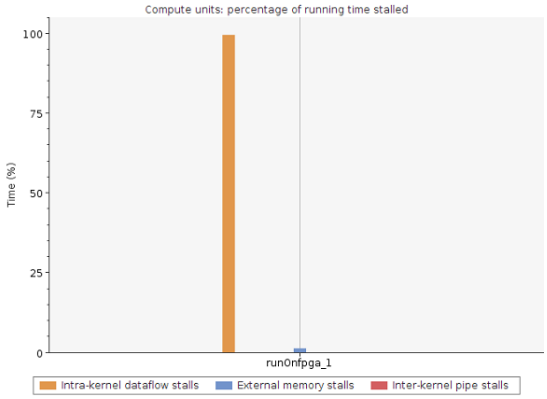
Top Kernel Transfer

Compute Unit	Device	Number of Transfers	Avg Bytes per Transfer	Transfer Efficiency (%)	Total Data Transfer (MB)	Total Write (MB)	Total Read (MB)	Total Transfer Rate (MB/s)
run0nfpga_1	xilinx_u50_gen3x4_xdma_base_2-0	1707725	1023.000	25.000	1748.700	349.726	1398.970	8791.360

Host Transfer

Context:Number of Devices	Transfer Type	Number of Buffer Transfers	Transfer Rate (MB/s)	Avg Bandwidth Utilization (%)	Avg Size (KB)	Total Time (ms)	Avg Time (ms)
context0:1	READ	8	2396.815	24.967	43715.800	145.913	18.239
context0:1	WRITE	9	2920.207	30.419	155441.000	479.065	53.229






```
Processing 1398.905 MBytes of data
Splitting data in 8 sub-buffers of 174.863 MBytes for FPGA processing
------------------------------------------------------------------
Executed FPGA accelerated version  |   581.8628 ms   ( FPGA 537.678 ms )
Executed Software-Only version     |   2204.2634 ms
------------------------------------------------------------------
Verification: PASS
```

Figura 33: Resultados de ejecución en Alveo U50 del proyecto "Bloom Filter", mediante procesamiento de 8 muestras en paralelo, con una transferencia del buffer global dividida en 8 iteraciones, con máximo solapamiento alcanzado

3. PARALLELIZATION=16, ITER=8, con solapamiento global:

Kernel Execution

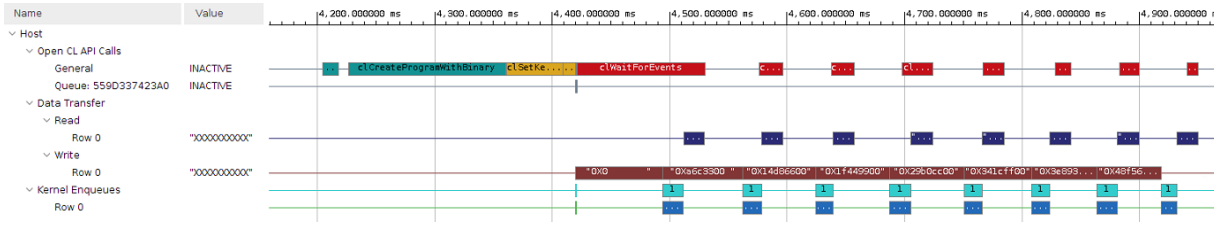
Kernel	Enqueues	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)
run0nfpga	9	128.876	0.366	14.320	18.346

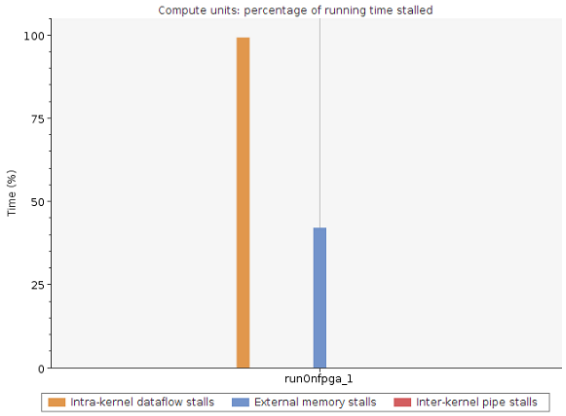
Top Kernel Transfer

Compute Unit	Device	Number of Transfers	Avg Bytes per Transfer	Transfer Efficiency (%)	Total Data Transfer (MB)	Total Write (MB)	Total Read (MB)	Total Transfer Rate (MB/s)
run0nfpga_1	xilinx_u50_gen3x4_xdma_base_2-0	1707725	1023.000	25.000	1748.700	349.726	1398.970	9794.400

Host Transfer

Context:Number of Devices	Transfer Type	Number of Buffer Transfers	Transfer Rate (MB/s)	Avg Bandwidth Utilization (%)	Avg Size (KB)	Total Time (ms)	Avg Time (ms)
context0:1	READ	8	2406.247	25.065	43715.800	145.341	18.168
context0:1	WRITE	9	2808.244	29.253	155441.000	498.165	55.352






```
Processing 1398.905 MBytes of data
Splitting data in 8 sub-buffers of 174.863 MBytes for FPGA processing
------------------------------------------------------------------
Executed FPGA accelerated version  |   586.4027 ms   ( FPGA 544.384 ms )
Executed Software-Only version     |   2219.4693 ms
------------------------------------------------------------------
Verification: PASS
```

Figura 34: Resultados de ejecución en Alveo U50 del proyecto "Bloom Filter", mediante procesamiento de 16 muestras en paralelo, con una transferencia del buffer global dividida en 8 iteraciones, con máximo solapamiento alcanzado

4. PARALLELIZATION=8, ITER=16, con solapamiento global:

Kernel Execution

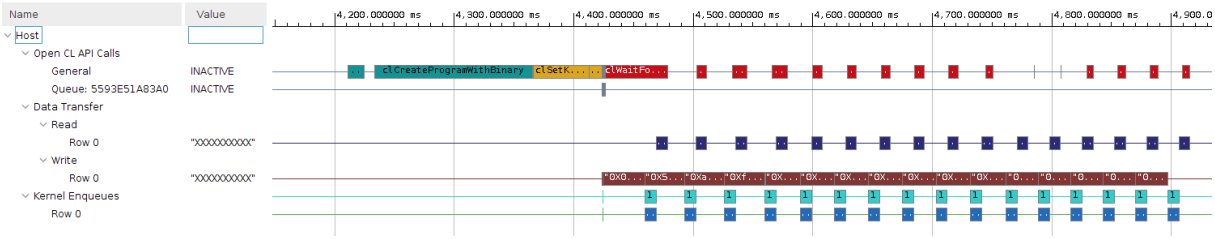
Kernel	Enqueues	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)
runOnfpga	17	151.274	0.324	8.898	9.505

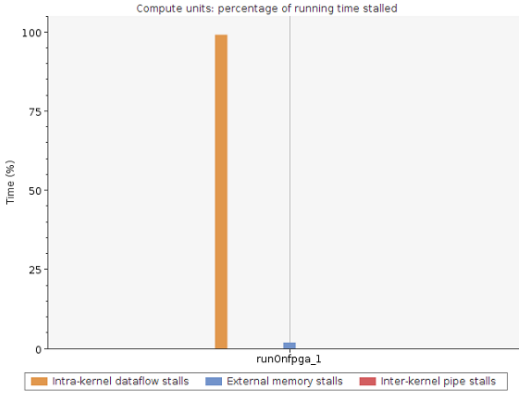
Top Kernel Transfer

Compute Unit	Device	Number of Transfers	Avg Bytes per Transfer	Transfer Efficiency (%)	Total Data Transfer (MB)	Total Write (MB)	Total Read (MB)	Total Transfer Rate (MB/s)
runOnfpga_1	xilinx_u50_gen3x4_xdma_base_2-0	1707736	1023.000	25.000	1748.700	349.727	1398.970	8760.410

Host Transfer

Context:Number of Devices	Transfer Type	Number of Buffer Transfers	Transfer Rate (MB/s)	Avg Bandwidth Utilization (%)	Avg Size (KB)	Total Time (ms)	Avg Time (ms)
context0:1	READ	16	2362.410	24.608	21857.900	148.038	9.252
context0:1	WRITE	17	2964.019	30.875	82292.500	471.985	27.764






```
Processing 1398.907 MBytes of data
Splitting data in 16 sub-buffers of 87.432 MBytes for FPGA processing
------------------------------------------------------------------
Executed FPGA accelerated version  |   548.6808 ms   ( FPGA 527.164 ms )
Executed Software-Only version     |   2210.9168 ms
------------------------------------------------------------------
Verification: PASS
```

Figura 35: Resultados de ejecución en Alveo U50 del proyecto "Bloom Filter", mediante procesamiento de 8 muestras en paralelo, con una transferencia del buffer global dividida en 16 iteraciones, con máximo solapamiento alcanzado

A partir de los resultados y del proceso seguido se pueden extraer una serie de conclusiones:

- La paralelización intra-kernel y extra-kernel es una herramienta potente para lograr acelerar aplicaciones, mediante la ejecución simultánea de tareas y el solapamiento de transferencias de datos y procesamientos, tanto en Host como en FPGA.
- Por norma general, el factor limitante para la aceleración es el canal de comunicaciones, en este caso el conector PCIe ya que, como se aprecia, su tiempo de transferencia supera el tiempo de ejecución del kernel y, por más que se acelere este, siempre supondrá un cuello de botella. Para solventar esta situación, se hace uso de solapamientos y trasferencias entre diferentes bancos de memoria, reduciendo tiempos de espera en CPU y FPGA, y eliminando contingencias en puertos I/O por accesos múltiples.
- Siempre hay que analizar cada algoritmo y aplicación y realizar paralelizaciones con algún objetivo en mente, ya que podría ser contraproducente. El claro ejemplo son los casos para “PARALLELIZATION” con valores de 8 y 16, así como para buffers de transmisión, de los mismos valores:
    - A mayor valor de “PARALLELIZATION”, pasando de 8 a 16, se consigue un peor tiempo de ejecución debido al aumento del tiempo de espera a acceso a memoria, ya que el cuello de botella se encuentra en las transferencias y en el acceso múltiple a los bancos de memoria.
    - A mayor valor de “ITER”, pasando de 8 a 16, se consigue un mejor tiempo de ejecución, gracias a la capacidad de mayor solapamiento de tareas, pudiéndose ejecutar los kernel, mientras se siguen transmitiendo datos tanto, de entrada, como de salida.
- Gracias a la aceleración se pueden conseguir grandes índices de mejora en cuanto a tiempo de ejecución y throughput, obteniendo un valor de 2.55 GB/s (1399MB/548ms) frente al valor inicial ejecutado en CPU de 0.63 GB/s (1399MB/2210ms). Esto supone un índice de mejora de 4.

Como se puede observar, el resultado difiere de los cálculos iniciales por varios motivos, principalmente por la diferencia teórico-práctica, siendo necesarias varias implementaciones e iteraciones para afinar el diseño a los requerimientos específicos. También influye que los

cálculos iniciales se realizan sobre valores teóricos proporcionados por el fabricante para un dispositivo concreto. Con el equipo utilizado y realizando los mismos tests, se obtienen resultados inferiores para los valores de transferencias de datos, que supone el principal cuello de botella en esta implementación. Dichos resultados, obtenidos para el Host y la tarjeta Alveo U50 utilizados son:

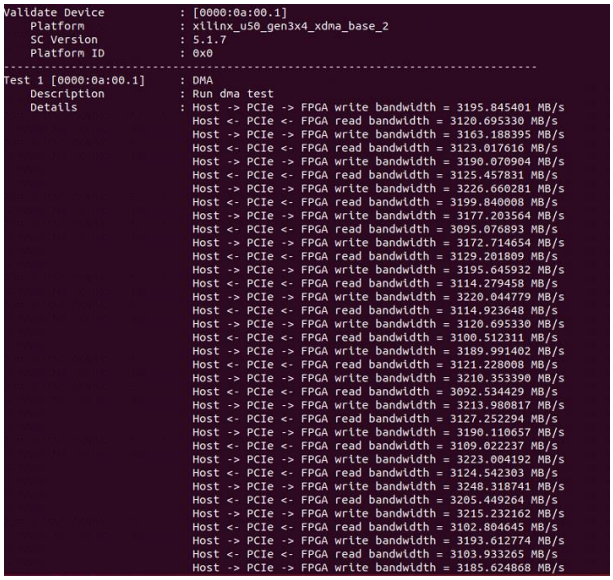


Figura 36: Resultado de test DMA de tarjeta Alveo U50 mediante comando "xbutil"

Se observa claramente que los valores medios se encuentran alrededor de los 3.1GB/s y no en torno a los 11.9 GB/s observados en el planteamiento teórico. Por ello, es fundamental realizar este tipo de pruebas en etapas previas al desarrollo, para determinar objetivos adecuados. En caso de que el equipo y los conectores PCIe sean demasiado limitantes, sería importante considerar su sustitución por otros de mayor calidad.

Un aspecto importante a destacar, es la necesidad de disponer de la tarjeta física para obtener unos resultados reales y en tiempos viables, ya que una emulación Hardware, para valores incluso menores a los 100000 documentos, requiere horas de duración. Además, se proporcionan valores idealizados para ciertas funcionalidades, como es el caso de las transferencias PCIe, mostradas en la siguiente figura:

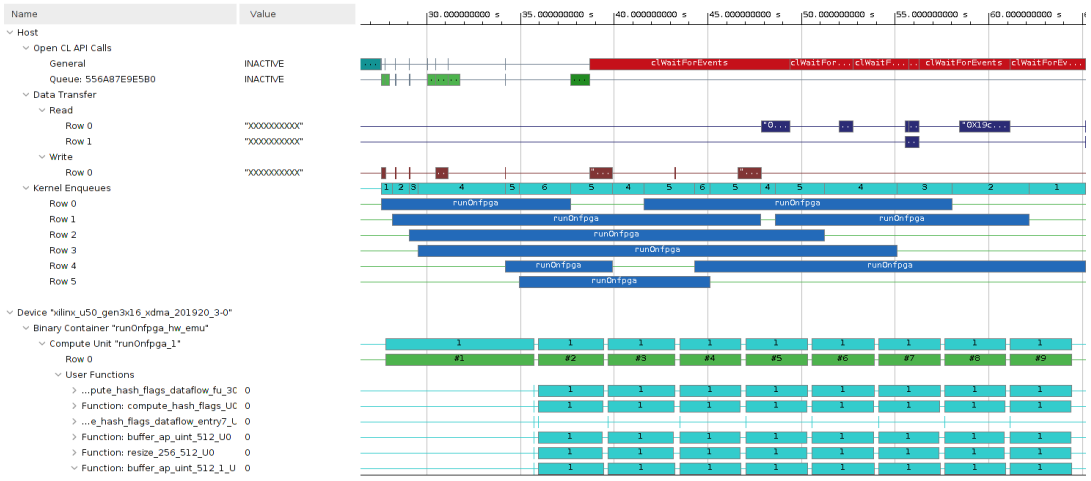


Figura 37: Resultados de emulación Hardware del proyecto "Bloom Filter", mediante procesamiento de 8 muestras en paralelo, con una transferencia del buffer global dividida en 8 iteraciones, con máximo solapamiento alcanzado

Otro ejemplo de ello, se obtiene al intentar emular el proyecto del apartado posterior, con un volumen de datos intermedio (sin llegar a la mitad de capacidad del kernel), obteniendo tiempos de emulación superiores al día, con un total de 2012 minutos (33 horas y media aproximadamente):

```
INFO::[ Vitis-EM 22 ] [Time elapsed: 1997 minute(s) 52 seconds, Emulation time: 10.4075 ms]
Data transfer between kernel(s) and global memory(s)
cholesky_kernel_1:m_axi_gmem0-HBM[0]          RD = 2048.000 KB            WR = 212.000 KB

INFO::[ Vitis-EM 22 ] [Time elapsed: 2002 minute(s) 54 seconds, Emulation time: 10.7056 ms]
Data transfer between kernel(s) and global memory(s)
cholesky_kernel_1:m_axi_gmem0-HBM[0]          RD = 2048.000 KB            WR = 889.000 KB

INFO::[ Vitis-EM 22 ] [Time elapsed: 2007 minute(s) 57 seconds, Emulation time: 11.0053 ms]
Data transfer between kernel(s) and global memory(s)
cholesky_kernel_1:m_axi_gmem0-HBM[0]          RD = 2048.000 KB            WR = 1570.000 KB

INFO: Finish kernel execution
INFO: FPGA execution time of 1 runs:376062462 us
INFO: Average execution time per run: 376062462 us
errA = 0
dataAN = 512
dataAM = 512
--------------
INFO: Result correct
--------------
INFO::[ Vitis-EM 22 ] [Time elapsed: 2012 minute(s) 7 seconds, Emulation time: 11.2565 ms]
Data transfer between kernel(s) and global memory(s)
cholesky_kernel_1:m_axi_gmem0-HBM[0]          RD = 2048.000 KB            WR = 2048.000 KB

INFO: [HW-EMU 06-0] Waiting for the simulator process to exit
```

Figura 38: Resultados de emulación Hardware del proyecto "Cholesky" para matrices de tamaño 512x512

Esto permite apreciar una discordancia con el resultado real visto anteriormente, implementado directamente sobre la tarjeta, aunque sí que permite analizar las capacidades de paralelización, así como analizar las señales internas del kernel, para validar la funcionalidad o el algoritmo en sí.

10.3. Implementación de algoritmo de Cholesky en varios lenguajes de programación.

El código relacionado con este apartado se indica en el Anexo D.2 en la página 200, y se adjunta en el archivo comprimido “TFM_source_files”.

Una vez comprendido el proceso general de aceleración de una aplicación y la metodología en que se basa, es posible realizar un análisis completo sobre el rendimiento de un algoritmo o programa concreto, implementado mediante diferentes lenguajes de programación y tecnologías. De esta forma, se consigue extraer una serie de conclusiones y resultados relacionados con la aceleración de aplicaciones mediante Hardware de aceleración, determinando su potencial, así como los posibles inconvenientes que acarree.

Para este análisis, se establece como base un ejemplo proporcionado por Xilinx, centrado en el algoritmo de descomposición de matrices de Cholesky, dentro del apartado de tutoriales de aceleración o “Acceleration Tutorial”. (Xilinx, Vitis In-Depth Tutorials (Repositorio GitHub), 2021)

Este algoritmo permite solucionar problemas de álgebra lineal en que se desea descomponer una matriz ‘M’ Hermitiana o Hermítica (cuadrada compleja cuya matriz traspuesta es igual a la matriz en sí) y positivamente definida (simétrica y con valores no nulos, para todo valor real no nulo de un vector columna ‘z’, obtenido según la expresión ‘$z^{T}Mz$’, en caso de matrices reales; y con valores no nulos, para todo valor no nulo complejo de ‘z’, obtenidos según la expresión ‘z*Mz’, en caso de matrices complejas). Gracias al algoritmo, se consigue descomponer la matriz ‘M’ de interés, como el producto de una matriz triangular inferior ‘L’ y su conjugada traspuesta ‘L^{T}’. (Wikipedia contributors, "Cholesky decomposition", 2021)

De esta forma, se puede expresar la matrix ‘M’ de la forma ‘LL^{T}’ cuando se trata de una matriz real, o de la forma ‘LL*’, cuando se trata de una matriz compleja. Toda matriz Hermítica definida positiva puede descomponerse de esta forma. También existen otras formas de descomposición, conocidas como ‘LDL’, aunque no se tienen en cuenta para este proyecto.

En cuanto a la computación relacionada con el algoritmo, que es el aspecto realmente interesante de cara al desarrollo del proyecto, se pueden destacar los siguientes puntos:

- Existen diferentes métodos de cálculo, pero en general, la complejidad logarítmica de los algoritmos más usados es de tipo $O(n^3)$. Los algoritmos proporcionados, por tanto, requieren unos (1/3) n^3 FLOPS ($n^3/6$ multiplicaciones y $n^3/6$ sumas) para valores reales

(en los cuales se centra este estudio para simplificar el proceso), siendo 'n' el tamaño de la matriz 'M'.

- El algoritmo seleccionado se conoce como algoritmo Cholesky-Crout y se basa en recorrer la matriz desde la parte superior izquierda, procesando la matriz, columna por columna.
- Las expresiones matemáticas que definen las bases del algoritmo se muestran a continuación, indicando cómo se descompone la matriz 'M' de interés en sus respectivas matrices triangulares 'L' y 'L^T', mostrando además cómo se calcula cada elemento de las matrices resultantes (únicamente para valores reales):

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{32} \\ 0 & 0 & L_{33} \end{pmatrix}$$

$$= \begin{pmatrix} L_{11}^2 & & \text{(symmetric)} \\ L_{21}L_{11} & L_{21}^2 + L_{22}^2 & \\ L_{31}L_{11} & L_{31}L_{21} + L_{32}L_{22} & L_{31}^2 + L_{32}^2 + L_{33}^2 \end{pmatrix}, \quad \mathbf{L} = \begin{pmatrix} \sqrt{A_{11}} & 0 & 0 \\ A_{21}/L_{11} & \sqrt{A_{22} - L_{21}^2} & 0 \\ A_{31}/L_{11} & (A_{32} - L_{31}L_{21})/L_{22} & \sqrt{A_{33} - L_{31}^2 - L_{32}^2} \end{pmatrix}$$

$$L_{j,j} = (\pm)\sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2},$$

$$L_{i,j} = \frac{1}{L_{j,j}} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} \right) \quad \text{for } i > j.$$

Figura 39: Ecuaciones matemáticas del algoritmo de descomposición de matrices de Choleskyt

Esto permite calcular cada entrada de la matriz 'L', conociendo las entradas anteriores, hacia su izquierda y hacia arriba.

Esta herramienta matemática permite solucionar de manera eficiente problemas numéricos complejos, como es el caso de aplicación de método de mínimos cuadrados, optimización no lineal o simulaciones de Monte Carlo. (Wikipedia contributors, "Monte Carlo method", 2021)

Para el análisis de los resultados, se escogen valores para las dimensiones de las matrices a procesar, de forma que puedan fácilmente incrementarse bajo factores concretos y, de manera intuitiva, identificar el impacto en el rendimiento global, al aumentar el volumen de datos. El factor de incremento utilizado se basa en multiplicar por dos cada una de las dimensiones de la matriz (duplicar filas y columnas, o lo que es lo mismo, multiplicar por cuatro el número de elementos). Partiendo de un valor lo suficientemente grande para analizar el

tiempo de ejecución, se obtienen los siguientes valores a estudiar, según el tamaño de la diagonal principal: 64, 128, 256, 512, 1024 y 2048. Para la obtención de las matrices Hermíticas positivamente definidas, se desarrolla un script de Octave propio, de forma que las matrices de entrada sean las mismas, intentando estandarizar las pruebas de la mayor manera posible.

10.3.1 Aceleración sobre Alveo.

Para este apartado, que es el bloque fundamental del estudio, se basa el desarrollo en el proyecto guiado de aceleración de la aplicación original del algoritmo, implementada sobre C++. Esta implementación forma parte de las librerías aceleradas de Xilinx. Los resultados de la implementación inicial puramente sobre CPU se muestran en el apartado de C/C++ con el resto de implementaciones sobre dicho lenguaje. En este apartado se centra el estudio en el proceso de aceleración y en el resultado final implementado sobre Alveo.

El proceso llevado a cabo se puede dividir en cinco módulos o fases diferenciadas del proceso de aceleración.

Una primera, en la que únicamente se realiza la modificación del código original, destinado a ejecución en CPU, hacia el modelo de aceleración Host-FPGA. Se establece, por tanto, el flujo de transmisión de datos Host-FPGA, ejecución de kernel con patrón “load-compute-store” y transmisión de datos procesados FPGA-Host. Al realizarse el mismo proceso secuencial que en CPU, aunque en FPGA se realice de manera más eficiente, al introducir las latencias de transmisión y carga y almacenamiento de datos, se dispara el tiempo de ejecución. Esto se debe a que no se realiza ninguna optimización.

La segunda fase se centra en dos fundamentos principales relacionados con las directivas del procesador. A pesar de su introducción manual, no se realiza ninguna optimización directa sobre el rendimiento. Sin embargo, esto permite que, en etapas posteriores y en caso de que el compilador no realizara estas labores automáticamente, se ejecutaran dichas tareas de optimización. Las directivas utilizadas son:

- “PIPELINE”: permite modificar el flujo secuencial normal de un bucle (no se ejecuta la siguiente iteración hasta completar la actual) para lograr ejecutar de manera escalonada y solapada diferentes iteraciones según están disponibles los datos de entrada en cada ciclo de reloj. Esto está determinado por el valor de ‘II’, que

óptimamente es 1 (una iteración iniciada por ciclo de reloj). Si el compilador no es capaz de lograr el 'II' deseado, se remite un informe sobre las posibles causas para su solución.

De esta forma, lo que por defecto requiere 12 ciclos de reloj para leer, procesar y escribir los datos en un bucle determinado, en un supuesto escenario de 4 iteraciones, pasa a requerir únicamente 6 ciclos de reloj. Esto se muestra gráficamente en la Figura 40. Aplicando esta directiva a varios bucles, se puede lograr optimizar la aplicación para evitar tiempos no deseados de espera, procesando, por ejemplo, la siguiente columna de la matriz cuando ya se tengan disponibles los datos de entrada, sin tener que esperar a la finalización de la iteración anterior.

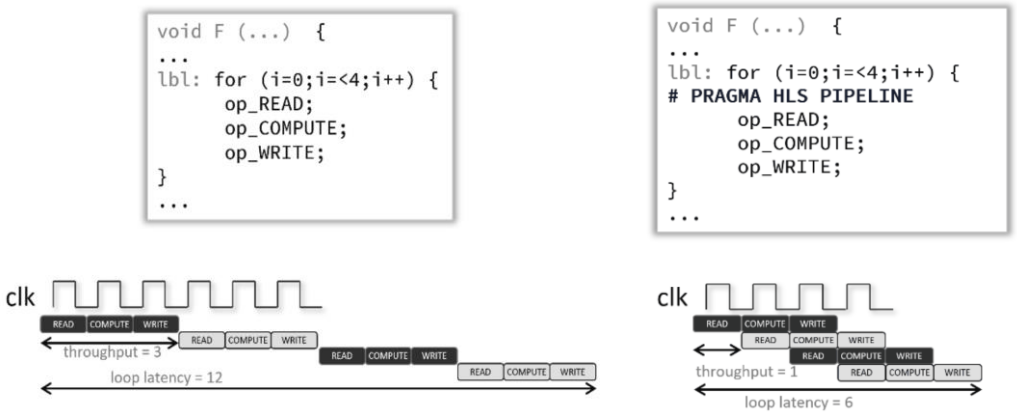


Figura 40: Ejemplo de implementación de directiva "PIPELINE" sobre bucle con ejecución secuencial

- "INTERFACE": permite especificar los adaptadores físicos para los puertos C del kernel, indicando cómo se vinculan con la plataforma, en la síntesis de interfaces en HLS. Mediante esta directiva se pueden definir propiedades como los protocolos para los puertos I/O o para sus interfaces, entre otras. Un aspecto importante es la posibilidad de especificación del modo ráfaga, de forma que se puedan leer varias muestras del mismo tipo, almacenadas de manera secuencial, de manera más eficiente (es necesario el uso de funciones concretas para la lectura de datos o el uso de la directiva "PIPELINE" si se trata de un bucle). Esto permite indicar en la compilación el mapeo de puertos, tanto para I/O como para bancos de memoria a utilizar.

En la tercera fase, se realiza una modificación temporal de la resolución de los tipos de datos utilizados, a fin de analizar el impacto en el rendimiento y en la utilización de recursos de la tarjeta Alveo, utilizando el mismo kernel. Para ello, se sustituyen los tipos de datos numéricos de punto flotante de 64 bits originales, por otros del mismo tipo, pero con una resolución de 32 bits.

En la siguiente fase, se realizan las optimizaciones finales para lograr una aplicación acelerada que mejore el rendimiento de la aplicación original. Se modifica el código, incluyendo las directivas de compilador necesarias, para lograr el paralelismo de tareas.

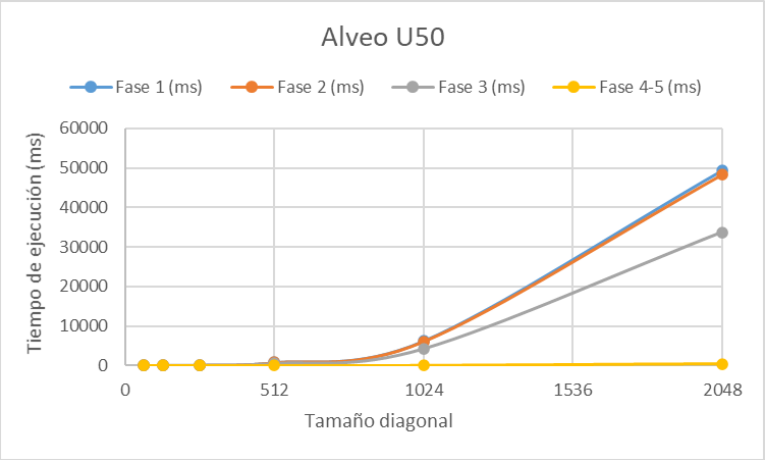
Para lograr la paralelización deseada, se hace uso de la directiva “DATAFLOW”, sobre el bucle encargado de ejecutar el algoritmo para cada columna, al que se aplica “unroll” con un factor ‘NCU’. Para garantizar que se aplica “DATAFLOW”, se divide el array total de entrada, en ‘NCU’ particiones, de forma que los datos estén disponibles para cada bucle en paralelo. De esta forma, se consigue ejecutar ‘NCU’ funciones del algoritmo de cálculo por columna, en paralelo, accediendo a la porción correspondiente de la matriz.

Para garantizar el funcionamiento adecuado y la aplicación correcta de la directiva DATAFLOW, se organiza y modifica el código, de forma que se cumplan las siguientes premisas:

- Único consumidor para único productor.
- No omisión de tareas (saltos en ejecución).
- No retroalimentación entre tareas (flujo solo hacia adelante).
- Ausencia de ejecución condicional de tareas.
- Supresión de bucles con múltiples condiciones de salida.

En la última fase, se realiza una modificación del código para generar ‘N’ unidades de computación o copias del kernel en paralelo, con el fin de analizar esta capacidad de paralelización y su flexibilidad, características del Hardware reconfigurable.

Tras el desarrollo del proyecto completo, se obtienen una serie de resultados referentes al tiempo de ejecución, para cada una de las fases comentadas previamente, mostrados en la Figura 41. El análisis de la computación en paralelo se muestra en el apartado de resultados.



Tam. Diagonal	64	128	256	512	1024	2048
Fase 1 (ms)	2,302	14,322	103,829	794,145	6220,437	49248,557
Fase 2 (ms)	2,230	14,461	101,616	777,450	6090,460	48221,130
Fase 3 (ms)	1,754	9,600	69,235	537,069	4234,104	33637,039
Fase 4-5 (ms)	0,468	1,030	2,659	11,510	63,068	398,372

Figura 41: Resultados de implementación de algoritmo Cholesky, mediante aceleración en Alveo U50

10.3.2. C/C++.

Para la ejecución sobre C/C++ se utilizan tres implementaciones diferentes:

- Librería de aceleración de Xilinx (potr): (Xilinx, Level II APIs: Matrix Decomposition (potrf), 2021)

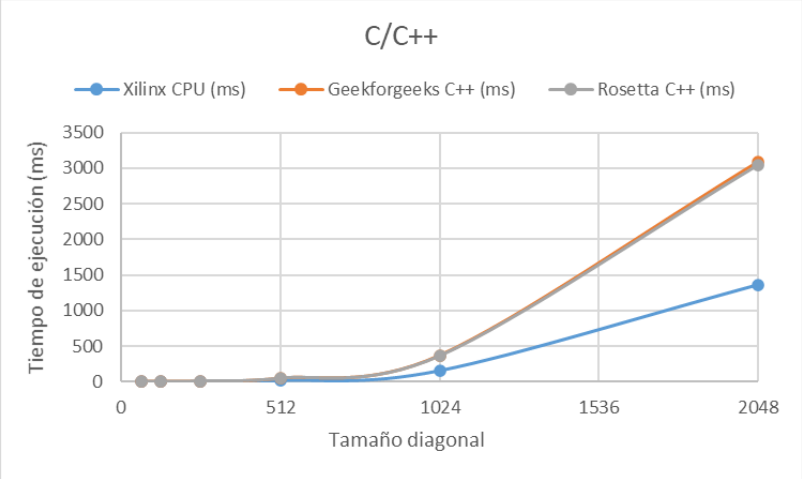
  En esta implementación no se realiza ninguna modificación y se ejecuta el programa con su propio conjunto de datos autogenerados.

- Geekforgeeks C++: (Geekforgeeks: Cholesky Decomposition C++, 2021)

  Se modifica la implementación para introducir medición de tiempos de ejecución y establecimiento del conjunto de datos de entrada.

- Rosetta Cholesky Decomposition C++: (Rosetta: Cholesky Decomposition C++, 2021)

  Se modifica la implementación para introducir medición de tiempos de ejecución y establecimiento del conjunto de datos de entrada.



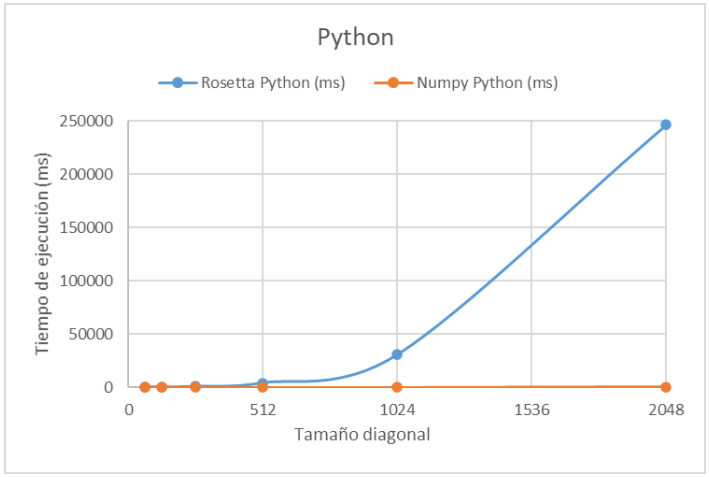
Tam. Diagonal	64	128	256	512	1024	2048
Xilinx CPU (ms)	0,031	0,242	2,091	18,684	157,294	1363,159
Geekforgeeks C++ (ms)	0,119	0,814	6,115	47,090	369,079	3089,930
Rosetta C++ (ms)	0,109	0,796	6,075	46,950	361,981	3049,770

Figura 42: Resultados de implementación de algoritmo Cholesky, mediante lenguaje C/C++

10.3.3. Python.

Para la ejecución sobre Python se utilizan dos implementaciones diferentes:

- Solución proporcionada por Rosetta (Rosetta: Cholesky Decomposition Python, 2021), realizando las modificaciones necesarias para medir tiempos de ejecución y establecer los datos de entrada generados mediante Octave.
- Librería Numpy, con función específica estándar para la descomposición de Cholesky. Se introducen marcación de tiempos y matrices generadas de entrada a procesar. (Numpy Cholesky - Python, 2021)

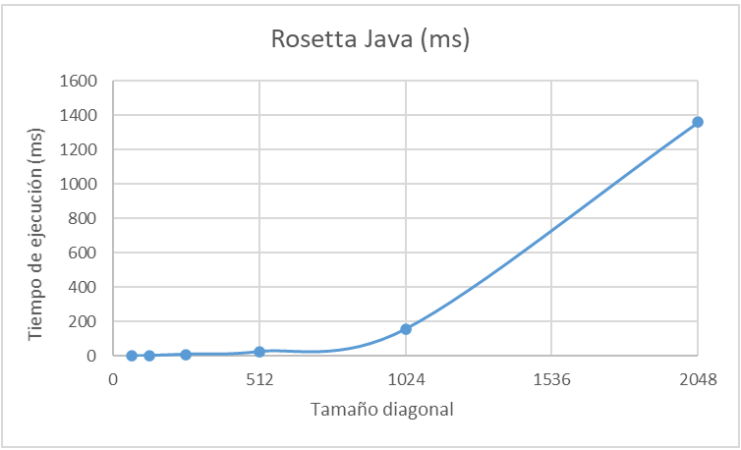


Tam. Diagonal	64	128	256	512	1024	2048
Rosetta Python (ms)	8,888	63,054	479,407	3889,256	30456,304	246397,238
Numpy Python (ms)	0,182	0,556	1,796	8,612	30,724	161,342

Figura 43: Resultados de implementación de algoritmo Cholesky, mediante lenguaje Python

10.3.4. Java.

Para la ejecución sobre Java se utiliza la solución proporcionada por Rosetta (Rosetta: Cholesky Decomposition Java, 2021), realizando las modificaciones necesarias para medir tiempos de ejecución y establecer los datos de entrada generados mediante Octave.

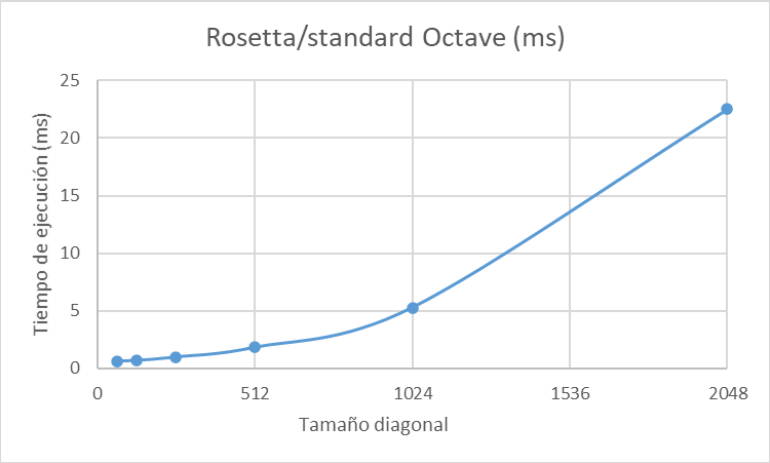


Tam. Diagonal	64	128	256	512	1024	2048
Rosetta Java (ms)	0,866	2,426	9,917	25,599	156,317	1358,313

Figura 44: Resultados de implementación de algoritmo Cholesky, mediante lenguaje Java

10.3.5. Octave

Para la ejecución sobre Octave se implementa la solución proporcionada por Rosetta (Rosetta: Cholesky Decomposition Octave, 2021), que simplemente se basa en el uso de la función estándar “chol” de Octave, realizando las modificaciones necesarias para medir tiempos de ejecución y establecer los datos de entrada generados mediante Octave.



Tam. Diagonal	64	128	256	512	1024	2048
Rosetta/standard Octave (ms)	0,619	0,672	0,967	1,810	5,273	22,516

Figura 45: Resultados de implementación de algoritmo Cholesky, mediante lenguaje Octave

11. Descripción de los resultados.

Tras la realización de todas las pruebas, para cada una de las implementaciones seleccionadas del algoritmo de Cholesky, se realiza una comparación de los resultados obtenidos, para extraer una serie de conclusiones relacionadas con el proceso de aceleración de aplicaciones y sus posibilidades.

Para el primer estudio, solamente se tienen en cuenta los tiempos de ejecución correspondientes al proceso de cómputo, dentro del paradigma "load-compute-store". Es decir, en esta fase se mide únicamente el tiempo requerido para procesar los datos de la matriz de entrada, obteniendo la matriz de salida, sin tener en cuenta los tiempos de carga y guardado de los datos, o de I/O. Este escenario se muestra en la Figura 46, donde el bloque de cómputo se corresponde con el algoritmo matemático de descomposición de Cholesky.

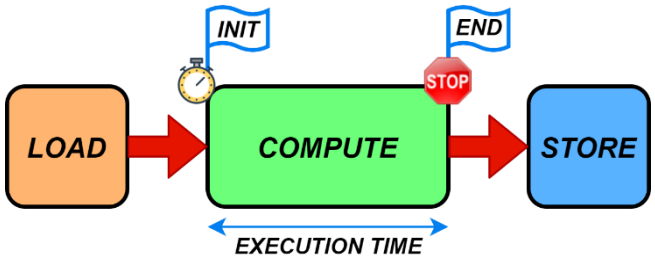
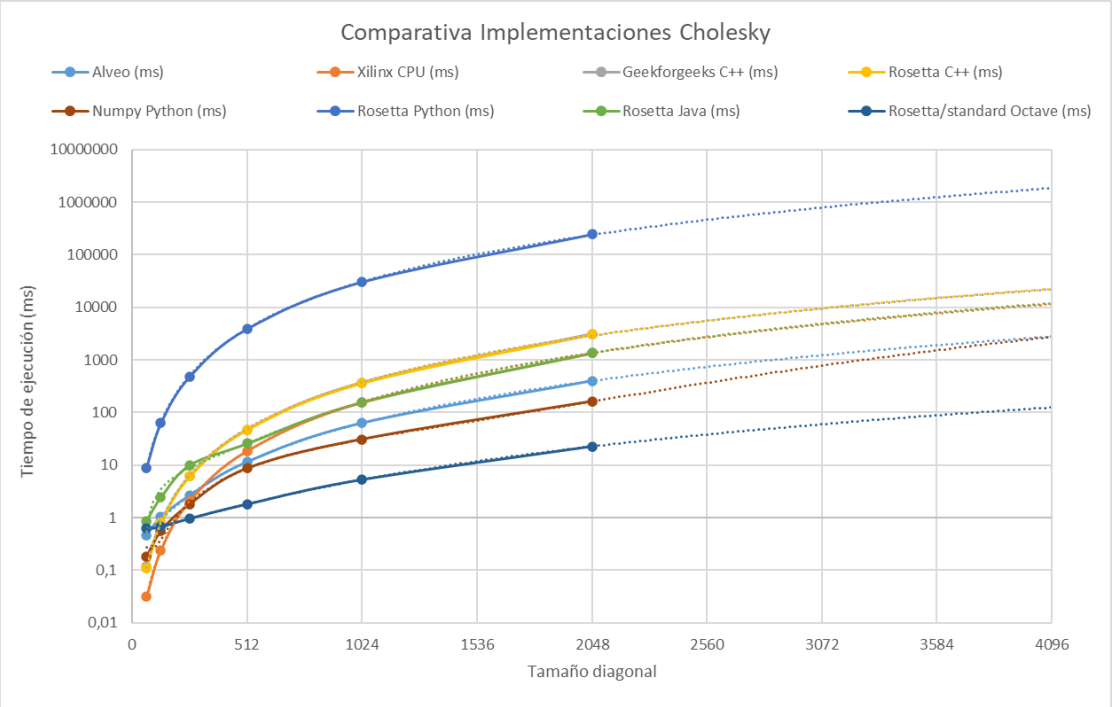


Figura 46: Escenario inicial para pruebas de medición de rendimiento de algoritmo Cholesky

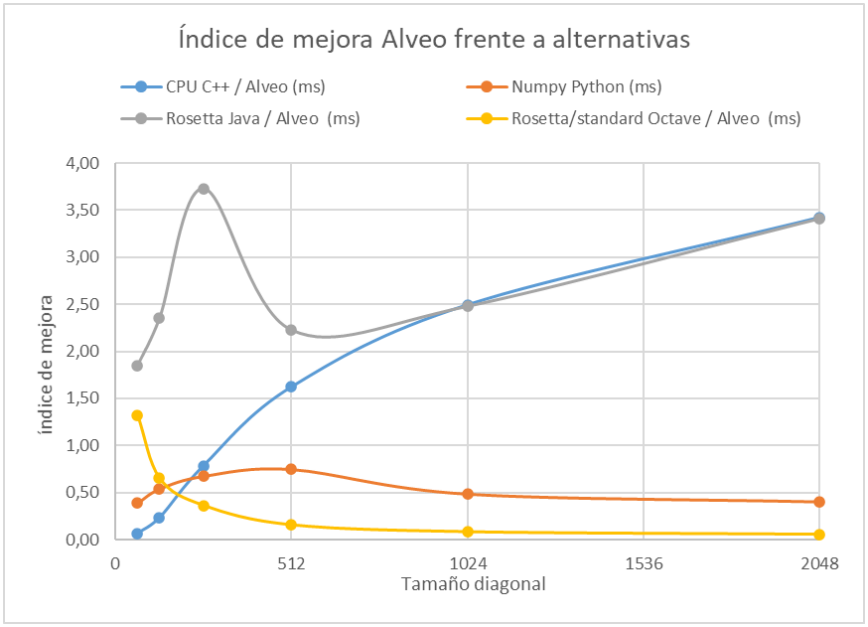
En la Figura 47 se muestra la evolución de los tiempos de ejecución, de cada una de las implementaciones, en función del tamaño de las matrices de entrada, según el escenario anterior. En este caso, lenguajes de alto nivel y de uso general (Pyhon, C++ y Java) presentan un rendimiento inferior al resto. A continuación, se encuentra el resultado de Alveo, viendo cómo se logra un mayor rendimiento, con una simple aceleración de una aplicación de lenguaje de mayor nivel (C++), obteniendo un resultado prometedor y perfectamente mejorable, si se realizan iteraciones de optimización sucesivas. Finalmente, se encuentran librerías desarrolladas para operaciones y algoritmos puramente matemáticos que, aunque se implementen sobre lenguaje de alto nivel, presentan un rendimiento realmente excelente, superando, en el caso de Octave, considerablemente al caso de aceleración con Alveo.



Tam. Diagonal	64	128	256	512	1024	2048
Alveo (ms)	0,468	1,030	2,659	11,510	63,068	398,372
Xilinx CPU (ms)	0,031	0,242	2,091	18,684	157,294	1363,159
Geekforgeeks C++ (ms)	0,119	0,814	6,115	47,090	369,079	3089,930
Rosetta C++ (ms)	0,109	0,796	6,075	46,950	361,981	3049,770
Numpy Python (ms)	0,182	0,556	1,796	8,612	30,724	161,342
Rosetta Python (ms)	8,888	63,054	479,407	3889,256	30456,304	246397,238
Rosetta Java (ms)	0,866	2,426	9,917	25,599	156,317	1358,313
Rosetta/standard Octave (ms)	0,619	0,672	0,967	1,810	5,273	22,516

Figura 47: Resumen y comparación de resultados de implementaciones de algoritmo Cholesky

Teniendo estos datos en cuenta, se puede extraer un índice de mejora para los casos analizados, ahora seleccionando los que mejor rendimiento presentan y se sitúan alrededor del caso Alveo. Este índice se muestra en la Figura 48, donde se puede observar que, los únicos casos en que no se consigue un mejor rendimiento mediante el proceso de aceleración, son aquellos en los que se utilizan las herramientas matemáticas NumPy y Octave.



Tam. Diagonal	64	128	256	512	1024	2048
CPU C++ / Alveo	0,07	0,23	0,79	1,62	2,49	3,42
Numpy Python	0,39	0,54	0,68	0,75	0,49	0,41
Rosetta Java / Alveo	1,85	2,36	3,73	2,22	2,48	3,41
Rosetta/standard Octave / Alveo	1,32	0,65	0,36	0,16	0,08	0,06

Figura 48: Comparativa de índices de mejora de resultados de implementaciones de algoritmo Cholesky

Sin embargo, esta situación no implica que la mejor opción sea el uso de estas herramientas matemáticas especializadas ya que, por una parte, en esta primera fase no se ha explotado una de las características fundamentales del Hardware reconfigurable, la capacidad de paralelización de unidades de cómputo, y, por otra, el kernel acelerado se encuentra en una primera etapa de aceleración (es funcional y permite aumentar el rendimiento original, pero todavía presenta mayor potencial de mejora).

Para comprobar que existe la posibilidad de aumentar el rendimiento global, sin modificar el kernel desarrollado, se sigue un proceso en el que se realizan diferentes pruebas, desplegando en paralelo, copias de dicho kernel. En este nuevo escenario, se establece como entrada de datos un número 'N' de matrices y, en cada prueba, se define un número 'K' de kernels desplegados sobre la FPGA. Para simplificar las pruebas, se establecen las siguientes condiciones:

- La evolución del tiempo de ejecución de las alternativas a Alveo, al ser todas de tipo secuencial, se supone que escala directamente proporcional a un factor 'X', siendo 'X'

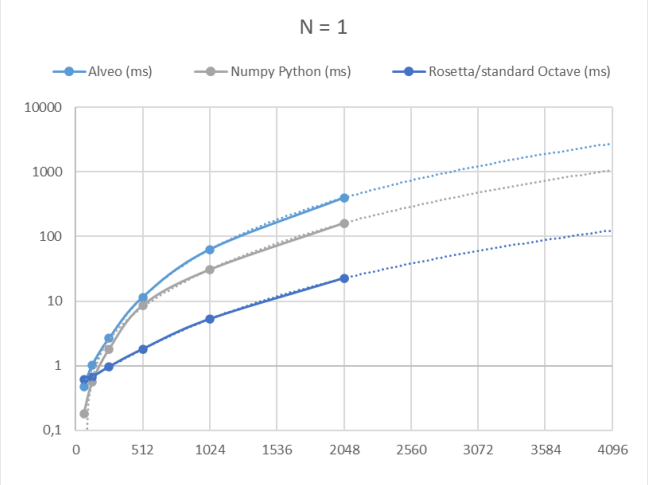
igual al número de matrices de entrada, ‘N’. Esto se debe a que se deben ejecutar ‘N’ algoritmos de Cholesky para procesar todas las matrices de entrada, uno detrás de otro.

- La evolución del tiempo de ejecución en Alveo, se supone que escala directamente proporcional a un factor ‘Y’, correspondiente al cociente del número de entradas ‘N’, entre el número de kernels en paralelo ‘K’. Es decir, cada ‘K’ matrices de entrada, se incrementa el tiempo de ejecución de la misma forma que el caso de las alternativas a Alveo. También se puede entender como que cada ‘K’ matrices de entrada, es necesario ejecutar los ‘K’ kernels en paralelo e iniciar una nueva iteración. Hay que tener en cuenta que, si se requiere de una nueva iteración (llega una nueva matriz y los ‘K’ kernels están ocupados), el tiempo de ejecución será prácticamente el mismo que si llegaran a la vez otras ‘K’-1 matrices.
- La limitación de la escalabilidad mediante este método de implementación de la misma copia del kernel ‘K’ veces, viene dada por la arquitectura y componentes lógicos de la tarjeta utilizada (al no existir proceso de optimización, se duplican los recursos requeridos por cada copia). Para el kernel de la última fase de optimización, se consigue desplegar un número máximo de 7 kernels en paralelo, siendo el valor límite utilizado para las pruebas. Esto se muestra en la tabla de la Figura 49, donde se aprecia que, entre la plataforma, con un 10.24% de las LUT, y las siete copias del kernel, con un uso del 80.03% de las LUT, no se dejan recursos suficientes para una nueva CU (11.4%).

Name	LUT	LUTAsMem	REG	BRAM	URAM	DSP
Platform	10.24%	2.31%	8.40%	9.52%	0.00%	0.07%
User Budget	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Used Resources	80.03%	12.61%	46.98%	11.51%	70.00%	22.83%
Unused Resources	19.97%	87.39%	53.02%	88.49%	30.00%	77.17%
cholesky_kernel (7)	80.03%	12.61%	46.98%	11.51%	70.00%	22.83%
cholesky_kernel_1	11.44%	1.80%	6.71%	1.64%	10.00%	3.26%
cholesky_kernel_2	11.42%	1.80%	6.71%	1.64%	10.00%	3.26%
cholesky_kernel_3	11.43%	1.80%	6.71%	1.64%	10.00%	3.26%
cholesky_kernel_4	11.43%	1.80%	6.71%	1.64%	10.00%	3.26%
cholesky_kernel_5	11.43%	1.80%	6.71%	1.64%	10.00%	3.26%
cholesky_kernel_6	11.44%	1.80%	6.71%	1.64%	10.00%	3.26%
cholesky_kernel_7	11.44%	1.80%	6.71%	1.64%	10.00%	3.26%

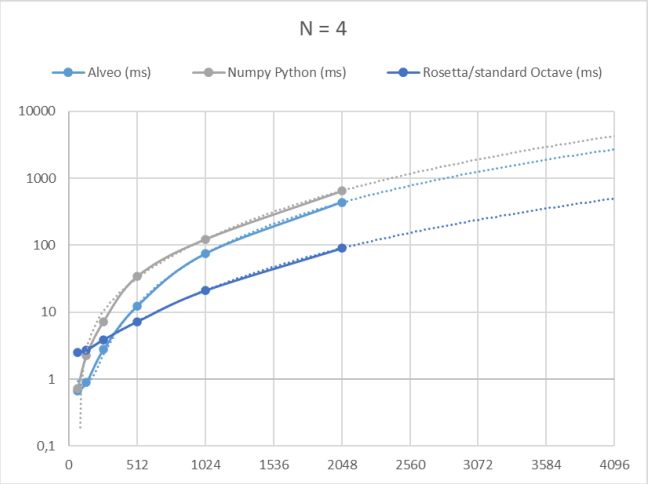
Figura 49: Resumen de utilización de recursos para implementación de 7 kernels en paralelo sobre Alveo U50

Los resultados de este escenario se pueden observar en las tablas y gráficas de las siguientes figuras:



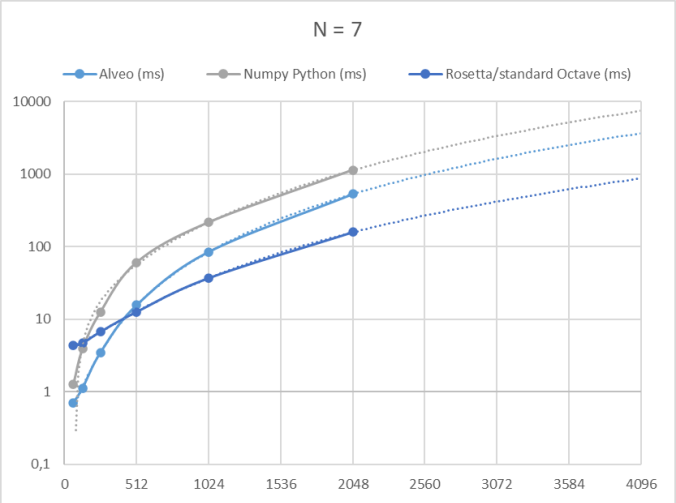
Tam. Diagonal	64	128	256	512	1024	2048
Alveo (ms)	0,468	1,030	2,659	11,510	63,068	398,372
Xilinx C++ (ms)	0,031	0,242	2,091	18,684	157,294	1363,159
Numpy Python (ms)	0,182	0,556	1,796	8,612	30,724	161,342
Rosetta/standard Octave (ms)	0,619	0,672	0,967	1,810	5,273	22,516

Figura 50: Comparativa de tiempos de ejecución de implementaciones de algoritmo Cholesky, para 1 matriz de entrada, desplegando 1 kernel sobre Alveo U50



Tam. Diagonal	64	128	256	512	1024	2048
Alveo (ms)	0,666	0,887	2,756	12,375	74,427	433,552
Xilinx C++ (ms)	0,124	0,968	8,364	74,736	629,176	5452,636
Numpy Python (ms)	0,729	2,226	7,184	34,448	122,896	645,368
Rosetta/standard Octave (ms)	2,476	2,687	3,867	7,240	21,091	90,064

Figura 51: Comparativa de tiempos de ejecución de implementaciones de algoritmo Cholesky, para 4 matrices de entrada, desplegando 4 kernels sobre Alveo U50



Tam. Diagonal	64	128	256	512	1024	2048
Alveo (ms)	0,704	1,116	3,470	15,584	84,425	529,040
Xilinx C++ (ms)	0,217	1,694	14,637	130,788	1101,058	9542,113
Numpy Python (ms)	1,275	3,895	12,572	60,283	215,068	1129,393
Rosetta/standard Octave (ms)	4,333	4,703	6,768	12,671	36,910	157,612

Figura 52: Comparativa de tiempos de ejecución de implementaciones de algoritmo Cholesky, para 7 matrices de entrada, desplegando 7 kernels sobre Alveo U50

Junto a los resultados numéricos, mediante la herramienta de Vitis Analyzer, es posible extraer también un gran número de parámetros de la aplicación y de la arquitectura Hardware, como el uso de recursos, las señales internas durante la ejecución o las líneas de tiempo de la aplicación, entre muchos otros.

Para ver de manera gráfica cómo se ha desplegado la arquitectura y por qué se logra esta mejora de rendimiento, se presentan las figuras Figura 53, Figura 54 y Figura 55, donde se observa la arquitectura desplegada y su comunicación con las interfaces PCIe y los bancos de memoria HBM, junto con la gráfica de la ejecución en el tiempo, donde se aprecia la ejecución simultánea y en paralelo, de tantos kernels como se ha definido en cada configuración.

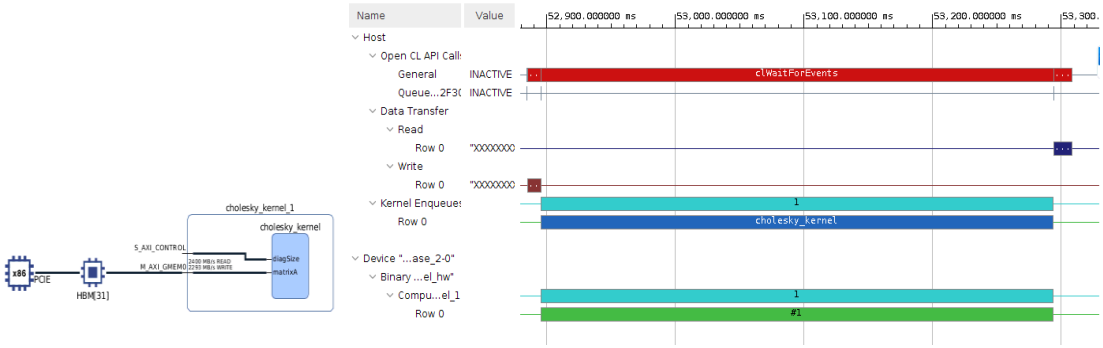


Figura 53: Resultado proporcionado por Vitis Analyzer para la ejecución de algoritmo Cholesky. con 1 kernel desplegado sobre AlveoU50

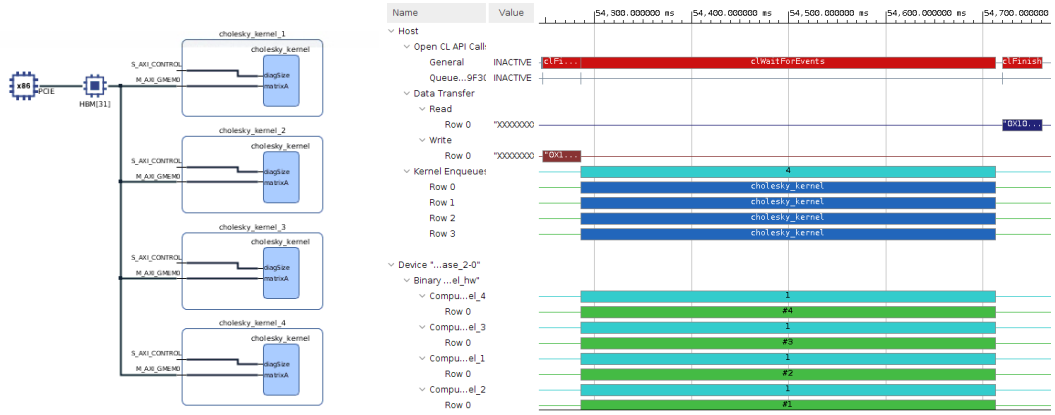


Figura 54: Resultado proporcionado por Vitis Analyzer para la ejecución de algoritmo Cholesky. con 4 kernels desplegados sobre AlveoU50

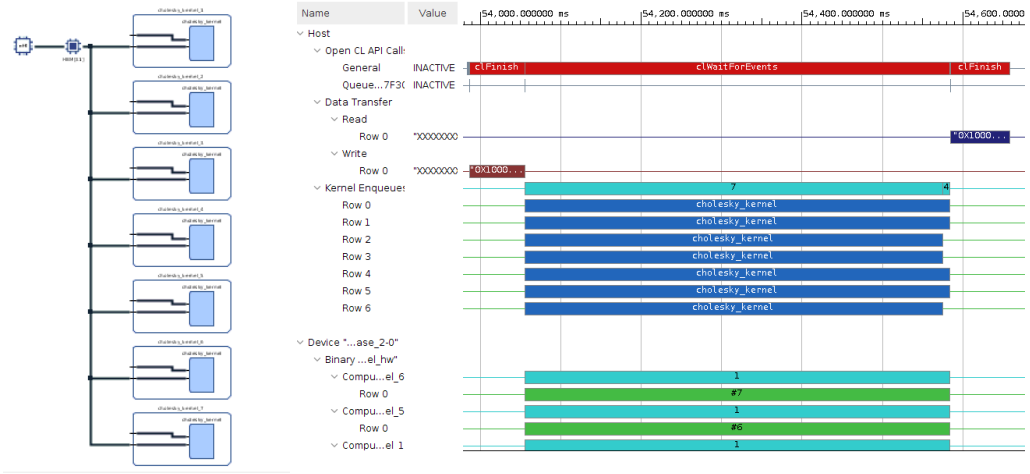
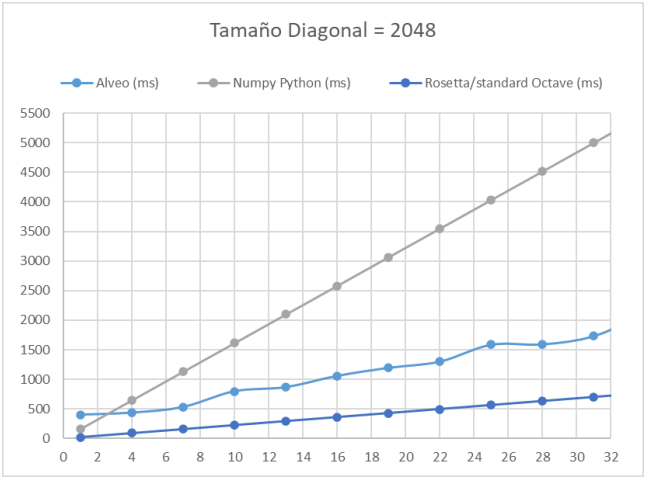


Figura 55: Resultado proporcionado por Vitis Analyzer para la ejecución de algoritmo Cholesky. con 7 kernels desplegados sobre AlveoU50

La evolución de los tiempos de ejecución, en función del número de matrices de entrada a procesar en paralelo, suponiendo que todas llegan simultáneamente, y para el caso de desplegar 7 kernels en Alveo U50, se muestra en la Figura 56.



Tam. Diagonal = 2048												
N	1	4	7	10	13	16	19	22	25	28	31	34
Alveo (ms)	398,372	433,552	529,040	796,744	867,104	1058,080	1195,116	1300,656	1587,120	1593,488	1734,208	2116,160
Numpy Python (ms)	161,342	645,368	1129,393	1613,419	2097,445	2581,470	3065,496	3549,522	4033,548	4517,573	5001,599	5485,625
Rosetta/standard Octave (ms)	22,516	90,064	157,612	225,160	292,708	360,256	427,804	495,352	562,900	630,448	697,996	765,544

Figura 56: Comparativa de la evolución de los tiempos de ejecución con el aumento del volumen de datos de entrada, teniendo en cuenta un despliegue de 7 kernels sobre Alveo U50

En este punto, se logra una mejora de rendimiento frente a las alternativas, sobre todo al aumentar el volumen de datos a procesar. Sin embargo, el caso de Octave sigue siendo más favorable en cuanto a rendimiento, por lo que es necesario analizar el escenario en un caso de implementación real. Para ello, se realizan la siguiente suposición: los procesos de carga de datos de entrada y guardado de datos procesados pasa a ser relevante, ya que, en una aplicación real, los datos deben ser introducidos y extraídos en el algoritmo para su procesamiento y posterior post-procesamiento.

Se analizan los tiempos solo para los casos más favorables siendo estos obtenidos de la siguiente manera:

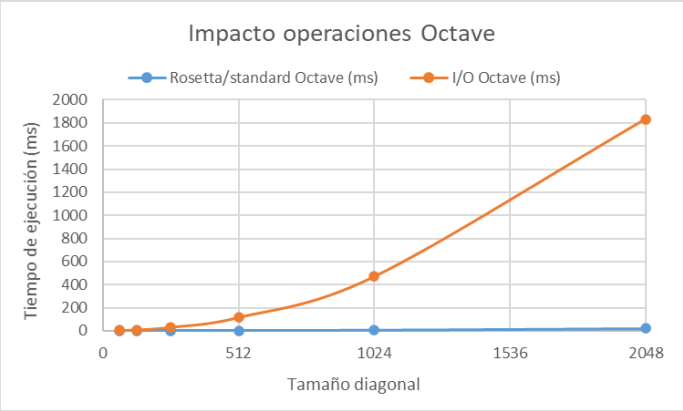
- Alveo: mediante herramienta Vitis Analyzer tras la ejecución en Hardware de la aplicación completa, en el caso más exigente, con 7 matrices de entrada, de tamaño 2048x2048, utilizando 7 kernels en paralelo. Estos valores, según se observa en la Figura 57, toman un valor de 68.546 ms para la función “load”, y de 73.84 ms para a función “store”, suponiendo un total de 142,39 ms.

Host Transfer

Context:Number of Devices	Transfer Type	Nu of	Transfer Rate (MB/s)	Avg Bandwidth Utilization (%)	Avg Size (KB)	Total Time (ms)	Avg Time (ms)
context0:1	READ	1	3180.967	33.135	234881.000	73.840	73.840
context0:1	WRITE	1	3426.604	35.694	234881.000	68.546	68.546

Figura 57: Datos y estadísticas de transferencias entre Host y FPGA proporcionados por Vitis Analyzer para el caso de Cho9lesky con 7 kernels en paralelo

- Octave: solo se tiene en cuenta la función “store”, mediante el almacenamiento en disco de los datos procesados mediante Cholesky. En la Figura 58 se muestran los tiempos de ejecución con y sin I/O para el mismo procesamiento. Para el caso de 1 matriz de entrada, de tamaño 2048x2048, supone un total de unos 1810 ms, lo que supone un impacto considerable.



Tam. Diagonal	64	128	256	512	1024	2048
Rosetta/standard Octave (ms)	0,619	0,672	0,967	1,810	5,273	22,516
I/O Octave (ms)	3,833	9,475	31,410	116,516	469,893	1832,098

Figura 58: Comparativa del tiempo de ejecución de Cholesky implementado sobre Octave, en función de las operaciones I/O

Esta diferencia se explica, entre otros aspectos, por dos causas principales:

- Aunque Octave trabaje de manera eficiente realizando operaciones matemáticas puras, presenta peor rendimiento en el resto de funcionalidades y presenta una mala escalabilidad. Es por ello que esta herramienta es utilizada generalmente para prototipos y procesamientos matemáticos exigentes, que serán portados a otros lenguajes más eficientes sobre implementaciones reales, como C/C++ o Python, si se trata de lenguajes de alto nivel y uso general.

- A pesar de que las interfaces PCIe pueden suponer el cuello de botella, presentan una gran capacidad de transmisión y, bajo una optimización adecuada, pueden solventar las latencias producidas en accesos a disco convencionales. Además, como Alveo trabaja con memorias DDR o HBM de gran ancho de banda, se consigue que los accesos a memoria sean mucho más eficientes.

De esta forma, se pueden definir los dos escenarios claramente diferenciados, uno ideal, en el que se mide únicamente el rendimiento del algoritmo o funcionalidad concreta que se implementa, y otro más fiel a la realidad, donde se tiene en cuenta la totalidad de las operaciones necesarias para la implementación sobre un entorno real. Gráficamente, se muestran ambos paradigmas en las figuras Figura 59 y Figura 60, respectivamente.

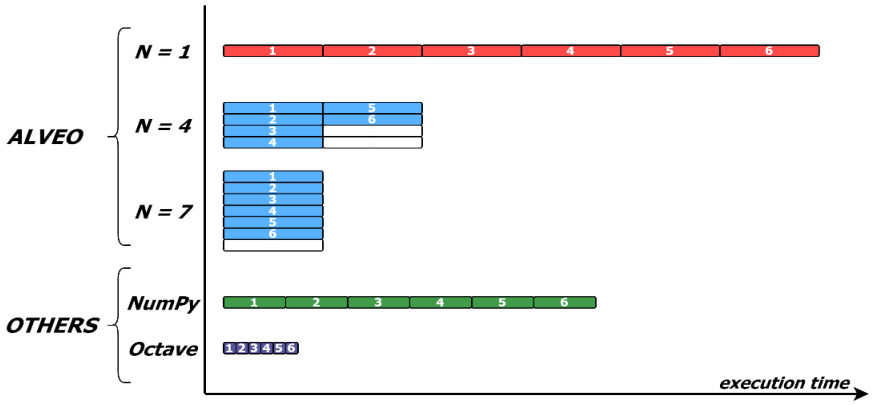


Figura 59: Diagrama conceptual de ejecución, para el algoritmo de Cholesky, con un volumen de entrada de 6 matrices, para el caso ideal sin operaciones I/O

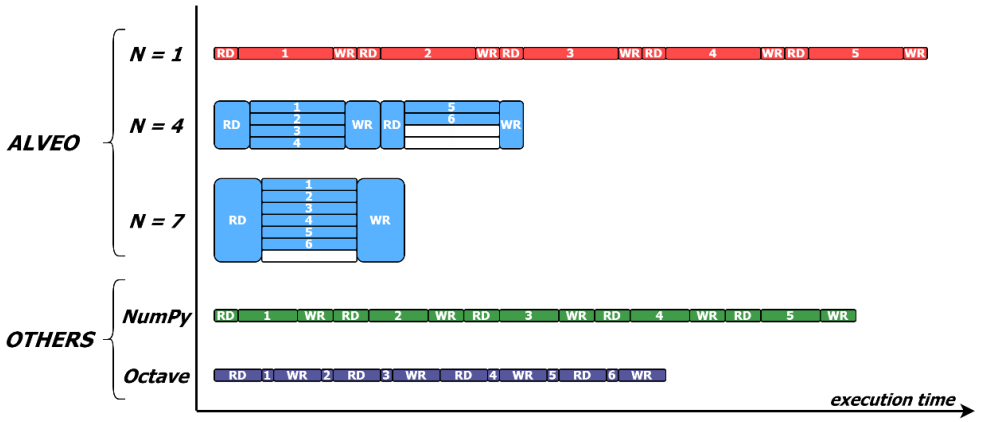


Figura 60: Diagrama conceptual de ejecución, para el algoritmo de Cholesky, con un volumen de entrada de 6 matrices, para el caso real con operaciones I/O

Como se pude observar, es necesario analizar ambos rendimientos, por separado y conjuntamente, ya que condicionan el rendimiento global de las aplicaciones y pueden determinar si las alternativas y elecciones son adecuadas o deben ser replanteadas.

Para profundizar un poco más en el desarrollo, se plantea un último escenario en el que se implemente un mayor número de kernels en paralelo, modificando la resolución del tipo de datos utilizados. Se modifican los datos de tipo numérico con punto flotante de 64 bits (double), por unos del mismo tipo de 32 bits (float). De esta forma, no solo se consigue aumentar el número de kernels desplegables sobre la tarjeta a Alveo, de 7 a 10, sino que, además, cada iteración del kernel se ejecuta en un menor tiempo. Los resultados obtenidos se muestran en las siguientes figuras, observando cómo, gracias a la ejecución simultánea de 3 algoritmos adicionales y, con un ligero menor tiempo de ejecución por cada ejecución, se consigue un índice de mejora de casi un 34%, cuando el volumen de datos crece.

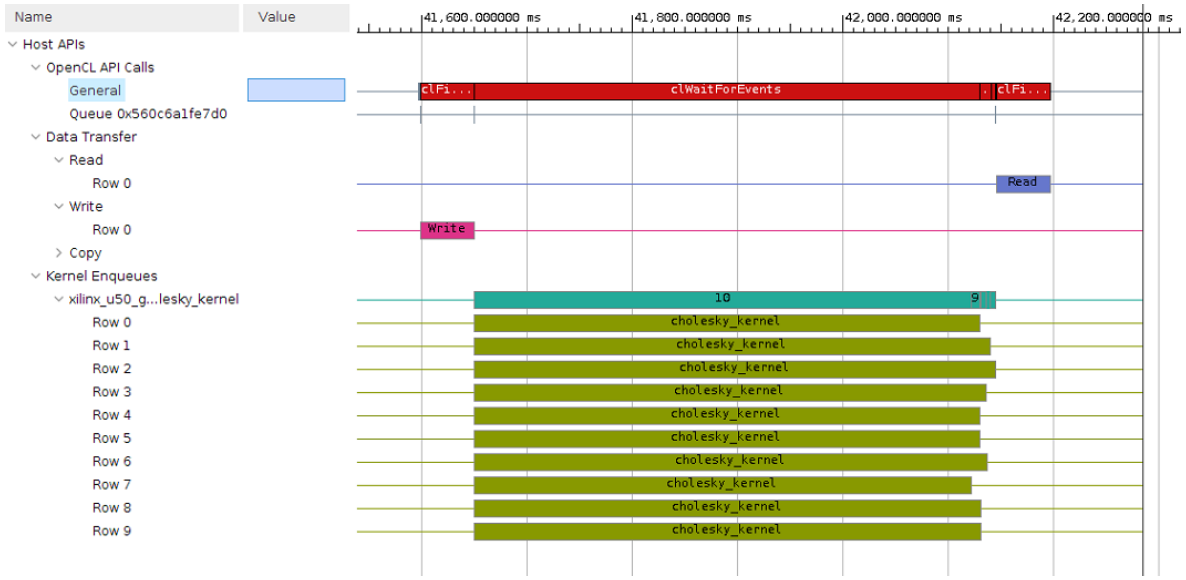
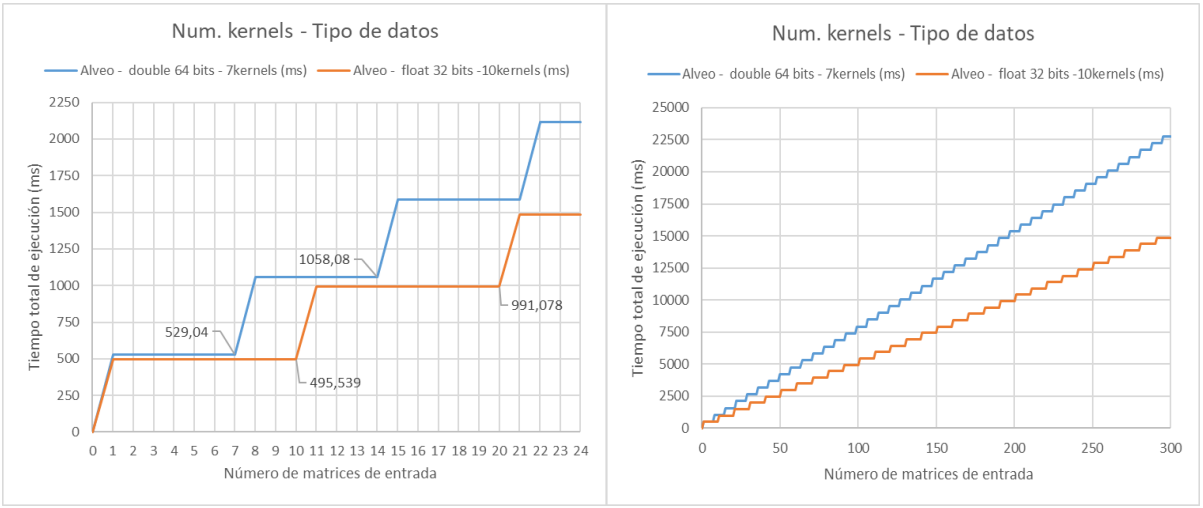


Figura 61: Resultado proporcionado por Vitis Analyzer para la ejecución de algoritmo Cholesky. con 10 kernels desplegados sobre AlveoU50, al disminuir resolución de tipo de datos



Tam. Diagonal = 2048									
N	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**
Alveo - double 64 bits - 7kernels (ms)	529,040	529,040	529,040	1058,080	1587,120	2645,200	5290,400	10051,760	19574,480
Alveo - float 32 bits -10kernels (ms)	495,539	495,539	495,539	495,539	991,078	1982,156	3468,773	6442,007	12884,014

Figura 62: Comparativa de evolución de tiempos de ejecución para alternativas de 7 y 10 kernels desplegados en paralelo sobre Alveo U50

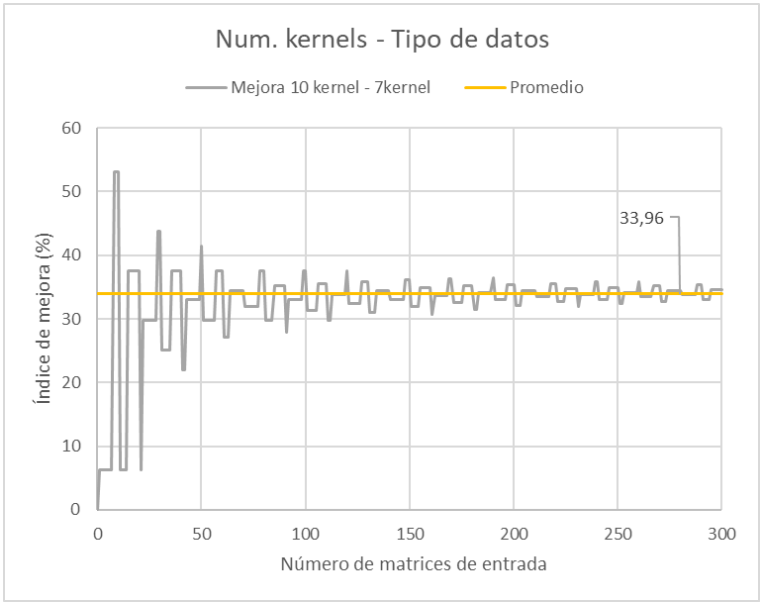


Figura 63: Índice de mejora para tiempos de ejecución de alternativa de 10 kernels en paralelo, sobre la versión de 7, desplegados sobre Alveo U50

Aunque en este caso se consigue un mejor rendimiento, es necesario analizar el impacto de la reducción de la resolución de los datos ya que, en función de cada aplicación, los decimales pueden ser relevantes para obtener unos resultados adecuados.

Tras el proceso de aceleración y adecuación de la aplicación y del propio Hardware, se consigue mejorar el rendimiento por encima del resto de alternativas, cuando se tiene en cuenta una implementación real. Además, este es el punto de partida ya que solamente se ha realizado una iteración del proceso de aceleración. En este punto, se deben replantear los objetivos y, si se requiere, volver a realizar el proceso de aceleración para lograr un mejor rendimiento.

Algunas de las posibles soluciones, incluidas como líneas futuras son:

- Optimización del kernel para una lectura/escritura más eficiente, haciendo uso de la totalidad de los bancos de memoria, dando un total de 8.192 GB de almacenamiento accesible con un ancho de banda de 316 GB/s, en el caso de Alveo U50.
- Adecuación de la resolución de los tipos de datos utilizados para permitir un mayor número de kernels en paralelo, adaptando a su vez la complejidad de las operaciones para ajustar el tiempo de cada ejecución.
- Fragmentación del flujo de datos para transmisiones eficientes a ráfagas, a cada kernel, en el momento adecuado.
- Solapamiento de transmisión de datos entre Host-FPGA y FPGA-Host, con el propio procesamiento en ambos elementos, logrando paralelizar las tareas a realizar y evitando tiempos de espera innecesarios.

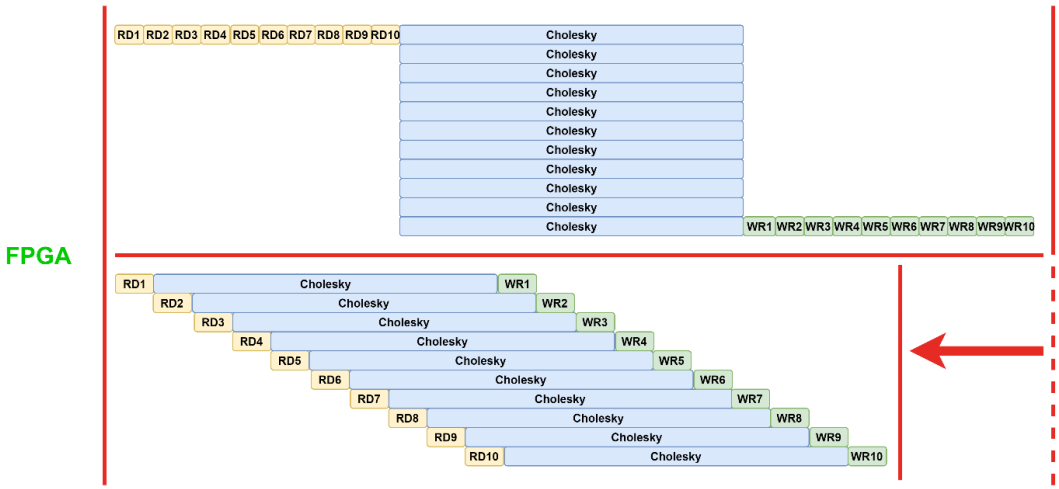


Figura 64: Representación de la posible mejora mediante la aceleración de la aplicación en el estado actual, logrando un solapamiento y paralelización adecuados

12. Conclusiones.

Tras el desarrollo completo del proyecto, gracias al estudio de la metodología propuesta y el análisis de las tecnologías implicadas, llevadas a la práctica mediante los proyectos específicos desarrollados, se concluye que se han logrado los objetivos propuestos, determinando, además, que el alcance planteado inicialmente ha sido el adecuado. Se ha conseguido estudiar en gran detalle los conceptos teóricos y prácticos fundamentales relacionados con el entorno de desarrollo para aplicaciones aceleradas, Xilinx Vitis, mediante Hardware de aceleración Xilinx Alveo. Además, se ha profundizado en la metodología y procesos de implementación, estableciendo una base y toma de contacto con la tecnología, de forma que, en proyectos futuros, sea posible sacar un mayor potencial a la aceleración de aplicaciones. Junto a esta síntesis conceptual, se ha conseguido desplegar un sistema de desarrollo completo, capaz de proporcionar todas las funcionalidades de aceleración planteadas. Finalmente, ha sido posible desarrollar varias iteraciones simples de aceleración sobre implementaciones prácticas reales.

Una vez llevados a cabo todos los procesos de investigación, análisis, desarrollo e implementación, se pueden extraer las siguientes conclusiones:

- Es fundamental analizar los posibles cuellos de botella, para adecuar el proceso de desarrollo. Al utilizar el paradigma Host-Kernel, es necesario tener en cuenta la latencia introducida por la conexión PCIe, siendo de manera general el cuello de botella que limita el potencial de aceleración. Su análisis debe ser el punto de partida crítico, para toda etapa conceptual de aceleración.
- Es importante analizar la naturaleza de los algoritmos y funcionalidades para determinar las capacidades de aceleración. Se logra un mayor potencial de mejora de rendimiento, cuanto mayor sea la complejidad computacional y el volumen de datos, gracias a la paralelización del procesamiento, con un hardware más eficiente, para operaciones específicas.
- Existen grandes limitaciones en emulación para el desarrollo de aplicaciones reales y es necesario disponer de una tarjeta física para conocer el funcionamiento real de la aplicación. Esto se debe a que el tiempo de emulación de cada prueba se dispara,

haciendo imposible depurar el código para grandes volúmenes de datos, y a que algunos parámetros se consideran ideales, como las transferencias de datos mediante PCIe.

- Deben establecerse objetivos de aceleración realistas, teniendo en cuenta las limitaciones proporcionadas por la tecnología.
- Gracias a la naturaleza del Hardware reprogramable, se logra una mayor flexibilidad y adaptabilidad a la evolución de las aplicaciones, pudiendo adaptar el propio Hardware a los requerimientos de cada etapa de desarrollo, con una serie de parámetros de compilación y reestructurando, en menor o mayor medida, según la aplicación, la arquitectura de la misma.

Para concluir y como síntesis general, se puede determinar que la aceleración de aplicaciones mediante Hardware de aceleración es un proceso iterativo costoso que requiere un gran esfuerzo y tiempo de desarrollo. Sin embargo, implementado de manera adecuada, permite mejorar el rendimiento de cada bloque funcional, así como de la aplicación global en sí, mediante la paralelización y el solapamiento de transmisiones de datos y procesamiento de los mismos, pudiendo implementar en paralelo varias unidades de procesamiento o kernels. Esto se traduce en un mejor rendimiento, utilizando el mismo Hardware para diferentes implementaciones, sin la limitación debida a la espera entre generaciones de Hardware, permitiendo un desarrollo de aplicaciones Software de calidad y eficientes, desde las etapas iniciales, sobre un Hardware configurado específicamente para ello. Esto permite unificar los desarrollos intrínsecamente relacionados, entre Software y Hardware, reduciendo la disonancia entre ambos elementos, de forma que ambos se beneficien conjuntamente.

Referencias

ADLINK. (2021). *Embedded Hardware for Processing AI at the Edge: GPU, VPU, FPGA, and ASIC Explained*. Obtenido de https://blog.adlinktech.com/en/2021/02/19/embedded-hardware-processing-ai-edge-gpu-vpu-fpga-asic/amp/

Alastruey, J., Briz, J. L., Ibáñez, P., & Viñals, V. (2006). Software Demand, Hardware Supply. *IEEE MICRO*, 72-82.

AWS, A. (2014). *Caso práctico de Baylor*. Obtenido de https://aws.amazon.com/es/solutions/case-studies/baylor/

AWS, A. (2018). *Toyota Research Institute accelerates safe automated driving with deep learning at a global scale on AWS*. Obtenido de https://aws.amazon.com/es/blogs/machine-learning/toyota-research-institute-accelerates-safe-automated-driving-with-deep-learning-at-a-global-scale-on-aws/

AWS, A. (2021). *AstraZeneca's Genomics Data Processing Solution Runs 51 Billion Tests in 1 Day on AWS*. Obtenido de https://aws.amazon.com/es/solutions/case-studies/astrazeneca/

AWS, A. (2021). *Capital One en AWS*. Obtenido de https://aws.amazon.com/es/solutions/case-studies/capital-one/

AWS, A. (2021). *FLYING WHALES Runs CFD on AWS to Quickly Launch Environmentally Friendly Cargo Transport Airships*. Obtenido de https://aws.amazon.com/es/solutions/case-studies/flying-whales/

AWS, A. (2021). *Informática de alto rendimiento*. Obtenido de https://aws.amazon.com/es/hpc/

AWS, A. (2021). *Instancias F1 de Amazon EC2*. Obtenido de https://aws.amazon.com/es/ec2/instance-types/f1/

Bode, A., & Dal Cin, M. (1993). *Parallel Computer Architectures: Theory, Hardware, Software, Applications.* Springer-Verlag Berlin Heidelberg GmbH.

Cope, B., Y.K. Cheung, P., Luk, W., & Howes, L. (2010). Performance Comparison of Graphics Processors to Reconfigurable Logic: A Case Study. *IEEE Transactions on Computers, 59*(4), 433-448.

Geekforgeeks: Cholesky Decomposition C++. (2021). Obtenido de https://www.geeksforgeeks.org/cholesky-decomposition-matrix-decomposition/

Gómez, B. (2020). *Qué son las FPGA y ASIC y en qué se diferencian de una CPU o GPU*. Obtenido de https://www.profesionalreview.com/2020/12/18/fpga-asic-cpu-gpu/amp/

insideHPC. (2021). *What is high performance computing?* Obtenido de https://insidehpc.com/hpc-basic-training/what-is-hpc/

K. Blum, E., & V. Aho, A. (2011). *Computer Science: The Hardware, Software and Heart of It.* Springer.

Morgan, G. (2020). *High-performance computing: The need for speed*. Obtenido de https://www.conocophillips.com/spiritnow/story/high-performance-computing-the-need-for-speed/

NetApp. (2021). *What is high performance computing?* Obtenido de https://www.netapp.com/data-storage/high-performance-computing/what-is-hpc/

Nielsom, M., & Rumsey, R. (2017). *SPI Configuration and Flash Programming in UltraScale FPGAs (XAPP1233).* Obtenido de https://www.xilinx.com/support/documentation/application_notes/xapp1233-spi-config-ultrascale.pdf

Nimbix. (2021). *Xilinx Alveo™ Accelerator Cards: Accelerate your workflows with Xilinx Alveo Accelerator Cards in the Cloud .* Obtenido de https://www.nimbix.net/alveo

Numpy Cholesky - Python. (2021). Obtenido de https://numpy.org/doc/stable/reference/generated/numpy.linalg.cholesky.html

NVIDIA. (2009). *Mythbusters Demo GPU versus CPU*. Obtenido de https://www.youtube.com/watch?v=-P28LKWTzrI&ab_channel=NVIDIA

Pauwels, K., Tomasi, M., Díaz, J., Ros, E., & M. Van Hulle, M. (2012). A Comparison of FPGA and GPU for Real-Time Phase-Based Optical Flow, Stereo, and Local Image Features. *IEEE TRANSACTIONS ON COMPUTERS, 61*(7), 999-1012.

Research, C. I. (2020). *Reference Samples for Human Genome Sequencing: Standardization to Ensure Accuracy*. Obtenido de https://www.coriell.org/1/NIGMS/Collections/NIST-

Reference-Materials?gclid=CjwKCAjw3MSHBhB3EiwAxcaEu4-DslTf_0KnSpVwCHYtVSluuz1CZl1F-j_PjpW4b6c_1CFgm8jbpxoCRqMQAvD_BwE

Rosetta: Cholesky Decomposition C++. (2021). Obtenido de https://rosettacode.org/wiki/Cholesky_decomposition#C.2B.2B

Rosetta: Cholesky Decomposition Java. (2021). Obtenido de https://rosettacode.org/wiki/Cholesky_decomposition#Java

Rosetta: Cholesky Decomposition Octave. (2021). Obtenido de https://rosettacode.org/wiki/Cholesky_decomposition#MATLAB_.2F_Octave

Rosetta: Cholesky Decomposition Python. (2021). Obtenido de https://rosettacode.org/wiki/Cholesky_decomposition#Python

Society, I. C. (2008). The Big Bang: 25 Years of Software History. *IEEE Software*, 6-14.

Thompson, A. (1998). *Hardware Evolution: Automatic Design of Electronic Circuits in.* Springer.

White, G. (2015). Hardware, Software, Humans: Truth, Fiction and Abstraction. *HISTORY AND PHILOSOPHY OF LOGIC, 0*(0), 1-25.

Wikipedia contributors, "Cholesky decomposition". (2021). Obtenido de https://en.wikipedia.org/wiki/Cholesky_decomposition

Wikipedia contributors, "Monte Carlo method". (2021). Obtenido de https://en.wikipedia.org/wiki/Monte_Carlo_method

Wirth, N. (2008). A Brief History of Software Engineering. *IEEE Annals of the History of Computing*, 32-39.

Xilinx. (2012). *Large FPGA Methodology Guide: Including Stacked Silicon Interconnect (SSI) Technology (UG872).* Obtenido de https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug872_largefpga.pdf

Xilinx. (2018). *Accelerating DNNs with Xilinx Alveo Accelerator Cards.* Obtenido de https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf

Xilinx. (2018). *Maxeler Technologies Inc. Partner Information*. Obtenido de https://www.xilinx.com/alliance/memberlocator/1-y77big.html#products

Xilinx. (2019). *Adaptable Accelerator Cards for Data Center Workloads: Alveo U50 Product Brief.* Obtenido de https://www.xilinx.com/publications/product-briefs/alveo-u50-product-brief-v2.pdf

Xilinx. (2019). *Alveo U50 Data Center Accelerator Card: User Guide (UG1371).* Obtenido de https://www.mouser.com/pdfDocs/u50-UserGuide.pdf

Xilinx. (2019). *Get Moving with Alveo: Acceleration Basics*. Obtenido de https://developer.xilinx.com/en/articles/acceleration-basics.html

Xilinx. (2020). *Alveo Card Out-of-Band Management Specification for Server BMC: User Guide (UG1363).* Obtenido de https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/ug1363-alveo-out-of-band-user-guide.pdf

Xilinx. (2020). *Alveo U50 Data Center Accelerator Card Data Sheet (DS956).* Obtenido de https://www.xilinx.com/support/documentation/data_sheets/ds965-u50.pdf

Xilinx. (2020). *Alveo U50 Data Center Accelerator Card Installation Guide (UG1370).* Obtenido de https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/1_7/ug1370-u50-installation.pdf

Xilinx. (2021). *Alveo Data Center Accelerator Card Platforms: User Guide (UG1120).* Obtenido de https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/ug1120-alveo-platforms.pdf

Xilinx. (2021). *Alveo Data Center Accelerator Card Test: User Guide (UG1361).* Obtenido de https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/5_0/ug1361-alveo-card-validation-test-solution.pdf

Xilinx. (2021). Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment.

Xilinx. (2021). Developer Program Secure Site: Using Xilinx Alveo Cards to Accelerate Dynamic Workloads.

Xilinx. (2021). *Downloads: Vitis Core Development Kit*. Obtenido de https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis.html

Xilinx. (2021). *Level II APIs: Matrix Decomposition (potrf)*. Obtenido de https://xilinx.github.io/Vitis_Libraries/solver/2020.1/guide_L2/L2_api.html#potrf

Xilinx. (2021). *Vitis In-Depth Tutorials (Repositorio GitHub)*. Obtenido de https://github.com/Xilinx/Vitis-Tutorials

Xilinx. (2021). *Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393).* Obtenido de https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/kme1569523964461.html

Xilinx. (2021). *Xilinx Qualified Servers Catalog*. Obtenido de https://www.xilinx.com/products/boards-and-kits/alveo/qualified-servers.html

Anexos.

A. Ampliación del estado del arte

A.1. Tecnologías implicadas en el estudio: CPU, GPU, FPGA y ASIC

Resumen conceptual en Apartado 5.1 en la página 33.

Los sistemas de tecnologías de la información y la comunicación evolucionan rápidamente, tanto a nivel empresarial como de mercado, de forma que la tendencia de crecimiento se ha disparado en los últimos años. Además, se espera que siga creciendo en los próximos años, con la irrupción de la inteligencia artificial y el volumen cada vez mayor de datos que se mueven a nivel mundial. (K. Blum & V. Aho, 2011) (Thompson, 1998)

Cuando se crearon los primeros sistemas embebidos, sus arquitecturas de sistema no estaban pensadas para los volúmenes de datos y requerimientos computacionales actuales, en campos como la inteligencia artificial o Internet of Things (IoT), entre muchos otros, actuales y por llegar. Ahora que ha cambiado el paradigma tecnológico, estos sistemas deben adaptarse y dar soporte a los nuevos requisitos computacionales.

Los principales requerimientos de los dispositivos para dar soporte a las aplicaciones y servicios en demanda actualmente, se centran en torno a términos de latencia, confiabilidad, movilidad, seguridad, eficiencia energética y costes de transmisión de datos. Para alcanzar estas demandas actuales, el Hardware debe evolucionar, desde sistemas basados en control y reglas estáticas, a entornos basados en datos y funcionalidades dinámicas.

En función del caso de uso, es necesario definir un núcleo de procesamiento adecuado, que condiciona en gran medida, todos los requerimientos a cumplir. Existen diferentes posibilidades de arquitecturas para sistemas en cuanto a núcleos de procesamiento, entre los que destacan: (ADLINK, 2021)

- CPU: unidad central de procesamiento (Central Processing Unit, en inglés). Se trata de una unidad de procesamiento de uso general que cuenta con una serie de núcleos, capaces de realizar ciertas tareas, de manera secuencial (una al mismo tiempo, en cada núcleo). Permiten ejecutar tareas complejas y facilita la gestión de recursos de los sistemas. Permiten ejecutar todo tipo de operaciones, aunque implica que no lo hacen de la forma más eficiente posible. Destinado a ejecución de tareas complejas y dinámicas de manera secuencial.

- GPU: unidad de procesamiento gráfico (Graphics Processing Unit, en inglés). Consiste en una unidad de procesamiento capaz de realizar numerosas tareas en paralelo, al contar con un mayor número de núcleos. Permiten desempeñar tareas con un mayor rendimiento, aunque, por normal general, con un mayor consumo y requiriendo un mayor tamaño físico. Son elementos de propósito general, aunque en este caso, destinados a una ejecución paralela de las tareas en cuestión, proporcionando mayor capacidad computacional por unidad de tiempo para aquellas tareas susceptibles de paralelización.
- FPGA: matriz de puertas lógicas programable en campo (Field-programmable gate array, en inglés). Se trata de sistemas que cuentan con una serie de puertas lógicas reconfigurables, que permite diseñar arquitecturas personalizadas para diferentes aplicaciones, con un menor coste energético que las opciones CPU y GPU. Permiten aportar flexibilidad y dinamismo a diseños Software y Hardware.
- ASIC: circuitos integrados para aplicaciones específicas (Application-specific integrated circuits, en inglés). Son circuitos lógicos personalizados, diseñados mediante librerías de circuitos de los propios fabricantes, que ofrecen ventajas como el bajo consumo, la alta velocidad y el pequeño tamaño. Se trata de diseños que requieren gran tiempo para su diseño y consumo, resultando en un proceso más costoso, por lo que se recomiendan para aquellos casos en que los productos se produzcan a gran escala. Existen diferentes tipos, en función de la aplicación, como las unidades de procesamiento de visión (VPU), unidades de procesamiento de tensores (TPU) o unidades de computación neuronal (NCU).

Un resumen de las características y principales ventajas e inconvenientes de cada uno de ellos se muestra en la tabla Tabla 10.

Tipo	Consumo	Descripción	Ventajas	Inconvenientes
CPU	Alto	Flexible, unidad de procesamiento de uso general	Tareas e instrucciones complejas. Gestión de sistemas	Posibles cuellos de botella en acceso a memoria. Pocos núcleos de procesamiento
GPU	Alto	Gran número de núcleos para paralelización de tareas	Procesamiento de alto rendimiento. Gran capacidad de paralelización	Elevado consumo energético. Gran tamaño
FPGA	Medio	Puertas lógicas configurables	Flexibilidad. Programabilidad in-field (por usuario, tras fabricación sin requerir desmontaje ni intervención de fabricante)	Elevado consumo energético. Complejidad de programación
ASIC	Bajo	Lógica personalizada diseñada mediante librerías	Rapidez y bajo consumo. Pequeño tamaño	Funcionalidad fija definida. Diseño personalizado caro

Tabla 10: Características principales CPU, GPU, FPGA y ASIC (ADLINK, 2021)

Estos cuatro elementos se pueden dividir en dos grandes grupos, en función de la filosofía de procesamiento. Esta filosofía se muestra en la Figura 65 y permite diferenciar los siguientes agrupamientos:

- Flujos de trabajo orientados a tareas, tanto a nivel secuencial como en paralelo: CPU y GPU.
- Flujos de trabajo orientados a datos, focalizando en la paralelización del procesamiento: FPGA y ASIC.

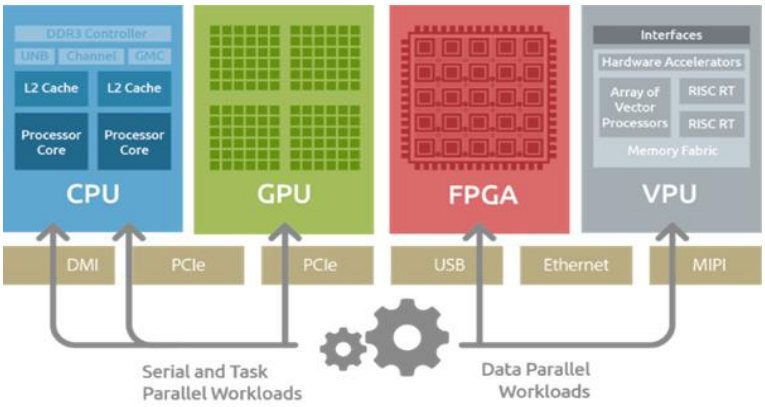


Figura 65: Diagrama general de arquitecturas CPU, GPU, FPGA y VPU (ADLINK, 2021)

Teniendo en cuenta las diferentes posibilidades de interés, se descartan los dispositivos ASIC, por no encajar en el contexto del proyecto, debido a su propósito tan específico. En cuanto al resto de tecnologías, es importante comprender su propósito principal y el funcionamiento de

su arquitectura interna. Al tratarse de aceleración de aplicaciones, se focaliza en la posibilidad de paralelización de tareas, por lo que las tecnologías CPU, GPU y FPGA son perfectamente válidas.

La diferencia principal, en cuanto a arquitectura, se muestra en la Figura 66. En ella, se aprecia lo siguiente, para cada tipo:

- CPU: cuentan con núcleos de procesamiento que se encargan de ejecutar conjuntos predefinidos de instrucciones y con rutas de datos con anchos fijos. La arquitectura está definida desde la fabricación y cada núcleo cuenta con una serie de elementos o bloques internos individuales predefinidos que se pueden comunicar o no, con los del resto de núcleos y sus componentes.
- GPU: cuentan con un número de núcleos de procesamiento mucho mayor que en el caso de la CPU, contando con el resto de elementos de la arquitectura para uso compartido entre estos núcleos. En este caso, la potencia computacional por núcleo es menor, por lo que es importante analizar el caso de uso. Arquitectura predefinida en fabricación.
- FPGA: cuenta con puertas lógicas programables, lo que significa que por defecto no cuenta con ninguna unidad de procesamiento. Son elementos personalizables a nivel electrónico, permitiendo el diseño de arquitecturas flexibles y adaptables sobre el mismo dispositivo Hardware. Esto significa que permiten lograr una optimización del rendimiento para cada aplicación.

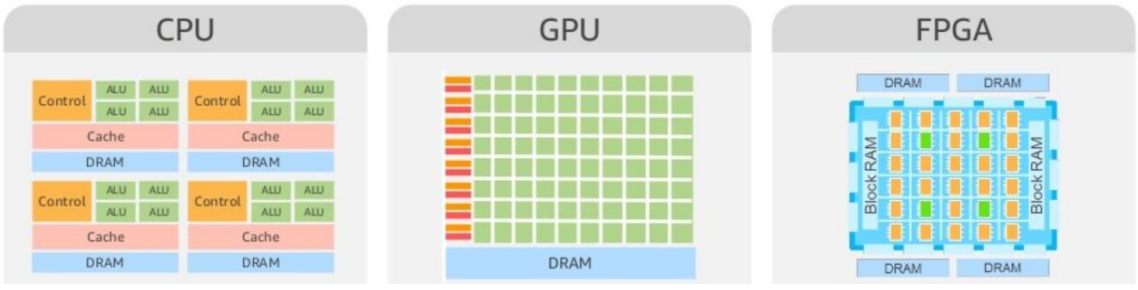


Figura 66: Componentes arquitecturas CPU, GPU y FPGA (Gómez, 2020)

La principal diferencia de diseño es que, tanto para CPU como para GPU, se diseña Software que se adecúe a un Hardware predefinido, mientras en que el caso de una FPGA, se desarrolla el Hardware que se adecúe al Software concreto de la aplicación.

Un aspecto importante, es la paralelización que permite cada arquitectura, permitiendo definir dos tipos de procesamiento computacional:

- Computación temporal: funcionamiento secuencial, de forma que en cada instante se ejecuta una única operación por unidad de procesamiento. Una instrucción tras otra, con cada ciclo de reloj.
- Computación espacial: funcionamiento paralelo mediante definición de caminos de datos que permiten ejecución simultánea de varias tareas, en un mismo ciclo de reloj.

Estos procesos, de forma gráfica, se pueden ver en la Figura 67. En ella se observa como en el caso de una CPU, el número de datos a procesar en cada momento está predefinido por el conjunto de instrucciones (ejecutándose una a una), mientras que, en el caso de una FPGA, se puede configurar la cantidad de datos a procesar en paralelo para permitir su procesamiento simultáneo (en este caso, permitiendo tres tareas simultáneamente sobre el flujo de datos de entrada). Además de poder definir los caminos de datos de manera flexible, se pueden optimizar las tareas a ejecutar para ser implementadas sobre un Hardware específico, configurado en la FPGA, con mayor potencia computacional. (NVIDIA, 2009)

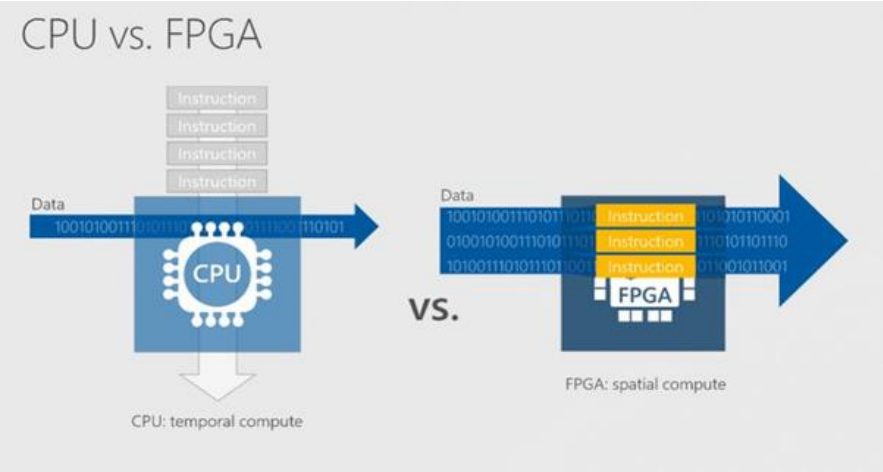


Figura 67: Procesamiento y secuencia de instrucciones para CPU y FPGA (Gómez, 2020)

En el ámbito de la aceleración del Software, estas son las principales ventajas que proporcionan las FPGA sobre las CPU y GPU y en torno a la que se centra el estudio de esta memoria.

A.2. Software Xilinx: Vitis Unified Software Platform

Resumen conceptual en Apartado 5.2 en la página 37.

(Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021) (Xilinx, Developer Program Secure Site: Using Xilinx Alveo Cards to Accelerate Dynamic Workloads, 2021)

La abstracción general del modelo de la plataforma propuesta por Xilinx, para la aceleración de aplicaciones Software, mediante aceleración Hardware, se observa en la Figura 68. En este diagrama se observa claramente la división del programa de la aplicación en dos partes diferentes, una, a la izquierda, correspondiente al programa de host y otra, a la derecha, correspondiente al kernel del Hardware acelerado. Es necesaria una comunicación entre ambos bloques, siendo la tecnología utilizada PCIe.

En cuanto al programa host, escrito en C/C++, haciendo uso de abstracciones mediante APIs (como OpenCL) es ejecutado sobre un procesador de host (como un procesador x86 o ARM). Por otra parte, el kernel de Hardware acelerado, es ejecutado en el interior de la región de lógica programable (PL) del dispositivo Xilinx correspondiente.

En la CPU del host, la aplicación personalizada, interactúa con los dispositivos FPGA mediante el uso de APIs OpenCL. La API correspondiente realiza llamadas, gestionadas por XRT, para realizar transacciones de datos entre el programa host y los aceleradores Hardware. Estos datos intercambiados, incluyen tanto transferencias de control, como de datos, a través de buses PCIe (en el caso de tarjetas de aceleración) o mediante interfaces AXI (en caso de plataformas embebidas). Mientras la información de control es intercambiada entre localizaciones de memoria específicas en el Hardware, la memoria global (Global Memory) es utilizada para las transferencias de datos entre el programa host y el kernel. La memoria global es accesible tanto por el procesador host como por el acelerador Hardware, mientras que la memoria del host, es únicamente accesible por este último.

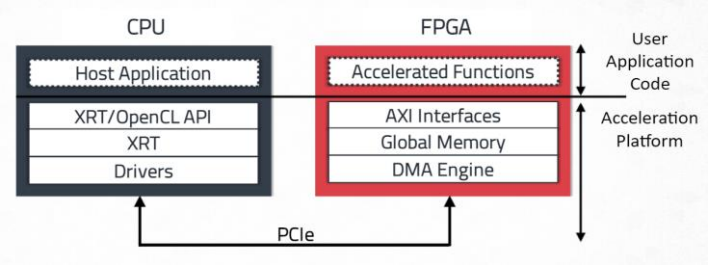


Figura 68: Modelo aceleración hardware propuesto por Xilinx (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

Por norma general, en una aplicación típica, la secuencia de transferencias sigue el siguiente orden:

1. El host transfiere información para ser procesada por el kernel, desde la memoria del host, hacia la memoria global.
2. El kernel, secuencialmente, realiza las operaciones y los procedimientos necesarios para procesar la información obtenida del host y obtener el resultado deseado. La información procesada se vuelve a almacenar en la memoria global.
3. Cuando el Kernel completa el procesamiento, se transfiere el resultado, desde la memoria global, hacia la memoria del host.

Este proceso se puede observar en la siguiente figura:

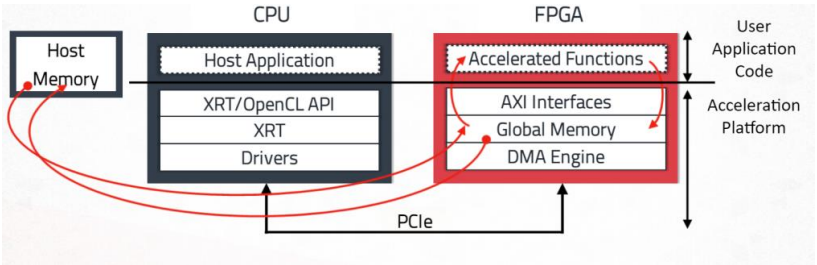


Figura 69: Secuencia de pasos a ejecutar en modelo de aceleración propuesto por Xilinx (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

Es importante tener en cuenta que estas transferencias introducen latencias no deseadas en el flujo del proceso, lo cual puede ser significativamente costoso para el rendimiento resultante de la aplicación. Para lograr una aceleración de un sistema real, una premisa clave a tener en cuenta es que la mejora y los beneficios obtenidos por los kernels de aceleración Hardware, superen la latencia añadida por las transferencias de datos.

A más bajo nivel, los drivers se encargan de gestionar las transferencias PCIe entre los dos dispositivos. La propia FPGA cuenta con la lógica necesaria precargada, para soportar el mecanismo DMA, para permitir una transferencia directa de los datos a la memoria DDR. De esta forma, los kernels de aceleración personalizados, permiten leer los datos, procesarlos y devolverlos a la memoria DDR, mediante el uso de interfaces estándar AXI4.

Para lograr una aceleración adecuada, es necesario comprender la naturaleza de la aplicación y, mediante las herramientas proporcionadas por la plataforma de aceleración, adecuar las funciones a cada dispositivo comentado previamente, para permitir ejecutar las operaciones más costosas, sobre el Hardware de aceleración.

En la Figura 70 se muestra un ejemplo de aceleración de una aplicación mediante Hardware de aceleración.

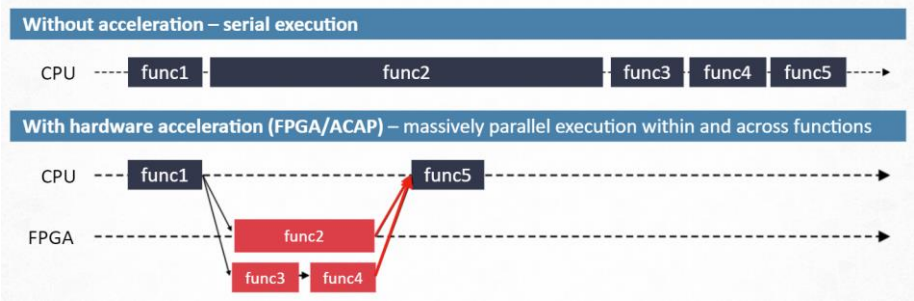


Figura 70: Ejemplo de aceleración de aplicación con ejecución secuencial (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

En este caso, se cuenta con 5 funciones diferentes, algunas de ellas requiriendo el resultado de una función anterior y alguna independiente de otras. Comprendiendo la naturaleza de la aplicación, se puede optimizar el caso inicial, sin aceleración, con ejecución secuencial de todas las funciones en la CPU, una a una, para conseguir no solo acelerar la ejecución de las funciones más costosas (como es el caso de la Función 2), sino también para permitir la ejecución en paralelo de funciones que no requieren la finalización de otras funciones. Esto se traduce en la reducción del tiempo de ejecución global de la aplicación y se consigue siguiendo dos vías:

1. Aceleración de funciones y procesos mediante el Hardware de aceleración
2. Paralelización del flujo de la aplicación en función de naturaleza y algoritmos de la aplicación

De esta forma, la función 1 se ejecuta en la CPU del host y tras obtener el resultado, se ejecutan en paralelo las funciones 2 y 3. Esta última, entrega su resultado a la función 4. Estas funciones son ejecutadas en el kernel de aceleración y los resultados obtenidos, se entregan de nuevo al host, que los procesa mediante la función 5.

Esta metodología permite lograr una aceleración masiva de ejecución paralela dentro y entre funciones, permitiendo mejoras significativas respecto al caso tradicional, siempre en función de la naturaleza y complejidad de la aplicación. La premisa principal es ejecutar en el kernel de la FPGA las tareas más intensas en cuanto a computación, las tareas profundamente segmentadas (pipelined) y las tareas con operaciones paralelas masivas. La CPU del host, ejecuta el resto de tareas.

Mediante los dispositivos Xilinx de aceleración se pueden lograr grandes ventajas sobre el caso tradicional de aceleración CPU/GPU, permitiendo implementar cualquier función ejecutable en un procesador, proporcionando un mejor rendimiento, con un menor consumo energético.

Para la construcción de los programas ejecutables tanto en el host como en el kernel del Hardware, se sigue un proceso estándar de compilación y linkado, para ambos programas. Estos programas son el programa del host y el binario de la FPGA. El esquema de este proceso se muestra en la Figura 71.

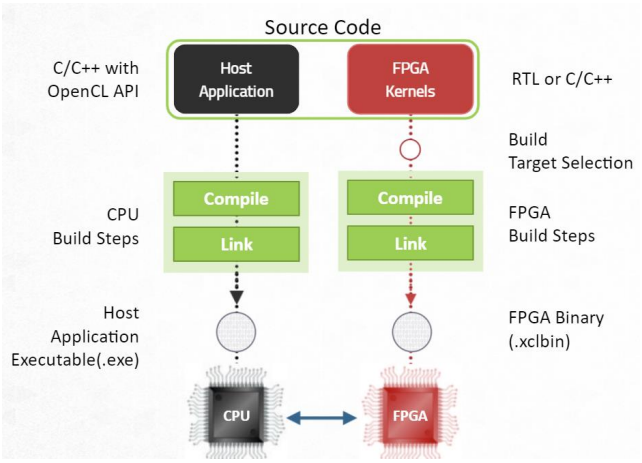


Figura 71: Proceso de generación de ejecutables host y kernel (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

En el caso de la aplicación del host, escrita en C/C++ mediante el uso de llamadas a API OpenCL, se construye usando el compilador de GNU C++ (g++) o el compilador cruzado GNU C++ ARM para dispositivos basados en MPSoC (basado en la colección de compilación GNU, GCC). Cada fichero fuente es compilado como un fichero objeto (.o) y linkado con la librería compartida Xilinx Runtime (XRT) para crear el ejecutable que corre sobre la CPU del host.

En cuanto al código del kernel, escrito en C, C++ o RTL, se construye mediante el compilador Vitis (v++) para generar el fichero objeto de Xilinx (.xo) y linka dichos ficheros con el fichero binario de FPGA (.xclbin).

Para permitir este proceso, Xilinx proporciona una plataforma completa de desarrollo, compuesta por diferentes soluciones Software, librerías, drivers e interfaces. Recibe el nombre de Plataforma Unificada de Software Vitis (Vitis Unified Software Platform en inglés) y es una herramienta que combina todos los aspectos de desarrollo de Xilinx en un único entorno unificado. Soporta tanto el flujo de desarrollo para Software embebido de Vitis (para usuarios que buscan tecnologías de próxima generación), como el flujo de desarrollo para aplicaciones aceleradas de Vitis (para usuarios que buscan la aceleración de Software usando lo último en aceleración de Software basado en FPGAs de Xilinx). Además, permite tanto desarrollo para equipos finales o para implementaciones on-premise o cloud. Un esquema de este entorno y los niveles de abstracción relacionados, se puede observar en la Figura 72, donde aparecen los paradigmas de aplicación, desarrollo y despliegue.

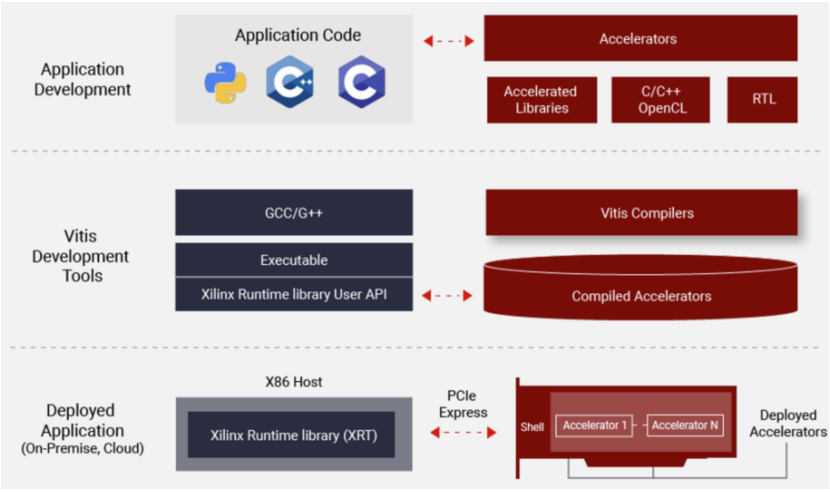


Figura 72: Niveles de abstracción de entorno de desarrollo unificado Xilinx Vitis (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

El flujo de desarrollo de aplicaciones de aceleración Vitis proporciona un entorno completo tanto para el desarrollo, como para el despliegue, de las aplicaciones aceleradas basadas en FPGAs, mediante el uso de lenguajes de programación estándar, tanto para los componentes Software como Hardware. En el caso concreto de desarrollo para tarjetas basadas en FPGAs, permite construir aplicaciones Software usando APIs (como APIs OpenCL) para ejecutar kernels Hardware sobre tarjetas de aceleración, como las tarjetas de aceleración de Xilinx Alveo Data Center.

Este kit de desarrollo proporciona, además, depuración y creación de perfiles de las aplicaciones desarrolladas. El analizador de Vitis es una utilidad que permite ver y analizar informes generados durante la compilación, construcción, depuración y ejecución de las aplicaciones. Un ejemplo de esta funcionalidad, se observa en la Figura 73, donde se aprecian elementos como llamadas a APIs, transferencias de datos o puestas en cola de kernels, entre otros.

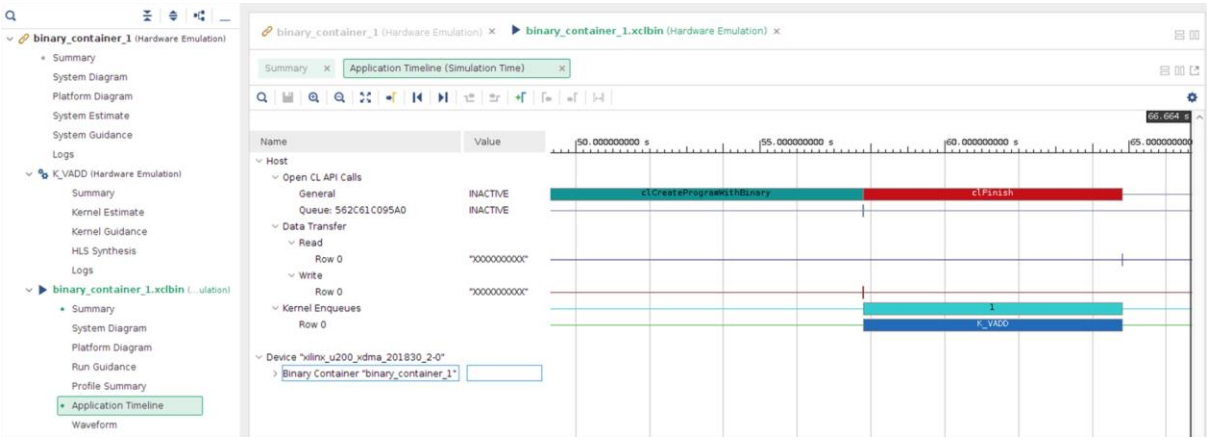


Figura 73: Ejemplo de resultado de Vitis Analyser (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

La plataforma Vitis soporta tanto funcionalidad GUI, como interacción mediante línea de comandos, como se aprecia en la Figura 74.

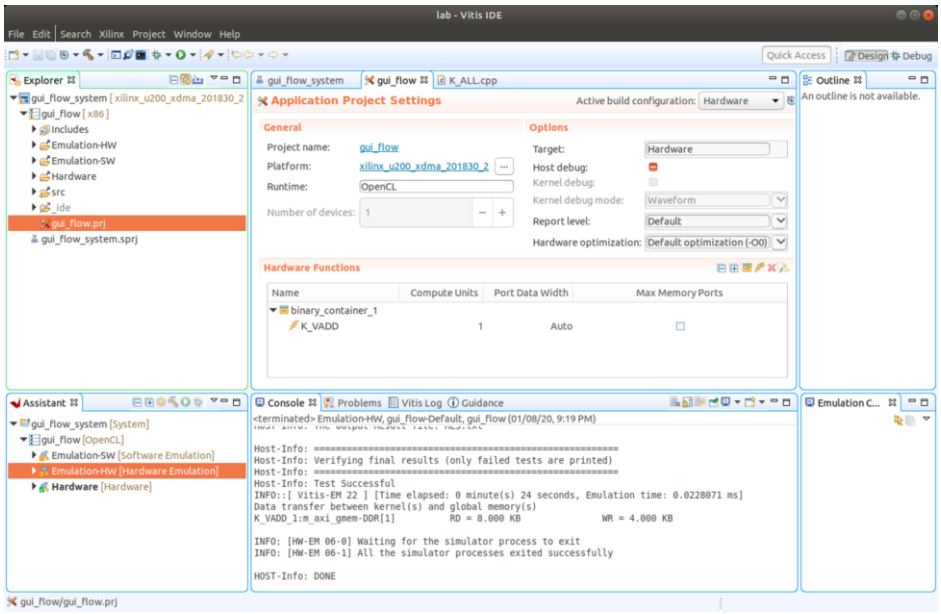


Figura 74: Entorno visual de Vitis IDE con consola integrada (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

Además, este entorno de desarrollo se basa en el estándar de código abierto de Eclipse.

La plataforma unificada de desarrollo Software de Vitis está enfocada hacia el desarrollo de aplicaciones de Software embebido en procesadores embebidos de Xilinx. Por ello, mantiene las características legacy del kit de desarrollo y añade nuevas funcionalidades destinadas a requerimientos de computación más heterogéneos. La tecnología Vitis se enfoca hacia las plataformas de aceleración Hardware, como las tarjetas de aceleración Alveo Data Center o los procesadores embebidos basados en Zynq Ultrascale+ MPSoc y Zynq-7000 SoC. De esta forma, proporciona una API y los drivers necesarios para conectar el programa host con la plataforma de destino, permitiendo manejar las transacciones entre las partes host y kernel.

El kit de desarrollo proporciona un conjunto de herramientas (como compiladores o compiladores cruzados) para construir los programas necesarios tanto para la parte host como para la parte kernel, los depuradores necesarios para pruebas y desarrollo, y analizadores, para generar perfiles de las aplicaciones y analizar el rendimiento.

Por otra parte, proporciona librerías de aceleración para optimizar el rendimiento de la aplicación sobre FPGAs, requiriendo el menor número de cambios en el código y sin necesitar recodificar el código completo para adecuarlo al Hardware, no perdiendo los beneficios de la computación adaptativa de Xilinx. Estas librerías proporcionan funciones para uso general

relacionadas con matemáticas, estadística, álgebra lineal, DSP, o incluso ciertos ámbitos específicos, como puede ser procesamiento de visión e imagen, transcodificación de vídeo, matemática financiera, bases de datos, análisis de datos, compresión de datos o machine learning.

Este conjunto de herramientas se representa gráficamente en la Figura 75, donde se muestran los niveles de abstracción, desde el nivel inferior, relacionado con los dispositivos físicos y el Hardware, hasta el nivel superior, relacionado con aplicaciones concretas con librerías de abstracción de alto nivel.

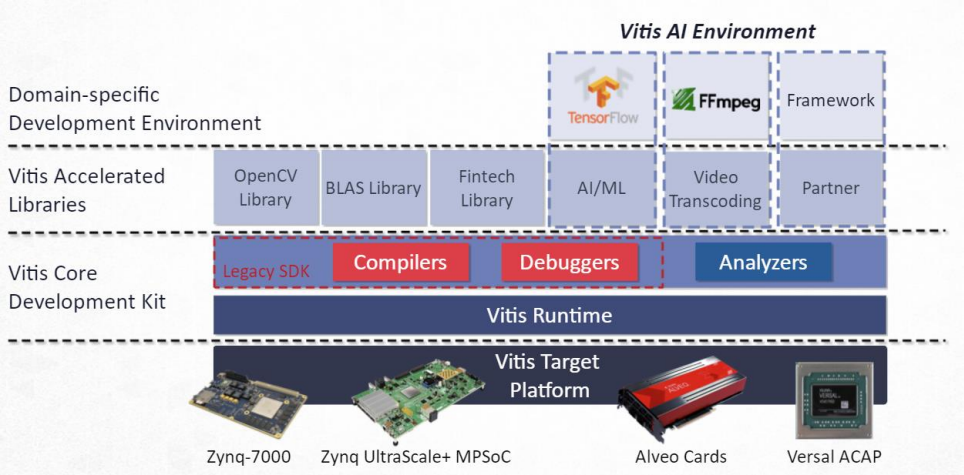


Figura 75: Herramientas de los diferentes niveles de abstracción del entorno de desarrollo unificado Vitis (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

En la parte intermedia, se encuentran todas las herramientas que permiten el desarrollo de aplicaciones para su ejecución directamente sobre el Hardware. Este es el principal objetivo del desarrollo, ejecutar las aplicaciones aceleradas sobre el Hardware de alto rendimiento basado en FPGAs. Para poder construir una solución desplegable de manera óptima y eficiente sobre el Hardware, es necesario seguir un proceso de desarrollo y depuración, como para cualquier otra aplicación. Para ello, el entorno unificado proporciona tres entornos diferentes para las pruebas de depuración, cada una satisfaciendo unas necesidades diferentes.

Existen dos modos de emulación, en que se permite la depuración para propósitos puramente de validación. El tercer modo es el de ejecución sobre el propio Hardware, es decir, generando el binario de la FPGA final.

Los tres modos indicados, mostrados en la Figura 76 y en la Figura 77, son:

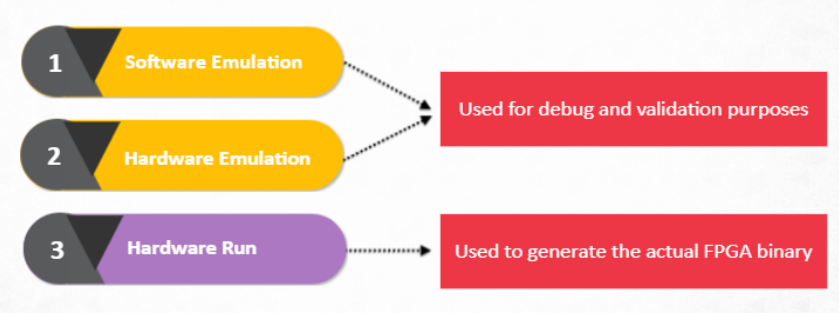


Figura 76: Modos de ejecución de aplicaciones aceleradas (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

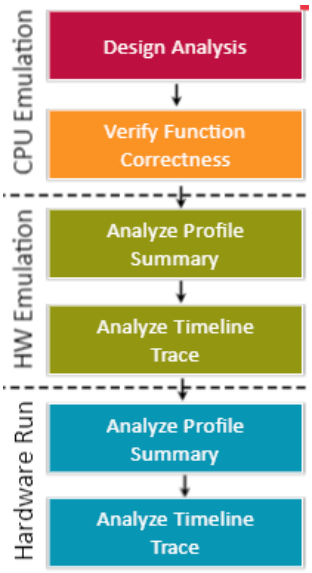


Figura 77: Secuencia de pasos realizados para cada modo de ejecución de aplicaciones aceleradas (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

1. Software emulation (o emulación Software):

   Tanto el código del host, como el del kernel, son compilados para su ejecución en el procesador del host. Esto permite un refinamiento de los algoritmos a través de un proceso iterativo de build-and-run (compilar y ejecutar), con tiempos entre iteraciones más cortos. Es útil para detectar errores de sintaxis y depurar a nivel de código el programa del kernel, junto al del host, para verificar el comportamiento del sistema completo, así como de cada una de sus partes integrantes.

2. Hardware emulation (o emulación Hardware):

   El código del kernel es compilado en un modelo de Hardware (RTL), que es ejecutado sobre un simulador dedicado. Este modelo build-and-run es algo más lento, pero proporciona una visión más detallada y precisa de la actividad del kernel. Es útil para probar el funcionamiento de la lógica que se introduce en la FPGA, permitiendo, además, obtener una primera aproximación del rendimiento real de la aplicación final. Es un punto intermedio hacia la ejecución completa en la tarjeta FPGA.

3. Ejecución en Hardware:

   El código del kernel es compilado en un modelo Hardware (RTL) que es implementado sobre la FPGA, dando como resultado un binario que puede ser ejecutado directamente en una FPGA real. Cuando el destino es este modo, el compilador de Vitis genera el fichero ".xclbin" correspondiente para la tarjeta de aceleración Hardware correspondiente, mediante la herramienta de diseño Vivado, realizando los procesos de síntesis e implementación necesarios. El kit de diseño permite abstraerse de esta complejidad, aunque es de interés profundizar en ella, para desarrollar aplicaciones y herramientas óptimas y eficientes, desarrolladas de la mejor manera posible, según las particularidades del Hardware implicado, obteniendo kernels de calidad.

Teniendo en cuenta este entorno y esta filosofía, se define una serie de pasos para lograr un sistema heterogéneo de ejecución de aplicaciones, en relación con el código del host. Se resumen en los siguientes puntos:

1. Arranque:

   En este punto, se carga la plataforma de destino en la FPGA. Una vez se encuentra lista para el uso, se comunica automáticamente con el host. Esta plataforma permite una abstracción de los recursos I/O de la FPGA, así como de las interfaces estándar AXI hacia los kernels. Además, proporciona características de confiabilidad y seguridad sobre el Hardware IP. En la Figura 78 se puede observar el esquema de la comunicación inicial del proceso de arranque, donde la plataforma de destino se encuentra entre la región de lógica programable y el host.

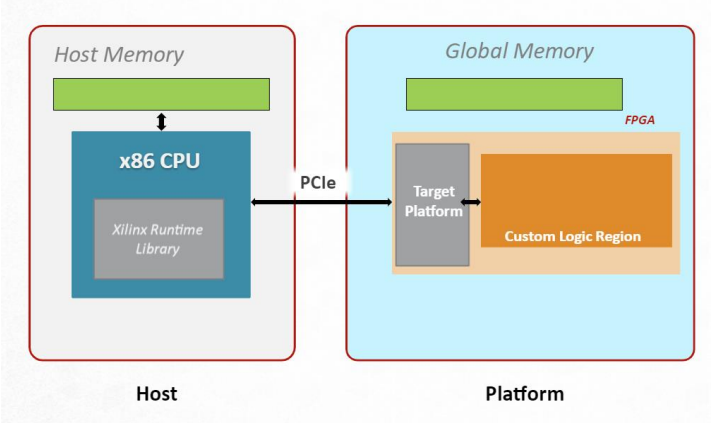


Figura 78: Bloques funcionales host y plataforma Xilinx de aceleración (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

2. Inicialización en tiempo de ejecución:

   En este punto, el programa host comienza a crear el contexto OpenCL, que consiste en registrar la plataforma y el dispositivo FPGA y abrir una cola de comandos, para comenzar a servir el dispositivo. Existen APIs OpenCL para descubrir plataformas y dispositivos, así como para crear los contextos y encolar llamadas o comandos. Un ejemplo de dicha funcionalidad sería:

```
errCode = clGetPlatformIDs(…);
clGetDeviceIDs(…, &device_id, …);
context = clCreateContextFromType(…);
queue = clCreateCommandQueue(context, device_id, …);
```

Código 1: Ejemplo de código para detección, inicialización y ejecución de kernel en tarjeta Alveo (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

3. Configuración de dispositivo:

Cuando el host ejecuta el comando de creación del programa binario mediante la API correspondiente, los kernel se cargan en el dispositivo. Es un proceso que no requiere la interacción con el usuario final ni el acceso al bitstream descargado. Los kernel se encuentran listos para su ejecución en la FPGA a partir de este punto. Mediante la Figura 79, se representa el flujo de este proceso, de forma que, tras la llamada a la API correspondiente, se genera el fichero binario xclbin que, mediante XRT y la comunicación PCIe, se comunica con la plataforma de destino, quien se encarga de configurar la lógica programable correspondiente a la aplicación concreta desarrollada.

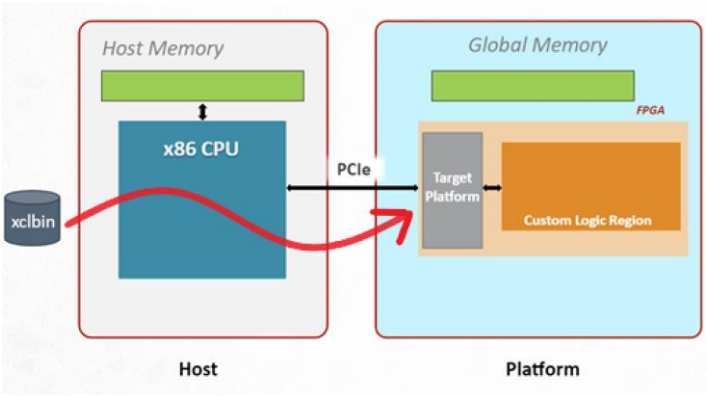


Figura 79: Diagrama de instalación de binario en plataforma destino (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

4. Asignación y reserva de buffers:

A continuación, la aplicación necesita comenzar a enviar datos. Para ello, es necesario definir los buffers para el intercambio de información, tanto en la memoria del host, como en la memoria del kernel. Mediante la llamada a la API correspondiente, se definen aquellos buffers espejo en el kernel que sean necesarios para el intercambio

correcto de información. Un ejemplo de la distribución de buffers, se muestra en la Figura 80.

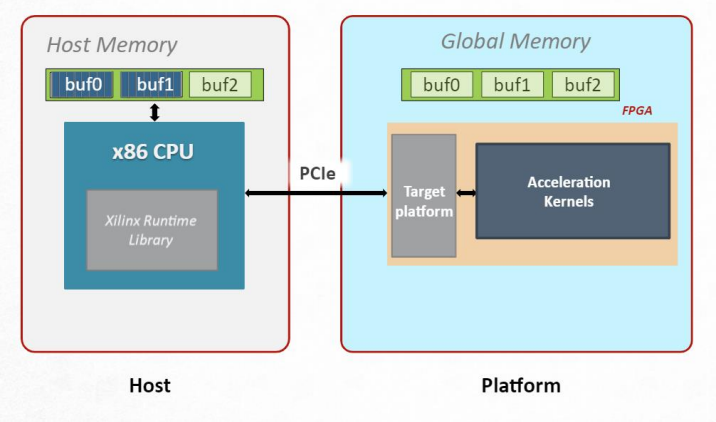


Figura 80: Asignación de buffers de entrada y salida en host y plataforma destino (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

5. Escritura de datos en memoria global:

Mediante la API de migración de objetos de memoria, se transfiere la información de cada buffer de la memoria del host, al buffer correspondiente de la memoria global del dispositivo.

6. Ejecución de acelerador:

Los buffers se asocian con los kernel mediante la API de asignación de argumentos. Una vez definidos y asociados, se puede utilizar la API de ejecución, para realizar las tareas o procesos necesarios, para leer los buffers de entrada, procesar la información y escribir el resultado en los buffers de salida. Para indicar al host que la operación está completada, el kernel envía automáticamente una notificación al host, indicando que el resultado está disponible. Esta señal se almacena en una cola de comandos, de forma que se puede definir si la acción es bloqueante o no, definiendo si el flujo debe esperar a la finalización o puede seguir en paralelo con otra ejecución. Esto permite controlar el flujo del programa, logrando una paralelización según los intereses del desarrollo. Este proceso se muestra en la Figura 81.

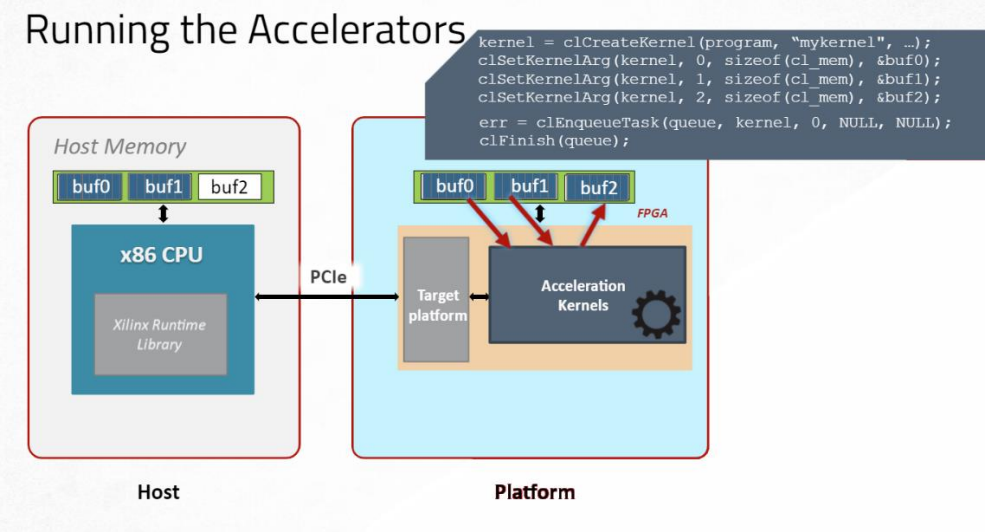


Figura 81: Ejemplo de ejecución de kernels en plataforma destino, con notificación hacia host (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

7. Lectura de datos de memoria global:

   Una vez finalizada la ejecución del kernel y habiéndose notificado su finalización, el host procede a la lectura de los resultados de los buffers de salida correspondientes, alojados en la memoria global. Para ello, hace uso de nuevo de la API de migración de objetos de memoria, utilizando una operación de encolado, de forma que se garantice que el valor leído es el esperado, tras la ejecución, y no valores de ejecuciones previas.

   Este paso completa el proceso de ejecución completo del kernel o los kernels implicados, conociendo en este punto, la secuencia de operaciones realizadas por el programa host y comprendiendo como los datos se mueven a través del sistema completo.

Una vez conocida la secuencia de pasos del proceso de ejecución, se puede analizar y entender de mejor manera y en mayor detalle cómo se logra la aceleración Hardware, focalizando el análisis en la ejecución de las funcionalidades y tareas de la aplicación. Como se ve en la Figura 70, la ejecución de funciones de una aplicación tradicional, ejecutada sobre CPU puramente, puede dividirse y ejecutarse parcial o totalmente en el Hardware de aceleración. Sin embargo, es algo más complejo que en dicho caso inicial, ya que es necesario tener en cuenta la introducción de funciones de llamada correspondiente a APIs y a procesos

de lectura/escritura. Esto introduce ciertas latencias, perfectamente distinguibles en la Figura 82.

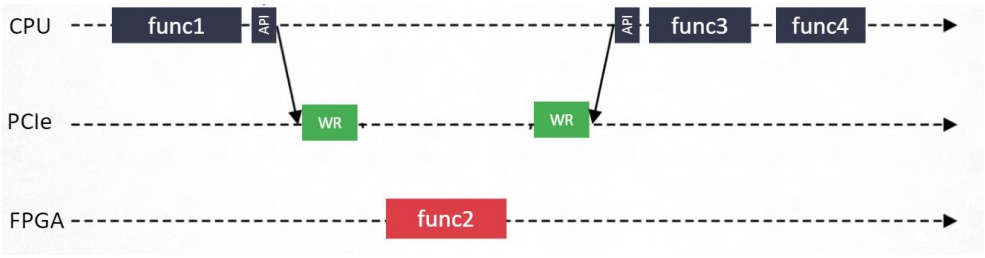


Figura 82: Representación de latencias introducidas por procesos de lectura/escritura mediante PCIe (Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021)

Teniendo en cuenta estas condiciones, se establecen varias reglas para definir cuándo se debe mover una función hacia la parte de aceleración Hardware:

1. Recordar la Ley de Amdahl:

   Tener en cuenta el rendimiento general de la aplicación, no solo de las funciones individuales. Comenzar con la medida de tiempo de ejecución y capacidad de rendimiento para identificar cuellos de botella de la aplicación ejecutándose sobre la plataforma utilizada. Estos datos de rendimiento deben ser generados tanto para la aplicación completa, como para cada una de las funciones individuales principales que la componen. La forma más eficaz es ejecutar la aplicación con herramientas de perfilado como valgrind, callgrind o GNU gprof.

   Las funciones que consumen un mayor tiempo de ejecución, son buenas candidatas para ser extraídas y aceleradas en el Hardware de las FPGAs. Un ejemplo de ello, es el mostrado en la Figura 70, donde la función 2 es la mejor candidata, siendo más favorable doblar el rendimiento de dicha función, que mejorar el rendimiento de la función 3, por ejemplo, en un orden de cincuenta.

2. Enfocarse en tareas largas y relacionadas con el ámbito de la computación:

   Es importante enfocarse en funciones cuyo tiempo de ejecución sea considerablemente superior al tiempo de transferencia de datos. Por tanto, son preferibles aquellas funciones que realizan muchas operaciones por cada llamada, frente a funciones que realizan pocas operaciones y son llamadas numerosas veces.

3. Conocer el máximo alcanzable:

En la aceleración de aplicaciones mediante FPGAs, es necesario transmitir los datos entre el host y el kernel, especialmente en aquellos casos basados en interfaces PCIe. Esto supone la adición de latencia, lo que puede suponer un elevado coste al rendimiento total de la aplicación. Es importante, por tanto, transferir los datos en el momento adecuado, para evitar que además del tiempo de transferencia, se añadan retardos mientras el kernel espera a recibir los datos a procesar.

Es crucial enviar los datos y que los kernels los tengan disponibles desde el momento en que vaya a comenzar su ejecución. Para lograr esto, se utiliza el solapamiento de la transferencia de datos, con la ejecución del kernel.

Aun realizando el proceso de la forma más óptima posible, existe un límite de la aceleración, determinado por el rendimiento de la propia interfaz PCIe, condicionado por diferentes aspectos como la placa base, los drivers, la plataforma de destino concreta o los tamaños de las transacciones de información.

Es importante realizar previamente un test DMA para medir la capacidad efectiva de las transferencias de la interfaz PCIe, determinando el límite superior alcanzable, si se consigue optimizar la aceleración de la aplicación. Toda aplicación que busque un objetivo de aceleración por encima de este valor, no puede alcanzarse por este método debido a limitaciones I/O. Este límite también hay que tenerlo en cuenta a la hora de desarrollar los propios kernels.

El valor máximo del potencial de aceleración viene dado por la ecuación de la Figura 83:

$$\text{Maximum acceleration potential} = \frac{\{\text{PCIe interface throughput}\}}{\{\text{SW throughput}\}}$$

Figura 83: Cálculo del máximo potencial teórico de aceleración de aplicación mediante plataformas Xilinx

4. Pensar en capacidad/rendimiento, no solo en latencia:

Una forma de optimizar las trasferencias de datos, es mediante la utilización de buffers de tamaño óptimo. La interfaz PCIe presenta un mayor rendimiento cuando los buffers

de transferencia tienen tamaños mayores. Cuanto mayor es el buffer, mejor es el rendimiento, ya que los aceleradores siempre tienen datos para operar y no se desperdician ciclos de reloj. Es recomendable realizar transferencias de 1 MByte o más. Hay que tener en cuenta el compromiso entre rendimiento y utilización de recursos, ya que cuanto mayor es el tamaño de los buffers, mayor será el número de recursos utilizado.

Mediante un test DMA, se puede conocer el tamaño óptimo de dichos buffers.

Siguiendo con esta filosofía, es interesante buscar como objetivo reducir los tiempos de espera de la CPU, mediante la maximización de la utilización del kernel, ya sea mediante la paralelización de tareas o mediante la transferencia de funciones con potencial de aceleración, a dicho dispositivo de aceleración Hardware.

Como resumen de la documentación, se pueden extraer unas conclusiones de cara a entender de manera general la filosofía de Xilinx para el entorno de desarrollo unificado de aceleración Hardware Vitis:

4. Mediante APIs, se trata de abstraer al desarrollador de la complejidad del propio Hardware y de las funcionalidades de bajo nivel de la aceleración de aplicaciones mediante dispositivos FPGA. Mediante llamadas a funciones definidas en la API OpenCL, el programa host puede descubrir dispositivos y contextos de aceleración, para enviar datos, solicitar ejecuciones de kernels y recuperar los datos procesados.
5. El paradigma se divide en dos grandes elementos: el programa host (que se ejecuta sobre plataformas x86/ARM, sobre procesadores) y el kernel (escrito en C/C++/RTL y ejecutado directamente sobre el Hardware de aceleración). Ambos elementos se comunican mediante una interfaz, en el caso de la tecnología utilizada, una interfaz PCIe.
6. No todos los algoritmos son susceptibles de ser acelerados, ya que es necesario entender el funcionamiento y comportamiento de los algoritmos implicados, para conocer si pueden existir beneficios de su aceleración y estableciendo un objetivo acorde a las plataformas e interfaces utilizadas (teniendo en cuenta el límite superior del potencial de la aceleración, determinado por la interfaz de comunicación, así como de los elementos Hardware y Software que interactúan con ella).

A.3. Hardware Xilinx: tarjetas de aceleración Alveo

Resumen conceptual en Apartado 5.3 en la página 42.

(Xilinx, Developer Program Secure Site: Accelerating Applications with the Vitis Unified Software Environment, 2021) (Xilinx, Developer Program Secure Site: Using Xilinx Alveo Cards to Accelerate Dynamic Workloads, 2021)

Para solventar la diferencia entre la demanda de computación de los data centers basados únicamente en CPUs, Xilinx ha desarrollado un conjunto de tarjetas, conocidas como Alveo. Estos dispositivos se definen como tarjetas de aceleración adaptables, preparadas para su despliegue en producción y mediante comunicaciones PCIe.

Con una rápida evolución de los algoritmos actuales, hacia modelos mucho más complejos, con un crecimiento más rápido que los propios ciclos de diseño del silicio, no es posible desarrollar dispositivos GPU o ASIC para funciones fijas y definidas, de manera que cumplan con los requisitos a largo plazo.

Ante esta situación, Xilinx proporciona sus tarjetas de aceleración Alveo, adecuadas para soluciones personalizadas, mediante su entorno de desarrollo unificado Vitis. Esto permite desarrollar aplicaciones adaptables al propio crecimiento computacional, respetando los dispositivos adquiridos y utilizados previamente.

Con la introducción de estas tarjetas de aceleración, se crea un ecosistema, actualmente en crecimiento, hacia aplicaciones y soluciones, tanto de Xilinx como de otros socios, implementadas sobre data centers comunes y de elevada potencia computacional.

En la Figura 84, se puede observar la tendencia del desarrollo de Xilinx, donde se aprecia la tendencia hacia un abanico más amplio de desarrollo que el inicial, centrado en dispositivos de silicio y tarjetas de evaluación. Dicha tendencia gira hacia un ecosistema completo, con soluciones formadas por servicios o aplicaciones de alto rendimiento, desarrolladas mediante un entorno y unas herramientas unificadas, e implementadas sobre Hardware reconfigurable de alto rendimiento.

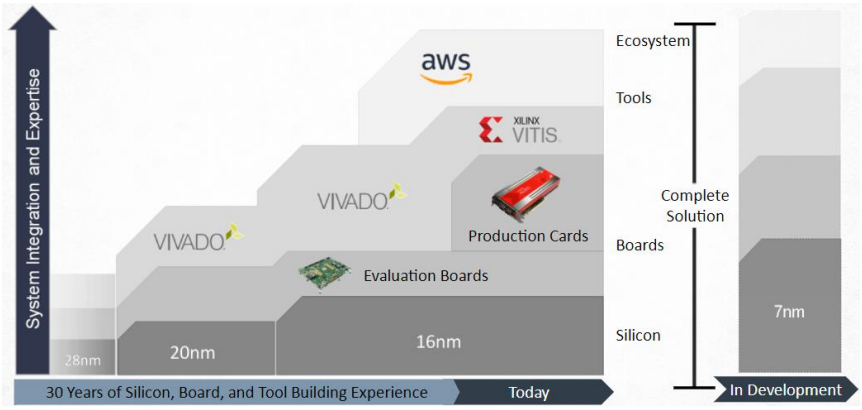


Figura 84: Evolución del silicio, tarjetas y herramientas Xilinx (Xilinx, Developer Program Secure Site: Using Xilinx Alveo Cards to Accelerate Dynamic Workloads, 2021)

Para satisfacer las diferentes demandas de las aplicaciones, Xilinx ha lanzado varias versiones de tarjetas de aceleración Alveo, cada una con una serie de recursos y características concretas. Las versiones actuales se pueden ver en la Figura 85.

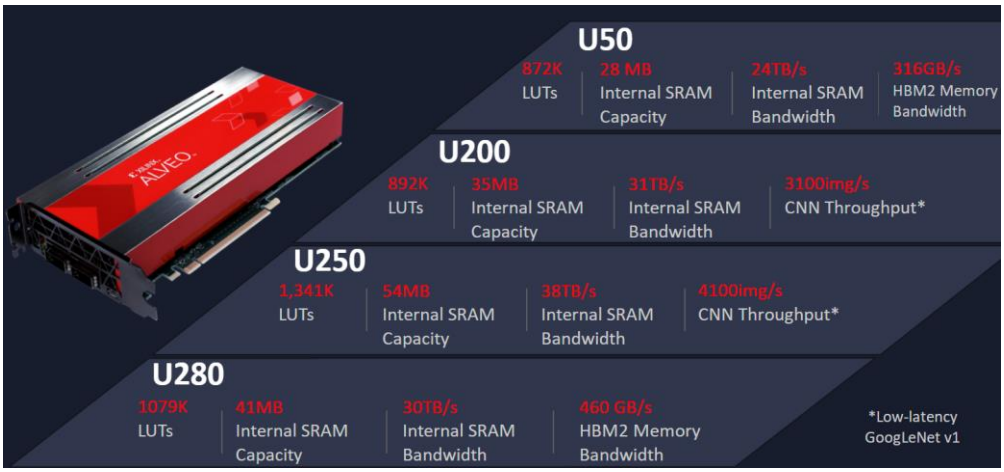


Figura 85: Modelos de tarjetas de aceleración Alveo (Xilinx, Developer Program Secure Site: Using Xilinx Alveo Cards to Accelerate Dynamic Workloads, 2021)

Un caso especial es el de la tarjeta mostrada en la Figura 86, que es una tarjeta de aceleración para data centers de tipo SmartNIC (Smart Network Interface Card). Esta tarjeta cuenta con dos conectores SFP28 ópticos para transmisiones de alta velocidad, permitiendo tasas de 10/25 Gbits/s. Además, cuenta con drivers y Software Onload (para aceleración de funciones de red sobre protocolos TCP/UDP), permitiendo aplicaciones sobre kernels de baja latencia, con mejoras de hasta un 400% para aplicaciones basadas en cloud. Cuenta también con las

tecnologías de Xilinx para PS y PL mediante FPGAs, bancos de memoria dedicados y un procesador ARM.

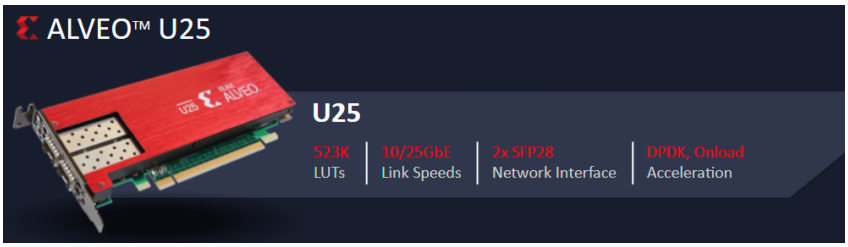


Figura 86: Modelo Alveo U25 (Xilinx, Developer Program Secure Site: Using Xilinx Alveo Cards to Accelerate Dynamic Workloads, 2021)

Mediante estos dispositivos, Xilinx satisface la demanda de los data centers modernos, que requieren cambios constantes, aportando un alto rendimiento, el cual supera en hasta 90 veces el rendimiento tradicional de los flujos de trabajo basados en CPU. Algunos de los ámbitos que más se favorecen de esta tecnología son machine learning, transcodificación de vídeo en tiempo real, búsquedas en bases de datos y análisis de datos. Las principales características de estos dispositivos, se muestran en forma de diagrama en la Figura 87.

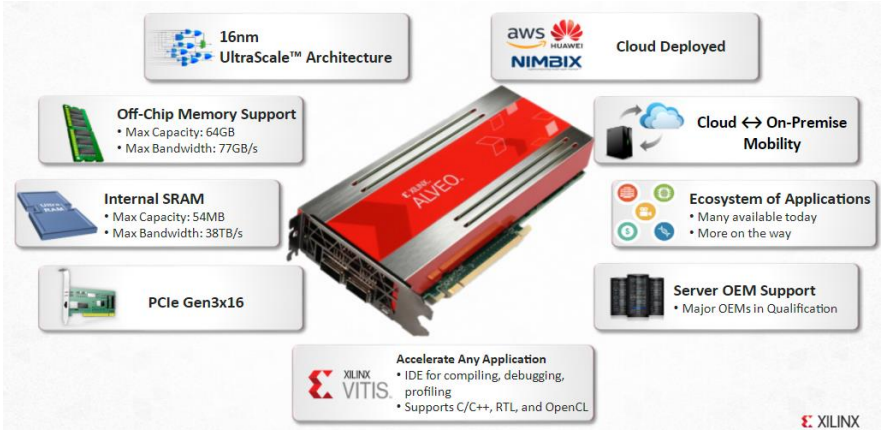


Figura 87: Diagrama de características generales de tarjetas Alveo (Xilinx, Developer Program Secure Site: Using Xilinx Alveo Cards to Accelerate Dynamic Workloads, 2021)

La filosofía de Xilinx para las tecnologías que forman la colección de tarjetas de aceleración comentadas, sigue tres aspectos clave:

1. Rapidez y rendimiento:

   Búsqueda de una mayor rapidez y rendimiento frente a CPUs y GPUs y una menor latencia respecto a las GPUs. Tras varios análisis realizados por el fabricante, se

obtienen factores de mejora importantes respecto a aplicaciones ejecutadas sobre CPU-GPU de manera tradicional. Algunos ejemplos del factor de mejora en diferentes ámbitos de aplicación, frente al caso tradicional de ejecución sobre CPU, se observan en la Figura 88.

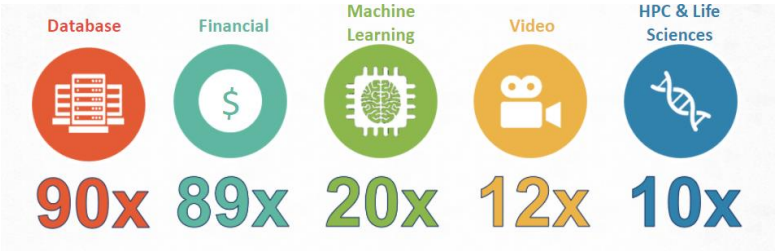


Figura 88: Índices de mejora de aceleración en ámbitos de aplicación probados (Xilinx, Developer Program Secure Site: Using Xilinx Alveo Cards to Accelerate Dynamic Workloads, 2021)

Estos factores de mejora, se obtienen de diferentes escenarios propuestos y realizados por Xilinx, en colaboración con empresas y entidades de los diferentes sectores:

- Bases de datos (database): comparación de pruebas de rendimiento en procesos de búsqueda elástica en instancias de Amazon AWS EC2 c4.8xlarge y la tarjeta Alveo U200. (AWS, Instancias F1 de Amazon EC2, 2021)
- Ámbito financiero (financial): cálculo del Valor de Riesgo (Value at Risk, VaR) en Intercambios de Tasas de Interés (Interest Rate Swap, IRS) usando una unidad Maxeler MPC-X conteniendo Maxelers de 5ª generación, sobre FPGAs de Xilinx VU9P (Xilinx, Maxeler Technologies Inc. Partner Information, 2018)
- Machine Learning. (Xilinx, Accelerating DNNs with Xilinx Alveo Accelerator Cards, 2018)
- Vídeo: comparación de la solución de codificación NG Codec U200 HEVC con un Dual Socket E5-2680 v3 2.5 GHz y una Alveo U200.
- Salud y Ciencia: (HPC & Life Sciences): con la tarjeta Alveo U200 se consigue reducir el tiempo de ejecución del GATK original, en más de 50 horas, en comparación con una CPU de 16 cores, para NA12878 WGS. (Research, 2020)

Para el ámbito concreto de la Inteligencia Artificial y del Machine Learning, Xilinx ha desarrollado Vitis AI, una plataforma de desarrollo encaminada a proporcionar rápidos despliegues de redes neuronales convolucionales (CNN), sobre Hardware de aceleración. La plataforma consiste en un conjunto optimizado de IPs, herramientas, librerías, modelos y diseños de ejemplo, encaminados a extender el gran potencial de la aceleración sobre FPGAs y ACAPs de Xilinx, a estos ámbitos concretos tan exigentes.

En la Figura 89, se ilustra la capacidad de Vitis AI en las tarjetas Alveo U200 y U250, en comparación con los casos de ejecución sobre CPU, GPU y FPGAs de otros fabricantes. En la Figura 90 se muestra la comparación entre consumos eléctricos.

La aceleración mediante estas tecnologías, permite alcanzar la menor latencia, el mayor rendimiento y la mayor eficiencia energética posibles, respecto al resto de aceleradores disponibles. En este caso, la tarjeta U250 es capaz de servir 4100 imágenes/segundo con una latencia de solo 1,8 ms, consumiendo un total de 110W de potencia. Esto se traduce en que mejora en un factor incremental de 20 al caso de ejecución puramente sobre CPU, un factor de incremento de 15 respecto a la FPGA Arria 10 y cerca de un factor incremental de 5 respecto a la solución GPU insignia de Nvidia, la tarjeta Tesla V100.

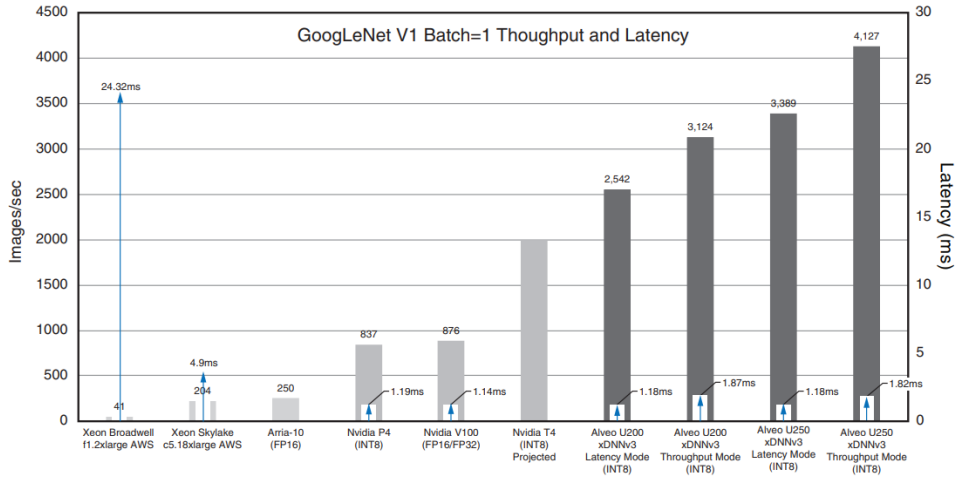


Figura 89: Comparativa de tasas de procesamiento y latencias en ejecución CNN para varios modelos CPU, GPU y FPGA (Xilinx, Accelerating DNNs with Xilinx Alveo Accelerator Cards, 2018)

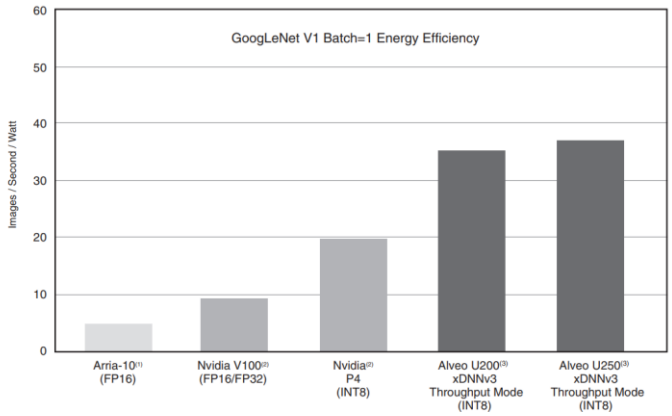


Figura 90: Comparativa de eficiencia energética en ejecución CNN para varios modelos CPU, GPU y FPGA (Xilinx, Accelerating DNNs with Xilinx Alveo Accelerator Cards, 2018)

Un gran número de industrias se ha beneficiado de las mejoras introducidas por la tecnología machine learning. Unidas a la mejora de la propia tecnología, la aceleración Hardware aporta nuevas características de interés, debido a la naturaleza de dicha tecnología. Machine Learning cuenta con dos fases principales diferenciadas: el entrenamiento, en que la red se entrena de forma offline; y la inferencia, fase en la que la red realiza la tarea en tiempo real. Ambas partes son exigentes en cuanto a requerimientos computacionales. Como resultado, soluciones únicamente ejecutadas sobre CPU son inviables, para satisfacer los requerimientos de latencia y capacidad, en relación a costes y consumo energético. La tecnología GPU ha mejorado estas condiciones para el caso de entrenamiento de ML, gracias a la mejora que aporta con la paralelización de la capacidad de computación, así como gracias a los altos ratios entre computación y transferencia de datos. Sin embargo, no se han consolidado como una solución adecuada en el caso de inferencia, debido a la combinación entre necesidad de latencias bajas con ratios entre computación y transferencia elevados. Es por ello que, mediante la utilización de tarjetas Alveo, junto a la CPU del host, se consiguen mejores resultados que en el caso de aplicar GPUs. Esto lo muestra el propio fabricante en la Figura 91, donde indica los resultados de la comparación, para el caso de conversión de voz a texto, para los casos de CPU-FPGA y CPU-GPU.

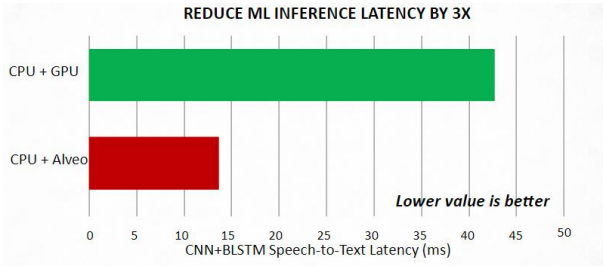


Figura 91: Comparación de latencia en inferencia Machine Learning en soluciones CPU+GPU y CPU+Alveo (Xilinx, Developer Program Secure Site: Using Xilinx Alveo Cards to Accelerate Dynamic Workloads, 2021)

La clave de la aceleración frente a otras tecnologías, reside en la capacidad de adaptar la arquitectura a la jerarquía de memoria y a los caminos de datos personalizados, óptimos para cada aplicación. Gracias a la adaptabilidad de la jerarquía de memoria, se consigue entregar los datos a cada proceso ejecutado, sin necesidad de utilizar un bloque de memoria común, que añade latencias y cuellos de botella. Esto se observa en la Figura 92, donde se aprecia cómo se entregan los datos al kernel A (leyendo una única vez de la memoria global), que procesa la información y la transmite al kernel B, que hace lo mismo y entrega su resultado a C, para que éste último devuelva el resultado final. En el caso de una GPU, es necesario leer, procesar y escribir los datos en cada bloque de procesamiento, ya sea en la memoria global, o en el mejor de los casos, en una memoria caché.

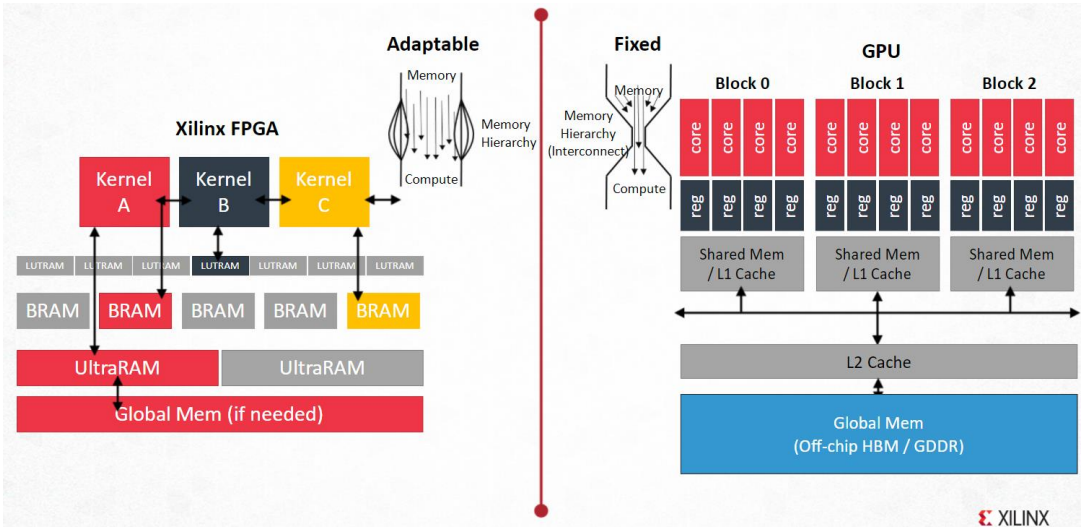


Figura 92: Diferencias en acceso a datos y flujo de los mismos a través del flujo de ejecución, para arquitecturas FPGA y GPU (Xilinx, Developer Program Secure Site: Using Xilinx Alveo Cards to Accelerate Dynamic Workloads, 2021)

2. Adaptabilidad para acelerar cualquier flujo:

El eje central es permitir la optimización de cualquier flujo o aplicación, de forma que se consiga una adaptabilidad a cambios constantes en algoritmos y computación, sin necesidad de sustituir el Hardware sobre el que se implementa.

Este aspecto es fundamental en ciertas aplicaciones actualmente en auge como es el caso de redes machine learning. Este tipo de tecnologías, se encuentran en constante evolución y las arquitecturas CPU y GPU pueden suponer un impedimento para su correcto desarrollo. Por ejemplo, las GPU sólamente pueden tratar un tipo específico de datos, por lo que, si se requiere una serie de datos con una cierta precisión concreta y mayor potencia computacional, capacidad y eficiencia, los usuarios deben esperar a las nuevas generaciones del Hardware para poder disponer de la nueva implementación de sus aplicaciones. Esto supone la introducción de retardos en la evolución de la tecnología, debido a las esperas entre generaciones, y suponen una mayor inversión económica, al tener que renovar el Hardware a medida que evoluciona la aplicación.

En el caso de las tarjetas Alveo, se consigue solucionar esta problemática gracias a que son dispositivos reprogramables, lo que permite adaptar el Hardware y sus funcionalidades, sobre el mismo dispositivo electrónico y sin tener que adquirir nuevo equipamiento, a las implementaciones de las nuevas versiones de las aplicaciones aceleradas, más exigentes e innovadoras.

El proceso de evolución de la implementación sobre Hardware Alveo, frente al uso de GPUs, se muestra de manera ilustrada en la Figura 93, donde se aprecia la continuidad del Hardware a lo largo de las diferentes versiones de la aplicación, sin necesidad de realizar mayor desembolso económico que el inicial para la adquisición de la tarjeta Alveo.

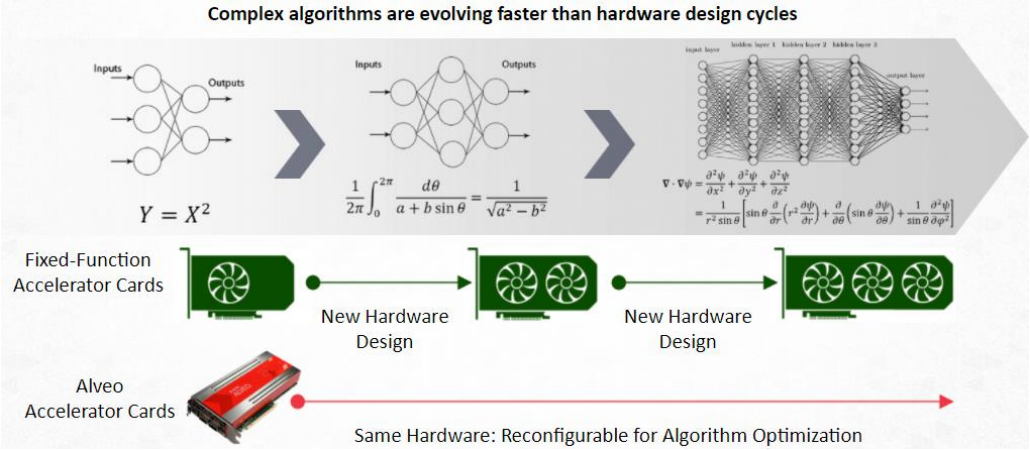


Figura 93: Evolución del hardware con el desarrollo de los requerimientos de los algoritmos implicados (Xilinx, Developer Program Secure Site: Using Xilinx Alveo Cards to Accelerate Dynamic Workloads, 2021)

3. Accesibilidad:

   Xilinx permite la movilidad entre cloud y on-premises, buscando siempre el despliegue de aplicaciones, de forma que estén disponibles en el menor tiempo posible.

   Siguiendo esta filosofía, Xilinx promueve un ecosistema de despliegue operativo, en colaboración con diferentes entidades y empresas a nivel mundial, para los diferentes ámbitos de aplicación que se pueden beneficiar de la aceleración Hardware. Actualmente existen numerosas soluciones operativas, proporcionadas por diferentes proveedores, de forma que es posible acceder a un mercado ya desarrollado si fuera necesario.

De esta forma, gracias a este planteamiento, se consiguen una serie de beneficios de la aplicación de aceleración sobre tarjetas Alveo:

1. Aplicabilidad sobre ámbitos y aplicaciones de diferente naturaleza.
2. Mejor rendimiento frente a CPU/GPU, con una mejora de hasta un factor de x90 en el caso de CPU y de hasta x5, en el caso de GPU.
3. Menor TCO (Total Cost of Ownership), ahorrando hasta un 65%.
4. Soluciones con visión a futuro, permitiendo adaptar el Software a la evolución de los flujos de trabajo y de las propias aplicaciones.

Para la solución propuesta de esta memoria, se elige la tarjeta Alveo U50, mostrada en la Figura 94. Algunas de las aplicaciones principales de dicho dispositivo, que el fabricante propone, son:

- Traducción de voz: alta tasa efectiva de procesamiento y aceleración de inferencia con baja latencia.
- Análisis de bases de datos: alta tasa efectiva de procesamiento en aceleración de consultas.
- Modelado de mercado financiero: alta eficiencia en modelado de precios derivados y riesgos.
- Operaciones de intercambio electrónico: ultra baja latencia en comunicaciones y computación aceleradas.
- Almacenamiento computacional: aceleración de compresión de datos con elevado rendimiento
- Aceleración de Hadoop: aceleración de compresión de datos con elevado rendimiento
- Transcodificación de vídeo en tiempo real: simplificación y reducción de costes de la infraestructura con un rendimiento superior, con menores latencias.

Figura 94: Tarjeta Alveo U50 (Xilinx, Developer Program Secure Site: Using Xilinx Alveo Cards to Accelerate Dynamic Workloads, 2021)

En cuanto a las opciones de instalación del Software, en relación a la localización física de las tarjetas Alveo, existen dos posibilidades principales:

1. On-Premises:

   Consiste en la adquisición del propio Hardware para su despliegue e instalación en las instalaciones propias de la empresa. Es necesario un desembolso inicial, pero el dispositivo pasa a ser propiedad de la empresa. Existen diferentes versiones de

tarjetas, con diferentes características y precios, siendo adquiridas directamente a través de Xilinx, o a través de otros proveedores colaboradores. Los precios de algunos modelos, siendo proporcionados por Xilinx, se pueden apreciar en la Figura 95, permitiendo hacerse una idea del coste económico que suponen este tipo de equipamiento. Además de los dispositivos Hardware, es necesario adquirir todas las licencias y el Software necesarios para el desarrollo y/o ejecución de las aplicaciones sobre estos dispositivos, según la naturaleza de los proyectos.

Figura 95: Costes adquisición modelos Alveo U50 y U250

En el caso de adquirir las tarjetas de manera física, es necesario comprender los diferentes modelos disponibles, tanto en especificaciones técnicas internas, como en la forma de refrigeración de las mismas, ya que esto condiciona el equipo final en el que se instala el equipamiento, así como el consumo esperado durante su funcionamiento.

Existen dos modos de refrigeración, según se observa en la Figura 96, para el caso de la tarjeta Alveo U200. Mientras la refrigeración activa está destinada a despliegues sobre entornos de ordenadores personales con ventilación no controlada (la tarjeta cuenta con ventiladores y disipadores), la refrigeración pasiva está destinada a data centers y habitáculos con sistemas de refrigeración propios.

Figura 96: Tipos de refrigeración para tarjetas Alveo

2. Cloud:

   Consiste en la suscripción a una serie de servicios proporcionados por terceros, que incluyen tanto el Hardware, accesible de manera remota, junto a las herramientas necesarias para el desarrollo y/o ejecución de las aplicaciones. Normalmente se proporcionan una serie de APIs para la integración del flujo de trabajo sobre FPGA y en algunos casos, las propias herramientas de Vitis a través de navegadores web.

   El proveedor propuesto por Xilinx es Nimbix, que permite este tipo de metodología Cloud, con soluciones ya operativas y con todas las herramientas necesarias para desarrollar nuevas aplicaciones aceleradas personalizadas, pudiendo elegir entre los diferentes dispositivos Alveo disponibles. (Nimbix, 2021)

Independientemente de la localización física de los dispositivos, Xilinx permite dos modos de instalación de Software para las aplicaciones de aceleración:

1. Despliegue (Deployment):

   Permite desplegar aplicaciones precompiladas directamente sobre el dispositivo. Consiste en Xilinx runtime (XRT) y la plataforma de despliegue. Se corresponde con los pasos 1 y 2 comentados posteriormente, en este apartado.

2. Desarrollo (Development):

   Permite desarrollar, compilar y depurar aplicaciones aceleradas, para posteriormente ser desplegadas en el Hardware. Consiste en la plataforma de desarrollo y en el entorno de desarrollo Software Xilinx Vitis. Se corresponde con los pasos 1, 2, 3 y 4 comentados posteriormente, en este apartado.

Gracias a esta división, se pueden establecer dos funciones o entornos diferenciados, que determinan los requerimientos en la instalación de cada uno de ellos.

Los pasos para la instalación de todas las herramientas y Software necesarios, son:

1. Xilinx Runtime (XRT):

   API y drivers necesarios para la abstracción de la capa de comunicación de bajo nivel, entre el host y la tarjeta de aceleración.

2. Plataforma de despliegue destino (deployment):

   Nivel físico de comunicación implementado e instalado en la tarjeta de aceleración.

3. Plataforma de desarrollo destino (development):

   Interfaz para desarrollo de aplicaciones que generan nuevos niveles físicos de comunicación implementados sobre la tarjeta de aceleración.

4. Entorno de diseño Vitis:

   Plataforma de desarrollo de Software de aceleración que proporciona el entorno y el conjunto de herramientas necesarias para aplicaciones sobre tarjetas de aceleración de Xilinx.

El proceso de instalación se muestra en la descripción de la solución con mayor detalle, indicando los procesos y fases necesarios para la correcta configuración del entorno.

A la hora de instalar la tarjeta de aceleración, es necesario tener un equipo lo suficientemente potente para soportar tanto las operaciones computacionales, como las comunicaciones desde/hacia la tarjeta. Los requerimientos mínimos que el fabricante recomienda para ordenadores personales son los mostrados en la Figura 97.

Component	Requirement
Motherboard	PCI Express 3.0 compatible with one dual-width x16 slot
System Power Supply	75 W(U25, U50), 225 W(U200, U250, U280)
Operating System	Linux, 64-bit • Ubuntu 16.04, 18.04, 20.04 • CentOS 7.4, 7.5, 7.6, 7,7, 7.8, 8,1, 8.2 • RHEL 7.4, 7.5, 7.6, 7.7, 7.8, 8.1, 8.2
System Memory	Minimum of 64 GB, but 80 GB is recommended
Hard Disk	Satisfy the minimum system requirements for your operating systems
Internet Connection	Required for downloading drivers and utilities

Figura 97: Requerimientos mínimos recomendados por Xilinx para equipos destinados a desarrollo y/o despliegue de aplicaciones aceleradas en tarjetas Alveo (Xilinx, Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393), 2021)

Para garantizar la correcta compatibilidad y la mejor optimización de las aplicaciones, el propio fabricante proporciona una lista de servidores completamente compatibles con las tecnologías requeridas por las tarjetas de aceleración. Estos servidores se muestran en la Figura 98, proporcionando las especificaciones principales que soportan. (Xilinx, Xilinx Qualified Servers Catalog, 2021)

Figura 98: Interfaz de visualización de servidores certificados por Xilinx para correcta compatibilidad con tarjetas Alveo (Xilinx, Xilinx Qualified Servers Catalog, 2021)

A.3.1. Xilinx Alveo U50

Resumen conceptual en Apartado 5.3.1 en la página 48.

La tarjeta de aceleración para centros de datos Xilinx Alveo U50, mostrada en la Figura 99, es una tarjeta de slot único, de factor de forma de pequeño tamaño, con alimentación pasiva, con una potencia máxima de operación de 75 W. Soporta comunicaciones PCI Express (PCIe) Gen3 x16 o dual Gen4 x8, presenta capacidad de red Ethernet y cuenta con una memoria de alto ancho de banda de 8 GB (HBM2). Con estas características, el fabricante recomienda este dispositivo para su aplicación en aceleración de aplicaciones ligadas a capacidades de memoria y a una computación intensiva, en ámbitos como la computación financiera, almacenamiento computacional, y búsqueda y analítica de datos. En concreto, la tarjeta Alveo U50LV, se recomienda para aceleración de flujos de trabajo de inferencias de machine learning. Las tarjetas Alveo U50 y U50LV son idénticas, exceptuando la tensión de trabajo del núcleo, diferenciadas por V_{NOM} (0.85V) y por V_{LOW} (0.72V), respectivamente. Ambas tarjetas, preparadas para su despliegue, cuentan con un conector QSFP28 óptico, para comunicaciones de hasta 4 canales a 25GB cada uno. (Xilinx, Alveo U50 Data Center Accelerator Card Data Sheet (DS956), 2020)

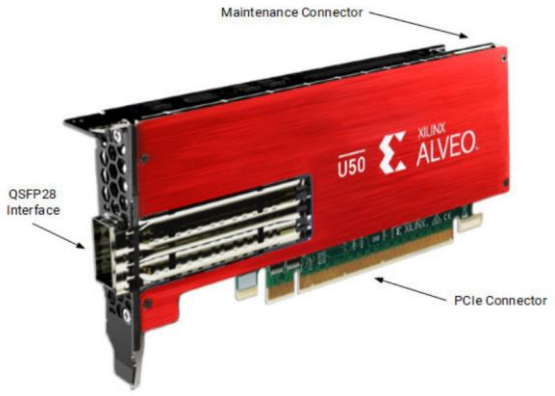


Figura 99: Representación física de la tarjeta Alveo U50 (Xilinx, Alveo U50 Data Center Accelerator Card Data Sheet (DS956), 2020)

Este dispositivo está preparado para su integración con la plataforma de desarrollo unificada de software Xilinx Vitis, que permite simplificar el proceso de diseño, permitiendo el uso de leguajes de alto nivel, como por ejemplo C, C++ y OpenCL. Mediante la plataforma correspondiente, se puede habilitar la configuración mediante la memoria flash interna, así como su actualización, mediante PCIe. Además, es posible integrarlo con Vivado Design

Suite, para desarrollar de manera detallada y a bajo nivel, todos los recursos de la lógica programable del circuito.

Los detalles y las especificaciones técnicas del producto se definen en la Tabla 11, diferenciando ambas versiones del modelo.

Specification	U50 Production[1]	U50 LV Production[1]
Product SKU	A-U50-P00G-PQ-G	A-U50-P00G-LV-G
Total electrical card load[2]	75W	75W
Thermal cooling solution	Passive	Passive
Weight	300g – 325g	300g – 325g
Form factor	Half height, half length	Half height, half length
Network interface	1x QSFP28 (100 GbE)	1x QSFP28 (100 GbE)
Network clock precision	IEEE 1588	IEEE 1588
PCIe interface[3, 4]	Gen3 x16, Gen4 x8, CCIX	Gen3 x16[5,6]
HBM2 total capacity	8 GB	8 GB
HBM2 bandwidth	316 GB/s[7]	316 GB/s[7]
Look-up tables (LUTs)	872K	872K
Registers	1,743K	1,743K
DSP slices	5,952	5,952
Max. Dist. RAM	24.6 Mb	24.6 Mb
36 Kb block RAM	1344 (47.3 Mb)	1344 (47.3 Mb)
288 Kb UltraRAM	640 (180.0 Mb)	640 (180.0 Mb)
GTY transceivers	20	20
V_{CCINT} supported	V_{NOM} (0.85V)	V_{LOW} (0.72V)
Vitis™ Development Enviroment	Yes	Yes
Vitis platform	Gen3 x16 XDMA, Gen3 x4 XDMA[8]	Gen3 x4 XDMA[9]
Vivado Design Suite	Yes	Yes
Target workloads	Fintech, video, database, and computational storage	Machine learning (ML) inference

Tabla 11: Especificaciones técnicas tarjetas Alveo U50 y U50LV (Xilinx, Alveo U50 Data Center Accelerator Card Data Sheet (DS956), 2020)

Como se puede observar, presentan las mismas características, exceptuando el caso comentado relacionado con la tensión de trabajo, además de la posibilidad de configuración de los interfaces PCIe en modo compatible con Gen4, solo en el caso de la Alveo U50.

El diagrama de bloques de las tarjetas U50/U50LV, mostrando los componentes internos de las mismas, se muestra en la Figura 100.

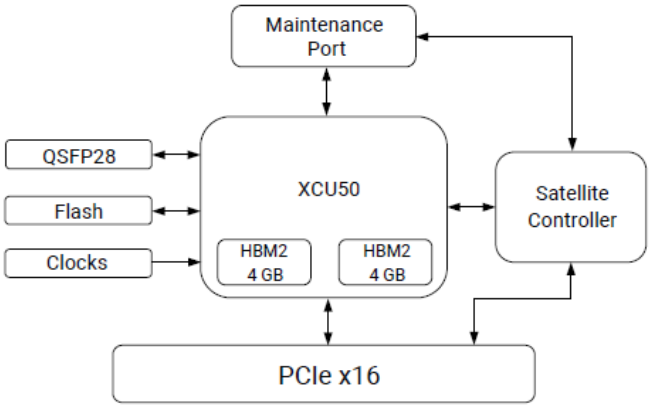


Figura 100: Diagrama de bloques tarjeta Alveo U50 (Xilinx, Alveo U50 Data Center Accelerator Card Data Sheet (DS956), 2020)

En dicho diagrama se muestras varios bloques de interés:

- Conector PCIe: La FPGA contiene un bloque PCIE4C regido por el estándar PCI Express Base Specification v3.1, que permite comunicaciones de hasta 8 GT/s (para el caso Gen3 x16) y compatible con el estándar PCI Express Base Specification v4.0, que soporta comunicaciones de hasta 16 GT/s (para el caso Gen4 x8). También se rige por el estándar CCIX Base Specification Revision 1.0 v0.9, que permite velocidades de hasta 16 GT/s. En el caso de la Alveo U50LV, solamente está soportado el modo PCIe Gen3 x4.
- Interfaces de comunicaciones y red: Cuenta con un elemento QSFP28, de cuatro líneas separadas, que pueden aceptar eléctricamente módulos de hasta 5W. Este QSFP28 puede conectar interfaces de hasta 100G, usando módulos o cables ópticos. Proporciona un reloj de 161,1328125 MHz a la interfaz QSFP28, de forma que diferentes núcleos Ethernet IP puedan ser habilitados. Además, cuenta con una memoria flash no volátil, de 1 Gb, para configuración, mediante comunicación Quad SPI. (Xilinx, Alveo U50 Data Center Accelerator Card: User Guide (UG1371), 2019)

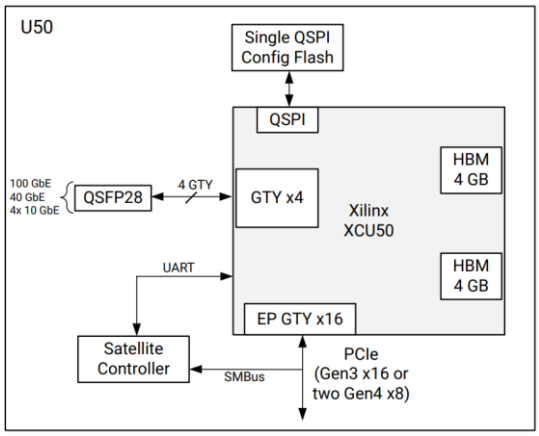


Figura 101: Diagrama de bloques Xilinx XCU50 FPGA (Xilinx, Alveo U50 Data Center Accelerator Card: User Guide (UG1371), 2019)

- Controlador satélite: Se trata de un TI MSP432 que permite controlar y monitorizar tensiones, corrientes y temperaturas. El controlador de la gestión de la tarjeta del servidor host (BMC) permite interactuar con el controlador satélite interno, para controlar y monitorizar la tarjeta Alveo mediante comunicaciones fuera de banda. Para ello, Xilinx soporta el protocolo PLDM sobre MCTP, sobre SMBUS, acorde a los estándares DMTF. Si se hace uso de la plataforma de Xilinx, se puede acceder de manera directa y con plena compatibilidad a esta funcionalidad, permitiendo detectar de manera fácil cualquier malfuncionamiento del sistema. (Xilinx, Alveo Card Out-of-Band Management Specification for Server BMC: User Guide (UG1363), 2020)
- Puerto de mantenimiento: Se trata de un puerto con un conector de 30 pines que permite acceder a diferentes funcionalidades y señales, incluyendo JTAG, UARTs, PMBus y resets, entre otras. Xilinx cuenta con una tarjeta independiente para su conexión a dicho puerto de mantenimiento, permitiendo acceder fácilmente a dichos puertos y funcionalidades. Este elemento recibe el nombre de Debug and Maintenance Board (DMB).
- FPGA: Es una FPGA UltraScale+ construida de manera personalizada para funcionar de manera óptima en la arquitectura Alveo. La tarjera Alveo U50/U50LV cuenta con una FPGA XCU50, que utiliza la tecnología de Xilinx Stacked Silicon Interconnect (SSI), para entregar una capacidad, ancho de banda y eficiencia energética innovadoras, a la FPGA. Esta tecnología permite incrementar la densidad, combinando

múltiples Super Logic Regions (SLR). La XCU50, comprende dos SLRs. La SLR inferior, cuenta con un controlador HBM2 que sirve de interfaz hacia la memoria HBM2 adyacente de 8 GB. Ambas SLR se muestran en la Figura 102, donde se ven las conexiones hacia PCIe y SFP-QSFP, en su correspondiente región, así como los controladores HBM2, conectados directamente con la SLR0.

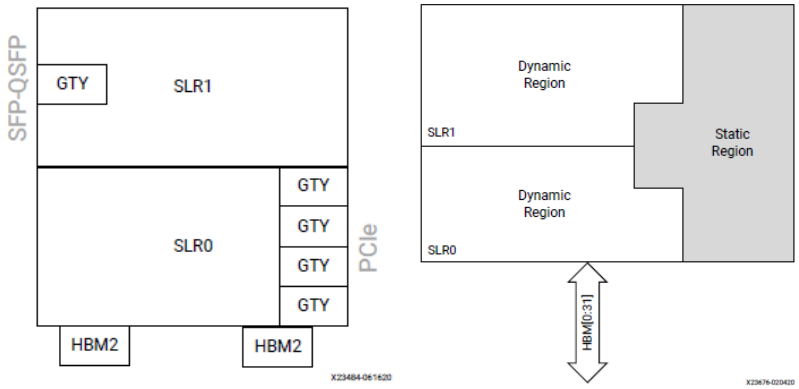


Figura 102: Esquema de distribución de regiones lógicas de FPGA XCU50 (Xilinx, Alveo U50 Data Center Accelerator Card Data Sheet (DS956), 2020) (Xilinx, Alveo Data Center Accelerator Card Platforms: User Guide (UG1120), 2021)

Cuando se utiliza el entorno de desarrollo de Xilinx Vitis, se crea una plataforma determinada para gestionar todas las interfaces PCIe, transferencias de datos e intercambio de información de estado de la tarjeta. Permite, además, cargar de manera remota los kernels y realizar diferentes funciones de control y gestión. Esta plataforma se encuentra sobre la región estática de la FPGA, no reconfigurable, que consume ciertos recursos de la tarjeta.

Además de la región estática, existe una región dinámica. La estructura interna de la tarjeta Alveo U50/U50LV, en relación a este aspecto y en comparación con el funcionamiento general de las plataformas Alveo, se muestra en la Figura 102 y en la Figura 103. En el dispositivo Xilinx o Alveo, la región estática de la plataforma, proporciona la infraestructura básica para la comunicación de la tarjeta con el host, así como el hardware necesario para soportar el kernel. Los kernels acelerados, se introducen en la región dinámica, mientras que la región estática cuenta con una serie de bloques definidos:

- Host Interface (HIF): terminal PCIe para comunicación con host externo PCIe.
- Direct Memory Access (DMA): bloques IP para XDMA y AXI Protocol Firewall.

- Clock, Reset and Isolation (CRI): gestión básica de relojes y resets para arranque y operación de la tarjeta. Estos elementos requieren un aislamiento durante la carga de los flujos de bits.
- Card Management Peripheral (CMP): periféricos responsables de la salud y diagnósticos de la tarjeta, tanto para depuración como programación.
- Card Management Controller (CMC): comunicaciones UART/I2C con controlador satélite MSP432, QSFP, sensores y para permitir actualizaciones de firmware desde el host (sobre PCIe).
- Embedded RunTime Scheduler (ERT): planificación y monitorización de las unidades de computación durante la ejecución de los kernels.

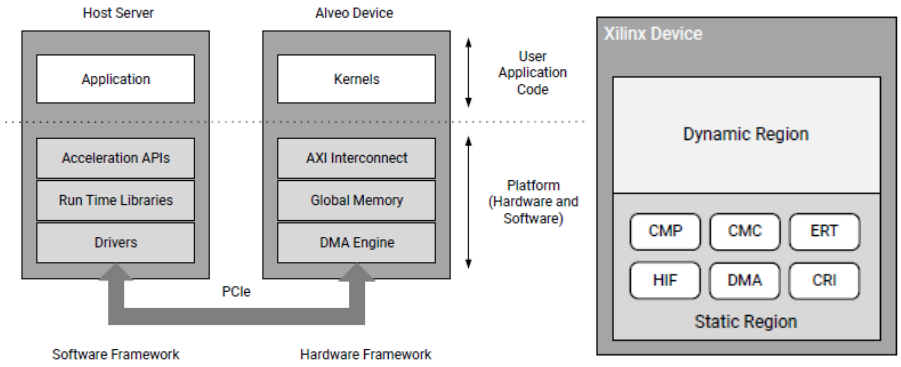


Figura 103: Esquema de distribución de componentes en regiones lógicas de plataforma destino Xilinx (Xilinx, Alveo Data Center Accelerator Card Platforms: User Guide (UG1120), 2021)

A un nivel más bajo, se puede definir un diagrama de bloques más complejo para las tarjetas Alveo, como el mostrado en la Figura 104. (Xilinx, Alveo Data Center Accelerator Card Test: User Guide (UG1361), 2021)

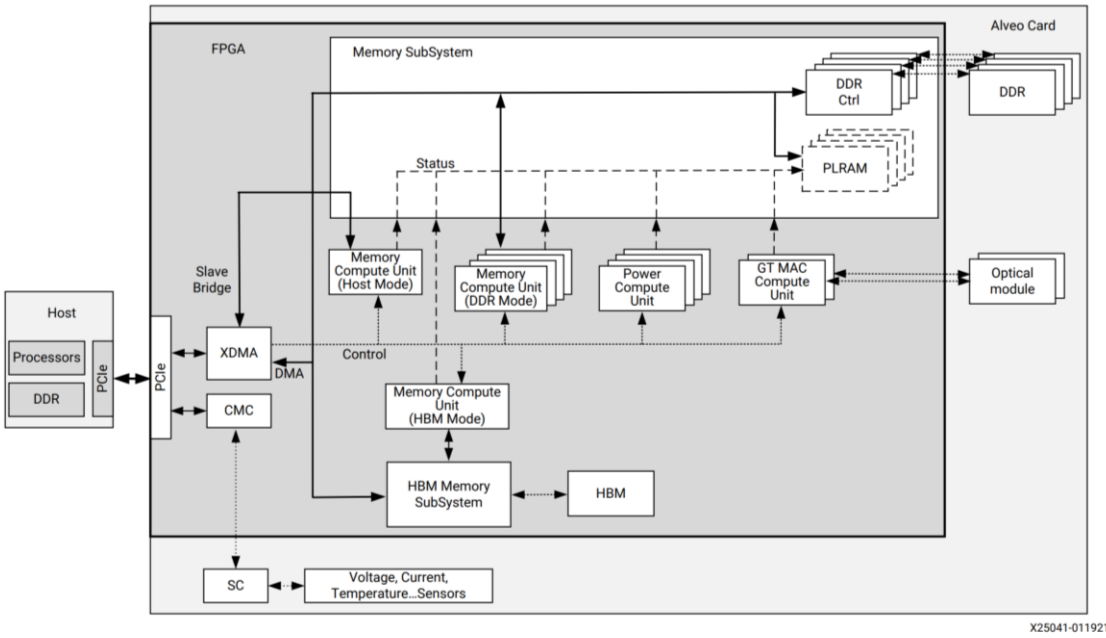


Figura 104: Diagrama de bloques detallado de tarjetas Alveo (Xilinx, Alveo Data Center Accelerator Card Test: User Guide (UG1361), 2021)

Para la instalación física de la tarjeta, se deben seguir los pasos definidos en el manual correspondiente, de forma que se conecten todos los puertos necesarios, mostrados en la Figura 99. Dicho manual es proporcionado por Xilinx e indica no sólo como conectar físicamente la tarjeta, sino también la instalación del software y plataformas para su correcto funcionamiento y comunicación con el equipo host. (Xilinx, Alveo U50 Data Center Accelerator Card Installation Guide (UG1370), 2020)

B. Instalación detallada del entorno de desarrollo Xilinx Vitis

B.1. Instalación.

Para instalar todas las herramientas y programas necesarios, Xilinx proporciona una serie de manuales, extensos, aunque con ciertas indicaciones ausentes. En este documento, se amplía dicha información y se trata de resumir la documentación, para facilitar el proceso a todo aquel interesado en desplegar el entorno unificado de desarrollo. Los principales pasos seguidos durante el despliegue del entorno son:

1. Instalar Sistema Operativo compatible: el SO utilizado para el proyecto es Ubuntu 20.04 LTS (última versión Ubuntu compatible). Esta labor escapa del alcance del proyecto, por lo que se deja al lector su instalación y configuración completa.
2. Preparación de software Xilinx, Vitis Unified Software Platform.
    a. Acceso a página web de descarga de software necesario e identificación de usuario o creación de cuenta de usuario (Xilinx, Downloads: Vitis Core Development Kit, 2021)
    b. Descarga del software necesario, obteniendo el instalador unificado para Linux (BIN), junto con los elementos de verificación indicados con los nombres “Digests”, “Signature” y “Public Key”.

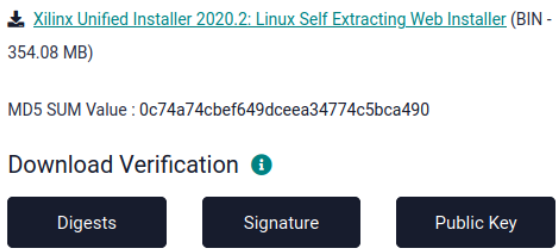


Figura 105: Descarga de archivos de instalación de Xilinx Unified 2020.2 para Linux (Xilinx, Downloads: Vitis Core Development Kit, 2021)

    c. Verificación del software descargado, para comprobar validez, autenticidad y confiabilidad de los archivos de instalación descargados.
        i. Clave pública de Xilinx: Importación de clave y asignación de nivel de confianza absoluta (ultimate).

```
user@ubuntu-pc:~/Xilinx$ gpg --import ./Downloads/xilinx-master-signing-key.asc
gpg: caja de claves '/home/user/.gnupg/pubring.kbx' creada
gpg: /home/user/.gnupg/trustdb.gpg: se ha creado base de datos de confianza
gpg: clave FD3DD2C355D2B701: clave pública "Xilinx, Inc. (Xilinx Software signing key)" importada
gpg: Cantidad total procesada: 1
gpg:               importadas: 1

user@ubuntu-pc:~/Xilinx$ gpg --list-keys
/home/user/.gnupg/pubring.kbx
------------------------------
pub   rsa4096 2018-04-23 [SC]
      4D27DDF2A08955CE241B4304FD3DD2C355D2B701
uid         [desconocida] Xilinx, Inc. (Xilinx Software signing key)
sub   rsa4096 2018-04-23 [S]
sub   rsa4096 2018-04-23 [E]
sub   rsa4096 2018-04-23 [A]

user@ubuntu-pc:~/Xilinx$ gpg --edit-key 4D27DDF2A08955CE241B4304FD3DD2C355D2B701
gpg (GnuPG) 2.2.19; Copyright (C) 2019 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.


pub  rsa4096/FD3DD2C355D2B701
     creado: 2018-04-23  caduca: nunca       uso: SC
     confianza: desconocido   validez: desconocido
sub  rsa4096/5DB253CE971AD24F
     creado: 2018-04-23  caduca: nunca       uso: S
sub  rsa4096/A9BDA2BBBCFF1F20
     creado: 2018-04-23  caduca: nunca       uso: E
sub  rsa4096/BCD820F8F76E0A9C
     creado: 2018-04-23  caduca: nunca       uso: A
[desconocida] (1). Xilinx, Inc. (Xilinx Software signing key)

gpg> trust
pub  rsa4096/FD3DD2C355D2B701
     creado: 2018-04-23  caduca: nunca       uso: SC
     confianza: desconocido   validez: desconocido
sub  rsa4096/5DB253CE971AD24F
     creado: 2018-04-23  caduca: nunca       uso: S
sub  rsa4096/A9BDA2BBBCFF1F20
     creado: 2018-04-23  caduca: nunca       uso: E
sub  rsa4096/BCD820F8F76E0A9C
     creado: 2018-04-23  caduca: nunca       uso: A
[desconocida] (1). Xilinx, Inc. (Xilinx Software signing key)

Por favor, decida su nivel de confianza en que este usuario
verifique correctamente las claves de otros usuarios (mirando
pasaportes, comprobando huellas dactilares en diferentes fuentes...)

  1 = No lo sé o prefiero no decirlo
  2 = NO tengo confianza
  3 = Confío un poco
  4 = Confío totalmente
  5 = confío absolutamente
  m = volver al menú principal

¿Su decisión? 5
¿De verdad quiere asignar absoluta confianza a esta clave? (s/N) s

user@ubuntu-pc:~/Xilinx$ gpg --list-keys
gpg: comprobando base de datos de confianza
gpg: marginals needed: 3  completes needed: 1  trust model: pgp
gpg: nivel: 0  validez:   1  firmada:   0  confianza: 0-, 0q, 0n, 0m, 0f, 1u
/home/user/.gnupg/pubring.kbx
------------------------------
pub   rsa4096 2018-04-23 [SC]
      4D27DDF2A08955CE241B4304FD3DD2C355D2B701
uid         [  absoluta ] Xilinx, Inc. (Xilinx Software signing key)
sub   rsa4096 2018-04-23 [S]
sub   rsa4096 2018-04-23 [E]
sub   rsa4096 2018-04-23 [A]
```

Figura 106: Gestión de clave pública de Xilinx (importación y definición de nivel de confianza)

ii. Firma: Verificación de firma del binario de instalación.

```
user@ubuntu-pc:~/Xilinx/Downloads$ gpg -v --verify Xilinx_Unified_2020.2_1118_1232_Lin64.bi
n.sig Xilinx_Unified_2020.2_1118_1232_Lin64.bin
gpg: Firmado el jue 19 nov 2020 06:03:38 CET
gpg:                usando RSA clave 5DB253CE971AD24F
gpg: usando subclave 5DB253CE971AD24F en vez de clave primaria FD3DD2C355D2B701
gpg: usando pgp como modelo de confianza
gpg: Firma correcta de "Xilinx, Inc. (Xilinx Software signing key)" [absoluta]
gpg: firma binaria, algoritmo de resumen SHA512, algoritmo de clave pública rsa4096
```

Figura 107: Verificación de firma de archivo binario de instalación Xilinx Unified

iii. Digests: Verificación de digests y obtención de valores numéricos

```
user@ubuntu-pc:~/Xilinx/Downloads$ gpg -v --verify Xilinx_Unified_2020.2_1118_1232_Lin64.bi
n.digests
gpg: cabecera de armadura: Hash: SHA512
gpg: nombre fichero original=''
gpg: Firmado el jue 19 nov 2020 06:03:44 CET
gpg:                usando RSA clave 5DB253CE971AD24F
gpg: usando subclave 5DB253CE971AD24F en vez de clave primaria FD3DD2C355D2B701
gpg: usando pgp como modelo de confianza
gpg: Firma correcta de "Xilinx, Inc. (Xilinx Software signing key)" [absoluta]
gpg: firma modotexto, algoritmo de resumen SHA512, algoritmo de clave pública rsa4096

user@ubuntu-pc:~/Xilinx/Downloads$ sha256sum -c Xilinx_Unified_2020.2_1118_1232_Lin64.bin.d
igests
Xilinx_Unified_2020.2_1118_1232_Lin64.bin: La suma coincide
sha256sum: ATENCIÓN: 22: líneas tienen un formato erróneo
```

Figura 108: Verificación de digests y hashes de Xilinx para binario de instalación

iv. Binario de instalación: Verificación de digests sobre instalador unificado.

```
user@ubuntu-pc:~/Xilinx/Downloads$ openssl dgst -sha256 Xilinx_Unified_2020.2_1118_1232_Lin64.bin
SHA256(Xilinx_Unified_2020.2_1118_1232_Lin64.bin)= 13f79e786902f0511a11da5b03ba302b2b66a95449a34cf348f785a8001d
0c04
user@ubuntu-pc:~/Xilinx/Downloads$ cat Xilinx_Unified_2020.2_1118_1232_Lin64.bin.digests | grep 13f79e786902f05
11a11da5b03ba302b2b66a95449a34cf348f785a8001d0c04
13f79e786902f0511a11da5b03ba302b2b66a95449a34cf348f785a8001d0c04 *Xilinx_Unified_2020.2_1118_1232_Lin64.bin
```

Figura 109: Obtención y comparación de digests y hashes del archivo de instalación, junto al proporcionado por Xilinx para garantizar autenticidad

3. Instalación de librerías adicionales requeridas: librerías necesarias, no indicadas en la documentación oficial cuya ausencia ocasiona errores en procesos de instalación. Adicionalmente, es necesario instalar las herramientas OpenCL.

```
sudo apt-get install libtinfo5 libncurses5 libncursesw5

sudo apt-get install git curl libcurl4-openssl-dev

sudo apt-get install ocl-icd-libopencl1 opencl-headers ocl-icd-opencl-dev
```

Código 2: Comandos para instalación de librerías adicionales requeridas para la instalación de Xilinx Vitis

4. Instalación de software Xilinx, Vitis Unified Software Platform

a. Obtención de licencia (solo necesaria en el caso de utilizar Vivado Design Suite)

b. Ejecución de instalador unificado de Xilinx

```
user@ubuntu-pc:~/Xilinx/Downloads$ sh ./Xilinx_Unified_2020.2_1118_1232_Lin64.bin
Verifying archive integrity... All good.
Uncompressing Xilinx Installer.....
.....
.....
.....
.....
.....
.....
..
```

Figura 110: Ejecución de instalados a través de terminal

c. Seguimiento de asistente de instalación.

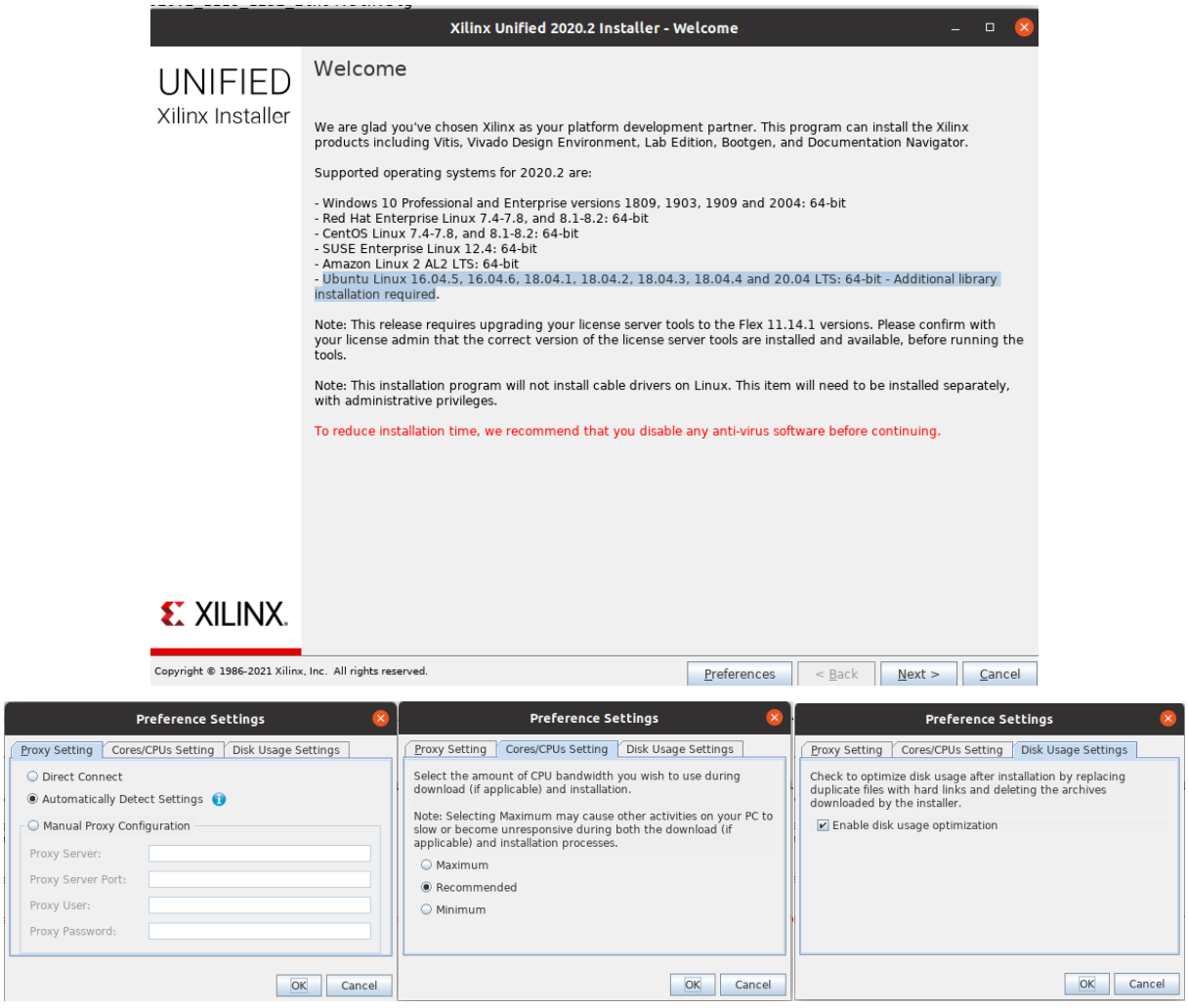


Figura 111: Primeros pasos del asistente de instalación

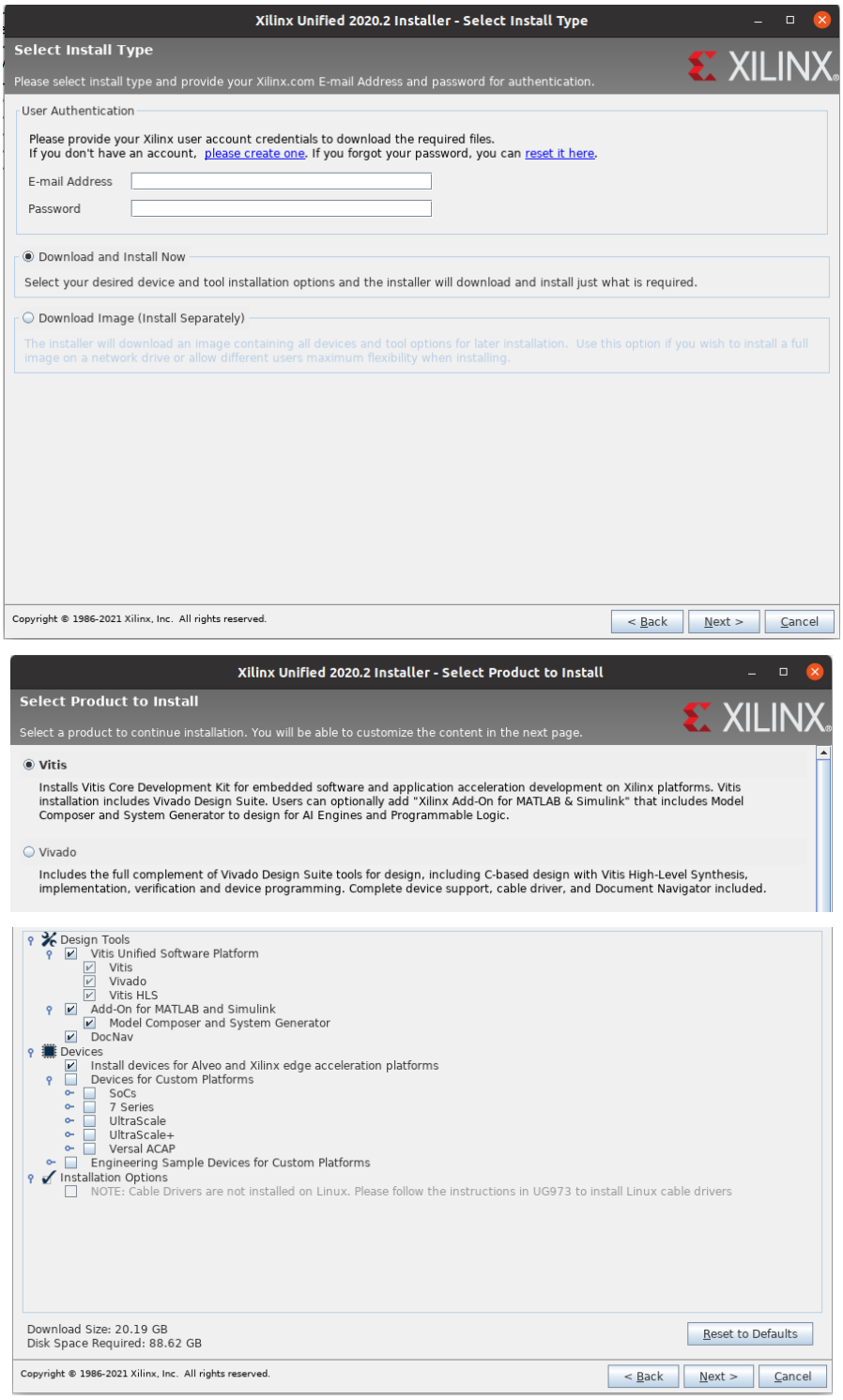


Figura 112: Autenticación mediante cuenta de usuario Xilinx y selección del kit de desarrollo Vitis, junto a herramientas necesarias, para su instalación

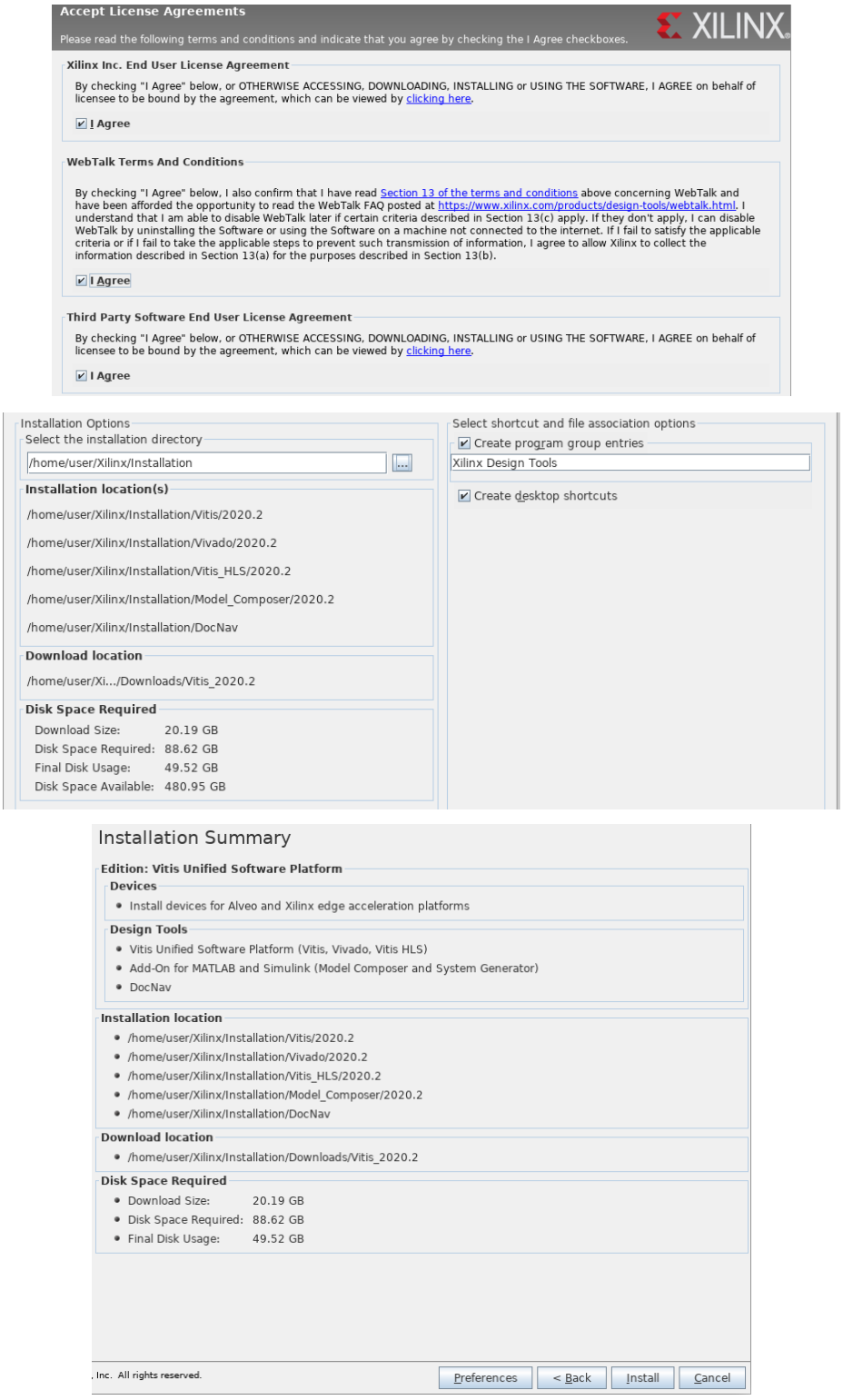


Figura 113: Aceptación de términos y condiciones, selección de directorios de instalación y resumen de parámetros de instalación

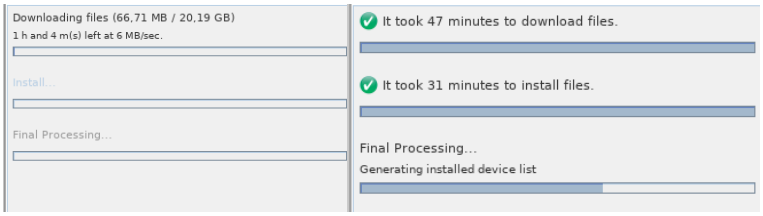


Figura 114: Descarga e instalación del software y de las herramientas Xilinx

d. Completar proceso de instalación

i. Finalizar correctamente

Figura 115: Finalización de instalación correcta

ii. Solución de posibles errores: Comprobar archivos de log indicados en el terminal de la instalación observando el último script ejecutado y fallido, para detener el proceso bloqueado correspondiente e instalar las dependencias y herramientas necesarias indicadas en el log (/home/user/.Xilinx/xinstall/xinstall_XXXXXXXXXX.log). Para poder continuar desde el punto anterior, salir del instalador con la opción de reiniciar el proceso desde la misma carpeta. Una vez instalado todo lo necesario, relanzar el instalador.

```
2021-04-08 00:26:09,872 DEBUG: n.t:? - Executing script Configure WebTalk: /home/user/Xilinx/Installation/Vivado/2020.2/data/webtalk/webtalk_install.sh [off]

2021-04-08 00:26:10,056 DEBUG: n.t:? - Executing script Generating installed device list: /home/user/Xilinx/Installation/Vivado/2020.2/bin/vivado [-nolog, -nojournal, -mode, batch, -source, /home/user/Xilinx/Installation/Vivado/2020.2/scripts/sysgen/tcl/xlpartinfo.tcl, -tclargs, /home/user/Xilinx/Installation/Vivado/2020.2/data/parts/installed_devices.txt]
```

Código 3: Ejemplo de salida en terminal para error de instalación

5. Instalación de librerías mediante script de Xilinx: localizado en directorio de instalación (<install_dir>/Vitis/<release>/scripts/installLibs.sh)

```
user@ubuntu-pc:~/Xilinx/Installation/Vitis/2020.2/scripts$ ls -al | grep install
-rwxr-xr-x  1 user user   3343 nov 18 17:34 installLibs.sh

user@ubuntu-pc:~/Xilinx/Installation/Vitis/2020.2/scripts$ sudo ./installLibs.sh
```

Figura 116: Instalación de librerías Xilinx para sistema operativo particular, tras instalación de Software Vitis

6. Inicialización y arranque de Vitis: ejecución mediante icono de programa Vitis localizado en dock de aplicaciones de Ubuntu.

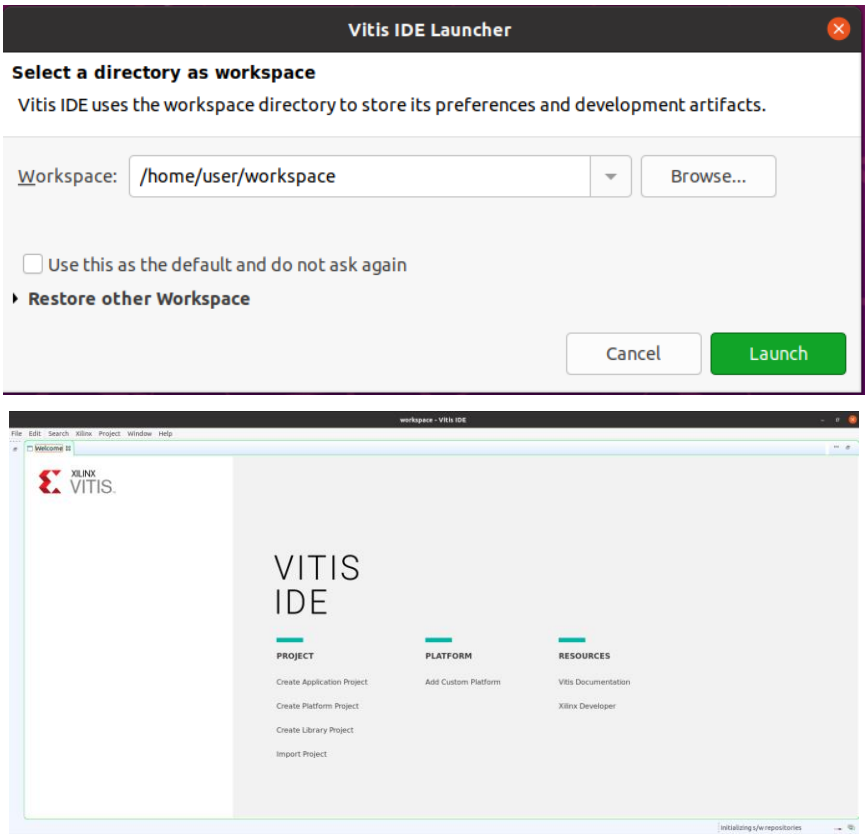


Figura 117: Primer arranque de Vitis IDE para verificación de instalación

7. Instalación Xilinx Runtime Library (XRT) y plataformas Xilinx Alveo U50:

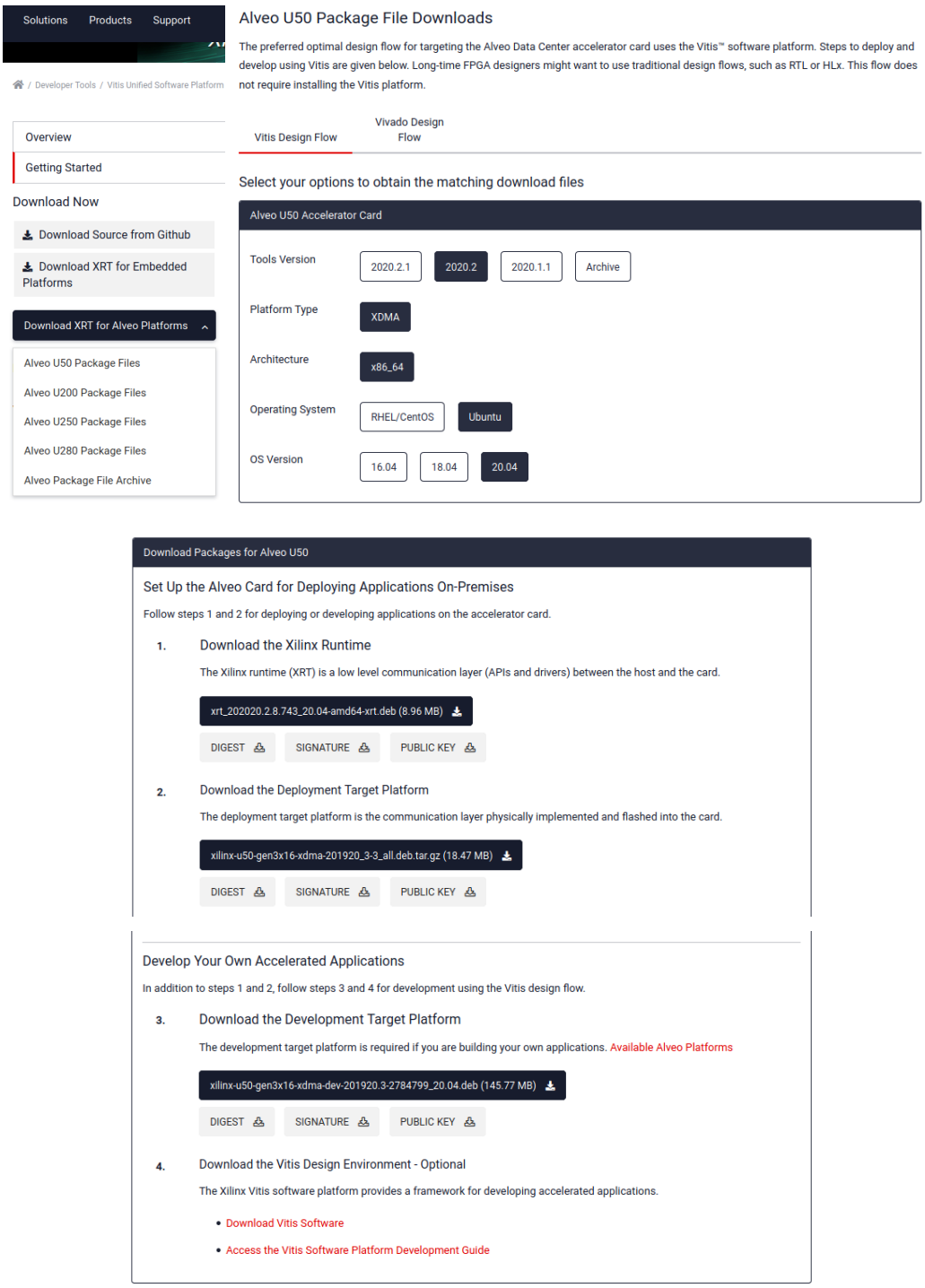


Figura 118: Selección y descarga de librerías, drivers y plataformas para tarjeta Alveo U50 (Xilinx, Downloads: Vitis Core Development Kit, 2021)

a. XRT

```
user@ubuntu-pc:~/Xilinx/Downloads/2_XRT$ gpg -v --verify xrt_202020.2.8.743_20.04-amd64-xrt.deb.sig xrt_20202
0.2.8.743_20.04-amd64-xrt.deb
gpg: Firmado el mié 02 dic 2020 09:41:08 CET
gpg:                usando RSA clave 5DB253CE971AD24F
gpg: usando subclave 5DB253CE971AD24F en vez de clave primaria FD3DD2C355D2B701
gpg: usando pgp como modelo de confianza
gpg: Firma correcta de "Xilinx, Inc. (Xilinx Software signing key)" [absoluta]
gpg: firma binaria, algoritmo de resumen SHA512, algoritmo de clave pública rsa4096

user@ubuntu-pc:~/Xilinx/Downloads/2_XRT$ gpg -v --verify xrt_202020.2.8.743_20.04-amd64-xrt.deb.digests
gpg: cabecera de armadura: Hash: SHA512
gpg: nombre fichero original=''
gpg: Firmado el mié 02 dic 2020 09:41:09 CET
gpg:                usando RSA clave 5DB253CE971AD24F
gpg: usando subclave 5DB253CE971AD24F en vez de clave primaria FD3DD2C355D2B701
gpg: usando pgp como modelo de confianza
gpg: Firma correcta de "Xilinx, Inc. (Xilinx Software signing key)" [absoluta]
gpg: firma modotexto, algoritmo de resumen SHA512, algoritmo de clave pública rsa4096

user@ubuntu-pc:~/Xilinx/Downloads/2_XRT$ sha256sum -c xrt_202020.2.8.743_20.04-amd64-xrt.deb.digests
xrt_202020.2.8.743_20.04-amd64-xrt.deb: La suma coincide
sha256sum: ATENCIÓN: 22: líneas tienen un formato erróneo
user@ubuntu-pc:~/Xilinx/Downloads/2_XRT$ openssl dgst -sha256 xrt_202020.2.8.743_20.04-amd64-xrt.deb
SHA256(xrt_202020.2.8.743_20.04-amd64-xrt.deb)= 6b944c45075d2e1df63d55209c8a311c94a81823273fee7e4e23a65833025
71c
user@ubuntu-pc:~/Xilinx/Downloads/2_XRT$ cat xrt_202020.2.8.743_20.04-amd64-xrt.deb.digests | grep 6b944c4507
5d2e1df63d55209c8a311c94a81823273fee7e4e23a6583302571c
6b944c45075d2e1df63d55209c8a311c94a81823273fee7e4e23a6583302571c *xrt_202020.2.8.743_20.04-amd64-xrt.deb
```

Figura 119: Verificación de firmas, digests, hashes y archivos binarios de instalación de Xilinx Runtime Library

```
sudo apt-get install xrt_XXXXXX.X.X.XXX-xrt.deb
```

Código 4: Comando para instalación de paquete XRT

b. Plataforma destino despliegue (Deployment)

```
user@ubuntu-pc:~/Xilinx/Downloads/3_AlveoU50$ gpg -v --verify xilinx-u50-gen3x16-xdma-201920_3-3_all.deb.tar.
gz.sig xilinx-u50-gen3x16-xdma-201920_3-3_all.deb.tar.gz
gpg: Firmado el mié 03 feb 2021 19:14:03 CET
gpg:                usando RSA clave 5DB253CE971AD24F
gpg: usando subclave 5DB253CE971AD24F en vez de clave primaria FD3DD2C355D2B701
gpg: usando pgp como modelo de confianza
gpg: Firma correcta de "Xilinx, Inc. (Xilinx Software signing key)" [absoluta]
gpg: firma binaria, algoritmo de resumen SHA512, algoritmo de clave pública rsa4096

user@ubuntu-pc:~/Xilinx/Downloads/3_AlveoU50$ gpg -v --verify xilinx-u50-gen3x16-xdma-201920_3-3_all.deb.tar.
gz.digests
gpg: cabecera de armadura: Hash: SHA512
gpg: nombre fichero original=''
gpg: Firmado el mié 03 feb 2021 19:14:04 CET
gpg:                usando RSA clave 5DB253CE971AD24F
gpg: usando subclave 5DB253CE971AD24F en vez de clave primaria FD3DD2C355D2B701
gpg: usando pgp como modelo de confianza
gpg: Firma correcta de "Xilinx, Inc. (Xilinx Software signing key)" [absoluta]
gpg: firma modotexto, algoritmo de resumen SHA512, algoritmo de clave pública rsa4096

user@ubuntu-pc:~/Xilinx/Downloads/3_AlveoU50$ sha256sum -c xilinx-u50-gen3x16-xdma-201920_3-3_all.deb.tar.gz.
digests
xilinx-u50-gen3x16-xdma-201920_3-3_all.deb.tar.gz: La suma coincide
sha256sum: ATENCIÓN: 22: líneas tienen un formato erróneo
user@ubuntu-pc:~/Xilinx/Downloads/3_AlveoU50$ openssl dgst -sha256 xilinx-u50-gen3x16-xdma-201920_3-3_all.deb
.tar.gz
SHA256(xilinx-u50-gen3x16-xdma-201920_3-3_all.deb.tar.gz)= c9897afe9eff9822489e52b2ca7dc39c7a97d6a1b76abf9993
f19ba6f5a7b40b
user@ubuntu-pc:~/Xilinx/Downloads/3_AlveoU50$ cat xilinx-u50-gen3x16-xdma-201920_3-3_all.deb.tar.gz.digests |
 grep c9897afe9eff9822489e52b2ca7dc39c7a97d6a1b76abf9993f19ba6f5a7b40b
c9897afe9eff9822489e52b2ca7dc39c7a97d6a1b76abf9993f19ba6f5a7b40b *xilinx-u50-gen3x16-xdma-201920_3-3_all.deb.
tar.gz
```

Figura 120:Verificación de firmas, digests, hashes y archivos binarios de instalación de plataforma de despliegue para Alveo U50

c. Plataforma destino desarrollo (Development)

```
user@ubuntu-pc:~/Xilinx/Downloads/3_AlveoU50$ gpg -v --verify xilinx-u50-gen3x16-xdma-dev-201920.3-2784799_20
.04.deb.sig xilinx-u50-gen3x16-xdma-dev-201920.3-2784799_20.04.deb
gpg: Firmado el mié 25 nov 2020 20:16:35 CET
gpg:                usando RSA clave 5DB253CE971AD24F
gpg: usando subclave 5DB253CE971AD24F en vez de clave primaria FD3DD2C355D2B701
gpg: usando pgp como modelo de confianza
gpg: Firma correcta de "Xilinx, Inc. (Xilinx Software signing key)" [absoluta]
gpg: firma binaria, algoritmo de resumen SHA512, algoritmo de clave pública rsa4096

user@ubuntu-pc:~/Xilinx/Downloads/3_AlveoU50$ gpg -v --verify xilinx-u50-gen3x16-xdma-dev-201920.3-2784799_20
.04.deb.digests
gpg: cabecera de armadura: Hash: SHA512
gpg: nombre fichero original=''
gpg: Firmado el mié 25 nov 2020 20:16:38 CET
gpg:                usando RSA clave 5DB253CE971AD24F
gpg: usando subclave 5DB253CE971AD24F en vez de clave primaria FD3DD2C355D2B701
gpg: usando pgp como modelo de confianza
gpg: Firma correcta de "Xilinx, Inc. (Xilinx Software signing key)" [absoluta]
gpg: firma modotexto, algoritmo de resumen SHA512, algoritmo de clave pública rsa4096

user@ubuntu-pc:~/Xilinx/Downloads/3_AlveoU50$ sha256sum -c xilinx-u50-gen3x16-xdma-dev-201920.3-2784799_20.04
.deb.digests
xilinx-u50-gen3x16-xdma-dev-201920.3-2784799_20.04.deb: La suma coincide
sha256sum: ATENCIÓN: 22: líneas tienen un formato erróneo
user@ubuntu-pc:~/Xilinx/Downloads/3_AlveoU50$ openssl dgst -sha256 xilinx-u50-gen3x16-xdma-dev-201920.3-27847
99_20.04.deb
SHA256(xilinx-u50-gen3x16-xdma-dev-201920.3-2784799_20.04.deb)= 0c38f7c4603f0149dea4a484a5a32d485196abdf82ba7
d3ea01b6a6972233189
user@ubuntu-pc:~/Xilinx/Downloads/3_AlveoU50$ cat xilinx-u50-gen3x16-xdma-dev-201920.3-2784799_20.04.deb.dige
sts | grep 0c38f7c4603f0149dea4a484a5a32d485196abdf82ba7d3ea01b6a6972233189
0c38f7c4603f0149dea4a484a5a32d485196abdf82ba7d3ea01b6a6972233189 *xilinx-u50-gen3x16-xdma-dev-201920.3-278479
9_20.04.deb
```

Figura 121:Verificación de firmas, digests, hashes y archivos binarios de instalación de plataforma de desarrollo para Alveo U50

```
sudo apt install Xilinx-u50-genXxXX-xdma-dev-XXX.deb
```

Código 5: Comando para instalación de plataforma de desarrollo

8. Configuración del entorno de trabajo y variables de Vitis

```
source <Vitis_install_path>/Vitis/2020.2/settings64.sh

source /opt/xilinx/xrt/setup.sh

export PLATFORM_REPO_PATHS=/home/user/Xilinx/Downloads/3_AlveoU50/deploy
```

Código 6: Comandos para configuración de variables de entorno Vitis y XRT, junto a exportación de directorio de archivos de plataforma de despliegue

```
user@ubuntu-pc:~/Xilinx/Installation/Vitis/2020.2$ source settings64.sh
user@ubuntu-pc:~/Xilinx/Installation/Vitis/2020.2$ source /opt/xilinx/xrt/setup.sh
XILINX_XRT        : /opt/xilinx/xrt
PATH              : /opt/xilinx/xrt/bin:/home/user/Xilinx/Installation/Vitis_HLS/2020.2/bin:/home/user/Xilinx/
Installation/Vivado/2020.2/bin:/home/user/Xilinx/Installation/Model_Composer/2020.2/bin:/home/user/Xilinx/Inst
allation/Vitis/2020.2/bin:/home/user/Xilinx/Installation/Vitis/2020.2/gnu/microblaze/lin/bin:/home/user/Xilinx
/Installation/Vitis/2020.2/gnu/arm/lin/bin:/home/user/Xilinx/Installation/Vitis/2020.2/gnu/microblaze/linux_to
olchain/lin64_le/bin:/home/user/Xilinx/Installation/Vitis/2020.2/gnu/aarch32/lin/gcc-arm-linux-gnueabi/bin:/ho
me/user/Xilinx/Installation/Vitis/2020.2/gnu/aarch32/lin/gcc-arm-none-eabi/bin:/home/user/Xilinx/Installation/
Vitis/2020.2/gnu/aarch64/lin/aarch64-linux/bin:/home/user/Xilinx/Installation/Vitis/2020.2/gnu/aarch64/lin/aar
ch64-none/bin:/home/user/Xilinx/Installation/Vitis/2020.2/gnu/armr5/lin/gcc-arm-none-eabi/bin:/home/user/Xilin
x/Installation/Vitis/2020.2/tps/lnx64/cmake-3.3.2/bin:/home/user/Xilinx/Installation/Vitis/2020.2/aietools/bin
:/home/user/Xilinx/Installation/DocNav:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
:/usr/local/games:/snap/bin
LD_LIBRARY_PATH   : /opt/xilinx/xrt/lib:
PYTHONPATH        : /opt/xilinx/xrt/python:
```

Figura 122: Proceso de configuración de variables de entorno VItis y XRT

9. Selección de plataforma en Vitis: en la creación de un nuevo proyecto de aplicación desde el interfaz de Vitis, se puede verificar que la instalación de la plataforma es correcta, apareciendo en la lista de plataformas disponibles. En caso de no aparecer, buscar los ficheros descargados e importarlos manualmente.

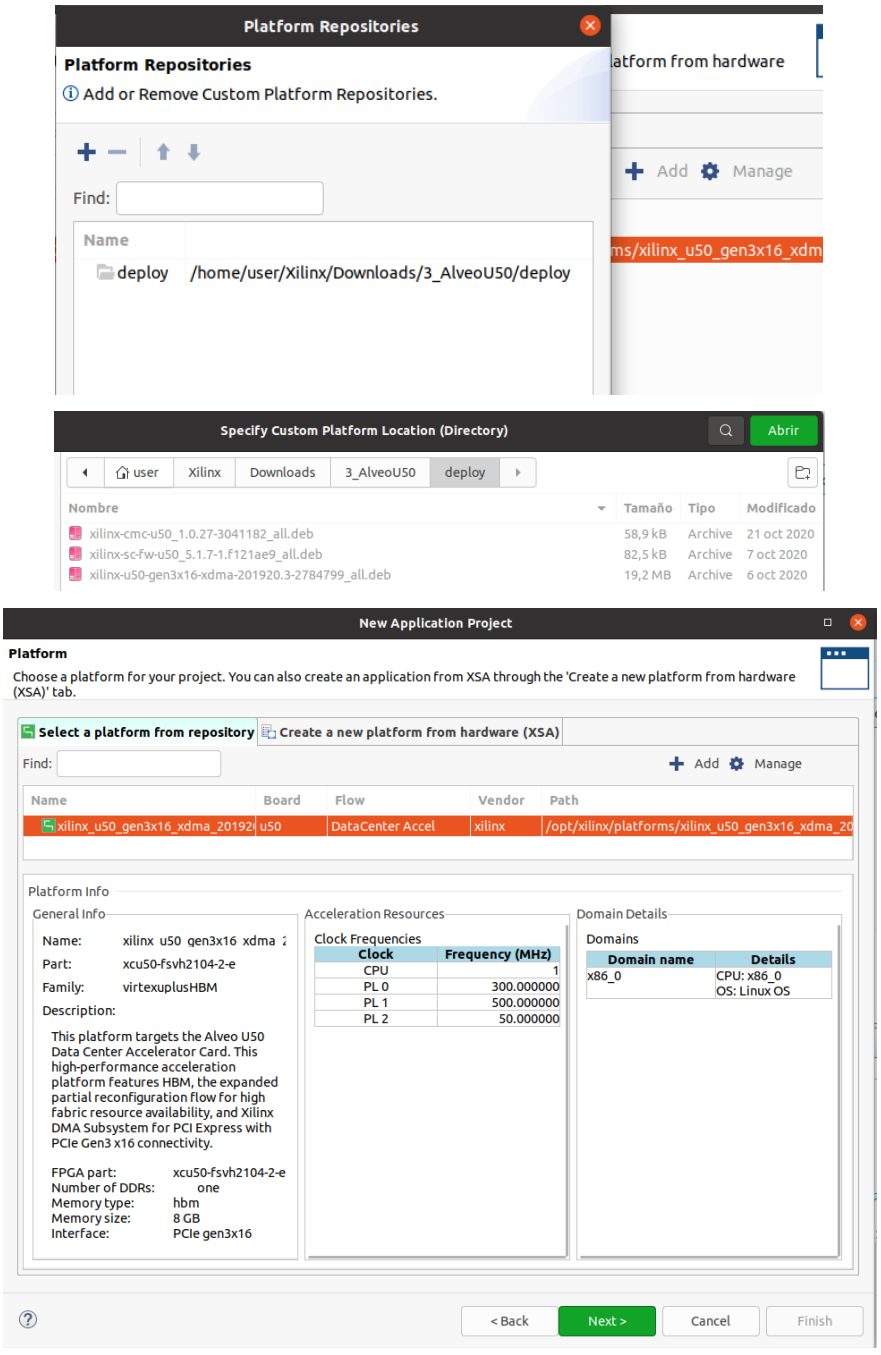


Figura 123: Selección de plataforma de destino en Vitis IDE, tras instalación correspondiente

10. Comprobación de instalación mediante importación de proyecto ejemplo:

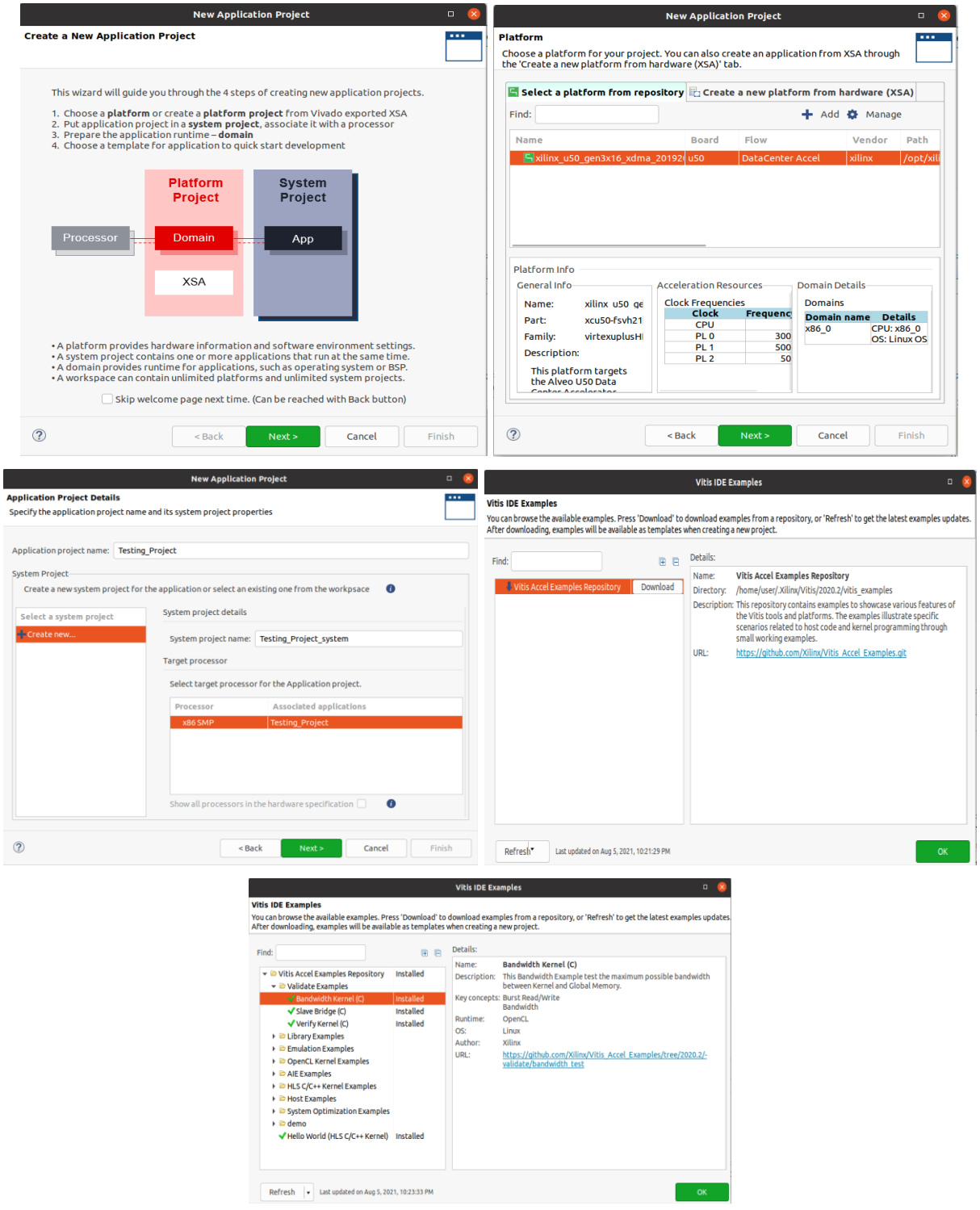


Figura 124: Creación de nuevo proyecto de aplicación, con plantillas de ejemplo de Xilinx

Figura 125: Selección de plantilla, modo activo de construcción de aplicación y salida del proceso de compilación.

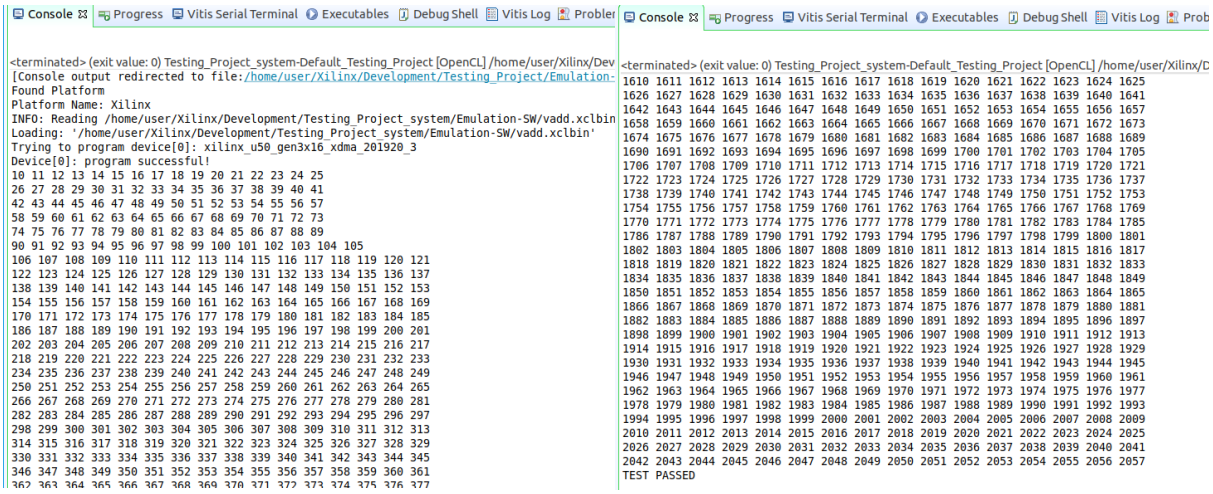


Figura 126: Ejemplo de salida de ejecución satisfactoria de proyecto de prueba importado

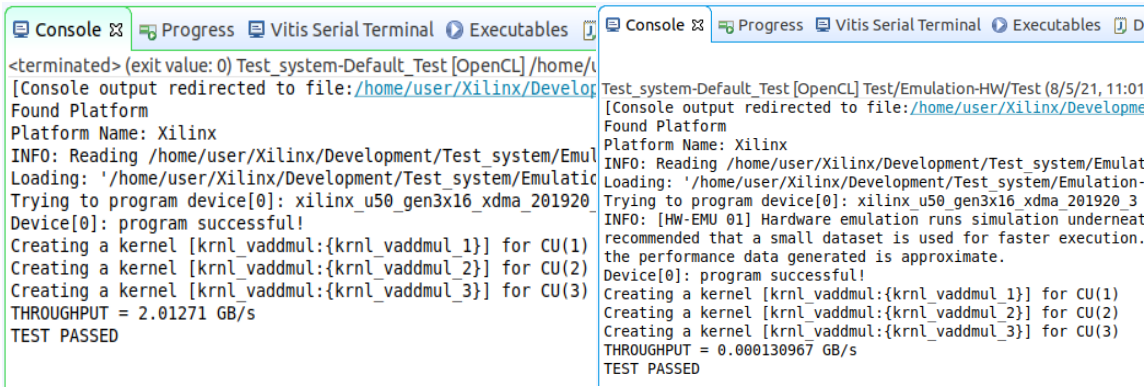


Figura 127: Diferencia de ejecución para modos de emulación Software y Hardware, respectivamente, observando la caída de rendimiento debida a la mayor cantidad de recursos necesarios en caso Hardware

Siendo capaces de ejecutar satisfactoriamente diversos proyectos de prueba, se puede verificar la correcta instalación del entorno, haciendo especial interés en la prueba de los componentes: plataforma de destino, programas y kernels C/C++, kernels OpenCL y comunicaciones AXI.

B.2. Actualización

Para este proceso, Xilinx cuenta con un Software denominado “Information Center” que proporciona notificaciones ante actualizaciones de los diferentes componentes instalados. Con un simple click, es posible actualizar cada uno de los componentes de manera individual, incluso instalar nuevas funcionalidades compatibles con el Software ya instalado.

Figura 128: “Information Center” con notificaciones de actualizaciones y nuevos componentes disponibles

B.3. Desinstalación.

a. Vitis, Documentación e Information Center: Iconos disponibles en dock de aplicaciones.

Figura 129: Iconos de desinstalación de Vitis IDE, DocNav e Information Center

b. XRT:

```
sudo apt remove xrt

… Desinstalando xrt (2.8.743)

… -------- Uninstall Beginning --------

Module:  xrt

Version: 2.8.743

Kernel:  5.8.0-48-generic (x86_64)

-------------------------------------

… DKMS: uninstall completed.

… Done.

Cleaning up …
```

Código 7: Comando para desinstalación de XRT

c. Plataformas:

```
sudo apt remove xilinx-u50-genXxXX-XXX-xdma-dev
```

Código 8: Comando para desinstalación de plataforma de desarrollo Alveo U50

d. Librerías adicionales:

```
sudo apt install synaptic

sudo synaptic

#Buscar xilinx y xrt y eliminar las dependencias instaladas
```

Código 9: Comandos y pasos para desinstalación de librerías adicionales instaladas por Vitis, Vivado y XRT

e. Limpieza final:

```
sudo apt-get autoremove

sudo apt-get autoclean
```

Código 10: Comandos para limpieza de dependencias y herramientas restantes

C. Ampliación de síntesis de fundamentos de aceleración Hardware mediante plataforma Xilinx Vitis y tarjetas Alveo.

C.1. Metodología para aceleración de aplicaciones.

La metodología de aceleración de aplicaciones mediante el uso de la plataforma Vitis, se puede agrupar en dos grandes fases, seguidas de una última fase complementaria, necesaria para verificar y analizar el funcionamiento y rendimiento de la aplicación acelerada. El proceso completo es iterativo y debe ser seguido hasta alcanzar los objetivos de rendimiento y latencia deseados.

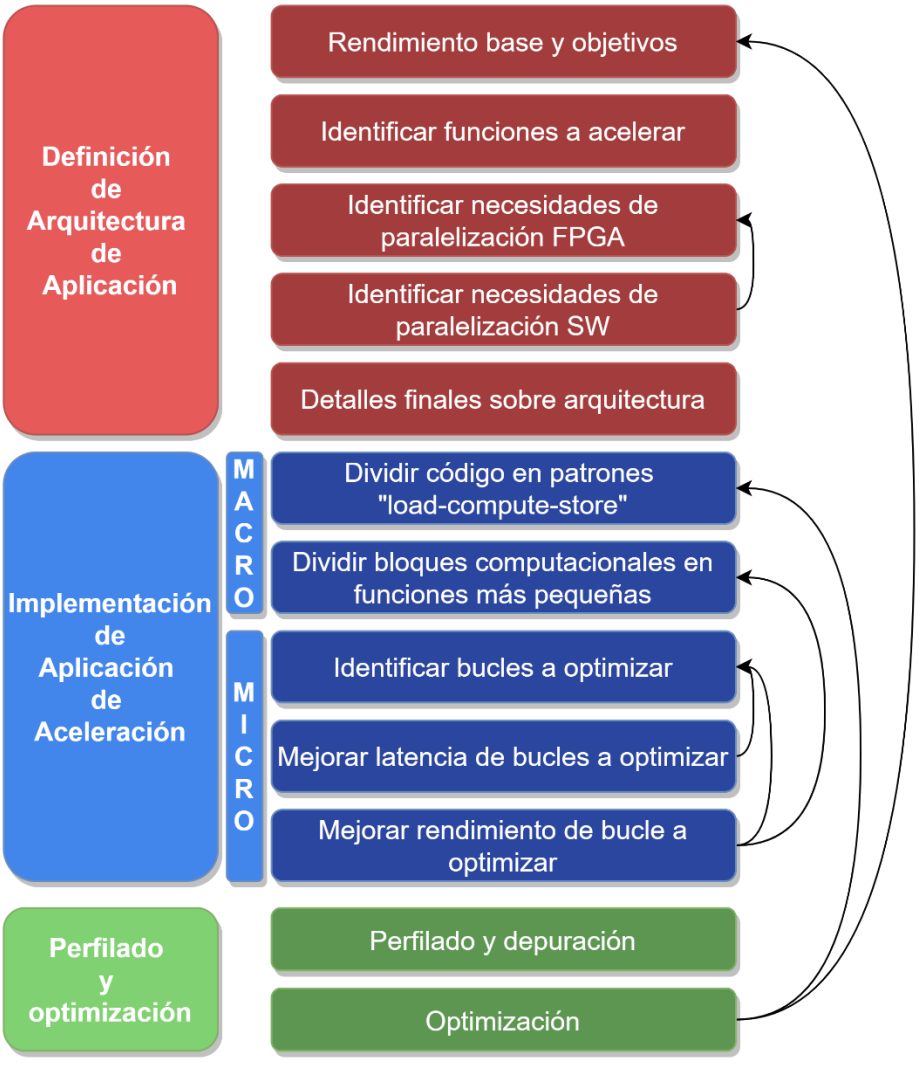


Figura 130: Esquema metodología de aceleración de aplicaciones mediante Xilinx Vitis

C.1.1. Definición de arquitectura de la aplicación.

En esta fase se toman las decisiones clave sobre la arquitectura de la aplicación, definiendo qué funciones software deben ser mapeadas en los kernels, qué grado de paralelismo es necesario y cómo debe ser llevado a cabo, para alcanzar una serie de objetivos de rendimiento establecidos de manera inicial.

La metodología planteada para esta fase sigue los siguientes pasos:

1. Rendimiento base y definición de objetivos:

   Se comienza con una ejecución para determinar el tiempo de ejecución y la capacidad de rendimiento, identificando posibles cuellos de botella, en las plataformas de despliegue. Estos datos deben ser generados tanto para la aplicación completa, como para las principales funciones internas. Para ello, se hace uso de herramientas de perfilado, obteniendo datos como el número de funciones que se ejecutan, cuántas veces lo hacen o cuales consumen más recursos, entre otros. Esto permite una primera aproximación para determinar qué puntos del flujo deben ser acelerados. Los parámetros a definir en esta sección son:

   - Tiempo de ejecución total y parcial de la aplicación y los bloques funcionales que la componen.
   - Capacidad o throughput de procesamiento de datos. Se calcula dividiendo el volumen de datos procesados entre el tiempo de ejecución de la función correspondiente.
   - Máxima capacidad o throughput alcanzable, basado en las tecnologías utilizadas en los dispositivos de despliegue. En el caso de Alveo, el factor limitante es principalmente la comunicación PCIe, de forma que es necesario saber el límite de esta tecnología. Para ello, existen herramientas de la propia plataforma Vitis, para obtener los datos necesarios y establecer el máximo potencial alcanzable de aceleración. Es un proceso fundamental ya que PCIe se ve influenciado por numerosos factores como la placa base, la plataforma de despliegue o los tamaños de transferencias.
   - Definición de los objetivos generales de la aceleración, basados en los tiempos y capacidades iniciales, buscando un valor realista alcanzable mediante la

aceleración, inferior al límite superior definido por la tecnología PCIe comentada anteriormente.

2. Identificación de funciones a acelerar:

   - Identificación de cuellos de botella de rendimiento: En ejecuciones puramente secuenciales, la identificación de cuellos de botella es bastante simple, perfectamente identificables con informes de perfilado. Sin embargo, la mayoría de las aplicaciones reales implican múltiples hilos y es necesario analizar el paralelismo en dicha ejecución. Para buscar candidatas a la aceleración, es necesario analizar el rendimiento de la aplicación completa y no únicamente el de funciones individuales. Un claro ejemplo es el mostrado en la Figura 131, donde se aprecia que no tiene sentido acelerar A2, independientemente del índice de mejora, sino que es importante centrar esfuerzos en acelerar A1, junto con B1, B2 y/o B3, de forma que se reduzcan los tiempos de ejecución parciales y, consiguientemente, el tiempo total.

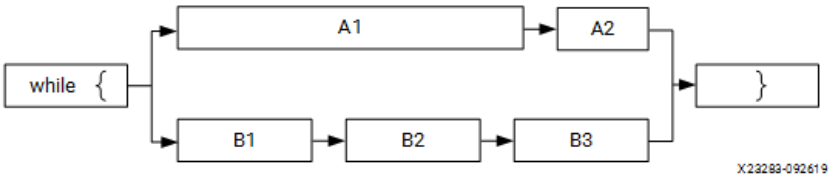


Figura 131: Ejemplo identificación de cuellos de botella en ejecución paralela de funciones (Xilinx, Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393), 2021)

   - Identificación de potencial de aceleración: Una función que suponga un cuello de botella no tiene por qué presentar un mayor rendimiento si se ejecuta sobre el Hardware de aceleración. Generalmente, es necesario realizar un análisis particular de la función correspondiente, para determinar el potencial real para la aceleración Hardware. Sin embargo, sí que existen una serie de factores que ayudan a establecer un punto de partida para determinar si las funciones pueden tener potencial de aceleración o no:

     1. Complejidad computacional: Indica el número básico de operaciones necesarias para ejecutarse. En dispositivos configurables, la aceleración se logra mediante la paralelización de tareas y de rutas de datos, de forma que, las mejores candidatas,

son aquellas funciones que tienen numerosas operaciones, que deben ser realizadas sobre cada muestra, para producir una muestra de salida determinada.

2. Intensidad computacional: Relación entre número total de operaciones y cantidad de datos de entrada y salida. Las funciones con un gran número de operaciones por volumen de datos son mejores candidatas.
3. Perfil de acceso a localización de datos: Es importante tener en mente los conceptos de reutilización de datos, localización espacial (refleja distancias entre accesos a memoria consecutivos) y localización temporal (refleja el número medio de operaciones necesarias para accesos a direcciones de memoria durante la ejecución). Cuanto menor sea el valor de localización espacial y localización temporal, mayor potencial de aceleración, ya que permite almacenar más eficientemente en la caché del acelerador, los datos utilizados.
4. Relación de la capacidad o rendimiento particulares, frente a los valores totales de la aplicación: Debido a que las aplicaciones aceleradas definen sistemas distribuidos basados en multi procesos, el rendimiento global, no podrá exceder el rendimiento de la función más lenta de todas. Esto permite definir el máximo potencial de aceleración como la ratio entre el rendimiento de dicha función más lenta y el rendimiento global. Para el caso de Alveo, al tener comunicaciones basadas en PCIe, se añade esta función de transferencia, que puede suponer un cuello de botella, o al menos, definir un máximo potencial de aceleración alcanzable en caso de no serlo.

3. Identificación de necesidades de paralelización en FPGA:
    - Estimar rendimiento Hardware sin paralelización: Para calcular una estimación, se divide la frecuencia de reloj del kernel, entre la intensidad computacional.

- Determinar grado de paralelización necesario: Una vez conocida la estimación del rendimiento en Hardware, se puede obtener una estimación inicial entre rendimiento Hardware y Software mediante el cociente entre ambos valores, para posteriormente definir el grado de paralelización requerido (cociente entre rendimiento total y rendimiento Hardware. Una vez conocido este valor, existen varias formas de lograr la paralelización:

    1. Determinar muestras a ser procesadas en paralelo para el ancho de rutas de datos definido: Mediante la creación de rutas de datos de mayor ancho que permitan procesar un mayor número de muestras por unidad de tiempo, permitiendo una mejora de rendimiento reduciendo la latencia (tiempo de ejecución) de la función.

    2. Determinar número de unidades de cómputo o kernels que deben ser instanciados: Si el ancho de ruta de datos no es suficiente, es posible añadir más instancias de kernels que se ejecuten en paralelo, mejorando el rendimiento al ejecutar la función un mayor número de veces por unidad de tiempo. El rendimiento escala linealmente con el número de instancias de kernel, siempre que el host sea capaz de mantener ocupados a dichos kernels, proporcionando y recuperando los datos necesarios a tiempo.

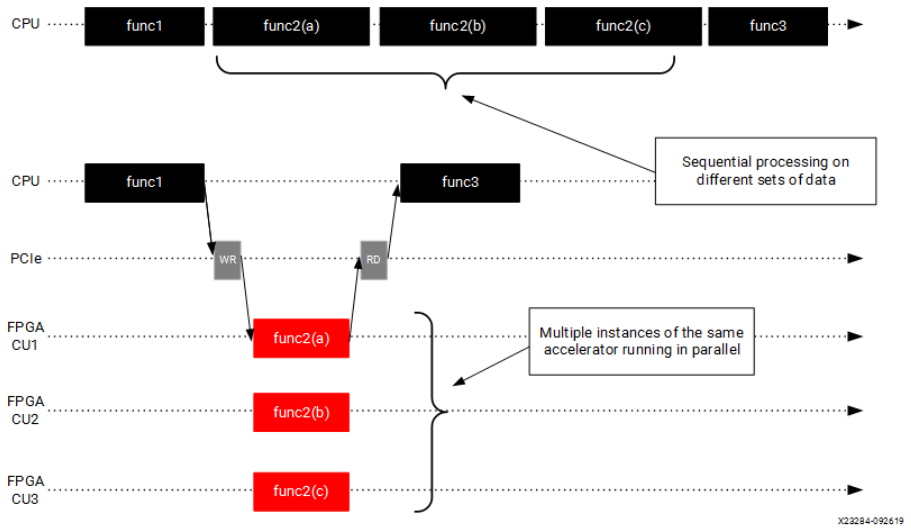


Figura 132: Ejemplo de aceleración mediante múltiples kernels en paralelo (Xilinx, Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393), 2021)

4. Identificación de necesidades de paralelización en aplicación Software: Mientras el dispositivo Hardware y sus kernels son diseñados para proporcionar el potencial del paralelismo, la aplicación Software debe ser diseñada para tomar ventaja de ello. Para ello, es necesario que el programa host cumpla con una serie de factores, mostrados en la Figura 133.

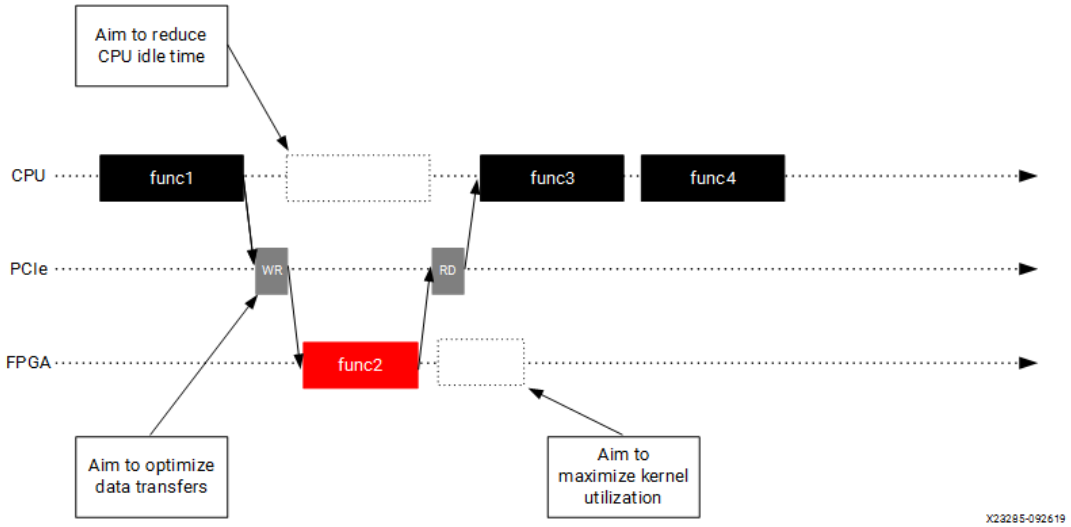


Figura 133: Factores a cumplir por programa host para correcto desarrollo de aplicación acelerada (Xilinx, Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393), 2021)

Estos aspectos se centran en 3 puntos:

- Minimización de tiempos de espera en CPU: Mientras el host espera a recibir los datos procesados por la FPGA, debe ser capaz de realizar en paralelo aquellas funciones que no dependan de dichos datos, mejorando el rendimiento global, al evitar tiempos de espera no productivos.
- Maximización del uso de los aceleradores FPGA: Evitar tiempos de espera entre ejecuciones de kernels, teniendo los datos disponibles y enviando las órdenes de ejecución, instantes previos a la finalización de la tarea actual, indicando al kernel la siguiente instrucción. Esto permite que la pueda llevar a cabo en el instante exacto en que termina con la tarea actual.

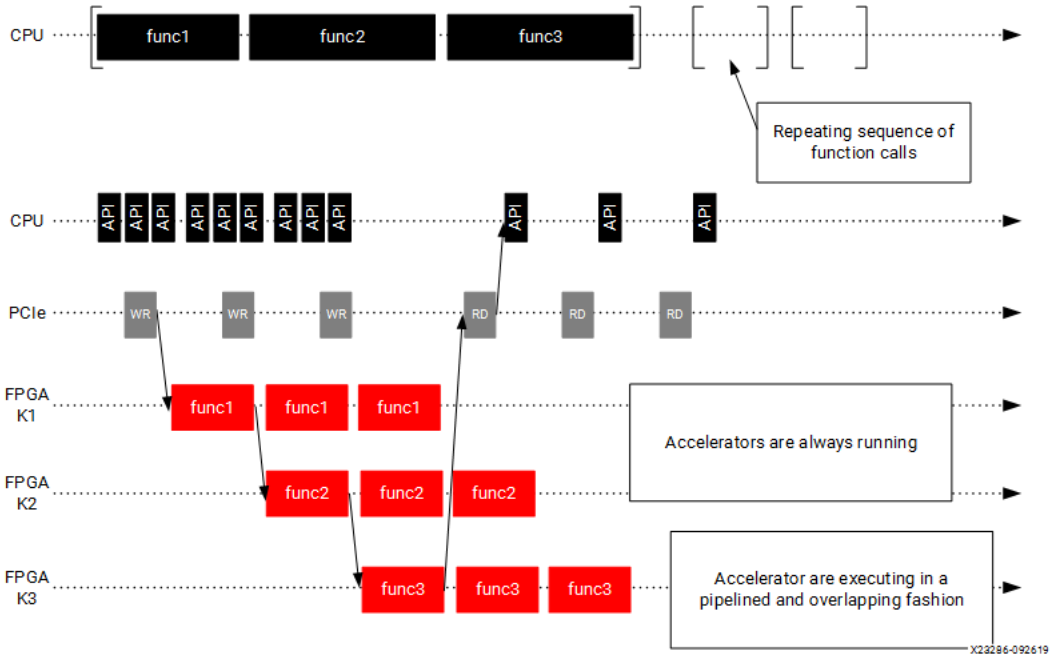


Figura 134: Ejemplo de maximización de uso de aceleradores en FPGA (Xilinx, Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393), 2021)

- Optimización de transferencias de datos CPU-FPGA y viceversa: Al ser necesaria la transmisión de los datos mediante PCIe, es inevitable la introducción de latencias no deseadas. Sin embargo, es posible minimizar el impacto de esta latencia, realizando la operación en el momento exacto, solapando transferencias y ejecución de kernels. Otro método utilizado es el uso de tamaños de buffer adecuados, adaptándolos a los requerimientos de la aplicación, de forma que, cuanto mayor sea el buffer, mejor rendimiento tendrá la transferencia. Por norma general, una buena estrategia, si la aplicación lo permite, es el uso de buffers de tamaño igual o superior a 1 MB. Es por ello que Xilinx recomienda agrupar conjuntos de datos en un buffer común para alcanzar la máxima capacidad posible.

Una vez analizados todos estos puntos, es posible conocer una primera aproximación de los requerimientos de aceleración y qué puntos son críticos para lograr los objetivos deseados. En este punto se puede realizar un nuevo diagrama de ejecución a lo largo del tiempo, para analizar los aspectos estudiados y determinar si es posible pasar a la etapa final, o si es necesario volver al punto 3 para volver a contemplar nuevos requisitos de aceleración.

5. Refinar detalles sobre arquitectura: Antes de comenzar con el desarrollo de la aplicación, es necesario realizar unos pasos finales derivados de las decisiones de alto nivel tomadas en los pasos previos.
    - Definir frontera de kernel de aceleración: En el caso de introducir múltiples instancias de kernels o unidades de computación, hay que tener en cuenta el número máximo de puertos que pueden ser utilizados según la plataforma de destino utilizada. Cada instancia supone consumo de puertos I/O, ancho de banda y recursos, de forma que es un aspecto importante. De cara a mejorar el rendimiento del ancho de banda de los puertos I/O, es mejor introducir varios motores dentro del mismo kernel (siempre que la ruta de datos no requiera la totalidad del ancho del puerto), que crear varias instancias de kernels. Sin embargo, esto supone codificar explícitamente la multiplexación, en el propio código fuente, suponiendo un mayor esfuerzo en el desarrollo. Es necesario analizar el compromiso necesidad-esfuerzo y actuar en función de ello.
    - Decidir localización y conectividad del acelerador: Tras definir los límites o la frontera del kernel, se conoce el número de kernels que deben ser instanciados y, por tanto, cuantos puertos son conectados a los recursos de la memoria global. Conociendo estos aspectos y teniendo en cuenta los recursos de la plataforma de destino, es posible mejorar el rendimiento garantizando un balanceo de transferencia de datos entre las diferentes regiones lógicas del dispositivo, mediante un mapeo adecuado entre los puertos del kernel y los bancos de memoria. Una posibilidad, es el uso de múltiples bancos de memoria para realizar un balance de carga de datos. Esta funcionalidad está disponible mediante un simple cambio en compilación.

A partir de este punto, se conoce toda la información necesaria para comenzar con el desarrollo e implementación de los kernels y el programa host, para finalmente construir la aplicación completa.

C.1.2. Implementación del acelerador: kernel y host.

En esta fase, se implementa la aplicación en sí, basando el desarrollo en la fase de creación de la arquitectura, buscando alcanzar los objetivos de rendimiento planteados y siguiendo los procesos definidos para ello. Requiere estructurar el código adecuadamente, definiendo tanto los programas host, como kernel, estableciendo el modo en que se transfiere la información entre ambos, y determinando los tiempos y secuencias de ejecución de cada elemento particular. Todo ello con el eje central focalizado en lograr el objetivo de aceleración de rendimiento definido en la primera etapa. Se trata de un proceso iterativo, que puede requerir numerosas fases de optimización hasta alcanzar el objetivo. Para lo referido a esta memoria, se centra el estudio en los modelos de kernel C/C++, de forma que el compilador Vitis genera las arquitecturas de Hardware optimizadas para las funcionalidades y los algoritmos desarrollados en dicho lenguaje, transformando el código en diseño RTL, que puede ser mapeado sobre la estructura del dispositivo de aceleración.

La metodología planteada para esta fase sigue los siguientes pasos:

- Macroestructura:

1. Dividir el código en patrones load-compute-store (cargar-computar-guardar):

   Un kernel es básicamente una ruta de datos personalizada (optimizada para la funcionalidad deseada) y que presenta una red de almacenamiento y movimiento de datos asociada. Esta red se encarga de mover los datos hacia dentro y hacia fuera del kernel, así como a través del mismo.

   Teniendo en cuenta que los accesos a memoria global son costosos y que el ancho de banda es limitado, la metodología de desarrollo recomienda establecer un patrón “load-compute-store” para la estructura del código del kernel. Esto se traduce en la creación de varias funciones de alto nivel con interfaces para los parámetros del kernel, estableciendo tres subfunciones para carga, computación y guardado, y una serie de arrays internos para pasar los datos entre dichas funciones. Gracias a esta estructuración, se posibilita el pipelining de las tareas, de forma que se puedan ejecutar de manera simultánea.

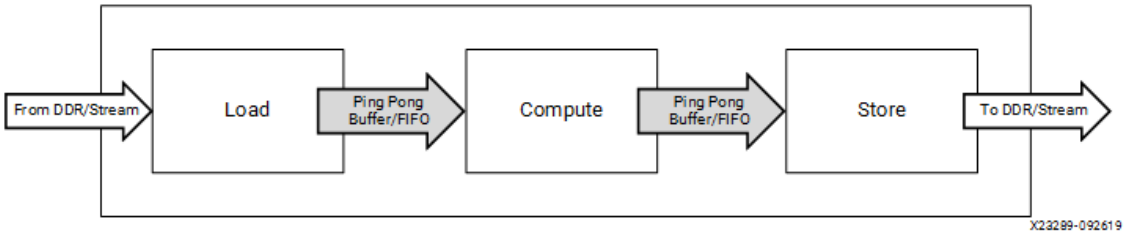


Figura 135: Esquema de bloques siguiendo patrón "load-compute-store" (Xilinx, Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393), 2021)

a. Crear funciones de alto nivel con la interfaz deseada: El punto inicial es una función de alto nivel cuyos parámetros coincidan con la interfaz deseada. Los parámetros de entrada deben pasarse como escalares y los bloques de datos de entrada o salida, como punteros. Para la finalización de la interfaz, se recurre a las directivas o pragmas del compilador.

b. Codificar funciones load y store: Son las funciones encargadas de introducir los datos de entrada en el kernel y de extraer los datos de salida del mismo, respectivamente. Es importante optimizar estas funciones para adecuar las transmisiones a la ejecución de los kernels. La distribución de la memoria global debe coincidir con la distribución de la memoria de la aplicación Software. Estos conceptos se traducen en:

   i. Debe coincidir el ancho del puerto del kernel con el ancho de la ruta de datos en la función de computación.

   ii. Uso de transferencias en ráfagas, evitando accesos atómicos (los más costosos).

   iii. Minimizar el número de transmisiones de datos desde la memoria global.

c. Codificar funciones compute: Son las funciones de computación en las que se realiza el procesamiento de los datos. Las primeras versiones deben estar centradas en una implementación de alto nivel, optimizando la movilidad de los datos, de forma que se tenga la funcionalidad correcta, con unas interfaces adecuadas.

d. Conectar funciones siguiendo el estilo de flujo de datos o dataflow: Mediante el uso de variables y arrays estándar C/C++, para conectar las interfaces de alto nivel de las funciones load-compute-store. Para la conexión, es necesario tener en cuenta la forma canónica del compilador HLS:

   i. Los datos sólo deben transmitirse hacia adelante (sin realimentaciones).

   ii. Cada conexión debe tener un único productor y un único consumidor.

   iii. Sólo las funciones load y store deben acceder a la interfaz primaria del kernel.

2. Particionar los bloques computacionales en funciones más pequeñas:

Se basa en refinar la función principal de computación, descomponiéndose en elementos funcionales más simples.

a. Descomponer para identificar objetivos de rendimiento: Como el cuello de botella, según el sistema de flujo de datos planteado, consiste en la función más lenta, esta tarea permite detectar y optimizar componentes funcionales individuales de manera eficaz, logrando encontrar los puntos con mayor potencial para aumentar el rendimiento global de manera más sencilla.

b. Búsqueda de funciones con bucles de anidación única: Si en una función hay varios bucles ejecutados secuencialmente, existe una pérdida de rendimiento por dicha secuencialidad. Sin embargo, si estos bucles son introducidos en funciones secuenciales, el compilador HLS es capaz de aplicar la optimización y generar implementaciones que permiten pipelining y solapamiento de ejecución de tareas. Solventando los problemas y mejorando el rendimiento general. Idealmente, deben introducirse los bucles secuenciales en funciones individuales, de forma que la de más bajo nivel, se componga de un bucle único perfectamente anidado.

c. Conectar funciones siguiendo el estilo de flujo de datos “canonical form”: Mismas reglas que para conexión load-compute-store, conectando bloques funcionales de bajo nivel, en bloques de mayor nivel.

- Microestructura:

1. Identificar bucles a optimizar:

    1. Calcular la latencia objetivo para cada bucle: Se obtiene como el cociente entre el volumen de datos procesados y el tiempo de ejecución.
    2. Generar informe HLS: Mediante el compilador Vitis HLS se pueden obtener este tipo de informes que indican el rendimiento y la latencia de las funciones y bucles. La latencia de un bucle se puede calcular de la siguiente manera:

    $$Latency_{Loop} = (Steps + II \times (TripCount - 1)) \times ClockPeriod$$

    Donde:

    - Steps: duración de una iteración simple del bucle, medida en número de ciclos de reloj.
    - TripCount: número de iteraciones en el bucle
    - II o Initiation Interval: número de ciclos de reloj entre el inicio de dos iteraciones consecutivas. Cuando un bucle no usa pipeline, II es igual Steps.

    3. Identificar bucles que excedan objetivo de latencia: Mediante los datos calculados y los proporcionados por los informes, es posible detectar qué funciones no cumplen con los requerimientos y afrontar la vía de reducción de latencia correspondiente. Las tres vías principales de mejora de la latencia de bucles, dado un ciclo de reloj determinado, son:

        i. Reducir el número de “Steps” o pasos por bucle, de forma que se requiera menos tiempo por iteración.
        ii. Reducir el “Trip Count”, de forma que el bucle tenga menos iteraciones.
        iii. Reducir “Initiation Interval”, de forma que las iteraciones puedan comenzar más rápido.

2. Mejorar latencia de bucle a optimizar:

a. "Unroll" de bucles: Esta técnica "desenrolla" el bucle de forma que varias iteraciones del mismo puedan ser ejecutadas conjuntamente, reduciendo el "Trip Count" global del bucle. Un ejemplo de ello puede verse en la Figura 136 donde, aplicando un factor de unroll, se consigue procesar en paralelo un mayor número de muestras por iteración, aunque requiriendo una ruta de datos también mayor. Se recomienda comenzar desde los bloques de más bajo nivel e ir subiendo según sea necesario (siempre hacer un "unroll" teniendo un objetivo en mente y nunca a ciegas). Este proceso permite ensanchar la ruta de datos resultante según el factor correspondiente, de forma que, generalmente, se incremente el ancho de banda al poder procesar más muestras en paralelo. Esto tiene dos implicaciones:

   i. El ancho de la función I/O debe coincidir con el de la ruta de datos y viceversa.

   ii. No se obtienen beneficios al aplicar "loop unrolling" más allá del punto en que los requerimientos I/O superan el tamaño máximo de los puertos del kernel (512 bits o 64 bytes).

   Esta técnica modifica los requerimientos I/O y los patrones de acceso de la función, de forma que, si una función hace accesos a arrays (como es en la mayoría de los casos), es fundamental asegurar que la ruta de datos resultante pueda acceder a la totalidad de los datos necesarios en paralelo.

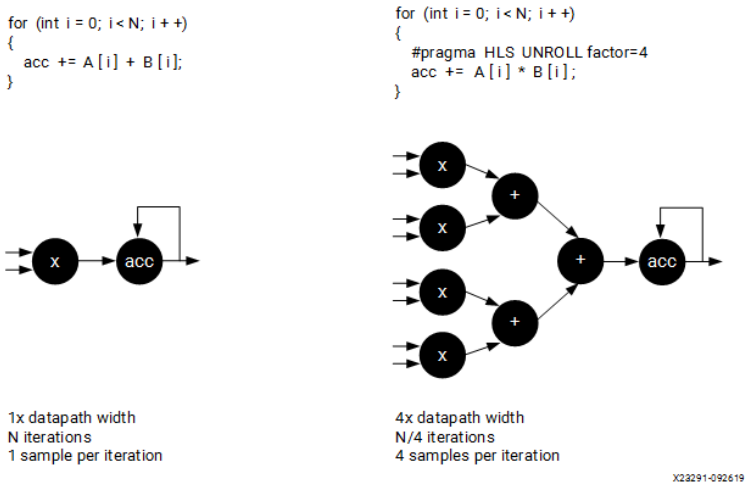


Figura 136: Ejemplo de "loop unrolling" con factor 4 para reducir TripCount (Xilinx, Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393), 2021)

b. Partición de arrays: Cuando el “unroll” no es suficiente, normalmente se debe a cuellos de botella debidos a accesos a memoria. Por defecto, el compilador Vitis HLS mapea arrays extensos a recursos de memoria con un tamaño de palabra igual al tamaño de un elemento del array. En la mayor parte de los casos, este valor debe ser modificado cuando se aplica “unrolling”. Mediante las directivas del compilador adecuadas, se pueden particionar y modificar el tamaño de los arrays, de forma que se cree una estructura de memoria que permita el nivel de paralelismo en acceso requerido.

Por norma general, con estos dos procesos es más que suficiente para alcanzar los objetivos deseados, de forma que se puede pasar al siguiente bucle a optimizar. En caso de no cumplir con los requisitos, se puede recurrir a las técnicas indicadas a continuación.

3. Mejorar rendimiento de bucle a optimizar:

Cuando mejorar la latencia reduciendo el “Trip Count” no es suficiente, es necesario intentar reducir el “Initiation Interval” o II.

a. Eliminar contenciones I/O: Aparecen cuando un puerto I/O determinado de los recursos internos de memoria debe ser accedido más de una vez por iteración del bucle. Es importante analizar si estos accesos son necesarios o pueden ser eliminados. Para evitar este problema se suelen utilizar dos métodos:

   i. Creación de estructuras caché internas

   ii. Reconfiguración de I/O y memorias

b. Eliminar dependencias de bucles: Existen dependencias cuando una iteración, depende de valores calculados en alguna otra iteración. Existen técnicas diferenciadas para los casos en que las dependencias sean sobre variables escalares o sobre arrays.

c. Técnicas avanzadas: Existen técnicas más complejas basadas en pragmas HLS, disponibles en la documentación de Xilinx. Sin embargo, una combinación de varias reducciones de II y de factores de “unroll” es más que suficiente para alcanzar la mayoría de los objetivos de aceleración.

D. Código de proyectos implementados.

D.1. Bloom Filter

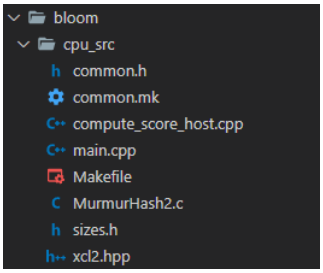


Figura 137: Esquema código fuente "Bloom Filter" (ficheros implementación CPU)

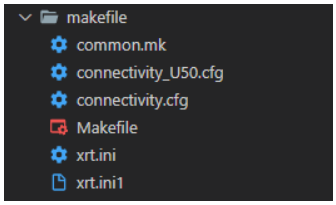


Figura 138: Esquema código fuente "Bloom Filter" (ficheros de configuración y compilación)

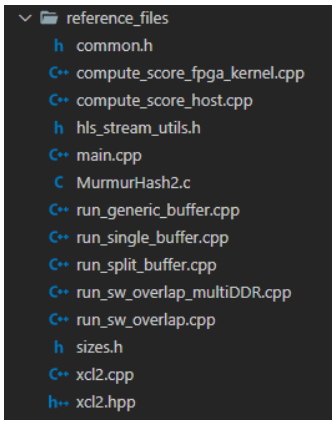


Figura 139: Esquema código fuente "Bloom Filter" (ficheros Host CPU y kernel FPGA)

Los ficheros se encuentran en la carpeta “bloom” y se dividen en tres grandes bloques:

- cpu_src: contiene los ficheros para la implementación puramente sobre CPU
- makefile: con los ficheros para los comandos make y la configuración y compilación de los programas Host y Kernel
- reference_files: contiene el código fuente para la implementación sobre el paradigma Host-FPGA, para los diferentes modos de compilación y ejecución

D.2. Cholesky Decomposition

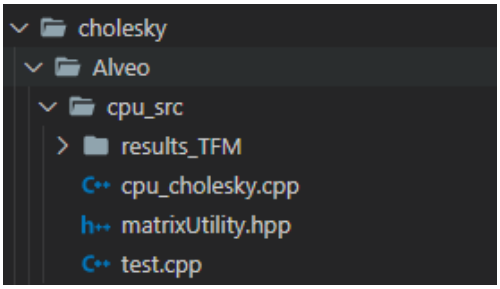


Figura 140: Esquema código fuente "Cholesky Decompositionr" (ficheros implementación CPU)

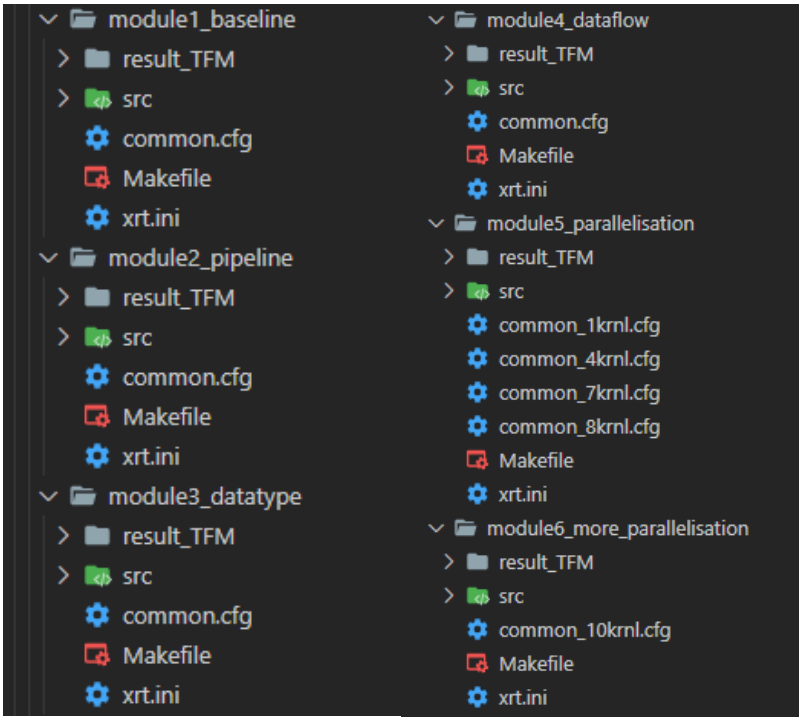


Figura 141: Esquema código fuente "Cholesky Decompositionr" (ficheros aceleración Alveo)

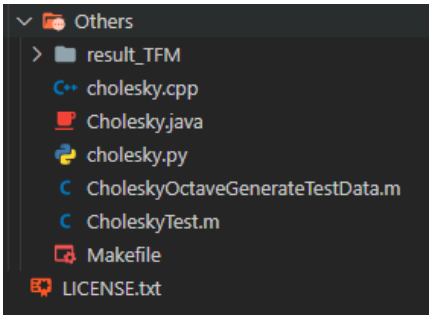


Figura 142: Esquema código fuente "Cholesky Decompositionr" (ficheros de alternativas de implementación)

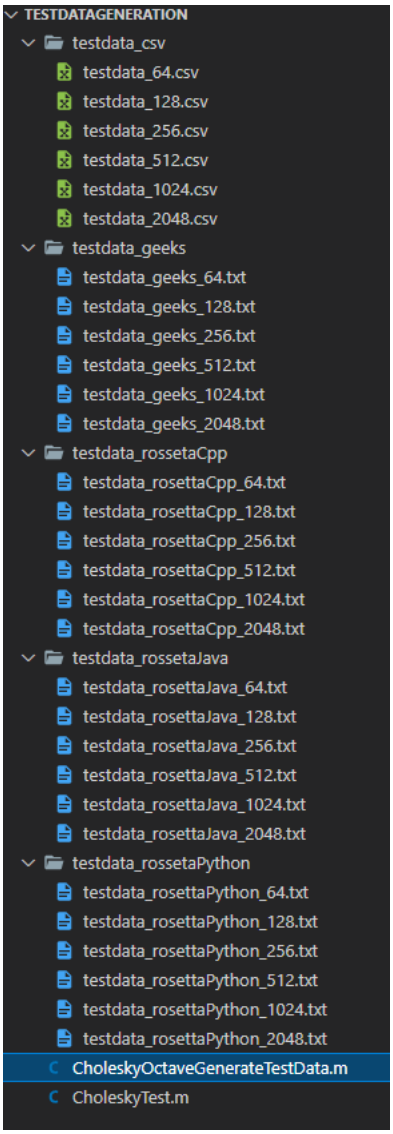


Figura 143: Esquema código fuente "Cholesky Decompositionr" (ficheros de generación y validación de datos de entrada, junto a ficheros de datos generados)

Los ficheros se encuentran en la carpeta “cholesky” y se dividen en dos grandes bloques, con diferentes subdirectorios:

- Alveo: contiene los proyectos completos para cada fase de aceleración sobre FPGA
    - cpu_src: implementación puramente en CPU (punto de partida de aplicación)
    - moduleX: conjunto de ficheros de cada fase ‘X’ de aceleración, partiendo de la implementación puramente en CPU, hasta el paradigma Host-FPGA con

múltiples kernels implementados en paralelo. En el directorio de cada fase se encuentran los ficheros de configuración y compilación, mientras que el código fuente del Host y el kernel, se encuentran dentro del subdirectorio “src”.

- Others: contiene los ficheros para las diferentes implementaciones alternativas del estudio: C++ (*.cpp), Java (*.java), Python (*.py) y Octave (*.m). Además, se incluye un fichero “Makefile” para automatización de compilación y configuración, así como para ejecución de pruebas. También se incluye un script de Octave (“CholeskyOctaveGenerateTestData.m”) para la generación de matrices Hermíticas positivamente definidas, del tamaño deseado, que sirvan de entrada para las diferentes implementaciones. El árbol de ficheros de salida de los datos generados, se corresponde con el mostrado en la Figura 143.

En cada uno de los directorios, se incluyen los resultados de las pruebas realizadas para cada implementación, dentro del subdirectorio “result_TFM”. Se incluye un fichero para cada ejecución y cada tamaño de las matrices de entrada ‘N’ (xxxx_N.log)