Implementación del algoritmo SUnSAL (Spectral Unmixing by Variable Splitting and Augmented Lagrangian) para desmezclado espectral en imágenes hiperespectrales de la superficie terrestre mediante síntesis de alto nivel sobre hardware reconfigurable.

Implementation of the SUnSAL (Spectral Unmixing by Variable Splitting and Augmented Lagrangian) algorithm for spectral unmixing in hyperspectral images of the Earth's surface through high level synthesis on reconfigurable hardware.

TRABAJO DE FIN DE GRADO DEL GRADO EN INGENIERÍA INFORMÁTICA

Enrique Rey Gisbert

Directores:
Daniel Báscones García
Carlos González Calvo

Facultad de Informática
Universidad Complutense de Madrid

Madrid, a 29 de mayo de 2023

A mis directores, Carlos González Calvo y Daniel Báscones García, por su ayuda y cercanía. A Daniel Mozos Muñoz, por su generosidad y respaldo. A mi familia, por todo lo que han hecho por mí y por seguir apoyándome a cada paso. A todas las personas que han contribuido para que este trabajo sea una realidad. Gracias.

Resumen

En este trabajo de fin de grado se ha realizado una implementación de un algoritmo de desmezclado espectral lineal conocido como SUnSAL (*Spectral Unmixing by Variable Splitting and Augmented Lagrangian*) sobre tarjetas de aceleración basadas en FPGAs a través de síntesis de alto nivel (HLS). Para ello, se ha utilizado el entorno de desarrollo *Xilinx Vitis Unified Software Platform*, a través del cual se ha acelerado por hardware el algoritmo con el objetivo de optimizar el rendimiento. Finalmente, y tras analizar en detalle los datos de latencia y consumo de hardware, se ha ejecutado el algoritmo en una tarjeta de aceleración *Xilinx* Alveo U250 utilizando los datos de una imagen hiperespectral capturada por el sensor hiperespectral AVIRIS (*Airborne Visible / Infrared Imaging Spectrometer*) y una librería espectral de *endmembers* del servicio geológico de Estados Unidos (USGS).

En primer lugar, se da una introducción a los conceptos básicos del análisis hiperespectral como disciplina científica, explorando sus múltiples aplicaciones y exponiendo de manera rigurosa la estructura de las imágenes hiperespectrales. A continuación, se presenta el concepto de hardware reconfigurable, dentro del cual se prioriza la explicación de la estructura y el diseño de las FPGAs, y se detallan sus ventajas e inconvenientes respecto de un ASIC, una CPU convencional o una GPU.

Seguidamente, se expone en detalle el concepto de síntesis de alto nivel y varias herramientas que permiten explotar esta tecnología, explorando las posibilidades que ofrece el entorno *Xilinx Vitis Unified Software Platform* junto con sus varios módulos, entre los cuales encontramos *Vitis HLS* o *Vitis Analyzer*. Adicionalmente, se explica la instalación de todo el entorno desde cero, se da una breve descripción del proceso de compilación que se lleva a cabo para la síntesis de alto nivel y se detalla con mucho rigor la estructura de los archivos de diseño y configuración.

Por último, y como objetivo principal del trabajo, se presenta el algoritmo SUnSAL como técnica de desmezclado espectral lineal, se desarrolla el proceso completo para su implementación y aceleración por hardware en *Xilinx Vitis Unified Software Platform* utilizando directivas de aceleración embebidas en el código a través de HLS, se analizan los resultados de latencia y consumo de hardware, y se ejecuta el algoritmo sobre una tarjeta de aceleración *Xilinx* Alveo U250 con datos hiperespectrales reales.

Como recursos adicionales, se incluyen los códigos utilizados a lo largo de la implementación y los reportes de síntesis obtenidos en las emulaciones y ejecuciones del algoritmo SUnSAL. Para la ejecución sobre la tarjeta de aceleración *Xilinx* Alveo U250 se han utilizado unos servidores de computación de alto rendimiento de la universidad ETH Zürich accesibles a través del programa HACC (*Heterogeneous Accelerated Compute Clusters*) de la empresa AMD (*Advanced Micro Devices*), que también se detallan.

Palabras clave — análisis hiperespectral, hardware reconfigurable, síntesis de alto nivel, desmezclado espectral, Vitis, Alveo, SUnSAL, AVIRIS.

Abstract

In this work, an implementation of a linear spectral unmixing algorithm known as SUnSAL (*Spectral Unmixing by Variable Splitting and Augmented Lagrangian*) has been performed on FPGA-based acceleration cards through high-level synthesis (HLS). For this purpose, the *Xilinx Vitis Unified Software Platform* development environment has been used, through which the algorithm has been hardware accelerated in order to optimize the performance. Finally, and after analyzing in detail the latency and hardware consumption data, the algorithm has been executed on a *Xilinx* Alveo U250 acceleration card using data from a hyperspectral image captured by the hyperspectral sensor AVIRIS (*Airborne Visible / Infrared Imaging Spectrometer*) and a spectral library from the United States Geological Survey (USGS).

First, an introduction to the basic concepts of hyperspectral analysis as a scientific discipline is given, exploring its multiple applications and rigorously exposing the structure of hyperspectral images. Then, the concept of reconfigurable hardware is presented, with priority given to the explanation of the structure and design of FPGAs and their advantages and disadvantages with respect to an ASIC, a conventional CPU and a GPU.

Next, the concept of high-level synthesis and several tools that allow exploiting this technology are explained in detail, exploring the possibilities offered by the *Xilinx Vitis Unified Software Platform* environment together with its various modules, among which we find *Vitis HLS* or *Vitis Analyzer*. Additionally, the installation of the entire environment from scratch is explained, a brief description of the compilation process that is carried out for high-level synthesis is given, and the structure of the design and configuration files is rigorously detailed.

Finally, and as the main objective of the work, the SUnSAL algorithm is presented as a linear spectral unmixing technique, the complete process for its implementation and hardware acceleration in *Xilinx Vitis Unified Software Platform* is developed using acceleration directives embedded in the code through HLS, the latency and hardware consumption results are analyzed, and the algorithm is executed on a *Xilinx* Alveo U250 acceleration card with real hyperspectral data.

As additional resources, codes used throughout the implementation and synthesis reports obtained from emulations and runs of the SUnSAL algorithm are included. For the execution on the acceleration card *Xilinx* Alveo U250, high-performance computing servers from ETH Zürich University were used, some of which are accessible through the HACC (Heterogeneous Accelerated Compute Clusters) program from AMD (Advanced Micro Devices). These are also detailed.

Keywords — hyperspectral analysis, reconfigurable hardware, high-level synthesis, spectral unmixing, Vitis, Alveo, SUnSAL, AVIRIS.

Índice general

Índice de figuras

Capítulo 1

Introducción

Objetivos

El objetivo principal de este trabajo está cuidadosamente condensado en su título: implementar un algoritmo de desmezclado espectral, conocido como SUnSAL (*Spectral Unmixing by Variable Splitting and Augmented Lagrangian*), mediante síntesis de alto nivel sobre hardware reconfigurable, aplicándolo específicamente a imágenes hiperespectrales de la superficie terrestre. Sin embargo, no es un objetivo que pueda afrontarse directamente sin antes haber estudiado con suficiente profundidad qué son, para qué sirven y cómo se procesan las imágenes hiperespectrales, a qué hacen referencia los conceptos de hardware reconfigurable y desmezclado espectral, o qué es la síntesis de alto nivel. Por ello, es necesario afrontar uno a uno los anteriores conceptos para, finalmente, llegar a la mencionada implementación del algoritmo SUnSAL entendiendo por completo el contexto en el que opera, la utilidad que tiene y ejecutándolo sobre hardware físico reconfigurable con datos de imágenes hiperespectrales reales.

De esta forma, los subobjetivos que se persiguen son, en primer lugar, familiarizarse con el propósito y los conceptos esenciales del análisis hiperespectral y, en concreto, de las técnicas de desmezclado espectral. En segundo lugar, estudiar en profundidad el enfoque de la síntesis de alto nivel y la aceleración por hardware para la implementación de algoritmos sobre hardware reconfigurable, aprendiendo a su vez a utilizar la herramienta más extendida para ello, conocida como *Vitis Unified Software Platform*. En tercer lugar, usar *Vitis Unified Software Platform* para implementar y acelerar por hardware el algoritmo SUnSAL para desmezclado espectral lineal, analizando en detalle los resultados de latencia y consumo de hardware. Por último, aprender a obtener y procesar datos hiperespectrales reales con los que ejecutar el algoritmo sobre tarjetas de aceleración *Xilinx* Alveo, analizando los resultados obtenidos.

Por supuesto, todo lo anterior conlleva incorporar conocimiento de múltiples áreas científicas como por ejemplo la física, para poder entender los conceptos básicos del análisis hiperespectral, las matemáticas, para establecer la axiomática y los teoremas que dan pie al algoritmo SUnSAL, y por supuesto la ingeniería informática, que se presenta en forma de diseño de hardware, algoritmia y gestión de sistemas operativos, programas y entornos de ejecución. Aunque se parte de un mínimo de conocimiento en cada una de estas áreas, los objetivos se centran en estudiar principalmente aquellos conceptos más cercanos a la ingeniería informática, por lo que se proporcionarán referencias bibliográficas para los temas que sean más extensos o escapen del ámbito de este trabajo.

Antecedentes

Los antecedentes y el estado del arte del análisis hiperespectral, el hardware reconfigurable, el desmezclado espectral y la síntesis de alto nivel se desarrollan en detalle en los correspondientes capítulos de este escrito. Sin embargo, el presente trabajo de fin de grado basa principalmente su realización en el algoritmo SUnSAL, que es de reciente aparición y requiere de mención especial. El algoritmo SUnSAL (*Spectral Unmixing by Variable Splitting and Augmented Lagrangian*) fue publicado por primera vez en el año 2012 por *José M. Bioucas-Dias* y *Mario A. T. Figueiredo* [8], dos investigadores del *Instituto Superior Técnico* de Lisboa, en Portugal. Desde entonces, se considera una de las técnicas de referencia en el ámbito del desmezclado espectral lineal moderno.

En un principio, se publicó como un algoritmo con un enfoque novedoso basado en técnicas de regularización para problemas inversos [33], y en poco tiempo se volvió uno de los algoritmos más utilizados en el ámbito del desmezclado espectral por su rápida convergencia y su baja latencia en comparación con métodos anteriores [52], aun teniendo algunas desventajas que se expondrán en detalle en el correspondiente capítulo. Aunque existen implementaciones del algoritmo SUnSAL en lenguajes de programación como Matlab o Java, hasta el momento no se ha realizado ninguna implementación sobre dispositivos de hardware reconfigurable basados en arquitecturas similares a las FPGAs. Esto puede deberse en parte a la alta complejidad de diseño del algoritmo SUnSAL, que es iterativo y requiere de una serie de pasos de alto coste computacional. Sin embargo, la síntesis de alto nivel proporciona las suficientes herramientas como para que esto sea posible, sin ser necesario un código de bajo nivel inabarcable utilizando lenguajes de descripción de hardware.

Plan de trabajo

El desarrollo del presente trabajo de fin de grado cubre todos los meses desde septiembre de 2022 hasta mayo de 2023, aunque la investigación inicial ha sido realizada desde febrero de 2022.

1. Desde febrero de 2022 hasta octubre de 2022 se dedica una primera etapa a investigar y aprender acerca del estado del arte del análisis hiperespectral, las técnicas de desmezclado espectral y la síntesis de alto nivel. Así, quedan estudiados y fijados el algoritmo concreto a implementar (SUnSAL) y las herramientas a utilizar (*Xilinx Unified Software Platform*), además de tener una visión clara de los conceptos anteriores, que son esenciales para proceder con las siguientes etapas.

2. Desde octubre de 2022 hasta febrero de 2023 se implementa y se acelera por hardware el algoritmo SUnSAL sobre *Xilinx Unified Software Platform*, asegurando su buen funcionamiento y optimizando el diseño a nivel de código. Tanto la propia implementación como la aceleración por hardware son procesos complejos, por lo que se dedican 2 meses a cada uno de ellos. Así, queda bien implementado el algoritmo SUnSAL con las correspondientes estimaciones de latencia y consumo de hardware.

3. Desde febrero de 2023 hasta abril de 2023 se obtienen y se procesan datos hiperespectrales reales con los que poder ejecutar el algoritmo SUnSAL sobre tarjetas de aceleración basadas en FPGAs, lo que requiere estudiar cómo utilizarlas, y aprender a recoger los datos de ejecución.

Consideraciones del autor

No sería una exageración afirmar que el autor de este trabajo hubiera pasado innumerables noches en vela durante su infancia si hubiera descubierto mucho antes varios de los temas que se tratan en este escrito. Aún recuerda el día en que se encontró de casualidad con una explicación muy sencilla de las bondades de las imágenes hiperespectrales y no pudo entender cómo no había pensado antes en la increíble idea de incorporar información de más allá del rango visible del espectro electromagnético en una imagen. O también el día en que entendió el funcionamiento de un procesador sencillo conectado con una memoria y unos cuantos registros, o incluso cuando le enseñaron por primera vez las amplias capacidades de la síntesis de alto nivel. Son conceptos que nacieron hace relativamente poco tiempo, no hay más que notar que el primer espectroscopio portátil no fue construido hasta el año 1974, pero que han supuesto una verdadera revolución en el mundo, aunque lo hayan hecho silenciosamente para la gran mayoría de las personas.

En cuanto a la redacción del presente trabajo, y con ánimo de facilitar lo más posible su lectura, el autor ha creído conveniente comenzar explicando las ideas básicas del análisis hiperespectral y el hardware reconfigurable, para después profundizar en conceptos más específicos como son la síntesis de alto nivel y el desmezclado espectral, tras los cuales se dedica un extenso capítulo a la implementación del algoritmo SUnSAL y se culmina ejecutándolo sobre hardware físico reconfigurable utilizando datos de imágenes hiperespectrales reales. A medida que avanzan los capítulos se va utilizando todo lo visto hasta ese momento, por lo que la complejidad de las explicaciones va en aumento a medida que se van introduciendo conceptos nuevos. A este respecto, y aunque los capítulos pueden leerse por separado gracias a las referencias incluidas en todo el texto, el autor recomienda proceder en orden para entender al completo la implementación final.

Introduction

Objectives

The main objective of this work is carefully condensed in its title: to implement a spectral unmixing algorithm, known as SUnSAL (*Spectral Unmixing by Variable Splitting and Augmented Lagrangian*), by means of high-level synthesis on reconfigurable hardware, applying it specifically to hyperspectral images of the Earth's surface. However, this is not an objective that can be tackled directly without first studying in sufficient depth what hyperspectral images are, what they are used for, how they are processed, what the concepts of reconfigurable hardware and spectral unmixing refer to, or what high-level synthesis is. Therefore, it is necessary to address the above concepts one by one to finally arrive at the aforementioned implementation of the SUnSAL algorithm, fully understanding the context in which it operates and its utility, and running it on physical reconfigurable hardware with real hyperspectral data.

Thus, the sub-objectives pursued are, firstly, to become familiar with the purpose and essential concepts of hyperspectral analysis and, in particular, of spectral unmixing techniques. Secondly, to study in sufficient depth the high-level synthesis and hardware acceleration approach for algorithm implementation on reconfigurable hardware, learning in turn to use the most widespread tool for this purpose, known as *Vitis Unified Software Platform*. Thirdly, to use *Vitis Unified Software Platform* to implement and hardware accelerate the SUnSAL algorithm for linear spectral unmixing, analyzing in detail the latency and hardware consumption results. Finally, learn how to obtain and process real hyperspectral data with which to run the algorithm on Alveo acceleration cards, analyzing the results obtained.

Of course, all of the above entails incorporating knowledge of multiple scientific areas such as physics, in order to understand the basic concepts of hyperspectral analysis, mathematics, to establish the axiomatics and theorems that give rise to the SUnSAL algorithm, and of course computer engineering, which is presented in the form of hardware design, algorithms and management of operating systems, programs and execution environments. Although a minimum of knowledge in each of these areas is assumed, the objectives focus mainly on studying those concepts closer to computer engineering, so bibliographic references will be provided for those topics that expand beyond the scope of this work.

Background work

The background work and state of the art of hyperspectral analysis, reconfigurable hardware, spectral unmixing and high-level synthesis are developed in detail in the corresponding chapters of this work. However, the present dissertation mainly bases its realization on the SUnSAL algorithm, which is of recent appearance and requires special mention. The SUnSAL algorithm (*Spectral unmixing by Variable Splitting and Augmented Lagrangian*) was first published in the year 2012 by *José M. Bioucas-Dias* and *Mario A. T. Figueiredo* [8], two researchers from the *Instituto Superior Técnico* of Lisboa, in Portugal. Since then, it has been considered one of the reference techniques in the field of modern linear spectral unmixing.

Initially, it was published as an algorithm with a novel approach based on regularization techniques for solving inverse problems [33], and it very quickly became one of the most used algorithms in the field of spectral unmixing due to its fast convergence and low latency compared to previous methods [52], even though it has some disadvantages that will be discussed in detail in the corresponding chapter. Although there are implementations of the SUnSAL algorithm in programming languages such as Matlab or Java, so far no implementation has been performed on reconfigurable hardware devices based on architectures similar to FPGAs. This may be partly due to the high complexity of the SUnSAL algorithm, which is iterative and requires a number of computationally expensive steps. However, high-level synthesis provides enough tools to make this possible, without requiring unmanageable low-level code using hardware description languages.

Work plan

The development of this bachelor's thesis covers all the months from September 2022 to May 2023, although the initial research has been carried out since February 2022.

1. From February 2022 to October 2022, a first stage is dedicated to researching and learning about the state of the art of hyperspectral analysis, spectral unmixing techniques and high-level synthesis. Thus, the specific algorithm to be implemented (SUnSAL) and the tools to be used (*Xilinx Unified Software Platform*) are decided and studied, in addition to having a clear vision of the previous concepts, which are essential to proceed with the following stages.

2. From October 2022 to February 2023, the SUnSAL algorithm is implemented and hardware accelerated on the *Xilinx Unified Software Platform*, ensuring its proper functioning and optimizing the design at code level. Both the implementation itself and the hardware acceleration are complex processes, so 2 months are dedicated to each of them. Thus, the SUnSAL algorithm is well implemented with the corresponding latency and hardware consumption estimates.

3. From February 2023 to April 2023, real hyperspectral data is obtained and processed with which to run the SUnSAL algorithm on FPGA-based accelerator cards, which requires studying how to use them and learning how to collect the execution data.

Author's remarks

It would not be an exaggeration to say that the author of this bachelor's thesis would have spent countless sleepless nights during his childhood if he had discovered much earlier several of the topics discussed in this paper. He still remembers the day when he happened to come across a very simple explanation of the goodness of hyperspectral imaging and could not understand how he had not thought of the incredible idea of incorporating information from beyond the visible range of the electromagnetic spectrum into an image before. Or the day he understood the workings of a simple processor connected to a memory and a few registers, or even when he was first taught the vast capabilities of high-level synthesis. These are concepts that were born relatively recently - one need only note that the first portable spectroscope was not built until 1974 - but which have brought about a real revolution in the world, even if they have done so silently for the vast majority of people.

In order to make this work as easy to read as possible, the author has decided to begin by explaining the basic ideas of hyperspectral analysis and reconfigurable hardware, and then delve into more specific concepts such as high-level synthesis and spectral unmixing, followed by an extensive chapter on the implementation of the SUnSAL algorithm, culminating in its execution on physical reconfigurable hardware using real hyperspectral image data. As the chapters progress, everything seen up to that point is used, so that the complexity of the explanations increases as new concepts are introduced. In this regard, and although the chapters can be read separately thanks to the references included throughout the text, the author recommends proceeding in order to fully understand the final implementation.

Capítulo 2

Análisis de imágenes hiperespectrales

2.1. Análisis hiperespectral

El análisis hiperespectral se encarga de recoger y procesar información a lo largo de todo el espectro electromagnético. Trabaja principalmente con un tipo de imágenes conocidas como imágenes hiperespectrales, que proporcionan información, no solo de los colores capturados, sino de la firma espectral del material que se está fotografiando, que es una medida que cuantifica los porcentajes de reflexión, absorción y transmisión de la radiación electromagnética que reciben, y que se expande más allá del rango visible del espectro electromagnético. Estos porcentajes dependen de la composición molecular y textura de cada material concreto, por lo que las imágenes hiperespectrales permiten potencialmente identificar cualquier superficie u objeto que se esté fotografiando.

Formalmente, la firma espectral de un material es la variación de su reflectancia en función de la longitud de onda de la radiación incidente. Fijado un cierto material E y una cierta radiación incidente con longitud de onda λ, la reflectancia $R_{E,\lambda}$ viene dada por la siguiente fórmula:

$$R_{E,\lambda} = \frac{\Phi^r_{E,\lambda}}{\Phi^i_{E,\lambda}}$$

donde $\Phi^r_{E,\lambda}$ es el flujo radiante reflectado y $\Phi^i_{E,\lambda}$ el flujo radiante recibido por el material. A su vez, el flujo radiante $\Phi_{E,\lambda}$ viene dado por:

$$\Phi_{E,\lambda} = \frac{dQ_E}{dt},\ Q_E = \int_\Sigma S \cdot \hat{n} dA$$

donde Q_E es la energía radiante expresada en Julios (J), que es una medida de la energía transportada por los fotones, S es el conocido como vector de Poynting, que es el producto vectorial del vector de campo eléctrico y el vector auxiliar de campo magnético y representa la densidad de corriente de la energía radiante, $\hat{n}$ es el vector normal a la superficie Σ del material y A representa el área de Σ. Para un desarrollo formal y detallado de los conceptos de radiación, reflectancia y firma espectral ver [28].

Comúnmente, la firma espectral de un material se representa en forma de gráfica que muestra, para cada valor de la longitud de onda λ de la radiación incidente, el respectivo valor de la reflectancia $R_{E,\lambda}$ del material, a veces como porcentaje de radiación reflectada respecto de la incidente. En la figura 2.1 se muestran, a modo de ejemplo, las firmas espectrales para el aluminio, la plata y el oro.

Con el objetivo de capturar la firma espectral de los materiales que estemos fotografiando, las imágenes hiperespectrales van a tener, a diferencia de las imágenes tradicionales, una dimensión adicional correspondiente a la longitud de onda, que va a estar compuesta de una serie de bandas espectrales que guardarán información sobre la reflectancia de los materiales presentes en el área de detección en distintos puntos del espectro electromagnético. De esta forma, cada píxel no contendrá un único valor, sino que estará compuesto por un vector de valores de reflectancia para radiación incidente de distintas longitudes de onda, lo que permite aproximar la firma espectral de los materiales presentes.

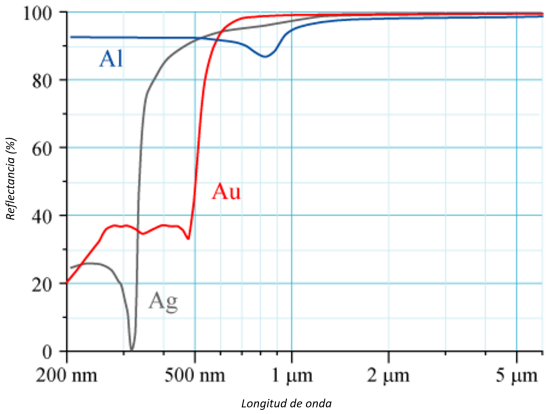


Figura 2.1: Reflectancia en función de λ para el aluminio (Al), la plata (Ag) y el oro (Au).

Debido precisamente a la estructura tridimensional que poseen las imágenes hiperespectrales, se suelen representar como un prisma rectangular donde el plano horizontal contiene las dimensiones espaciales, y el eje vertical la dimensión espectral. Esto es lo que precisamente se muestra en la figura 2.2, que simula una imagen hiperespectral de la superficie terrestre tomada desde un sensor a bordo de un satélite.

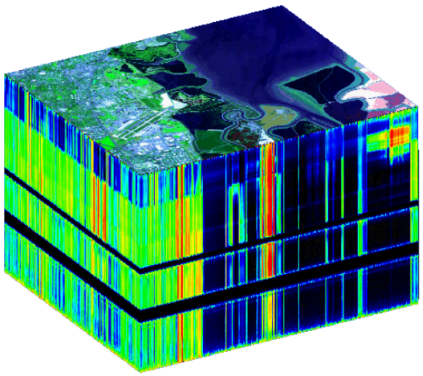


Figura 2.2: Ejemplo de una imagen hiperespectral representada en forma de prisma tridimensional. [30]

Respecto a la utilidad del análisis hiperespectral, sus aplicaciones son muy variadas y se pueden encontrar casos de uso en multitud de ámbitos. Desde control de calidad de alimentos, pasando por autenticación de cuadros y obras artísticas, seguridad y defensa, determinación de la composición de rocas y minerales, y hasta monitorización del cambio climático y de desastres naturales a escala global. Por supuesto, dependiendo de la situación, se requerirán imágenes hiperespectrales tomadas con sensores próximos al elemento que se desee analizar, sensores que operen desde satélites o sensores acoplados a medios de trasporte como barcos o aviones, los cuales deberán ser capaces de tomar las imágenes y, en muchos casos, procesarlas en el momento. Para un recorrido detallado de muchas de las aplicaciones que tiene el análisis hiperespectral ver [36].

No obstante, las imágenes hiperespectales también presentan muchas problemáticas en cuanto a su manejo. En primer lugar, el coste y complejidad computacional de su tratamiento es mucho mayor que en el caso de imágenes tradicionales, puesto que suelen involucrar conjuntos de datos mucho más grandes, de al menos decenas o cientos de megabytes [32]. En segundo lugar, al ser comúnmente tomadas por sensores a bordo de satélites, deben ser transmitidas a Tierra por canales que poseen muy poco ancho de banda, lo que fuerza a realizar un preprocesamiento a bordo del satélite que disminuya el tamaño de estas, a través de técnicas de compresión, reducción dimensional o simplemente selección de los datos relevantes para el estudio que se quiera hacer. Por último, el coste de los sensores suele ser mucho más alto que el de una cámara que solo captura datos en el rango visible del espectro, por lo que el acceso a imágenes hiperespectrales es mucho más restringido, incluso para investigadores. Por suerte, existen ya diversas bases de datos hiperespectrales para uso público, que facilitan cada vez más su estudio.

Por último, y como se puede ver también en la figura 2.2, en un escenario real es muy difícil que en el área de detección de una imagen hiperespectral encontremos únicamente una cantidad finita de materiales claramente diferenciados, y suele ocurrir que habrá múltiples sustancias mezcladas de distinta manera, incluso en un mismo píxel. Esto es una de las principales dificultades que presenta el análisis hiperespectral, por lo que será imprescindible disponer de técnicas que nos permitan hacer frente a esta problemática, que introduciremos en el capítulo 5, centrándonos principalmente en lo que se conoce como desmezclado espectral.

2.2. Estructura de una imágen hiperespectral

Con el objetivo de explicar y tener claros los datos que componen una imagen hiperespectral, vamos a detallar su estructura partiendo de las imágenes tradicionales que solemos utilizar cuando tomamos una fotografía con una cámara. De esta forma, veremos que las imágenes hiperespectrales no son inventos rebuscados, sino que surgen como extensión natural de las imágenes tradicionales.

El tipo de imágenes más básico son las imágenes monocromáticas. Estas poseen tan solo dos dimensiones espaciales, y cada píxel guarda un único valor que da información de su intensidad. Si aumentamos el valor de un píxel, este se verá más claro y blanquecino debido al aumento de la intensidad de la luz, y si lo disminuimos se verá más oscuro. Si a esto le sumamos que cada píxel pueda tener valores relativos a la proporción de diferentes tipos de colores, obtenemos las conocidas como imágenes RGB, en las que cada píxel tiene tres valores correspondientes a tres puntos del espectro visible (rojo, azul y verde) que se combinan para mostrar diferentes colores.

Aunque no nos demos cuenta, en las imágenes RGB ya tenemos una tercera dimensión espectral, puesto que cada píxel es en realidad un vector de tres valores en varios puntos del espectro, aunque solo nos restrinjamos al visible. Si esta idea la extendemos a todo el espectro electromagnético, surgen las conocidas como imágenes multiespectrales, que suelen tener vectores de 5 a 50 valores en múltiples rangos del espectro, extendiéndose al infrarrojo y ultravioleta.

El problema que encontramos con las imágenes multiespectrales es que por un lado los rangos de espectro monitorizados en cada valor son bastante amplios, y por otro lado tenemos un número de valores que en muchos casos es insuficiente porque discretiza de manera demasiado gruesa el espectro electromagnético. Con el objetivo de capturar la información suficiente como para aproximar la firma espectral de un material, surgen las imágenes hiperespectrales, que tienen vectores con muchos más valores en cada píxel, usualmente entre 200 y 1000, y rangos del espectro electromagnético mucho más estrechos, que cubren zonas de entre 10 y 20 nm [25].

Fijada una posición de los vectores que conforman los píxeles, al conjunto de valores que ocupan esa posición se le llama banda espectral, y en la figura 2.3 vemos un esquema de la cantidad de bandas espectrales que poseen los diferentes tipos de imágenes, siendo las hiperespectrales las que poseen mayor cantidad. De esta forma, tenemos una serie de bandas espectrales que de manera individual corresponden a la reflectancia de los píxeles para diferentes valores de longitud de onda, y que se representan en forma de *slices* rectangulares formando un cubo tridimensional cuando se ponen una detrás de otra.

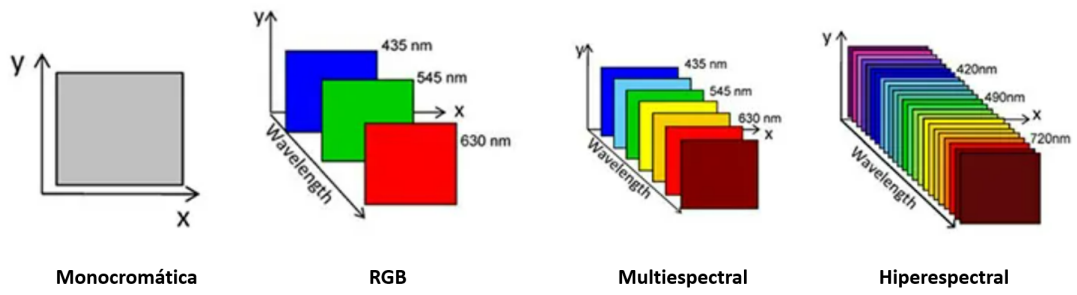


Figura 2.3: Esquema de la estructura de diversos tipos de imágenes en función de sus bandas espectrales.

Para reconstruir de manera aproximada la firma espectral de los materiales que caen bajo el área de detección de un píxel, basta con considerar su vector de valores en las diferentes bandas espectrales, que dan la reflectancia de los materiales en función de la longitud de onda, como se muestra en la figura 2.4.

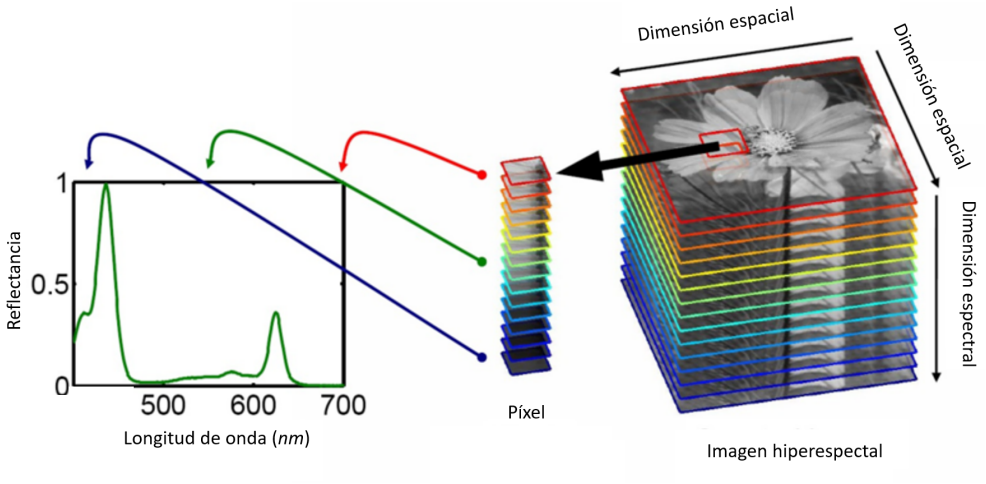


Figura 2.4: Construcción de la aproximación a la firma espectral presente en cada píxel.

Por supuesto, el paso de imágenes multiespectrales a hiperespectrales conlleva un considerable aumento en el peso de estas, y por tanto un aumento en la complejidad de los algoritmos que las procesan, lo que hace que hoy en día sea un tema de investigación muy candente.

2.3. Captura y procesado de imágenes hiperespectrales

Para obtener una imagen hiperespectral con la estructura explicada en la sección 2.2, se utilizan sensores cuyas características dependen de aquello que se esté fotografiando. Por concreción, nos centraremos en la toma de imágenes hiperespectrales de la superficie terrestre, y en concreto en el *modus operandi* de sensores similares al AVIRIS (*Airborne Visible / Infrared Imaging Spectrometer*) [40].

Como punto de partida, independientemente del sistema empleado, un sensor básico para la obtención de imágenes hiperespectrales requiere de tres mecanismos imprescindibles: un sistema de generación o calibración de radiación con diferente longitud de onda que dependerá de las bandas espectrales que queramos obtener, un método de dispersión respecto de la longitud de onda, como por ejemplo un prisma, y un detector. Hoy en día hay muchos sensores disponibles para investigación, y existen bases de datos con grandes cantidades de imágenes hiperespectrales almacenadas.

Una vez tenemos los anteriores elementos, existen diversas técnicas para la obtención de imágenes hiperespectrales, que se suelen clasificar según el orden en el que se obtienen los datos, lo que también implica utilizar diferentes métodos espectroscópicos en el sensor. Existen cuatro sistemas principales: *whiskbroom*, *pushbroom*, *staring* y *snapshot*.

Los sistemas *whiskbroom* adquieren la información espectral píxel a píxel, consiguiendo uno a uno todos los valores de reflectancia de todas las bandas espectrales en un píxel antes de pasar al siguiente. Utilizan un dispositivo de dispersión de la longitud de onda, como por ejemplo un prisma de difracción, y ofrecen, en general, mejor resolución espectral que espacial. El sensor LandSat [56] (hasta el modelo LandSat 8) es un ejemplo de sistema *whiskbroom*.

Por su parte, los sistemas *pushbroom* funcionan igual que los *whiskbroom*, pero obtienen la información completa de múltiples píxeles a la vez, parecido a como los procesadores vectoriales hacen las operaciones de *load* en comparación con los escalares. Como no puede ser de otra forma, utilizan sensores bidimensionales para ello y, aunque son considerablemente más rápidos, suelen perder precisión por el camino. Ejemplo de sistemas *pushbroom* hay muchos, como el propio AVIRIS ya introducido, o el sensor HYDICE (*Digital Imagery Collection Experiment*) [6].

En cuanto a los sistemas *starring*, estos obtienen todos los valores de reflectancia de una banda espectral concreta antes de pasar a la siguiente banda. Esta técnica mejora la resolución espacial a costa de reducir la espectral, y no requiere de cambios rápidos en la longitud de onda de la radiación emitida ya que va banda a banda.

Por último, y sin entran en detalle por su alta complejidad, los sistemas *snapshot* obtienen el cubo tridimensional completo de una sola vez con ayuda de múltiples sensores integrados que permiten emitir radiación de distintas longitudes de onda simultáneamente y realizar detecciones de todas las bandas espectrales para todos los píxeles a la vez. A pesar de sus evidentes ventajas, obtienen resoluciones espaciales muy bajas y suelen requerir de procesados posteriores muy potentes y costosos.

La figura 2.5 muestra un diagrama con los cuatro sistemas descritos para obtener todos los datos de una imagen hiperespectral. Para más detalle y un extensa explicación en relación al proceso completo de detección y los elementos requeridos, ver [4].

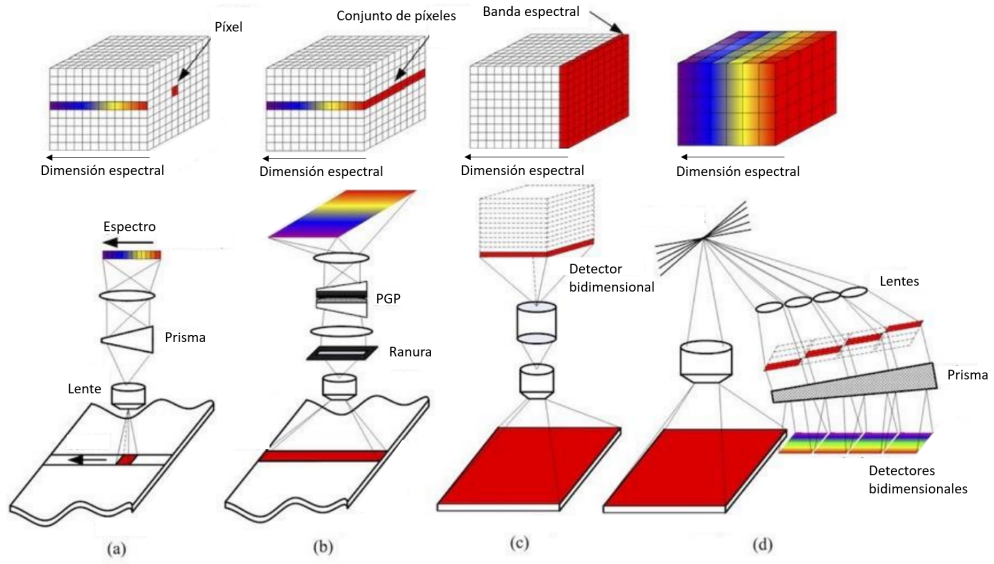


Figura 2.5: Sistemas de obtención de imágenes hiperespectrales según el orden de captura de los datos: a) *whiskbroom* b) *pushbroom* c) *staring* d) *snapshot*. [38]

Una vez que se han obtenido las imágenes hiperespectrales, y por supuesto dependiendo del estudio que se desee hacer, hay que procesar los datos que la conforman. Existen multitud de algoritmos para conseguir información relevante de ellos, de entre los cuales, a pesar de que sirven distintos propósitos, hay algunos que suelen ser muy recurridos por la propia naturaleza de las imágenes hiperespectrales y sus aplicaciones: preprocesado para compresión o reducción dimensional, identificación y tratamiento del ruido, desmezclado espectral y finalmente clasificación. Una extensa recopilación del estado actual en el estudio de estos métodos se puede encontrar en [25].

Como primer paso, siempre suele ser indispensable comprimir o reducir la cantidad de datos que componen una imagen hiperespectral, debido a que el aumento de la resolución espectral que se va produciendo a medida que mejoran los sensores supone un verdadero reto para la capacidad de las memorias actuales de los ordenadores personales y las técnicas convencionales de procesado de imágenes. Existen multitud de algoritmos de compresión para imágenes hiperespectrales, que se pueden dividir en dos tipos: los métodos de selección de variables (*feature selection*) y los métodos de selección de características (*feature extraction*) [25].

Por un lado, los métodos de selección de variables se limitan a elegir, sin modificarlas, aquellas bandas espectrales más relevantes para el estudio que desea hacer. Por otro lado, los métodos de selección de características tienen como objetivo encontrar una función f, que puede ser lineal o no, que transforme los vectores asociados a cada píxel de la imagen en otros vectores con menos dimensiones que concentren toda la información posible de la imagen original. La figura 2.6 muestra un esquema sencillo de la

diferencia entre los dos métodos. Es común encontrar técnicas de compresión, como los algoritmos de reducción dimensional, que combinan ambos métodos empezando por una selección de variables para después realizar una selección de características de la imagen reducida.

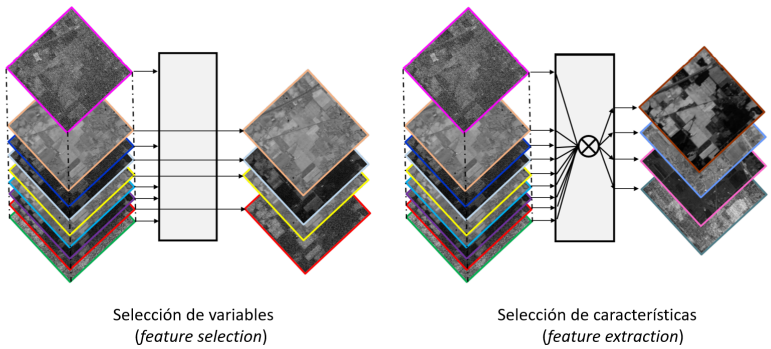


Figura 2.6: Esquema de los métodos de selección de variables (*feature selection*) y selección de características (*feature extraction*) para compresión de imágenes hiperespectrales. [25]

En segundo lugar, las imágenes hiperespectrales suelen venir con alteraciones o anomalías, que conocemos como ruido, producidas por el sensor utilizado, las condiciones atmosféricas y otros factores. De esta forma, necesitamos técnicas de restauración o eliminación del ruido que nos permitan reconstruir la imagen hiperespectral real en base a la imagen corrompida, intentando mitigar los efectos de pequeños errores en los datos y mejorar la relación señal-ruido. Para ello, se establece un modelo de ruido, generalmente ruido blanco gaussiano aditivo, y se emplean técnicas de regularización, enfoques basados en estadística bayesiana o redes neuronales convolucionales.

Sin embargo, la detección y mitigación del ruido presente en una imagen hiperespectral sigue siendo una cuestión de suma importancia que presenta muchos problemas, como la selección del modelo de ruido a utilizar, la estimación de sus parámetros, la identificación de las causas del ruido mayoritario, la eficiencia de los métodos de restauración, y un largo etcétera. En [7] se puede encontrar una detallada comparación entre diversas técnicas de restauración de imágenes hiperespectrales pertenecientes a un *dataset* producido por el sensor AISA [2] de imágenes de un pueblo italiano llamado Trento. Los resultados visuales de una banda se muestran en la figura 2.7.

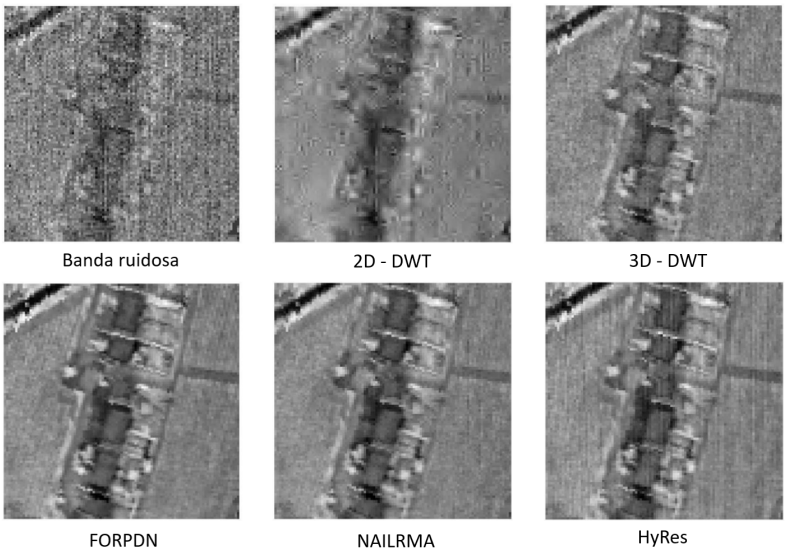


Figura 2.7: Resultados visuales de diversas técnicas de eliminación del ruido sobre una banda espectral de un *dataset* del sensor AISA: transformada discreta Wavelet 2D y 3D (2D-DWT y 3D-DWT) [5], FORPDN [55], NAILRMA [83] y HyRes [59].

Una vez se ha realizado la compresión y posiblemente la detección y eliminación del ruido, queda aplicar ya los algoritmos que nos van a permitir obtener la información buscada, que dependerán de la aplicación para la que se vayan a utilizar las imágenes hiperespectrales. Hoy en día, las técnicas más extendidas y utilizadas son las de desmezclado espectral, que se introducen en detalle en el capítulo 5, y las de clasificación, que vemos a continuación.

El objetivo principal de las técnicas de clasificación espectral es el de agrupar las imágenes hiperespectrales en base a, en el caso de imágenes de la superficie terrestre, diferentes tipos de terrenos o materiales presentes en el área de detección, al igual que se hace en muchas otras áreas relacionadas con la inteligencia artificial, que poco a poco se va aplicando también al análisis hiperespectral. Sus aplicaciones son muy variadas, desde validación de las técnicas de compresión y eliminación del ruido, hasta detección de patrones en series temporales de imágenes hiperespectrales, pasando por la aplicación obvia de agrupar imágenes en base a la localización del área de detección en la superficie terrestre.

Podemos catalogar estos métodos en dos grupos: clasificadores espectrales y clasificadores espaciales, que a su vez se pueden dividir en tres categorías, que se definen igual que en el caso de clasificadores en inteligencia artificial clásica: clasificación supervisada, no supervisada y parcialmente supervisada. Los clasificadores espectrales difieren de los espaciales en el hecho de que no tienen en cuenta la organización espacial de los píxeles, mientras que los espaciales estudian también las dependencias entre píxeles y mediciones cercanas. Para ello, se utilizan comúnmente los conocidos como campos aleatorios de Markov, o se crean sistemas dinámicos de vecindades entre píxeles que ayudan a incorporar la información espacial en los datos.

Recopilaciones y explicaciones detalladas de los mencionados métodos de clasificación de imágenes hiperespectrales se pueden encontrar en [1] y [16], que también contienen infinidad de referencias útiles en este ámbito, el cual está en pleno auge, como se muestra en la figura 2.8 [25], en donde se puede ver que más de un cuarto de los artículos de análisis hiperespectral publicados en las revistas de IEEE son de clasificación hiperespectral.

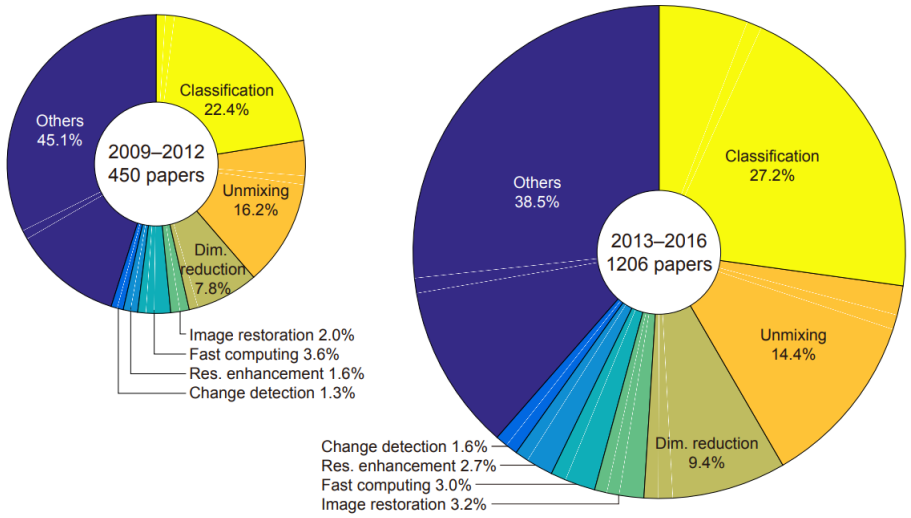


Figura 2.8: Estadísticas del porcentaje de artículos de análisis hiperespectral en las revistas de IEEE para cada tipo de técnica de procesado, en los periodos (a) 2009 - 2012 (b) 2013 - 2016. [25]

Debido al considerable aumento en la complejidad de los algoritmos, el gran tamaño de los datos que componen las imágenes hiperespectrales y los requerimientos de procesado en tiempo real que cada vez son más importantes para multitud de aplicaciones, hoy en día una gran parte de las técnicas de análisis hiperespectral se implementa en lo que se conoce como hardware reconfigurable, concretamente en dispositivos electrónicos basados en FPGAs (*field-programmable gate arrays*) que pasamos a introducir en el capítulo 3.

Capítulo 3

Hardware reconfigurable

3.1. Historia y evolución

La electrónica digital se encarga del estudio de circuitos que manejan información en forma de una cantidad finita de estados, que suelen identificarse con la corriente eléctrica que llega, sale o transita un cierto componente electrónico. En la mayoría de las aplicaciones se tienen dos estados representados por un 0 o un 1 correspondientes a la ausencia o no de corriente respectivamente. Para establecer relaciones entre los estados de entrada y los de salida, se utilizan agrupaciones de transistores organizadas en bloques o puertas lógicas, que permiten realizar operaciones sobre los estados en base a las interconexiones que se utilicen. Este método de diseño se conoce como *transistor - transistor logic* (TTL) [47].

Los circuitos de los primeros chips electrónicos eran estáticos en el sentido de que una vez que se diseñaba y organizaba el hardware, este era utilizado para un propósito concreto y no podía ser adaptado para otros usos sin que ello requiriera manipular físicamente los elementos que lo componían. En el momento en que se requería otra funcionalidad, se diseñaba un circuito distinto.

A medida que mejoraba la tecnología y aumentaban las necesidades del hardware, se empezaron a considerar circuitos estáticos modulares que permitieran aumentar la reconfigurabilidad de los sistemas. Un ejemplo que representa esta deriva son los circuitos electrónicos *7400 series* de *Texas Instruments* [57] presentados en la década de 1960, que usan un tipo de transistores MOSFET (*metal-oxide–semiconductor field-effect transistor*) para implementar puertas lógicas que pueden ser interconectadas para diseñar multitud de circuitos sencillos [39].

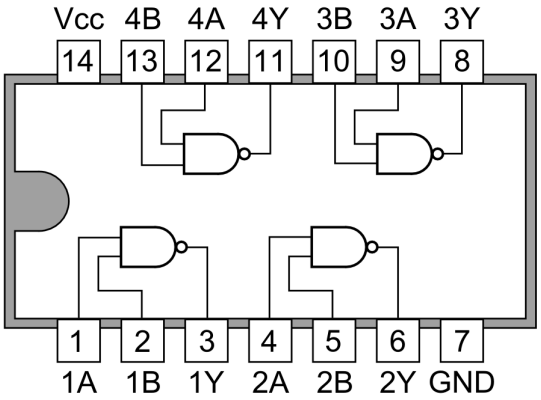


Figura 3.1: Esquema de un circuito integrado de *7400 series* con NANDs de 2 entradas. [47]

En base a esta idea, en 1980 surgen los primeros PLDs (*programmable logic devices*) y, en 1984, Altera produce el primer circuito de lógica reprogramable, el EP300 [13]. La idea de tener hardware que permitiera ser reprogramado a gusto del usuario empezaba a tener cada vez más peso en la industria. Poco tiempo después, en el año 1984, Ross Freeman y Bernard Von Der Schmitt presentan la famosa FPGA (*field-programmable gate array*), que no solo permite reprogramar las puertas lógicas, sino que da

la posibilidad de modificar también las interconexiones entre ellas, dando una libertad de reconfiguración muy alta e incorporando en muchos casos otros elementos hardware como memorias más complejas, dispositivos de entrada y salida, etcétera [58].

3.2. Estructura de una FPGA

Una FPGA es un conjunto de bloques lógicos y elementos de memoria cuyas interconexiones pueden ser programadas, lo que hace que se consideren hardware reconfigurable. Los bloques lógicos se implementan usando una tabla de correspondencia o *look-up table* (LUT), que toma una cantidad finita de *inputs* en forma de conexiones que transportan la información de varios bits, y genera un *output* a partir de unos bits de configuración que se programan en base a la tabla de verdad de la operación booleana que se quiera realizar, como se muestra en la figura 3.2.

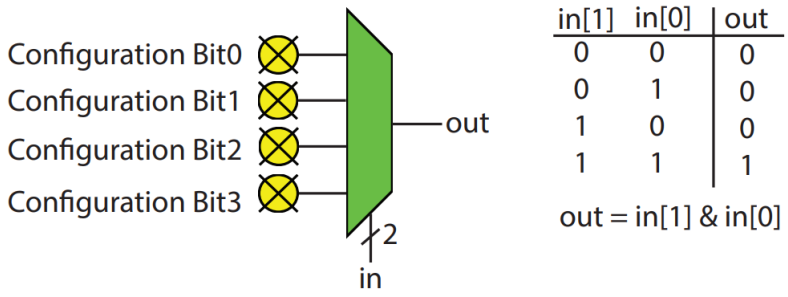


Figura 3.2: Esquema de una 2-LUT con un ejemplo de configuración para una puerta AND. [34]

El elemento básico de memoria en una FPGA es el biestable, *flip-flop* (FF) o *latch*, que se combina de diversas maneras con las LUTs y algunos multiplexores para formar elementos lógicos más complejos denominados bloques lógicos configurables (CLB), bloques de arrays lógicos (LAB), o simplemente *slices*. Su tamaño depende de la arquitectura y el fabricante, pero es común integrar, por ejemplo, un sumador completo o funciones básicas en un único *slice*.

Para interconectar los *slices* de manera que haya un alto grado de reconfigurabilidad, se utilizan canales de enrutado programables que manejan la conexión y desconexión de los *inputs* y *outputs* de los *slices*, que a su vez se controlan con bloques de conmutadores, los cuales se organizan para generar matrices de conexión entre los canales de enrutado, como muestra el esquema de la figura 3.3.

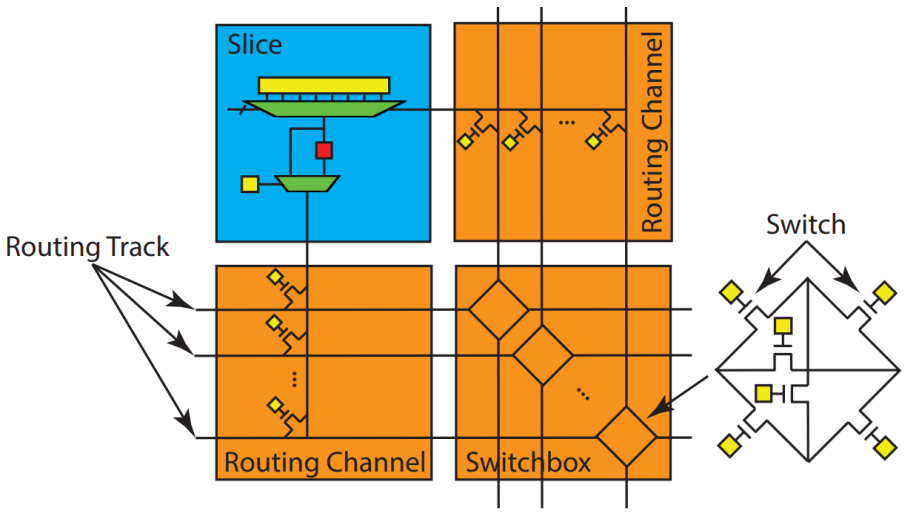


Figura 3.3: Esquema de un *slice* y diversos canales de enrutado (*routing Channels*) conectados por un bloque de conmutadores (*switchbox*). [34]

Según se van fabricando FPGAs cada vez más complejas y con más transistores por unidad de área, es muy común que se introduzcan elementos hardware *ad-hoc* distintos a los *slices*. Ejemplos de esto son manejadores de entrada/salida, o los módulos o bloques de memoria de acceso aleatorio (BRAM), que permiten diferentes organizaciones de sus interfaces y puertos, suelen tener alrededor de 32 KBytes de memoria, y se pueden interconectar para formar agrupaciones de memoria aún más grandes.

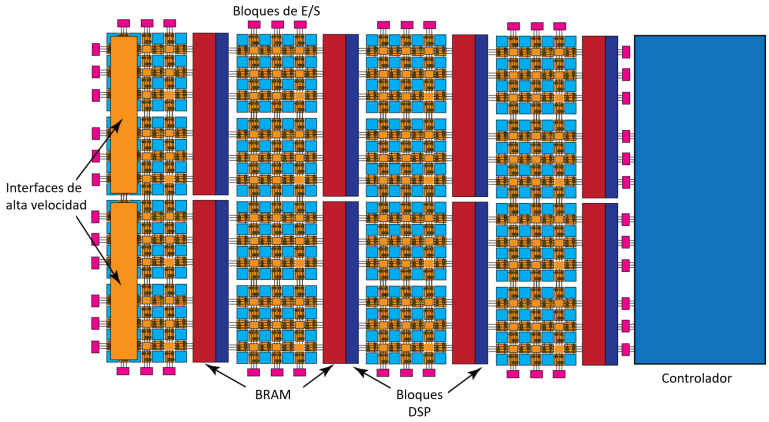


Figura 3.4: Esquema de la organización de LUTs, biestables y canales de enrutado en una FPGA con diversos elementos *ad-hoc* como BRAMs, controladores, bloques de E/S, etcétera. [34]

3.3. Ventajas y desventajas de las FPGAs

La reconfigurabilidad de las FPGAs es la principal ventaja que presentan respecto a otras unidades de procesamiento como las CPUs (*central processing units*) o las GPUs (*graphics processing units*). Además, ocupan poco espacio en comparación con otros circuitos de propósito general haciendo la misma función, lo que las hace idóneas para procesamiento de datos a bordo de medios aerotransportados o satélites, permiten una alta paralelización y son adecuadas para diseños modulares. Añadido a lo anterior, tienen un consumo energético considerablemente más bajo en comparación con una GPU o CPU para la misma carga computacional, lo que es muy importante para su uso en satélites [25].

Sin embargo, también tienen sus desventajas, entre las que destacan la complejidad que a veces requiere su reconfiguración a través de lenguajes de descripción de hardware (HDL), o un aumento de la latencia derivado de la propia flexibilidad que aporta la FPGA. Este último punto es especialmente importante puesto que evidencia un hecho que viene de la mano con el hardware reconfigurable: la mayor flexibilidad de programación del hardware tiene un coste de latencia y consumo energético respecto a un circuito integrado para una aplicación específica (ASIC).

A la hora de evaluar la conveniencia de usar una FPGA, es importante también analizar tanto el algoritmo o aplicación concreta que se quiera desarrollar como el proceso de implementación que se vaya a realizar. Por ejemplo, si el algoritmo es paralelizable, modular y debe poderse adaptar con el tiempo, entonces las FPGAs son una muy buena opción, mientras que si el algoritmo es puramente secuencial y estático entonces quizás convenga utilizar una CPU de propósito general más económica. Por ello, las FPGAs suelen ser una opción muy recomendada para implementar algoritmos de análisis hiperespectral que requieren de cierto paralelismo por la propia estructura de las imágenes hiperespectrales, y más aún si son parte de procesado a bordo de un satélite o se dividen en módulos de implementación independientes.

Para hacer frente a las complicaciones de la configuración de una FPGA a través de lenguajes de descripción de hardware u otros sistemas de más bajo nivel, en las últimas décadas se ha empezado a desarrollar una nueva tecnología bajo el nombre de síntesis de alto nivel (HLS), que busca agilizar y facilitar la implementación de algoritmos sobre hardware reconfigurable a través del uso de lenguajes de alto nivel como Python, C++ o Java. En el capítulo 4 vemos en detalle los conceptos básicos de la síntesis de alto nivel, así como una introducción a una de las principales herramientas para implementación de algoritmos en hardware reconfigurable con HLS: *Xilinx Vitis Unified Software Platform*.

Capítulo 4

Síntesis de alto nivel en Vitis

Presentamos a lo largo de este capítulo lo que se conoce como síntesis de alto nivel (HLS) sobre tarjetas de aceleración basadas en FPGAs. Para ello, introducimos primero los conceptos básicos de esta tecnología y las herramientas que permiten explotarla, para después detallar concretamente los recursos sobre los que implementaremos en el capítulo 6 un algoritmo de desmezclado espectral con HLS, que es uno de los objetivos principales de este trabajo. Por esta razón, entramos en gran detalle en el funcionamiento de la síntesis de alto nivel, que constituye la base del desarrollo de capítulos posteriores.

4.1. Introducción a la síntesis de alto nivel (HLS)

Desde los inicios del hardware reconfigurable y la creación de las FPGAs en la década de 1980, se han venido desarrollando lenguajes de descripción del hardware (HDL), como Verilog [46] o VHDL [51], bajo los conocidos como sistemas VLSI (*Very Large Scale Integration*) [64]. Estos sistemas permiten definir la estructura, diseño y operación de circuitos electrónicos con una cierta abstracción respecto de los componentes físicos [44], pero con un mayor enfoque al diseño de hardware que un lenguaje de programación convencional como Java o C++. Permiten, a diferencia de un lenguaje *netlist*, no solo especificar los elementos de hardware necesarios, sino también su comportamiento. Añadido a esto, dan la posibilidad de simular y modelar los componentes electrónicos antes de que sean construidos físicamente.

Sin embargo, la creciente complejidad de los algoritmos y del propio hardware han hecho que sean necesarios lenguajes de descripción del hardware con un nivel de abstracción cada vez mayor [62]. Esto ha supuesto y sigue suponiendo un reto enorme en cuanto al balance entre la precisión de descripción del hardware y la facilidad de uso del lenguaje. Un paso en esta dirección se hizo con la abstracción de diseño RTL (*register-transfer level*), que buscaba poder modelar circuitos electrónicos síncronos en términos de flujos de señales digitales entre registros hardware, y las operaciones realizadas sobre ellas. Esto permite que el programador hardware pueda indicar los registros y las operaciones que se requieren para un cierto algoritmo sin preocuparse por cómo se implementan físicamente. Para más detalles acerca del diseño RTL ver [15].

Sin embargo, en 1998 la empresa de automatización *Forte Design Systems*, comprada en 2014 por la tecnológica *Cadence* [12], presenta una herramienta llamada *Cynthesizer* para diseño de hardware sobre SystemC [45] que supuso el inicio del auge de un método de diseño de hardware que aporta un nivel de abstracción todavía mayor: la síntesis de alto nivel (HLS). Esta herramienta fue inicialmente adoptada por muchas empresas japonesas debido a la gran comunidad de programadores de SystemC que había en Japón en aquel momento, pero no es hasta el año 2008 cuando esta nueva tecnología empieza a darse a conocer en el resto del mundo de parte de empresas como la tecnológica japonesa *Sony* o la propia *Cadence*.

La síntesis de alto nivel (HLS) tiene como objetivo principal que el diseñador de hardware pueda centrarse en el comportamiento abstracto de un programa o algoritmo durante el proceso de diseño, sin preocuparse por los registros individuales, las operaciones que se llevan a cabo ciclo a ciclo o los manejos específicos de las memorias utilizadas. Para ello, el diseñador captura el comportamiento de un algoritmo y lo plasma en una herramienta de HLS, normalmente bajo un lenguaje de programación de alto nivel como Java o C++, que se encarga de crear una microarquitectura RTL de manera automática.

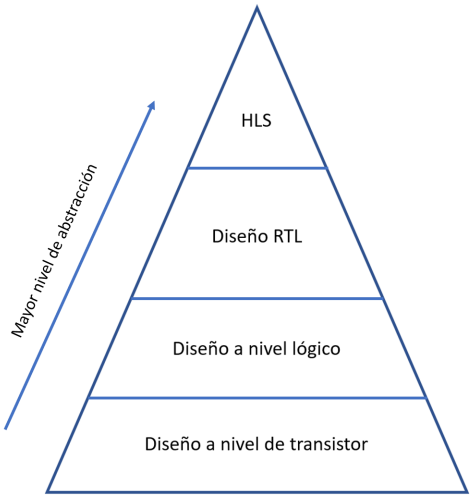


Figura 4.1: Niveles de abstracción en el diseño de circuitos electrónicos [62].

La posibilidad de tener una herramienta HLS con esta funcionalidad conlleva muchas ventajas para un diseñador de hardware. Tan solo se requiere una especificación de un algoritmo en un lenguaje de programación de alto nivel para que se genere un diseño RTL ejecutable en un dispositivo reconfigurable. Esto hace que el proceso de implementación del algoritmo se pueda hacer de forma mucho más rápida, sencilla y directa, sin tener que preocuparnos por los siguientes aspectos generales, que se procesan de manera automática:

- Lógica de control
- Uso de recursos hardware
- Paralelismo / Concurrencia del algoritmo
- Gestión de memoria
- Interfaces de conexión de los módulos hardware
- Uso de registros para limitar caminos críticos

Sin embargo, las herramientas HLS también suelen permitir especificaciones en forma de restricciones de diseño, latencia y grado de paralelismo, como veremos más adelante en este capítulo. Al manejo y uso de estas especificaciones a través de comandos o directivas sobre el propio código de alto nivel para obtener el mejor rendimiento en la ejecución de un algoritmo sobre un dispositivo reconfigurable se le conoce como aceleración por hardware, y es un concepto esencial en el diseño de algoritmos a través de HLS.

El núcleo de una herramienta HLS es un compilador que se encarga de transformar el código de alto nivel en la microarquitectura RTL. Por la propia naturaleza de la tarea que deben llevar a cabo, estos compiladores son muy complejos, lo que supone un verdadero reto desarrollarlos, y las pocas empresas que tienen recursos para hacerlo, como *Xilinx* [70] y *Cadence*, llevan años puliendo y mejorando los suyos. Esto no solo se traduce en tiempos de compilación muy largos y costosos en cuanto a recursos del procesador, sino que es uno de los principales cuellos de botella de esta nueva tecnología.

Adicionalmente, el desarrollo de estos sistemas para la implementación de algoritmos con HLS tiene dos importantes problemas añadidos: la optimalidad de las configuraciones hardware generadas y la capacidad del diseñador de detallar cómo quiere que se implemente su algoritmo. El hecho de abstraer el diseño hardware de un algoritmo hace que perdamos precisión, lo que muchas veces resulta en implementaciones menos óptimas. Además, aunque el proceso de implementación sea más sencillo y rápido, el diseñador pierde la posibilidad de indicar detalladamente los recursos hardware a utilizar, así como el grado de concurrencia y otros factores importantes.

En resumen, el diseño hardware utilizando HLS trae muchas ventajas a la hora de implementar algoritmos a través de lenguajes de programación de alto nivel, aportando un enfoque basado en el comportamiento de estos y no en la implementación concreta que se haga sobre el hardware. Sin embargo, también viene acompañado de múltiples desventajas relacionadas con la precisión y optimalidad de la implementación, que en los últimos años se han ido afrontando con nuevas ideas que pueden hacer que el diseño por HLS suponga una verdadera revolución en cuanto a la manera de reconfigurar circuitos electrónicos para implementar algoritmos sobre hardware.

4.2. *Xilinx Vitis Unified Software Platform*

Una de las herramientas HLS más importantes de los últimos años ha sido *Xilinx Vivado HLS* [68], creada por la empresa tecnológica Americana *Xilinx*, que a partir de la versión 2020.1 se pasó a conocer como *Vitis HLS* [67]. Actualmente se integra dentro de un entorno llamado *Xilinx Vitis Unified Software Platform* [65] [66], y, básicamente, se encarga de sintetizar una función en C / C++ a código RTL, que posteriormente se implementa en la región de lógica programable de una FPGA de *Xilinx*. El diseñador de hardware tan solo interactúa con el código C / C++ y los parámetros que *Vitis HLS* permite ajustar, y el resto del proceso se hace internamente en base a las anteriores indicaciones.

Su origen se remonta a abril del año 2012, cuando la compañía tecnológica *Xilinx* introduce *Vivado Design Suite* [69] como un entorno para la síntesis y el análisis de lenguajes de descripción de hardware (HDL), que además incluía módulos para HLS, integración de IPs, etcétera. El módulo encargado de dar capacidades de HLS se pasó a conocer como *Xilinx Vivado HLS*, hasta que en octubre del año 2019 *Xilinx* anuncia [3] una nueva plataforma conocida como *Xilinx Vitis Unified Software Platform* en donde incluye todas las capacidades de HLS que anteriormente estaban presentes en *Vivado Design Suite*, separando de manera efectiva el desarrollo de hardware en dos plataformas diferenciadas: una enfocada en desarrollo de más bajo nivel con HDL y capacidades RTL, y otra centrada en HLS y aplicaciones aceleradas.

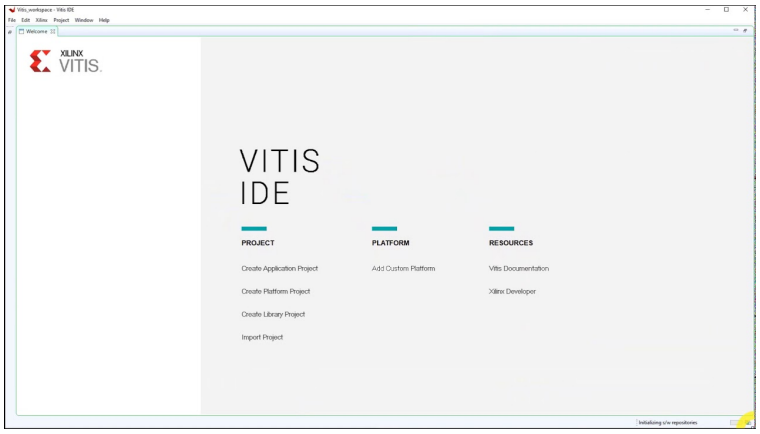


Figura 4.2: Pantalla principal y logo de *Xilinx Vitis Unified Software Platform*.

4.2.1. Proceso de diseño, modelo de ejecución y análisis de los resultados

Para desarrollar un algoritmo e implementarlo con *Vitis HLS* bajo *Xilinx Vitis Unified Software Platform*, el diseñador debe proporcionar y configurar los archivos que se describen a continuación, además de poseer una FPGA de *Xilinx* compatible.

- Kernel: Función especificada en C / C++ / RTL (Verilog o VHDL) que contiene el algoritmo que se quiere implementar sobre la parte programable de la FPGA. Se procesa con un compilador propio de Vitis (v++) para generar un archivo de objeto *Xilinx* (*.xo*), el cual posteriormente se integra con la información de la FPGA utilizada para obtener un archivo binario de dispositivo Vitis (*.xclbin*). Este último es el que finalmente se utiliza para lanzar el algoritmo en la FPGA a través de interfaces AXI (*Advanced eXtensible Interface*).

- Host: Programa escrito en C / C++ que interactúa con el Kernel proporcionándole los datos de ejecución y otros recursos. Una vez compilado con GNU C++ (g++), se ejecuta sobre una CPU convencional externa a la FPGA y utiliza unas APIs propias de *Xilinx* implementadas en la librería XRT (*Xilinx Runtime Library*) para interactuar con el Kernel.

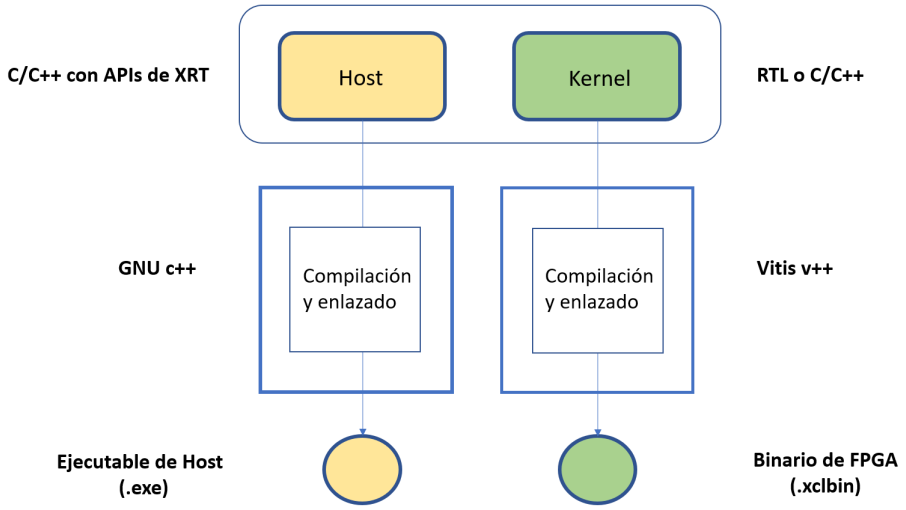


Figura 4.3: Proceso de compilación y enlazado del Host y Kernel en *Xilinx Vitis Unified Software Platform*. Obtención de los archivos *.exe* y *.xclbin* a partir del código fuente.

A modo de ejemplo sencillo de diseño y comunicación entre el Kernel y el Host, veamos cómo se implementaría un sumador de vectores sobre *Xilinx Vitis Unified Software Platform*. Comenzamos por analizar un ejemplo sencillo de código para el Kernel en C++.

```
extern "C" {

        void vadd (const int *in1, const int *in2, int *out, int size) {

            #pragma HLS INTERFACE m_axi port=in1 bundle=aximm1
            #pragma HLS INTERFACE m_axi port=in2 bundle=aximm2
            #pragma HLS INTERFACE m_axi port=out bundle=aximm1

            for(int i = 0; i < size; ++i)
                      out[i] = in1[i] + in2[i];

        }

}
```

Como se muestra en el código anterior, el Kernel no es más que una simple función en C++, que debe ser declarada como externa para evitar problemas con los cambios de nombre durante el enlazado, lo que muestra la potencia de las herramientas HLS a la hora de diseñar de forma rápida y eficaz algoritmos que se implementen después sobre hardware reconfigurable. Además, en el anterior código aparecen también, bajo el nombre de *pragma HLS*, lo que se conoce como directivas de aceleración, que serán muy importantes a la hora de trabajar con herramientas HLS.

Las directivas de aceleración o *pragmas* permiten dar indicaciones al compilador HLS de Vitis (v++) acerca de restricciones de latencia, concurrencia, refactorizaciones de código, manejo del *pipeline*, comportamiento de bucles y funciones, etcétera. Son una forma de controlar y dirigir el proceso de compilación según requerimientos de implementación que queramos que se tengan en cuenta, de forma que podamos controlar en cierta medida el diseño RTL resultante, y todo el proceso no sea una mera caja negra que no podamos controlar mas que en base a los inputs y outputs.

Por ejemplo, podemos indicar al compilador que no permita paradas de *pipeline* en una cierta región de código a costa de utilizar más recursos hardware de la FPGA, o podemos, como se está haciendo en el código anterior, asignar los distintos argumentos a diferentes puertos del Kernel de tal forma que se puedan leer y escribir en paralelo. Existen multitud de directivas de aceleración, que se pueden encontrar en [73], y en la implementación que se realiza en el capítulo 6 detallamos y utilizamos varias de ellas.

Sin embargo, y aunque *Xilinx Vitis Unified Software Platform* tenga compatibilidad casi completa con todas las estructuras propias de lenguajes como C++, hay ciertas restricciones respecto al uso de memoria dinámica u otros tipos de utilidades como *templates* que no están soportados por el compilador de Vitis v++. En el capítulo 6 nos encontraremos con algunas de estas restricciones, y en [76] pueden consultarse con gran nivel de detalle.

Para proporcionar los valores de los argumentos del Kernel y controlar su correcta ejecución desde la máquina host que estemos utilizando, veamos a continuación un ejemplo de código Host en C++ con la librería XRT que, debido a su gran extensión, explicamos en varios pasos. En el apéndice 9.2 puede encontrarse una versión más compleja de código Host usando librerías de OpenCl.

1. En primer lugar, incluimos en el preámbulo las librerías de C++ básicas que necesitemos y los diferentes módulos de la librería XRT de *Xilinx* para interactuar con el Kernel. El *main* comienza procesando los argumentos de entrada, localiza el dispositivo hardware que tengamos instalado en base a una convención de índices que identifican a los dispositivos, carga el binario *.xclbin*, y crea un objeto Kernel asociado al sumador de vectores.

```
// Librerías básicas de C++
#include <iostream>
#include <cstring>

// Módulos de la librería XRT
#include "xrt/xrt_bo.h"
#include <experimental/xrt_xclbin.h>
#include "xrt/xrt_device.h"
#include "xrt/xrt_kernel.h"

int main(int argc, char** argv) {

    // Procesado de los argumentos
    std::cout << "argc = " << argc << std::endl;
    for(int i=0; i < argc; i++)
            std::cout << "argv[" << i << "] = " << argv[i] << std::endl;

    // Localización del archivo .xclbin
    std::string binaryFile = "./vadd.xclbin";

    // Índice del dispositivo hardware instalado que se va a utilizar
    // Si sólo hay uno, por defecto se asigna al índice 0
    int device_index = 0;

    // Dispositivo hardware
    std::cout << "Open the device" << device_index << std::endl;
    auto device = xrt::device(device_index);

    // Carga del .xclbin en el dispositivo hardware
    std::cout << "Load the xclbin " << binaryFile << std::endl;
    auto uuid = device.load_xclbin("./vadd.xclbin");

    // Creacion del objeto Kernel
    auto krnl = xrt::kernel(device, uuid, "vadd",
                            xrt::kernel::cu_access_mode::exclusive);
    ...
```

2. Como segundo paso, se crean los *buffers* necesarios para compartir datos entre la memoria de la FPGA y el Kernel. Para ello, se conectan los argumentos del Kernel con el diseño RTL generado en base al dispositivo hardware que se utilice, y se asignan los *buffers* a regiones de memoria de la máquina Host. Por supuesto, las regiones de memoria utilizadas variarán según el tamaño de las entradas y salidas del Kernel.

```
...

// Asignación de argumentos del Kernel al diseño RTL
std::cout << "Allocate Buffer in Global Memory\n";
auto boIn1 = xrt::bo(device, vector_size_bytes, krnl.group_id(0));
auto boIn2 = xrt::bo(device, vector_size_bytes, krnl.group_id(1));
auto boOut = xrt::bo(device, vector_size_bytes, krnl.group_id(2));

// Asignación de los buffers a memoria de la máquina Host
auto bo0_map = boIn1.map<int*>();
auto bo1_map = boIn2.map<int*>();
auto bo2_map = boOut.map<int*>();
std::fill(bo0_map, bo0_map + DATA_SIZE, 0);
std::fill(bo1_map, bo1_map + DATA_SIZE, 0);
std::fill(bo2_map, bo2_map + DATA_SIZE, 0);

...
```

3. Por último, se cargan en los *buffers* los argumentos del Kernel y se realizan cuatro operaciones: transferir los valores de los argumentos a la memoria del dispositivo hardware, ejecutar el Kernel, esperar a que termine y recuperar las salidas obtenidas como resultado de la ejecución del algoritmo.

```
...

// Valores de los argumentos del Kernel
int bufReference[DATA_SIZE];
for (int i = 0; i < DATA_SIZE; ++i) {
    bo0_map[i] = i;
    bo1_map[i] = i;
}
std::cout << "synchronize input buffer data to device global memory\n";
boIn1.sync(XCL_BO_SYNC_BO_TO_DEVICE);
boIn2.sync(XCL_BO_SYNC_BO_TO_DEVICE);

// Ejecución del Kernel
std::cout << "Execution of the kernel\n";
auto run = krnl(boIn1, boIn2, boOut, DATA_SIZE);
run.wait();

// Recuperación de las salidas del Kernel
std::cout << "Get the output data from the device" << std::endl;
boOut.sync(XCL_BO_SYNC_BO_FROM_DEVICE);

}
```

Con los archivos Kernel y Host que acabamos de detallar tendríamos todo lo necesario para que el compilador de *Vitis HLS* sea capaz de generar un modelo RTL y gestionar las memorias de la máquina Host y el dispositivo hardware que estemos utilizando. Así, podemos ejecutar en una FPGA un cierto algoritmo que tengamos implementado en C++ en el Kernel con unos argumentos concretos que especifiquemos en el código del Host, que se envían y se recogen de memoria de la FPGA a través de la tecnología PCIe de *Xilinx* [11].

Una vez hayamos terminado el diseño y configuración del algoritmo que queramos implementar a través del Kernel y el Host, *Xilinx Vitis Unified Software Platform* nos proporciona tres modos distintos con los que ejecutar nuestra aplicación. Existen dos formas de emulación, que no requieren siquiera de una FPGA para lanzarse, y un modo de ejecución sobre hardware, que es el encargado de generar el *.exe* y el *.xclbin* que irán directamente a ejecutarse sobre la CPU de la máquina Host y la región reconfigurable de la FPGA respectivamente.

Es importante tener la posibilidad de emular nuestras aplicaciones por varios motivos, pero principalmente porque nos permite diferenciar dos etapas a la hora de implementar los algoritmos: una primera etapa de diseño y depuración utilizando entornos controlados bajo emulación para comprobar el correcto funcionamiento del algoritmo y detectar fácilmente errores de sintaxis, y una segunda etapa de despliegue y análisis de la ejecución real. Detallemos uno a uno los tres modos de ejecución:

- Emulación Software: Los códigos del Kernel y del Host son compilados para su ejecución sobre el procesador de la máquina host. Permite depurar de manera rápida y sencilla a nivel de código, pero no es representativo del comportamiento que tendrá el algoritmo al ejecutarse sobre la FPGA.
- Emulación Hardware: El código del Kernel se compila para generar un modelo hardware RTL que se ejecuta en un simulador dedicado. A pesar de que es más lenta, proporciona datos detallados de cómo se comportará la aplicación cuando se ejecute sobre el hardware físico, por lo que es una estimación muy buena de los resultados finales de ejecución sobre la FPGA.
- Hardware: El código del Kernel se compila para generar un modelo de hardware RTL y se implementa sobre la FPGA a través de un archivo de hardware binario *.xclbin*. Representa el paso final del proceso de implementación de un algoritmo en *Xilinx Vitis Unified Software Platform*, durante el cual se obtienen los datos reales de su implementación física.

Al terminar de ejecutar nuestra aplicación en alguno de los tres modos, obtendremos un reporte de síntesis correspondiente a la ejecución realizada, que nos permitirá analizar en detalle todo tipo de datos acerca de latencias, consumo de hardware en la FPGA, periodo de reloj, pasos de control, etcétera. En el capítulo 6 analizaremos en detalle varios reportes de síntesis completos.

4.3. Tarjetas de aceleración *Xilinx* Alveo

Aunque hasta ahora nos hemos referido al dispositivo de hardware reconfigurable que utiliza *Vitis Unified Software Platform* de modo genérico como FPGA, en realidad *Xilinx* ha desarrollado unos dispositivos conocidos como tarjetas de aceleración Alveo que están precisamente dedicados a flujos de diseño por HLS y a aceleración por hardware. Junto a estas tarjetas, *Xilinx* a creado un ecosistema completo de desarrollo de aplicaciones de alto rendimiento, que permite optimizar el proceso de implementación para hardware de alto rendimiento reconfigurable. La figura 4.4 muestra un esquema de este ecosistema.

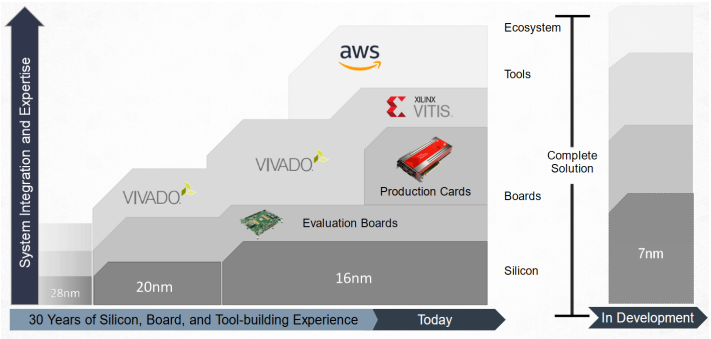


Figura 4.4: Evolución del ecosistema de *Xilinx*.

En concreto, a lo largo de este trabajo vamos a utilizar la tarjeta de aceleración Alveo U250. Sus manuales pueden encontrarse en [72]. La figura 4.5 muestra un diagrama de bloques de la Alveo U250, y en la tabla 4.1 se detallan sus especificaciones.

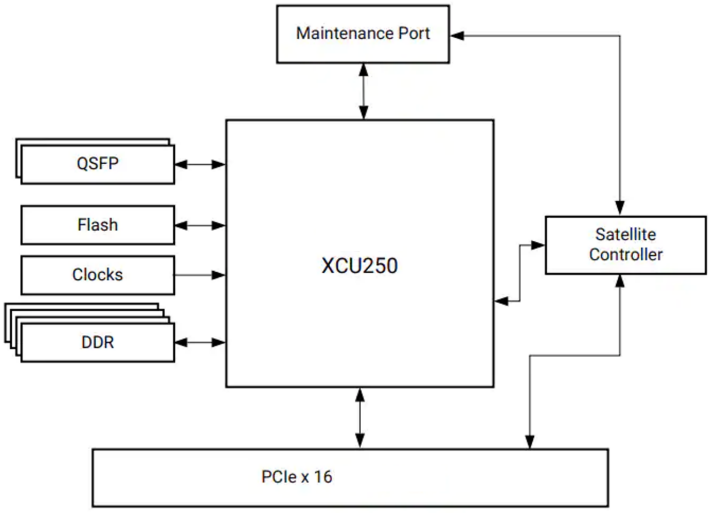


Figura 4.5: Diagrama de bloques de una tarjeta de aceleración Alveo U250.

La Alveo U250 es una tarjeta de aceleración de dos slots, con un factor de forma pequeño y un peso de alrededor de 1 Kg, alimentación activa (modelo A-U250-P64G-PQ-G) o pasiva (modelo A-U250-A64G-PQ-G) [17], y una potencia máxima de 225 W. Posee una FPGA UltraScale+ XCU250-2LFIGD2104E [74] con alrededor de 1,341,000 LUTs y más de 3 millones de registros, cuatro memorias RAM DDR4 de 16 GB con ancho de banda de 77 GB/s, ancho de banda SRAM interno de 38 TB/s, y posibilidad de llegar hasta los 33,3 INT8 TOPs. Soporta comunicaciones PCI Express (PCIe) Gen3x16 [11] de *Xilinx* para las conexiones con la máquina Host, tiene dos interfaces de red QSFP28 [24], y posee otros módulos con controladores y puertos de mantenimiento, así como la posibilidad de configurar una memoria flash QSPI adicional de 1 GB.

Especificaciones	Alveo U250
INT8 TOPs (Máx.)	33,3
Slots	2
Memoria externa DDR4	64 GB
Ancho de banda DDR4 externo	77 GB/s
Memoria interna SRAM	54 MB
Ancho de banda SRAM interno	38 TB/s
PCI Express	Gen3x16
Interfaces de red	2x QSFP28
LUTs	1,341,000
Potencia máxima	225 W
Alimentación	Activa (A-U250-P64G-PQ-G) / Pasiva (A-U250-A64G-PQ-G)

Tabla 4.1: Especificaciones de la tarjeta de aceleración Alveo U250.

Existen multitud de modelos de tarjetas de aceleración *Xilinx* Alveo con diferentes prestaciones y coste. Una detallada comparación entre las versiones U200, U250, U280, U50 y U55C puede encontrarse en [75]. En el caso de este trabajo, se ha seleccionado el modelo U250 por cuatro motivos principales: accesibilidad a través del programa HACC de ETH Zürich (ver sección 9.3), alta capacidad de cómputo, amplio almacenamiento, y compatibilidad con librerías de *Vitis HLS*. El algoritmo de desmezclado espectral (ver sección 2.3) que implementamos en *Xilinx Vitis Unified Software Platform* sobre esta tarjeta de aceleración *Xilinx* Alveo U250 se presenta en el capítulo 5.

Capítulo 5

Desmezclado espectral. Algoritmo SUnSAL

En este capítulo pasamos a estudiar en profundidad lo que se conoce como desmezclado espectral, y detallamos tanto los diferentes tipos que hay como varios usos importantes, de entre los cuales nos centramos en la detección de cambios aplicada a imágenes hiperespectrales. Por último, introducimos el algoritmo SUnSAL para desmezclado espectral, cuya implementación a través de HLS en el capítulo 6 es uno de los objetivos principales de este trabajo.

5.1. Desmezclado espectral

Como ya se expuso con detalle en la sección 2, trabajamos con imágenes hiperespectrales que dan información de las firmas espectrales de los materiales que están presentes en el área de detección, de forma que cada píxel cubre un área específica. Sin embargo, esto no quiere decir que cada píxel solo proporcione información de la firma de un material concreto, sino que prácticamente en la totalidad de los casos un píxel ha capturado el grado de reflectancia de múltiples materiales que caen bajo su área de detección.

Por tanto, a diferencia de otros tipos de imágenes, los píxeles de las imágenes hiperespectrales son en realidad una combinación de firmas espectrales de múltiples materiales, lo que dificulta saber la verdadera naturaleza de estos, llegando a ser un problema común incluso saber qué materiales son los que están presenten en cada píxel [54].

En concreto, este fenómeno ocurre por dos causas principales. La primera se debe a la baja resolución espacial de las imágenes hiperespectrales tomadas, que hace que en un mismo píxel se mezclen materiales que ocupan diferentes lugares, como se muestra en la figura 5.1. Así, la medición resultante en la imagen hiperespectral será la composición de las firmas individuales de cada material presente. La segunda se debe a la propia área detectada, que puede no tener materiales claramente definidos, sino que sea una mezcla homogénea de diferentes sustancias, por ejemplo arenas en una playa, cada una con su propios valores de reflectancia, como se muestra en la figura 5.2.

Lo que intenta precisamente el desmezclado espectral es determinar las firmas espectrales individuales de cada uno de los materiales que hay presentes en un píxel, a través de la determinación de las firmas espectrales de las sustancias puras presentes (que llamaremos *endmembers*) y sus abundancias, lo cual puede ser una tarea más fácil o más difícil según las características del área de detección. En general, las técnicas de desmezclado espectral se suelen dividir en dos grupos: las lineales y las no lineales.

Las técnicas de desmezclado espectral lineales suponen que la mezcla de firmas espectrales en un píxel es una combinación lineal de los *endmembers*, con una cierta distribución de pesos basada en la fracción de área en la que está presente cada sustancia pura. Esta suposición suele estar cerca de la realidad cuando los materiales de un píxel se localizan en zonas fijas separados unos de otros en forma de tablero de ajedrez, como se muestra en la figura 5.3. El algoritmo de desmezclado espectral SUnSAL que veremos más adelante es justamente una de estas técnicas.

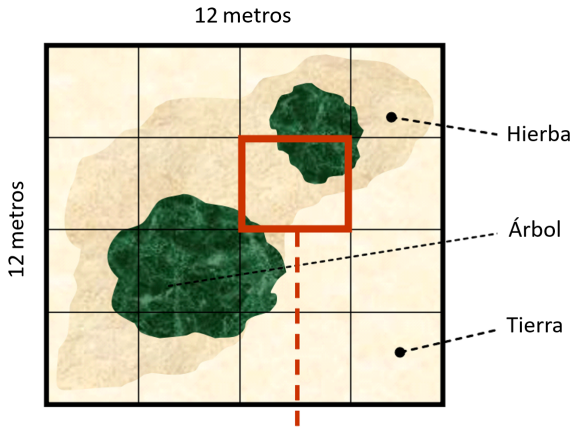


Figura 5.1: Mezcla macroscópica de 15 % tierra, 25 % árbol y 60 % hierba en un mismo píxel de 3 x 3 metros. Debido a la baja resolución, no hay ningún píxel que recoja solamente hierba. [54]

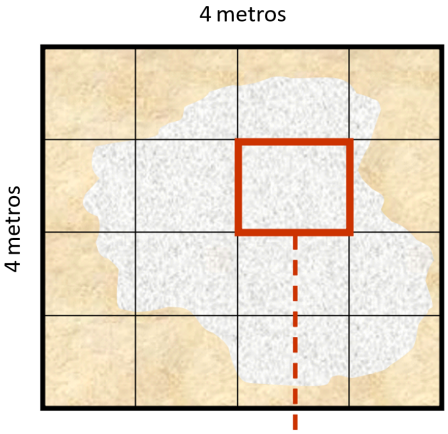


Figura 5.2: Mezcla homogénea con diferente grado de reflectancia en un mismo píxel. No es posible obtener una píxel con el grado de reflectancia de una sola sustancia en este área de detección. [54]

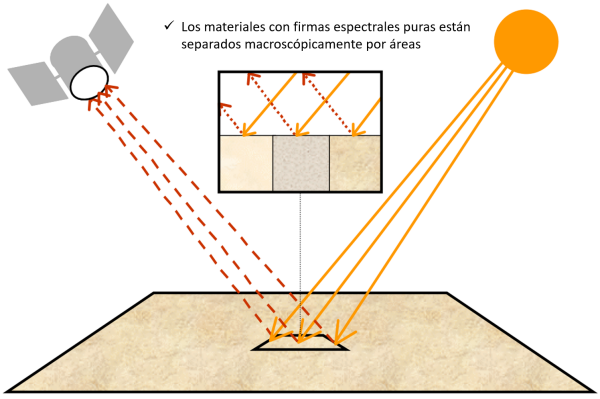


Figura 5.3: Escenario idóneo para la aplicación de técnicas de desmezclado espectral lineal. [54]

Por el contrario, las técnicas de desmezclado espectral no lineales suponen que las sustancias puras están distribuidas aleatoriamente en el área de detección de un píxel, que suele acercarse a la realidad, por ejemplo, en lugares con arenas mezcladas de distinta composición. Además, pueden tener en cuenta otros fenómenos externos a los materiales presentes en el área de detección, como interferencias atmosféricas, sombras, etcétera. La figura 5.4 muestra un esquema de este tipo de mezclas homogéneas.

Sin embargo, estas técnicas suelen basarse principalmente en redes neuronales y técnicas de inteligencia artificial avanzadas, que son más complicadas de implementar.

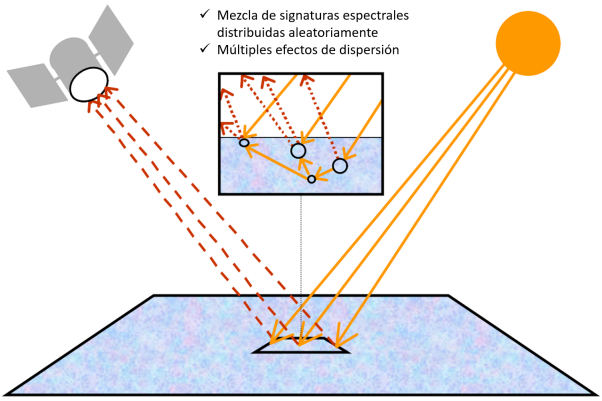


Figura 5.4: Escenario que requiere de la aplicación de técnicas de desmezclado espectral no lineal. [54]

El proceso completo de desmezclado espectral sobre una imagen suele comenzar con un preprocesamiento inicial que se encarga de comprimir la imagen utilizando técnicas de reducción dimensional mencionadas en la sección 2, para después hacer una selección de *endmembers* relevantes para el estudio que se vaya a hacer, y por último terminar obteniendo las abundancias de cada *endmember* seleccionado en el píxel concreto, tal y como se muestra en el esquema de la figura 5.5.

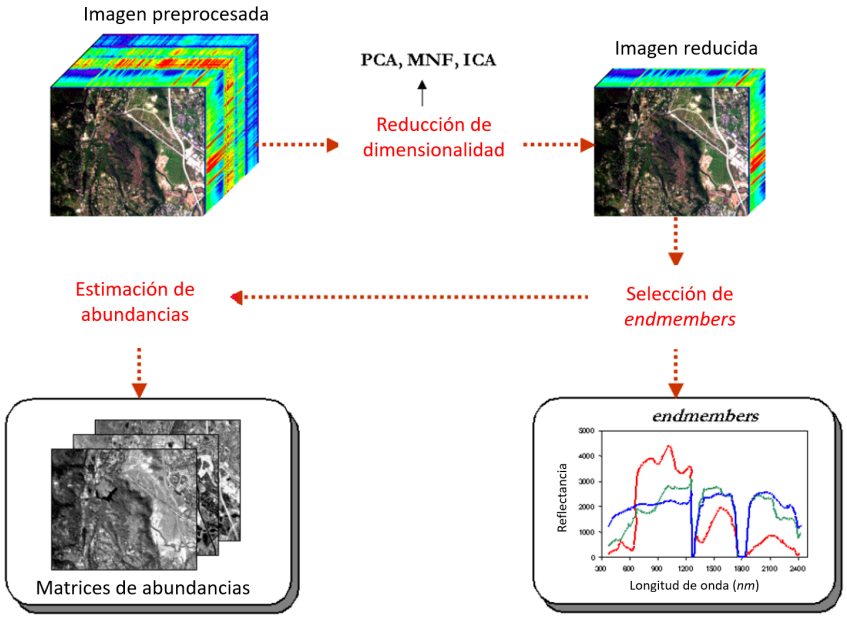


Figura 5.5: Proceso completo de desmezclado espectral desde la imagen hiperespectral original. [54]

5.2. Aplicaciones del desmezclado espectral

La aplicación principal del desmezclado espectral es, por tanto, la identificación de las sustancias puras que se encuentran en el área de detección de las imágenes hiperespectrales que estemos estudiando. Esto tiene como casos de uso importantes el análisis de rocas y minerales, la detección de explosivos y minas, el análisis del fondo marino, etcétera.

Sin embargo, hay multitud de aplicaciones derivadas que no son tan directas, de entre las que destaca la detección de cambios sobre el área de detección. El objetivo concreto que buscamos es determinar qué cambios ha sufrido un cierto área de la superficie terrestre a lo largo del tiempo, para medir, por ejemplo, los efectos del cambio climático sobre una zona concreta, la evolución del grado de urbanización o la monitorización de los efectos de catástrofes naturales.

Para ello, se parte de una serie de imágenes hiperespectrales tomadas en intervalos de tiempo fijos sobre un mismo área de detección y se aplica sobre cada una de las imágenes la misma técnica de desmezclado espectral en todos los píxeles para construir una matriz que da información sobre los *endmembers* presentes. Después, se superponen las matrices obtenidas para cada una de las imágenes y se comparan los resultados para analizar los cambios que ha habido en el área de detección a lo largo del tiempo en términos de los mencionados *endmembers*.

Hay infinidad de casos de uso para la detección de cambios a través de desmezclado espectral. Desde la monitorización de desastres naturales, pasando por la obtención de datos sobre la velocidad de expansión sobre superficies rocosas de plantas como las briofitas, hasta el control de los efectos del cambio climático como, por ejemplo, el sistema CoastNet (*Portuguese Coastal Monitoring Network*) [20].

Sin embargo, la detección de cambios también conlleva muchos problemas en su ejecución, puesto que la necesidad de tomar múltiples imágenes hiperespectrales de un mismo área de detección a lo largo del tiempo requiere de un control muy preciso del ruido atmosférico que pueda haber, de posibles descuadres de los píxeles, para lo cual son necesario sensores muy precisos y estables, y de los cambios que haya sobre la propia superficie que se está fotografiando.

Figura 5.6: Ejemplo del resultado de un proceso de detección de cambios a través de desmezclado espectral sobre imágenes tomadas por el satélite Sentinel-2.

La figura 5.6 muestra dos imágenes hiperespectrales captadas por el satélite Sentinel-2 [53] como parte del programa de ONERA (*Office National d'Etudes et de Recherches Aérospatiales*) [49]. La imagen de la derecha representa la diferencia entre los mapas de abundancias, con píxeles de color blanco para aquellos que han mostrado un cambio sustancial en sus *endmembers*.

5.3. Desmezclado espectral lineal. Algoritmo SUnSAL

5.3.1. Técnicas de desmezclado espectral lineal

Una vez vistos los conceptos del desmezclado espectral y varias de sus aplicaciones, nos centramos en el desmezclado espectral lineal, al que pertenece el conocido como algoritmo SUnSAL (*Spectral Unmixing by Variable Splitting and Augmented Lagrangian*), cuya implementación supone uno de los objetivos principales de este trabajo. Fue publicado por primera vez en [8], y se han desarrollado ya multitud de variantes enfocadas en mejorar aspectos concretos de este, como las presentadas en [29] y [63].

A lo largo de los últimos años se han desarrollado una gran variedad de técnicas para afrontar el problema del desmezclado espectral lineal, entre las que encontramos técnicas estadísticas desde el enfoque bayesiano como las que se pueden encontrar en [33], técnicas geométricas basadas en búsquedas de los vértices de un politopo correspondientes a los *endmembers* de modo semejante a cómo opera el simplex, y un largo etcétera. Una recopilación detallada de ellas se puede encontrar en [52].

El algoritmo SUnSAL, en cambio, busca atacar el problema del desmezclado espectral a través de una combinación de las conocidas como técnicas de regularización explicadas en [33] y el uso de una librería con las firmas espectrales de unos *endmembers* elegidos *a priori*, que va a ser la encargada de concentrar la información que se va a utilizar para estimar los *endmembers* en los píxeles de la imagen hiperespectral que estemos estudiando. De este modo, no es necesario hacer un procesamiento previo de la imagen para seleccionar los *endmembers*, sino que estos se deben determinar en el momento de construir la librería.

Comencemos presentando formalmente los elementos con los que vamos a trabajar y la base matemática sobre la que se construye el algoritmo SUnSAL, a través de tres problemas de optimización que vamos a detallar de manera rigurosa. En realidad, el algoritmo SUnSAL tiene varias versiones según el problema de optimización que se esté resolviendo, pero nosotros nos vamos a centrar solamente en una de ellas, que definiremos más adelante y que es la que nos interesa para el desmezclado espectral.

5.3.2. Base matemática y derivación del algoritmo SUnSAL

Sea $k \in \mathbb{N}$ el número de bandas espectrales de las imágenes hiperespectrales consideradas, $n \in \mathbb{N}$ el número de *endmembers* considerados, y $A \in \mathbb{R}^{k\times n}$ una matriz que representa la librería con la firma espectral de los n *endmembers*. Supongamos fijado un vector $y \in \mathbb{R}^k$ con la información de un píxel, y consideremos el problema de hallar el vector de abundancias $x \in \mathbb{R}^n$ tal que

$$Ax = y. \tag{5.1}$$

Si consideramos los conceptos definidos en [35] en relación a lo que se conoce como problemas directos e inversos, básicamente estamos proponiendo un problema inverso en términos de una ecuación de Fredholm de primer tipo sobre los espacios de Hilbert finitos y separables $\mathbb{R}^k$ y $\mathbb{R}^n$, con el operador $A\colon \mathbb{R}^n \to \mathbb{R}^k$ inducido por la matriz A.

Ahora, con el objetivo de resolver la ecuación (5.1) sin caer en los errores detallados en [33] que muchas veces van asociados al estudio de los problemas inversos, aplicamos diversas técnicas de regularización para obtener los diferentes problemas que van a servir de base para la construcción del algoritmo SUnSAL. Según si aplicamos la regularización *L2* [10] con la restricción $x \geq 0$, la regularización *L2* con las restricciones $x \geq 0$ y $1^T x = 1$, o un tipo de regularización de Tikhonov [33] con $x \geq 0$, obtenemos los problemas CLS (*Constrained Least Squares*), FCLS (*Fully Constrained Least Squares*) y CSR (*Constrained Sparse Regression*) respectivamente, definidos de la siguiente manera:

$$\begin{aligned}
(P_{CLS})\colon\ & \min_x \frac{1}{2}\|Ax-y\|_2^2 \\
& \text{sujeto a}\colon x \geq 0, \\
\\
(P_{FCLS})\colon\ & \min_x \frac{1}{2}\|Ax-y\|_2^2 \\
& \text{sujeto a}\colon x \geq 0,\ 1^T x = 1, \\
\\
(P_{CSR})\colon\ & \min_x \frac{1}{2}\|Ax-y\|_2^2 + \lambda\|x\|_1 \\
& \text{sujeto a}\colon x \geq 0,
\end{aligned} \tag{5.2}$$

donde $\|\cdot\|_1, \|\cdot\|_2 : \mathbb{R}^k \to \mathbb{R}$ denotan las normas *L1* y *L2*, y $\lambda > 0$ es un parámetro de regularización. Notar que el problema CLS no es más que un caso particular del problema CSR con $\lambda = 0$.

Para resolver este tipo de problemas de minimización, vamos a recurrir a un algoritmo iterativo conocido como el ADMM (*Alternating Direction Method of Multipliers*), que se deduce del siguiente teorema, cuyo desarrollo teórico puede encontrarse en [9].

Teorema 1. *Consideremos el problema*

$$\min_{x\in\mathbb{R}^n} f_1(x) + f_2(Gx), \tag{5.3}$$

donde $f_1\colon \mathbb{R}^n \to \mathbb{R}$ *y* $f_2\colon \mathbb{R}^p \to \mathbb{R}$ *son cerradas, y* $G \in \mathbb{R}^{p\times n}$ *tiene rango* n. *Sean* $\mu > 0$ *y* $u_0, d_0 \in \mathbb{R}^p$. *Finalmente, tomemos tres sucesiones* $(x_k)_{k=1}^{\infty} \subset \mathbb{R}^n$, $(u_k)_{k=1}^{\infty} \subset \mathbb{R}^p$ *y* $(d_k)_{k=1}^{\infty} \subset \mathbb{R}^p$ *que satisfagan*

$$\begin{aligned} x_{k+1} &= \arg\min_x f_1(x) + \frac{\mu}{2}\left\|Gx_k - u_k - d_k\right\|_2^2, \\ u_{k+1} &= \arg\min_u f_2(u) + \frac{\mu}{2}\left\|Gx_{k+1} - u_k - d_k\right\|_2^2, \\ d_{k+1} &= d_k - (Gx_{k+1} - u_{k+1}). \end{aligned}$$

Entonces, si (5.3) tiene alguna solución $x_s \in \mathbb{R}^n$, $\lim_{k\to\infty} x_k = x_s$. *En caso contrario, alguna de las sucesiones* $(u_k)_{k=1}^{\infty}$ *o* $(d_k)_{k=1}^{\infty}$ *diverge.*

Algoritmo 1: Algoritmo ADMM

Data: $\mu > 0$, $u_0, v_0 \in \mathbb{R}^p$
$k \leftarrow 0$
repeat
  $x_{k+1} \leftarrow \arg\min_x f_1(x) + \frac{\mu}{2}\left\|Gx - u_k - d_k\right\|_2^2$
  $u_{k+1} \leftarrow \arg\min_u f_2(u) + \frac{\mu}{2}\left\|Gx_{k+1} - u - d_k\right\|_2^2$
  $d_{k+1} \leftarrow d_k - (Gx_{k+1} - u_{k+1})$
  $k \leftarrow k + 1$
until *criterio de parada*;

El problema que va a resolver la versión del algoritmo SUnSAL que vamos a desarrollar es justamente el (P_{FCLS}) definido en (5.2), por lo que tenemos que ajustarlo para poder aplicar el teorema 1. Con este objetivo, escribimos de manera equivalente su formulación como

$$(P_{FCLS})\colon \min_x \frac{1}{2}\left\|Ax - y\right\|_2^2 + \iota_{\{1\}}(1^T x) + \iota_{\mathbb{R}^n_+}(x),$$

donde ι_S es la función indicatriz del conjunto S, que sustituye a las restricciones de $x \geq 0$ y $1^T x = 1$. Ahora, tomamos

$$\begin{aligned} f_1(x) &= \frac{1}{2}\left\|Ax - y\right\|_2^2 + \iota_{\{1\}}(1^T x), \\ f_2(u) &= \iota_{\mathbb{R}^n_+}(u), \\ G &= I, \end{aligned}$$

y aplicando el teorema 1 obtenemos la siguiente versión del algoritmo 1 anterior.

Algoritmo 2: Algoritmo ADMM para FCLS

Data: $\mu > 0$, $u_0, v_0 \in \mathbb{R}^p$
$k \leftarrow 0$
repeat
  $x_{k+1} \leftarrow \arg\min_x \frac{1}{2}\left\|Ax - y\right\|_2^2 + \iota_{\{1\}}(1^T x) + \frac{\mu}{2}\left\|x - u_k - d_k\right\|_2^2$
  $u_{k+1} \leftarrow \arg\min_u \iota_{\mathbb{R}^n_+}(u) + \frac{\mu}{2}\left\|x_{k+1} - u - d_k\right\|_2^2$
  $d_{k+1} \leftarrow d_k - (x_{k+1} - u_{k+1})$
  $k \leftarrow k + 1$
until *criterio de parada*;

Analizándolo línea a línea, vemos que u_{k+1} y d_{k+1} ya podemos calcularlos de manera sencilla, puesto que claramente

$$\arg\min_u \iota_{\mathbb{R}^n_+}(u) + \frac{\mu}{2}\|x_{k+1} - u - d_k\|_2^2 = x_{k+1} - d_k,$$

y solo resta encontrar la forma de resolver el cálculo de x_{k+1}, para lo cual recurrimos de nuevo a la resolución del problema de minimización de la regularización extendida de Tikhonov presentada en [33], que nos dice que

$$\min_x \frac{1}{2}\|Ax - y\|_2^2 + \iota_{\{1\}}(1^T x) + \frac{\mu}{2}\|x - u_k - d_k\|_2^2 = B^{-1}w - C(1^T B^{-1}w - 1),$$

donde

$$\begin{aligned} B &= A^T A + \mu I, \\ C &= B^{-1}1(1^T B^{-1}1)^{-1}, \\ w &= A^T y + \mu(u_k + d_k), \end{aligned}$$

de forma que llegamos por fin al algoritmo SUnSAL para la resolución del problema FCLS.

Algoritmo 3: Algoritmo SUnSAL para FCLS

Data: $\mu > 0$, $u_0, v_0 \in \mathbb{R}^p$
$k \leftarrow 0$
repeat
  $w \leftarrow A^T y + \mu(u_k + d_k)$
  $x_{k+1} \leftarrow B^{-1}w - C(1^T B^{-1}w - 1)$
  $v_k \leftarrow x_{k+1} - d_k$
  $u_{k+1} \leftarrow \max(0, v_k)$
  $d_{k+1} \leftarrow d_k - (x_{k+1} - u_{k+1})$
  $k \leftarrow k + 1$
until *criterio de parada*;

En el capítulo 6 vamos a implementar esta versión del algoritmo SUnSAL con HLS a través de la plataforma de *Xilinx Vitis Unified Software Platform*, utilizando la tarjeta de aceleración *Xilinx* Alveo U250 que ya se detalló en el capítulo 4. El pseudocódigo del algoritmo 3 nos servirá como punto de partida teórico, y aseguraremos la corrección de la implementación en base a él.

Capítulo 6

Implementación en *Vitis HLS* del algoritmo SUnSAL

En este capítulo damos una explicación detallada de la implementación en *Xilinx Vitis Unified Software Platform* del algoritmo SUnSAL para desmezclado espectral presentado en la sección 5.3.2. Principalmente, nos centraremos en el proceso de diseño de los archivos asociados al Kernel y al Host introducidos en la sección 4.2.1, la aceleración por hardware llevada a cabo con directivas de aceleración en base a los reportes de síntesis, y el análisis de los resultados considerando todo lo presentado en capítulos anteriores. La tabla 6.1 muestra los datos exactos de configuración para la implementación, incluyendo las versiones exactas de las herramientas y programas utilizados. Adicionalmente, todo el código desarrollado que se detalla a lo largo de este capítulo puede encontrarse completo en [26] y en el apéndice 9.4.

Entorno	*Xilinx Vitis Unified Software Platform* 2022.1
Sistema operativo	Linux, 64-bit, Ubuntu 20.04.4 LTS
Tarjeta de aceleración	*Xilinx* Alveo U250
Instalación	Ver sección 9.1

Tabla 6.1: Configuración detallada para la implementación del algoritmo SUnSAL.

6.1. Implementación inicial en el lenguaje de alto nivel Python

Partiendo del pseudocódigo presentado en el algoritmo 3 y previa la implementación de SUnSAL para el problema FCLS sobre *Xilinx Vitis Unified Software Platform*, desarrollamos una versión sobre Python que mantenga la corrección del mencionado pseudocódigo. Esto hará más sencillo la posterior adaptación a código para HLS, permitirá sobrepasar el primer escollo de traducir el pseudocódigo a un lenguaje iterativo, ayudará a detectar la modularidad del algoritmo y pondrá de manifiesto la necesidad de incluir ciertos pasos de normalización de las matrices y vectores involucrados, así como un control de la norma de estos a través del cálculo de residuos que tenemos que manejar durante las sucesivas iteraciones.

En primer lugar, añadimos las librerías de Python necesarias para facilitar esta implementación inicial e incluimos los argumentos relativos al criterio de parada ($iters$), la tolerancia de error (tol), los *inputs* que representan la librería espectral ($A \in \mathbb{R}^{k \times n}$) y el vector de un píxel concreto ($y \in \mathbb{R}^k$), y un valor inicial $x_0 \in \mathbb{R}^n$ para la incógnita.

```
import sys
import scipy as sp
import numpy as np
import scipy.linalg as splin
from numpy import linalg as LA

def sunsal(A, y, iters = 1000, tol = 1e-4, x0 = None):

        ...
```

En segundo lugar, debido a que la matriz A y el vector y dados como argumentos pueden no tener la misma escala de medida y el algoritmo SUnSAL no hace suposiciones sobre la distribución de los datos, es importante normalizar los datos. Para ello, calculamos la norma de la matriz A corregida por un cierto factor que depende del número de componentes $n \geq 0$ del vector x, y reescalamos acordemente.

```
...

# Normalización
norm_A = splin.norm(A) * (25 + n) / float(n)
# Reescalar A e y
A = A / norm_A
y = y / norm_A

...
```

Ahora, escogemos un valor adecuado para el parámetro $\mu > 0$ proveniente del teorema 1, y precalculamos todas las matrices y vectores que no varían a lo largo de las iteraciones del algoritmo.

- Para calcular B^{-1} vamos a utilizar la descomposición en valores singulares (SVD) [33] aplicada a la expresión
$$B = A^T A + \mu I.$$
Con esto en mente, hacemos el producto $A^T A$ y lo descomponemos de la siguiente forma:
$$A^T A = USV^T,$$
donde $U, V^T \in \mathbb{R}^{n \times n}$ son matrices ortogonales que cumplen $U = V$ por ser $A^T A$ simétrica, y $S \in \mathbb{R}^{n \times n}$ es una matriz diagonal con elementos en la diagonal $\lambda_1, \ldots, \lambda_n \geq 0$. Ahora, B^{-1} es sencilla de calcular y viene dada por:
$$B^{-1} = U\overline{S}U^T,$$
donde
$$\overline{S} = \begin{bmatrix} \frac{1}{\lambda_1+\mu} & & & \\ & \frac{1}{\lambda_2+\mu} & & \\ & & \ddots & \\ & & & \frac{1}{\lambda_n+\mu} \end{bmatrix}.$$

- La matriz $C = B^{-1}1(1^T B^{-1}1)^{-1}$ puede parecer compleja de calcular, pero si analizamos bien su expresión, $1^T B^{-1}1$ no es más que la suma de todos los elementos de B^{-1} y $B^{-1}1$ su suma por filas. Luego, C es el cociente entre la suma por filas de B^{-1}, que es un vector, y el valor de la suma de todos sus elementos.

- Por último, en el algoritmo 3 el valor que toma la solución x_{k+1} en cada iteración viene dada por la expresión
$$x_{k+1} = B^{-1}w - C(1^T B^{-1}w - 1),$$
donde
$$w = A^T y + \mu(u_k + d_k).$$
Hay una forma compacta de manejar estos valores y tener precalculada una matriz $\overline{B^{-1}}$ (denotada por IB1 en el código Python) que simplifique las operaciones a realizar. En efecto, si denotamos por B_f, B_c y B_t a la suma por filas, la suma por columnas y la suma total de B^{-1} respectivamente, tenemos que
$$x_{k+1} = B^{-1}w - C(1^T B^{-1}w - 1) = B^{-1}w - \frac{B_f}{B_t}(B_c w - 1) = (B^{-1} - \frac{B_f B_c}{B_t})w - \frac{B_f}{B_t}1.$$
Tomando ahora $\overline{B^{-1}} = B^{-1} - \frac{B_f B_c}{B_t}$ y $\text{Aux} = \frac{B_f}{B_t}1$ basta únicamente un producto matriz-vector y una suma de vectores para poder calcular x_{k+1} en cada iteración.

```
...

mu = 0.01

# Descomposición en valores singulares (svd)
[U,S] = splin.svd(sp.dot)(A.T,A))[:2]

# Inversa de B
IB = sp.dot( sp.dot(U,sp.diag(1./(S+mu))) , U.T )

# C
C = (1./IB.sum()) * sp.sum(IB,axis=1,keepdims=True)

# Aux
Aux = sp.sum(C,axis=1,keepdims=True)

# Matriz auxiliar IB1
IB1 = IB - sp.dot(C,sp.sum(IB,axis=0,keepdims=True))

# Traspuesta de A por y
yy = sp.dot(A.T,y)

...
```

A continuación, damos valores a los primeros elementos de las sucesiones del teorema 1 y fijamos una solución inicial x_0. Además, con el objetivo de evitar que por errores de precisión la solución no tome valores muy grandes que puedan incluso saturar los tipos de datos con los que trabajamos, y para ir midiendo el error cometido en cada iteración, introducimos dos residuos (residuo primal y residuo dual) que van a servir para controlar la norma de los vectores que vamos actualizando en cada iteración.

El desarrollo teórico del efecto y la importancia de los residuos primal y dual puede encontrarse en [9] y está ligado al teorema 1. Para cada iteración $k = 1, 2, ...$ vienen dados por

$$res_p = x_{k+1} - u_{k+1}, \quad res_d = \mu(u_{k+1} - u_k),$$

y tan solo los calcularemos cada diez iteraciones completas. Si en algún momento difieren en un factor multiplicativo mayor que 10, actualizaremos B^{-1} y Aux a través de la modificación del parámetro μ, que reescalará los anteriores elementos.

```
...

# Valor inicial de la solución
if x0 is None:
        x = sp.dot( sp.dot(IB,A.T) , y)
else:
        x = x0

# Valores iniciales de u y d
u = x
d  = 0

# Tolerancia de error. Si conseguimos
# mayor precisión que este valor en alguna
# iteración, paramos.
tol = sp.sqrt(k*n)*tol

# Índice de las iteraciones
i=1
```

```
                # Residuos primal y dual
                res_p = sp.inf
                res_d = sp.inf

                mu_changed = 0

                ...
```

Por último, tan solo queda escribir el cuerpo del bucle principal que realiza las iteraciones en base al algoritmo 3. Esto lo hacemos considerando lo comentado anteriormente en relación al control de los residuos, los vectores y matrices precalculados, y las condiciones de parada.

```
        ...

        while (i <= iters) and ((abs(res_p) > tol) or (abs(res_d) > tol)))

                # Guardar u_{k} para calcular posteriormente los residuos
                if (i%10) == 1:
                        u0 = u

                # u_{k+1}
                u = sp.maximum(x - d,0)
                # x_{k+1}
                x = sp.dot(IB1,yy + mu*(u+d)) + Aux
                # d_{k+1}
                d -= (x - u)

                # Actualizar mu para mantener los residuos bajo un factor de 10
                if (i%10) == 1:

                        # Residuo primal
                        res_p = splin.norm(x - u)

                        # Residuo dual
                        res_d = mu * splin.norm(u - u0)

                        # update mu
                        if res_p > 10*res_d:
                                mu = mu*2
                                d = d/2
                                mu_changed = True
                        elif res_d > 10*res_p:
                                mu = mu/2
                                d = d*2
                                mu_changed = True

                        if  mu_changed:
                                # Actualizar IB e IB1
                                IB = sp.dot( sp.dot(U,sp.diag(1./(S+mu))) , U.T )
                                C = (1./IB.sum()) * sp.sum(IB,axis=1,keepdims=True)
                                Aux = sp.sum(C,axis=1,keepdims=True)
                                IB1 = IB - sp.dot(Aux,sp.sum(IB,axis=0,keepdims=True))
                                mu_changed = False

                i+=1

        }
}
```

6.2. Adaptación a Vitis HLS

Una vez tenemos implementado en Python el algoritmo SUnSAL en su versión para resolver el problema FCLS, pasamos a diseñar ya sobre *Xilinx Vitis Unified Software Platform* el archivo Kernel de nuestra aplicación. Esto implica dos pasos importantes: traducir el código en Python a C++ y adaptar el código resultante en C++ a las restricciones que impone el compilador de Vitis (v++).

El primer paso es sencillo, pues requiere tan solo traducir las librerías de Python en módulos o librerías de C++ y adaptar la sintaxis del resto del código para que sea compatible con el compilador GNU C++ (g++). Sin embargo, el segundo paso es más complicado debido a que perdemos la posibilidad de utilizar muchas librerías y estructuras de datos propias de C++, teniendo que trabajar bajo las restricciones del compilador de Vitis (v++) que se detallan en [76]. En general, esto no supone un problema por lo altamente compatible que es v++ con código C++ de todo tipo, pero sí que habrá que tener cuidado con ciertas cuestiones que detallamos según presentamos el código.

Para facilitar el seguimiento de todo el proceso de traducción del código a C++ adaptado a Vitis (v++) iremos poniendo conjuntamente el extracto de código en Python al que corresponde cada sección de código en C++, de forma que se puede comprobar fácilmente cómo se ha hecho para diseñar todo el algoritmo en *Xilinx Vitis Unified Software Platform*.

Comenzamos por implementar las funciones necesarias para hacer el producto de matrices y vectores que antes realizábamos con ayuda de la librería *Scipy* de Python, así como las funciones para trasponer matrices y realizar la normalización previa al bucle principal del algoritmo. Además, aprovechamos para introducir dos cuestiones importantes a tener en cuenta a la hora de ir escribiendo el código del Kernel: el nombrado de bloques de código y la asignación de memoria.

Por un lado, dar nombres a ciertos bloques de código como bucles o grupos de instrucciones secuenciales es una manera sencilla de facilitar el posterior análisis de los resultados expuestos en los reportes de síntesis que obtendremos. Si no identificamos a estas regiones de código, el compilador de Vitis (v++) asignará nombres de forma interna a las instrucciones según la modularidad del código, y será complicado determinar qué valores de latencia se corresponden con cada bucle, función, estructura condicional, etcétera. Sin embargo, si nombramos los bloques de código en los que estamos interesados, el reporte de síntesis mostrará todos los datos en función de estos identificadores.

Por otro lado, una de las restricciones más importantes del compilador de Vitis (v++) es que no permite asignación de memoria dinámica. El código del Kernel debe permitir la reserva de unos recursos hardware concretos y una región de memoria fija, por lo que no podemos utilizar funciones como *malloc* o *free* que creen y destruyan zonas de memoria de tamaño variable en tiempo de ejecución. Todos los tipos y estructuras de datos que utilicemos deben tener un tamaño prefijado que no varíe a lo largo del código. Esto no significa que no se puedan usar punteros, pero sí restringe el uso que se les puede dar. Para más información, ver [76].

```
R = sp.dot(A,B)         # Producto de matrices A y B
```

```
static void producto_NporKporM(float A[N][K], float B[K][M], float R[N][M]) {
        producto_NporKporM_outer_loop: for (int i = 0; i < N; i++) {
                producto_NporKporM_inner_loop: for (int j = 0; j < M; j++) {
                        float Rij = 0;
                        producto_NporKporM_innermost_loop: for (int k = 0; k < K; k++)
                                Rij += A[i][k] * B[k][j];
                        R[i][j] = Rij;
                }
        }
}
```

```
R = sp.dot(A,y)         # Producto de una matriz A con un vector y
```

```
static void producto_vector_NporK(float A[N][K], float y[K], float R[N]) {
        producto_vector_NporK_outer_loop: for(int i = 0; i < N; i++) {
                float Ri = 0;
                producto_vector_NporK_inner_loop: for(int j = 0; j < K; j++) {
                        Ri += A[i][j] * y[j];
                }
                R[i] = Ri;
        }
}
```

```
A.T                # Matriz traspuesta
```

```
static void traspuesta_KporN(float A[K][N], float R[N][K]) {
        traspuesta_KporN_outer_loop: for(int i = 0; i < N; i++)
                traspuesta_KpoN_inner_loop: for(int j = 0; j < K; j++)
                        R[i][j] = A[j][i];
}
```

```
norm_A = splin.norm(A) * (25 + n) / float(n)        # Normalización
A = A / norm_A
y = y / norm_A
```

```
static void reescalado(float A[K][N], float y[K]) {
        float norm_f = 0;
        norma_frobenius_outer_loop: for(int i = 0; i < K; i++) {
                norma_frobenius_inner_loop: for(int j = 0; j < N; j++) {
                        float Aij = A[i][j];
                        norm_f += Aij * Aij;
                }
        }
        norm_f = hls::sqrt(norm_f);
        norm_f = norm_f * (25 + N) / N;
        reajuste_A_outer_loop: for(int i = 0; i < K; i++)
                reajuste_A_inner_loop: for(int j = 0; j < N; j++)
                        A[i][j] = A[i][j] / norm_f;
        reajuste_y_loop: for(int i = 0; i < K; i++)
                y[i] = y[i] / norm_f;
}
```

A continuación, debemos hallar una forma de calcular la descomposición en valores singulares que tenga el mismo comportamiento que el correspondiente método de la librería *Scipy* de Python. Por suerte, *Vitis HLS* posee actualmente algunas librerías recopiladas bajo el nombre de *Vitis Libraries* [77] con funciones básicas aceleradas de álgebra lineal, entre las que se incluye la descomposición en valores singulares. Sin embargo, debido a que se encuentran en desarrollo activo en *Xilinx*, todavía no hay incluidas librerías para, por ejemplo, el producto de matrices. En concreto, la librería para la descomposición en valores singulares se encuentra dentro del módulo con nombre *Solver* [78].

```
[U,S] = splin.svd(sp.dot)(A.T,A))[:2]        # Descomposición en valores singulares (svd)
```

```
float v[N][N] = {{}};
xf::solver::svdTop<N, N, xf::solver::svdTraits<N, N, float, float>,
        float, float>(AtA, S, U, v);
```

Ahora, y una vez calculadas las matrices de la descomposición en valores singulares $A^TA = USU^T$, tenemos que determinar la matriz inversa de B, el vector C y la matriz auxiliar que llamamos IB1.

```
IB = sp.dot( sp.dot(U,sp.diag(1./(S+mu))) , U.T )        # Inversa de B
C = (1./IF.sum()) * sp.sum(IF,axis=1,keepdims=True)
Aux = sp.sum(C,axis=1,keepdims=True)
IB1 = IB - sp.dot(C,sp.sum(IB,axis=0,keepdims=True))
```

```
static void calcular_IB1(float U[N][N], float Ut[N][N], float S[N][N],
        float mu, float C[N], float IB[N][N], float IB1[N][N]) {
        float S[N][N] = {{}}, SUt[N][N] = {{}}, Cspc[N][N] = {{}};
        float suma_por_filas[N] = {}, suma_por_columnas[N] = {};
        float suma_total_IB = 0;
        calculo_S_loop: for(int i = 0; i < N; i++)
                S[i][i] = 1 / (S_0[i][i] + mu);
        producto_NporNporN(S, Ut, SUt);
        producto_NporNporN(U, SUt, IB);
        sumas_outer_loop: for(int i = 0; i < N; i++) {
                sumas_inner_loop: for(int j = 0; j < N; j++) {
                        suma_por_filas[i] += IB[i][j];
                        suma_por_columnas[i] += IB[j][i];
                }
        }
        suma_total_loop: for(int i = 0; i < N; i++)
                suma_total_IB += suma_por_filas[i];
        vector_C_loop: for(int i = 0; i < N; i++)
                C[i] = (1/suma_total_IB) * suma_por_filas[i];
        Cspc_outer_loop: for(int i = 0; i < N; i++)
                Cspc_inner_loop: for(int j = 0; j < N; j++)
                        Cspc[i][j] = C[i] * suma_por_columnas[j];
        IB1_outer_loop: for(int i = 0; i < N; i++)
                IB1_inner_loop: for(int j = 0; j < N; j++)
                        IB1[i][j] = IB[i][j] - Cspc[i][j];
}
```

El resto del código principal se traduce directamente desde la versión en Python, usando las funciones que acabamos de presentar. En términos generales, podemos dividirlo conceptualmente en dos partes, una relativa a todos los cálculos previos de matrices y vectores que no varían a lo largo de las iteraciones del algoritmo (preprocesado) y otra con el bucle principal y el control de residuos.

```
static void compute_SUNSAL(float A[K][N], float y[K], float x_sol[N])  {

        // REESCALADO

        reescalado(A,y);
```

```
// INICIALIZACION

float mu = 0.01;
bool mu_changed = false;
float const tol = hls::sqrt(N) * 0.0001;
float res_p = INF;
float res_d = INF;

float At[N][K] = {{}}, IBAt[N][K] = {{}}, AtA[N][N] = {{}};

traspuesta_KporN(A, At);

float At_aux[N*K], A_aux[K*N], AtA_aux[N*N];

producto_NporKporN(At, A, AtA);

float U[N][N], S_0[N][N], Utr[N][N], IB[N][N], IB1[N][N];
float C[N] = {}, Aty[N] = {};

// yy = At * y

producto_vector_NporK(At, y, Aty);

// SVD

float v[N][N] = {{}};
xf::solver::svdTop<N, N, xf::solver::svdTraits<N, N, float, float>,
        float, float>(AtA, S_0, U, v);

traspuesta_NporN(U,Utr);

calcular_IB1(U, Utr, S_0, mu, C, IB, IB1);

// Inicializar x_sol

producto_NporNporK(IB, At, IBAt);
producto_vector_NporK(IBAt, y, x_sol);

// Inicializar u, d

float u[N], d[N] = {}, W[N] = {}, u_0[N] = {};
inicializar_x_loop: for(int i = 0; i < N; i++)
        u[i] = x_sol[i];

// ITERACIONES

int iters = 1;

main_loop: for (int iters = 1; iters < M; iters++) {
        if (!((res_d > tol) || (res_d < 0 - tol) ||
                (res_p > tol) || (res_p < 0 - tol)))
                break;

        if ((iters % 10) == 1) {
           residuo_u_loop: for(int i = 0; i < N; i++)
           u_0[i] = u[i];
        }
```

```
// u = max(x - d, 0)

actualizar_u_loop: for (int i = 0; i < N; i++) {
        float r = x_sol[i] - d[i];
        if (r > 0)
                u[i] = r;
        else
                u[i] = 0;
}

// x

w_loop: for(int i = 0; i < N; i++)
        W[i] = Aty[i] + mu * (u[i] + d[i]);

actualizar_x_outer_loop: for(int i = 0; i < N; i++) {
        x_sol[i] = C[i];
        actualizar_x_inner_loop: for(int j = 0; j < N; j++)
                x_sol[i] += IB1[i][j] * W[j];
}

// d = d - (x - u)

actualizar_d_loop: for (int i = 0; i < N; i++)
        d[i] = d[i] - (x_sol[i] - u[i]);

// control de los residuos (res_p y res_d)

if ((iters % 10) == 1) {

        residuos: {

                // residuo primal (x - z)

                res_p = 0;
                res_p_loop: for(int i = 0; i < N; i++) {
                        res_p += (x_sol[i] - u[i]) * (x_sol[i] - u[i]);
                }
                res_p = hls::sqrt(res_p);

                // residuo dual (u - u_0)

                res_d = 0;
                res_d_loop: for(int i = 0; i < N; i++)
                        res_d += (u[i] - u_0[i]) * (u[i] - u_0[i]);
                res_d = mu * hls::sqrt(res_d);

        }

        // actualizar mu

        if (res_p > 10 * res_d) {
                mu = mu * 2;
                d_res_p_loop: for(int i = 0; i < N; i++)
                        d[i] = d[i] / 2;
                mu_changed = true;
        }
```

```
                        else if (res_d > 10 * res_p) {
                                mu = mu / 2;
                                d_res_d_loop: for(int i = 0; i < N; i++)
                                        d[i] = d[i] * 2;
                                mu_changed = true;
                        }

                        if (mu_changed) {
                                calcular_IB1(U, Utr, S_0, mu, C, IB, IB1);
                                mu_changed = false;
                        }

                }

        }

}
```

Por último, vamos a optimizar el flujo de datos que manejan las funciones anteriores con ayuda de dos elementos que son clave a la hora de diseñar un buen código Kernel en *Xilinx Vitis Unified Software Platform*: la directiva *dataflow* y la estructura de datos *hls::stream*. Si bien es cierto que estos conceptos ya entran dentro de lo que llamaríamos aceleración por hardware, consideramos que son esenciales para cualquier algoritmo que implementemos en *Vitis HLS*, por lo que los introducimos anticipadamente.

La directiva de aceleración *dataflow* permite que diferentes funciones o bucles se ejecuten de manera solapada compartiendo datos según van estando disponibles. De esta forma, un bucle puede ir generando valores de un vector y otra función puede ir consumiéndolos según están disponibles sin tener que esperar a que el bucle anterior acabe de producir todos los elementos del vector. Así, conseguimos un diseño RTL con mayor concurrencia y mejoramos la latencia del diseño completo. La figura 6.1 muestra un esquema del efecto de la directiva *dataflow* sobre el código de la función *top*.

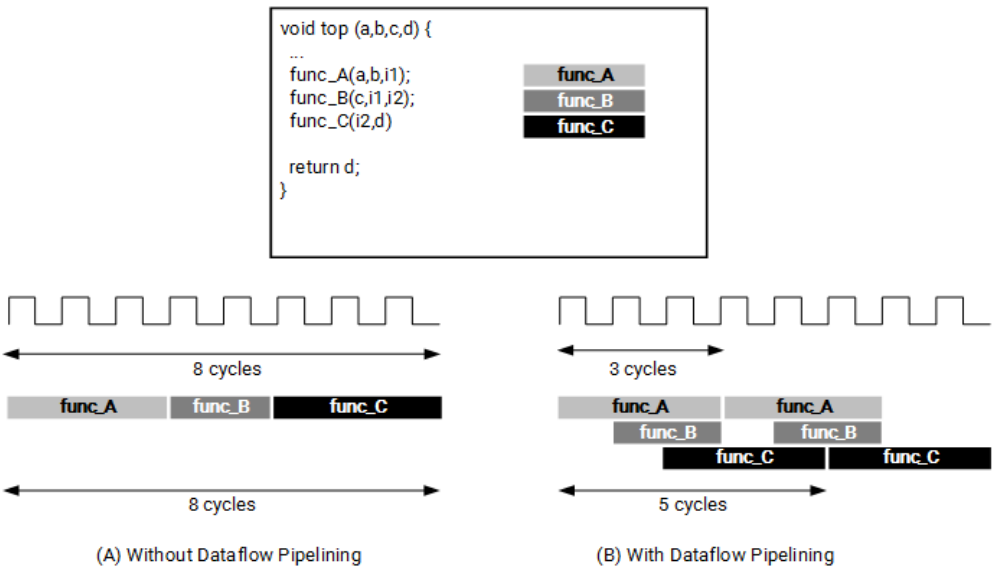


Figura 6.1: Esquema del efecto de la directiva *dataflow*.

En conjunción con la directiva *dataflow*, existe una estructura de datos nativa de *Vitis HLS* llamada *hls::stream* que permite simular una cola FIFO infinita, facilitando así el manejo de datos generados y consumidos en un *dataflow*. Además, ayuda a evitar problemas con manejos de punteros y memoria dinámica que puede ser sensible para el compilador de Vitis (v++) cuando queremos transferir datos de una función a otra. Al usar esta estructura de datos con una directiva *dataflow*, obtenemos el código para la función principal del archivo Kernel que se muestra a continuación.

```
static void load_input_A(float* A, hls::stream<float>& A_stream) {
        load_input_A_loop: for (int i = 0; i < K*N; i++)
                A_stream << A[i];
}

static void load_input_y(float* y, hls::stream<float>& y_stream) {
        load_input_y_loop: for (int i = 0; i < K; i++)
                y_stream << y[i];
}

static void store_result_x(float* x, hls::stream<float>& x_stream) {
        store_result_x_loop: for (int i = 0; i < N + 5*N*N; i++)
                x[i] = x_stream.read();
}

extern "C" {
        void krnl_SUNSAL(float* A, float* y, float* x) {
                #pragma HLS INTERFACE m_axi port = A bundle = gmem0
                #pragma HLS INTERFACE m_axi port = y bundle = gmem1
                #pragma HLS INTERFACE m_axi port = x bundle = gmem2

                static hls::stream<float> A_stream("input_A");
                static hls::stream<float> y_stream("input_y");
                static hls::stream<float> x_stream("output_x");

                #pragma HLS dataflow
                load_input_A(A, A_stream);
                load_input_y(y, y_stream);
                compute_SUNSAL(A_stream, y_stream, x_stream);
                store_result_x(x, x_stream);
        }
}
```

Al emular por hardware en *Xilinx Vitis Unified Software Platform* el archivo Kernel anterior para diversos valores de los tamaños $K, N \in [1, 500]$ de las matrices y vectores involucrados, así como para un rango amplio de número de iteraciones $M \in [10, 20, \ldots, 100]$, obtenemos reportes de síntesis que muestran con exactitud todos los detalles de latencias, paradas de *pipeline*, grado de concurrencia, etcétera. A continuación se detalla toda la información que podemos extraer para el caso $K = N = 500, M = 100$, que utilizaremos como base para acelerar por hardware nuestro algoritmo en la sección 6.3.1. Adicionalmente, para poder simular ejecuciones con datos concretos, utilizamos un código Host con librerías OpenCl que se detalla en la sección 9.2.

Lo primero que nos indica el reporte de síntesis son datos relativos a la propia simulación por hardware realizada y al archivo Kernel. En orden de aparición, podemos saber: la versión del compilador Vitis (v++) utilizada, el tipo de simulación y la fecha en que se realizó, el copyright de *Xilinx*, la fecha de creación del reporte de síntesis, el nombre del Kernel, el dispositivo hardware utilizado (en nuestro caso una Alveo U250), la frecuencia de reloj objetivo y el número de archivos Kernel simulados con librerías OpenCl sobre memoria de la FPGA.

```
================================================================
Version:                        v++ v2022.1 (64-bit)
Build:                          HW Build 3524075 on 2022-04-13-17:42:45
Copyright:                      Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.
Created:                        Sat Mar 4 14:04:59 2023
================================================================
```

```
================================================================
```

Design Name:	krnl_SUNSAL
Target Device:	xilinx:u250:gen3x16_xdma_4_1:202210.1
Target Clock:	300.000000MHz
Total number of kernels:	1 fpga0:OCL_REGION_0

```
================================================================
```

Seguido a lo anterior, se nos indica la frecuencia de reloj estimada para cada módulo de código, y lo que es más importante, las estimaciones de latencia, ciclos de reloj y hardware consumido. A modo de ejemplo, analicemos los datos asociados al módulo con nombre *compute_SUNSAL_Pipeline_inicializar_x_loop*, que se encargaba de rellenar un vector auxiliar con sus valores iniciales para las iteraciones del algoritmo.

```
inicializar_x_loop: for(int i = 0; i < N; i++) {
        u[i] = x_sol[i];
}
```

```
================================================================
```

Kernel Name:	krnl_SUNSAL
Module Name:	compute_SUNSAL_Pipeline_inicializar_x_loop
Target Frequency:	300.3 MHz
Estimated Frequency:	418 MHz
Best (cycles):	502
Avg (cycles):	502
Worst (cycles):	502
Best (absolute):	1.673 ns
Avg (absolute):	1.673 ns
Worst (absolute):	1.673 ns
FF:	21
LUT:	71
BRAM:	0

```
================================================================
```

Como podemos observar, se consigue una frecuencia de reloj superior al objetivo de 300.3 MHz, requiere de exactamente 502 ciclos para las 500 iteraciones que realiza, y se estima que tarda alrededor de 1.673 ns considerando la frecuencia objetivo. En cuanto al hardware que consume, en este caso solo requiere de 21 *flip-flops* y 71 LUTs, sin llegar a necesitar memoria RAM u otro tipo de memoria auxiliar.

Sin embargo, hay casos en los que el reporte de síntesis no es tan preciso y el compilador no puede inferir ciertas estimaciones por la naturaleza del módulo que se estudia. Esto es justamente lo que ocurre con la región de código de la librería *Solver* encargada de la descomposición en valores singulares. Como se muestra a continuación, en este caso la frecuencia estimada es menor, no tenemos estimaciones de ciclos o tiempo medio de ejecución, y los casos peor y mejor varían significativamente (desde 0.135 segundos hasta casi 70 segundos). Claramente, esto se debe a la alta variabilidad que existe a la hora de calcular la descomposición en valores singulares, que puede tardar mucho o poco dependiendo de las propiedades de la matriz que se considere. Asimismo, podemos comprobar cómo, a diferencia del bucle anterior, no solo utiliza *flip-flops*, sino que también requiere de 48 bloques de memoria BRAM.

```
================================================================
```

Kernel Name:	krnl_SUNSAL
Module Name:	svdBasic_500_500_svdTraits_500_500_float_float
Target Frequency:	300.3 MHz
Estimated Frequency:	377 MHz
Best (cycles):	40414011

Avg (cycles):	undef
Worst (cycles):	20398191861
Best (absolute):	0.135 s
Avg (absolute):	undef
Worst (absolute):	67.987 sec
FF:	9290
LUT:	10072
BRAM:	48

==========

En cuanto al tiempo total de ejecución del código Kernel, si no utilizamos la directiva de aceleración *dataflow* y el código se ejecuta todo de manera secuencial, obtenemos aproximadamente la suma de las latencias de cada bloque de código que compone el algoritmo. En cambio, si incluimos esta directiva, la latencia total del Kernel resulta ser, tras todo el procesado interno del compilador Vitis (v++), mucho más baja y en el caso peor es de menos de 5 segundos, como nos indican los respectivos reportes de síntesis. Además, el aumento de consumo de hardware es casi inexistente. Aquí es donde se ve lo esencial que resulta utilizar bien el *dataflow*. Para consultar en detalle todo el resto de datos para los diferentes módulos del Kernel implementado, ver [26] o el apéndice 9.4.

==========

Kernel Name:	krnl_SUNSAL
Module Name:	krnl_SUNSAL
Dataflow:	No
Target Frequency:	300.3 MHz
Estimated Frequency:	377 MHz
Best (cycles):	4226183013
Avg (cycles):	undef
Worst (cycles):	224835678365
Best (absolute):	14.086 s
Avg (absolute):	undef
Worst (absolute):	749 sec
FF:	28281
LUT:	38957
BRAM:	266

==========

==========

Kernel Name:	krnl_SUNSAL
Module Name:	krnl_SUNSAL
Dataflow:	Yes
Target Frequency:	300.3 MHz
Estimated Frequency:	376.9 MHz
Best (cycles):	1250709
Avg (cycles):	undef
Worst (cycles):	1495877819
Best (absolute):	4.169 ms
Avg (absolute):	undef
Worst (absolute):	4.986 sec
FF:	28326
LUT:	39212
BRAM:	266

==========

6.3. Aceleración sobre hardware con HLS

Una vez implementado el algoritmo SUnSAL para el problema FCLS sobre *Xilinx Vitis Unified Software Platform* en lenguaje C++ compatible con el compilador v++, pasamos a acelerar por hardware el código con directivas de aceleración o *pragmas* de *Vitis* que ya se introdujeron en la sección 4.2.1. El objetivo principal será mejorar los resultados de HLS obtenidos en el reporte de síntesis que se comenta al final de la sección 6.2. Para ello, analizamos primero las regiones de código con mayor latencia y los puntos donde ocurren paradas de *pipeline* indeseadas. En base a esta información, diseñamos estrategias de paralelización o reestructuraciones de código que incluyan directivas de aceleración.

6.3.1. Análisis del reporte de síntesis inicial. Detección de latencias, paradas de *pipeline* y consumo de recursos hardware

Al analizar en profundidad el reporte de síntesis obtenido al final de la sección 6.2 con la directiva de aceleración *dataflow* incluida, podemos determinar qué módulos tienen más latencia y cuales pueden ser acelerados por tener muchas paradas de *pipeline* o tener potencial de paralelización. En concreto, tenemos tres puntos en los que acelerar por hardware con directivas de aceleración puede suponer una mejora sustancial por motivos que se desarrollan en los párrafos sucesivos: los productos de matrices, y en concreto el producto de matrices $N \times N$ que se realiza en todas las iteraciones, la sucesión de operaciones secuenciales de la función *calcularIB1*, y por último los bucles que leen y escriben sobre la misma variable en cada iteración, como por ejemplo *actualizar_x_inner_loop* o *producto_vector_NporK_inner_loop*.

En primer lugar, la multiplicación de matrices es una operación fundamental en los algoritmos numéricos que puede llevar mucho tiempo, sobre todo si las matrices son de tamaño muy grande. Por lo tanto, es una parte importante de muchos problemas de cálculo numérico tener un producto de matrices con buen rendimiento, baja latencia, bajo gasto de memoria y un consumo de hardware no muy elevado. El algoritmo SUnSAL en su versión para el problema FCLS no es una excepción, e incluye una serie de productos de matrices que lastran considerablemente su rendimiento.

Si analizamos los reportes de síntesis y excluimos la descomposición en valores singulares, el producto de matrices es el módulo que tiene más latencia (3.333 s) y debido a que en cada iteración se realiza al menos un producto de matrices $N \times N$, acelerar estos módulos es clave para mejorar significativamente el rendimiento del algoritmo. De esta forma, identificamos como muy prioritario la aceleración de los productos de matrices. Sin embargo, y como veremos en la sección correspondiente, mejorar la latencia de estos módulos supondrá un aumento considerable en consumo de hardware, lo que nos obligará a seleccionar con cuidado los módulos concretos a acelerar en base a su efecto sobre la latencia total.

==========

Kernel Name:	krnl_SUNSAL
Module Name:	calcular_IB1_Pipeline_producto_NporNporN
Target Frequency:	300.3 MHz
Estimated Frequency:	426.98 MHz
Best / Worst (cycles):	1000000010
Best / Worst (absolute):	3.333 s
FF:	418
LUT:	528
BRAM:	0

==========

En segundo lugar, un bloque de código que también se ejecuta en varias iteraciones es la función *calcularIB1*. Está compuesto por una serie de operaciones secuenciales que, a pesar de que individualmente no tienen demasiada latencia, al traducirse a diseño RTL generan una larga cadena de dependencias que resultan en un cuello de botella significativo. Para solucionar esto veremos una directiva de aceleración muy útil con nombre *expression_balance*, que se encargará de reorganizar la secuencia de operaciones en forma de un árbol balanceado generado a partir del análisis de las dependencias del código, para reducir la latencia y aumentar la concurrencia a costa de un consumo mayor de hardware.

Finalmente, hay varios ejemplos de módulos en los que una misma variable está involucrada en operaciones de lectura y escritura, como por ejemplo el bucle *producto_vector_NporK_inner_loop* en el que se van sumando cantidades sobre la misma variable en cada iteración, lo que resulta en muchas paradas de *pipeline* que aumentan el número de ciclos necesarios para hacer los cómputos. Veremos cómo hacer frente a estas situaciones, y nos centraremos en un caso concreto que involucra a los bucles *res_p_loop* y *res_d_loop* en donde mostraremos cómo, analizando bien la situación, podemos resolver varios problemas de latencia a la vez con una sola directiva de aceleración.

6.3.2. Proceso de aceleración sobre hardware del algoritmo SUnSAL

Producto de matrices

Partimos del siguiente bloque de código que vamos a acelerar de manera razonada. Implementa la que quizás es la manera más común de realizar un producto de matrices. Tenemos dos bucles externos que iteran sobre las filas y columnas de la matriz resultante $R \in \mathbb{R}^{n\times m}$, y un bucle interno que computa el producto escalar de cada fila de $A \in \mathbb{R}^{n\times k}$ con la correspondiente columna de $B \in \mathbb{R}^{k\times m}$.

```
static void producto_NporKporM(float A[N][K], float B[K][M], float R[N][M]) {
        producto_NporKporM_outer_loop: for (int i = 0; i < N; i++) {
                producto_NporKporM_inner_loop: for (int j = 0; j < M; j++) {
                        float Rij = 0;
                        producto_NporKporM_innermost_loop: for (int k = 0; k < N; k++)
                                Rij += A[i][k] * B[k][j];
                        R[i][j] = Rij;
                }
        }
}
```

Debido a que el producto escalar de dos vectores conlleva acumular los productos de elementos que se van calculando, tenemos una variable *Rij* que va guardando el valor de la suma total, lo que produce muchas paradas de *pipeline* debido a su lectura y escritura en cada iteración del bucle interno. Para evitar esto y mejorar la latencia, introducimos la directiva de aceleración *pipeline*.

$$pragma\ HLS\ pipeline\ II = K,\ \ K \in \mathbb{N}$$

Esta directiva *pipeline* permite forzar que el intervalo de inicialización de una instrucción sea de una determinada duración en ciclos, aumentando la concurrencia para conseguirlo. Por defecto, el compilador de Vitis (v++) intenta priorizar una frecuencia de reloj que sea superior a la especificada, pero la directiva *pipeline* tiene una opción (II) que permite forzar el número de ciclos de iniciación para cada instrucción, aumentando la concurrencia a base de mayor consumo de hardware. La figura 6.2 muestra el efecto de esta directiva con II = 1.

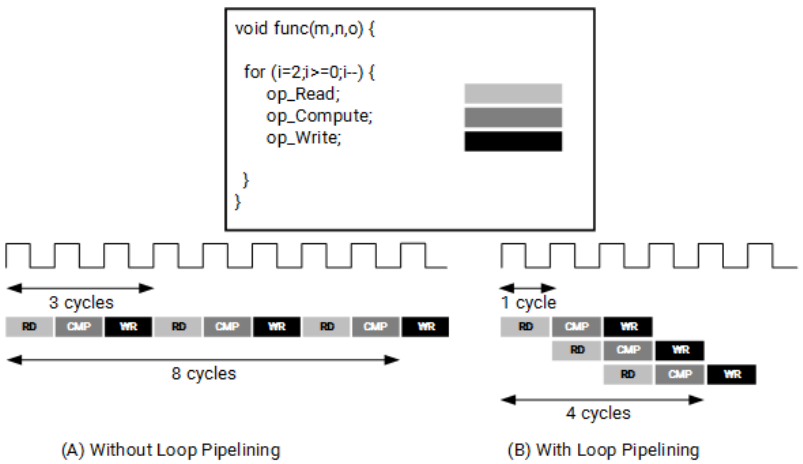


Figura 6.2: Diagrama del efecto de la directiva *pipeline* con II = 1.

Al incluir la directiva *pipeline* para el bucle más interno, el efecto que conseguimos es desenrrollar entero este bucle resultando en un circuito con N módulos de multiplicación-suma que se ejecutan concurrentemente. El aumento del consumo de hardware es considerable, pero la mejora de latencia también. Sin embargo, esta mejora de la concurrencia requiere también poder leer y escribir varios operandos de manera simultánea, para lo que vamos a introducir unas directivas adicionales.

La directiva de aceleración *array_partition* permite separar el espacio de direcciones asociado a una matriz en diversos bloques que permiten lectura simultánea en cada uno de ellos. Así, conseguimos que en un mismo ciclo de reloj tengamos leído un conjunto de datos como, por ejemplo, una fila o columna entera de la matriz, que es justo lo que necesitamos.

Sin embargo, existe otra directiva más con el nombre de *array_reshape* que, además de separar en diversos espacios de memoria la matriz, los recombina en un solo bloque de direcciones con un ancho de direcciones mayor, sin modificar la cantidad de bits utilizados. Su ventaja respecto a *array_partition* es que muchas veces es más sencillo asignar el bloque grande a recursos de memoria de una FPGA que diversos bloques independientes más pequeños, además de que reduce la latencia por las interconexiones entre las memorias. La figura 6.3 muestra el efecto en memoria de estas directivas sobre una matriz.

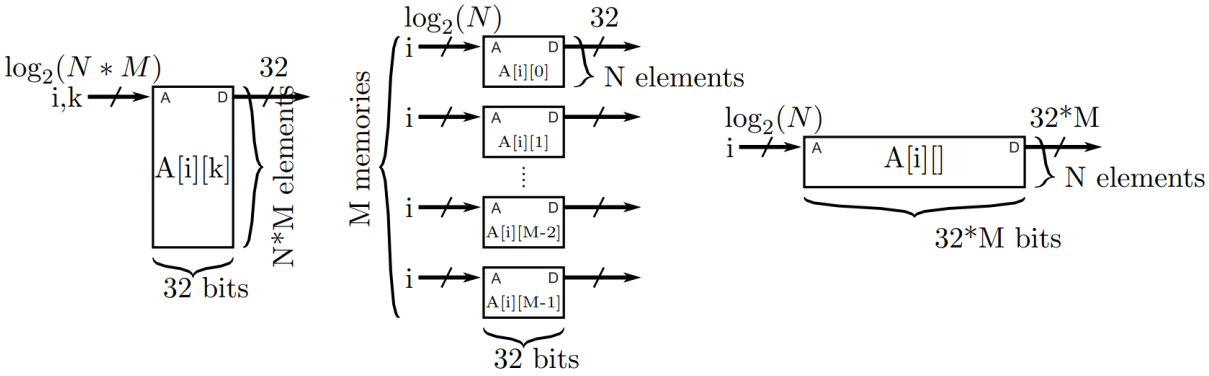


Figura 6.3: Diagrama del efecto de las directivas *array_partition* (centro) y *array_reshape* (derecha).

Por supuesto, la optimalidad de las directivas *array_partition* y *array_reshape* depende en gran medida del tamaño concreto de los vectores o matrices que estemos utilizando. Por ejemplo, el tamaño mínimo de los bloques RAM (BRAM) en las FPGA *Xilinx Virtex UltraScale+* suele ser de 16 Kbits en diversas configuraciones de profundidad y anchura. Si tenemos una matriz de tamaño 1024×4, cabría en una memoria BRAM de tamaño 4 Kbits $\times$ 4, pero al aplicar la directiva *array_partition* a la segunda dimensión de la matriz necesitaríamos cuatro memorias de 1 Kbits $\times$ 4, que haría que consumamos mucha más memoria y aumentara la latencia. En cambio, al aplicar la directiva *array_reshape* podríamos utilizar una única memoria BRAM de 1 Kbits $\times$ 16. En caso de estar utilizando una librería espectral real, el control que tenemos sobre su tamaño es reducido, pero aun así es bueno tener en cuenta este tipo de cuestiones para optimizar nuestro diseño del algoritmo.

```
static void producto_NporKporM(float A[N][K], float B[K][M], float R[N][M]) {
        #pragma HLS array_reshape variable=A complete dim=2
        #pragma HLS array_reshape variable=B complete dim=1
        producto_NporKporM_outer_loop: for (int i = 0; i < N; i++) {
                producto_NporKporM_inner_loop: for (int j = 0; j < M; j++) {
                        #pragma HLS pipeline II=1
                        float Rij = 0;
                        producto_NporKporM_innermost_loop: for (int k = 0; k < N; k++)
                                Rij += A[i][k] * B[k][j];
                        R[i][j] = Rij;
                }
        }
}
```

La mejora en la latencia del producto de matrices al introducir conjuntamente las directivas de aceleración *pipeline* y *array_reshape* es muy notable. En comparación con los resultados mostrados en el reporte de síntesis de la sección 6.3.1, los cuales indicaban una latencia de alrededor de 3.333 segundos, ahora conseguimos un tiempo de apenas 0.845 microsegundos para los módulos encargados de estos productos de matrices. Por supuesto, esto viene de la mano de un considerable aumento de hardware, pasando de usar 512 LUTs y 418 *flip-flops* a usar 36544 LUTs y 103134 *flip-flops*. No llegamos a necesitar BRAM por estar utilizando matrices de tamaño 500×500 (más que suficiente para los tamaños estándar de las librerías espectrales a utilizar) pero si cambiáramos a tamaño de 1024×1024 ya se empezarían a requerir módulos BRAM para realizar los manejos de memoria.

Kernel Name:	krnl_SUNSAL
Module Name:	calcular_IB1_Pipeline_producto_NporNporN
Target Frequency:	300.3 MHz
Estimated Frequency:	426.98 MHz
Best / Worst (cycles):	253507
Best / Worst (absolute):	0.845 ms
FF:	103134
LUT:	36544
BRAM:	0

Reducción de riesgos LDE

Las paradas de *pipeline* causadas por lecturas y escrituras de una misma variable dentro de un bucle son muy comunes, y hay multitud de formas de lidiar con ellas más allá de usar la directiva de aceleración *pipeline*. Un ejemplo de ello es el que vemos a continuación. Tenemos dos bucles con nombres *res_p_loop* y *res_p_loop* que calculan los residuos a lo largo de las iteraciones del algoritmo SUnSAL, lo que requiere, al igual que en el bucle interno del producto de matrices anterior, sumar a una misma variable una serie de valores de los vectores involucrados. Sin embargo, en este caso los dos bucles se realizan uno detrás del otro y tienen el mismo número de iteraciones, lo que nos va a permitir resolver dos problemas de latencia simultáneamente con ayuda de la directiva *loop_merge*.

```
residuos: {
        #pragma HLS loop_merge

        res_p = 0; res_d = 0;

        res_p_loop: for(int i = 0; i < N; i++)
                res_p += (x_sol[i] - u[i]) * (x_sol[i] - u[i]);

        res_d_loop: for(int i = 0; i < N; i++)
                res_d += (u[i] - u_0[i]) * (u[i] - u_0[i]);

        res_d = mu * hls::sqrt(res_d); res_p = hls::sqrt(res_p);

}
```

Como su propio nombre indica, la directiva de aceleración *loop_merge* fusiona bucles consecutivos en un único bucle para reducir la latencia total, aumentar la compartición de recursos y mejorar la lógica de las iteraciones. Esto lo consigue a través de una reducción en el número de ciclos de reloj necesarios para pasar de un bucle al siguiente y, si es posible, paralelizando el bucle. Al aplicar esta directiva a los bucles *res_p_loop* y *res_d_loop* anteriores, no solo mejoramos la latencia por las optimizaciones de la fusión de los bucles, sino que también impedimos que haya tantas paradas de *pipeline* al paralelizar las lecturas y escrituras de las variables *res_p* y *res_d* en el bucle resultante. Es un ejemplo claro de cómo muchas veces compensa pensar bien el diseño de la aceleración, y no solo limitarse a incluir la directiva *pipeline* a los bucles por separado.

En este caso, la mejora de latencia y número de ciclos estimada por el reporte de síntesis al incluir la directiva *loop_merge* es menor que en el caso de la aceleración del producto de matrices, pero aun así sigue siendo significativa. Pasamos de requerir 11.715 microsegundos para cada uno de los dos bucles, a tan solo tener una latencia de 11.719 microsegundos para el bucle unificado resultante (notemos que se utiliza exactamente un ciclo más en el bucle resultante que en los bucles originales por separado). En total, es como reducir practicamente a la mitad la latencia total, y de paso conseguimos una reducción del consumo de hardware, como bien muestran los reportes de síntesis.

```
================================================
Kernel Name:                krnl_SUNSAL
Module Name:                compute_SUNSAL_Pipeline_res_p_loop
Loop Merge:                 No
Target Frequency:           300.3 MHz
Estimated Frequency:        426.98 MHz
Best / Worst (cycles):      3515
Best / Worst (absolute):    11.715 us
FF:                         183
LUT:                        183
BRAM:                       0
================================================

================================================
Kernel Name:                krnl_SUNSAL
Module Name:                compute_SUNSAL_Pipeline_res_d_loop
Loop Merge:                 No
Target Frequency:           300.3 MHz
Estimated Frequency:        426.98 MHz
Best / Worst (cycles):      3515
Best / Worst (absolute):    11.715 us
FF:                         183
LUT:                        183
BRAM:                       0
================================================

================================================
Kernel Name:                krnl_SUNSAL
Module Name:                compute_SUNSAL_Pipeline_residuos
Loop Merge:                 Sí
Target Frequency:           300.3 MHz
Estimated Frequency:        426.98 MHz
Best / Worst (cycles):      3516
Best / Worst (absolute):    11.719 us
FF:                         313
LUT:                        280
BRAM:                       0
================================================
```

Otra directiva de aceleración que va más allá de simplemente evitar paradas de *pipeline* causadas por lecturas y escrituras consecutivas es la directiva *expression_balance*. Cuando tenemos una serie de sentencias secuenciales en el código de C++, se pueden generar cadenas de operaciones muy largas en el diseño RTL compilado que, junto a un periodo de reloj pequeño, puede dar lugar a latencias altas. La directiva *expression_balance* utiliza las propiedades asociativas y conmutativas que permite el código para reorganizarlo en forma de árbol balanceado, reduciendo la longitud de la cadena de operaciones en el diseño RTL y por tanto la latencia, a coste de un mayor consumo de hardware.

```
static void calcular_IB1(float U[N][N], float Ut[N][N], float S_0[N][N],
                         float mu, float C[N], float IB[N][N], float IB1[N][N]) {
        #pragma HLS expression_balance

        ...
```

Los resultados de aplicar conjuntamente la directiva *expression_balance* y la aceleración del producto de matrices detallada en la sección anterior sobre la función *calcular_IB1* son bastante buenos, con una reducción de la latencia desde los 6.674 segundos hasta los 10.053 microsegundos (recordemos que *calcular_IB1* se utiliza en múltiples iteraciones, lo que hace que se reduzca considerablemente la latencia global). El aumento de consumo de hardware también es notable tanto en LUTs como en memoria, como muestran los siguientes reportes de síntesis.

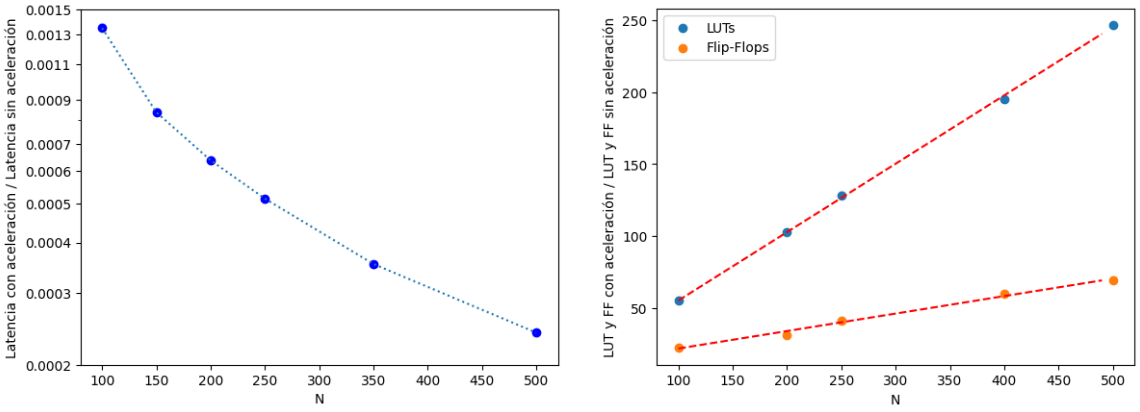
Kernel Name:	krnl_SUNSAL
Module Name:	calcular_IB1
Expression Balance:	No
Target Frequency:	300.3 MHz
Estimated Frequency:	426.98 MHz
Best / Worst (cycles):	2002509101
Best / Worst (absolute):	6.674 s
FF:	2862
LUT:	3705
BRAM:	50

Kernel Name:	krnl_SUNSAL
Module Name:	calcular_IB1
Expression Balance:	Sí
Target Frequency:	300.3 MHz
Estimated Frequency:	426.98 MHz
Best / Worst (cycles):	3016095
Best / Worst (absolute):	10.053 ms
FF:	438434
LUT:	259984
BRAM:	34

6.3.3. Análisis final de los resultados de aceleración

Hasta ahora hemos estudiado las mejoras de ciclos y latencia, así como el aumento del consumo de hardware para un tamaño fijo de matrices con $K = N = 500$ y un número de iteraciones $M = 100$. Para validar correctamente la aceleración por hardware realizada hasta ahora con directivas de aceleración, comparemos los resultados para diversas variaciones de los anteriores parámetros en los rangos $K, N \in [100, 1000]$ y $M \in [10, 1000]$, representando varias medidas de mejora y viendo como evolucionan según modificamos los tamaños de las matrices y el número de iteraciones del algoritmo SUnSAL.

Empezamos analizando la mejora de latencia y el aumento del consumo de hardware tras la aceleración por hardware del producto de matrices realizada en la sección 6.3.2. Más allá de que reducimos la latencia a un valor que es alrededor de 0.0004 veces la latencia original cuando trabajamos con matrices de tamaño $N \times N$ con $N \in [100, 500]$, esta reducción es más significativa cuanto más grandes son las matrices, como se muestra en la figura 6.5. En cuanto al hardware, el aumento es lineal en N debido al desenrollado del bucle interno del producto de matrices, que hace que se requieran elementos hardware proporcionales al número de operandos.

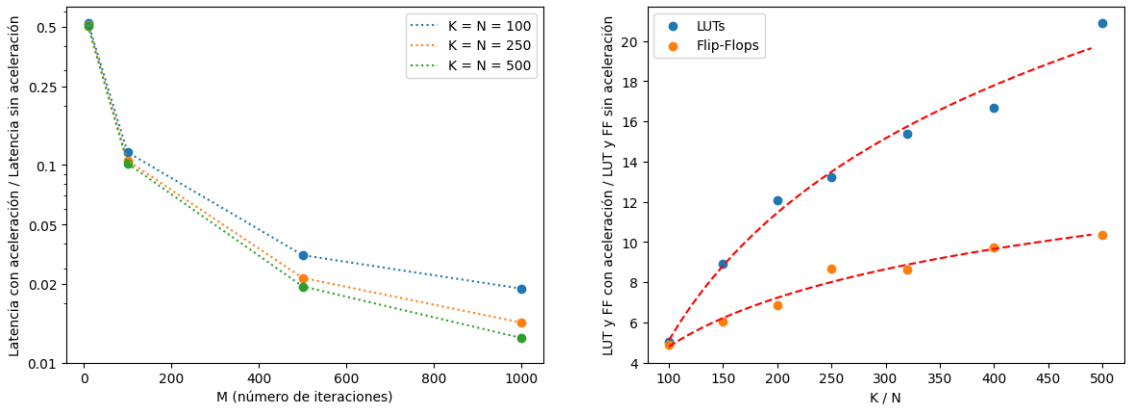


(a) Mejora de la latencia.

(b) Aumento lineal del consumo de hardware.

Figura 6.4: Mejora en la latencia del producto de matrices y aumento del consumo de hardware para $N \in [100, 500]$ tras la acelereación por hardware de la sección 6.3.2.

Para el caso del algoritmo SUnSAL completo tras toda la aceleración por hardware realizada, obtenemos un aumento del consumo de hardware total que supone alrededor de 5 veces el consumo original para tamaños de matrices de 100×100 elementos, pero que evoluciona en una escala logarítmica más allá de 20 veces el hardware consumido original para tamaños de 500×500 en adelante. En cuanto a la mejora de latencia, esta es muy significativa, reduciéndose a la mitad para un número de iteraciones bajo $M \in [10, 50]$, pero reduciéndose hasta 0.01 veces la latencia original para un número de iteraciones mayor $M \in [500, 1000]$. Por ello, tanto la reducción de la latencia al considerar un alto número de iteraciones cómo el aumento de consumo de hardware guiado por una escala logarítmica nos permiten asegurar que la aceleración por hardware realizada sobre el algoritmo SUnSAL resiste bien ante tamaños de matrices cada vez mayores y tiene un buen comportamiento cuando dejamos que itere con valores altos de M.



(a) Mejora de la latencia variando M.

(b) Aumento del consumo de hardware variando K, N.

Figura 6.5: Mejora en la latencia del algoritmo SUnSAL completo para $M \in [10, 1000]$ y aumento del consumo de hardware para $K = N \in [100, 500]$ tras la acelereación por hardware de la sección 6.3.2.

Para ver los reportes de síntesis completos que se han utilizado a la hora de obtener los datos anteriores, consultar [26]. Asimismo, en el apéndice 9.4 se encuentran los reportes para el caso $K = N = 500$ y $M = 100$ que se utilizó como base para la aceleración por hardware de la sección 6.3.2. Por último, la emulación por hardware del algoritmo puede replicarse siguiendo los pasos detallados en las secciones 4.2.1, 9.2 y 9.4, que tratan sobre el manejo de *Xilinx Vitis Unified Software Platform* y los códigos de los archivos Kernel y Host utilizados. La ejecución sobre hardware físico del algoritmo utilizando imágenes y librerías hiperespectrales reales la pasamos a tratar en el siguiente y último capítulo.

Capítulo 7

Resultados Experimentales

Una vez terminada la implementación y la aceleración por hardware del algoritmo SUnSAL para el problema FCLS introducido en la sección 5.3.2 sobre *Xilinx Vitis Unified Software Platform* utilizando HLS (ver los capítulos 4 y 6), pasamos a ejecutarlo sobre hardware físico con datos obtenidos de una imagen hiperespectral real tomada por el sensor AVIRIS (*Airborne Visible / Infrared Imaging Spectrometer*) [40] y seleccionando las firmas espectrales de unos *endmembers* para construir la librería espectral (ver detalles del proceso completo de desmezclado espectral en la sección 5.1). En concreto, vamos a utilizar la ya introducida en la sección 4.3 tarjeta de aceleración *Xilinx* Alveo U250 y analizaremos los resultados de ejecución obtenidos comparándolos con las estimaciones que obtuvimos en el capítulo 6 tras la simulación hardware realizada.

7.1. Obtención y análisis de imágenes hiperespectrales tomadas por el sensor AVIRIS

El JPL (*Jet Propulsion Laboratory*) [30] de la NASA (*National Aeronautics and Space Administration*) [19] tiene abierto al público un portal [32] para acceder de manera libre a una base de datos de imágenes hiperespectrales tomadas por el sensor AVIRIS desde el año 2006 hasta la actualidad. Todas ellas poseen 224 bandas espectrales en el rango de 400 a 2500 nm de longitud de onda, y están muy bien organizadas con datos acerca de su resolución, tamaño y localización. A continuación, vamos a ver cómo analizar estas imágenes hiperespectrales y cómo obtener la información de los píxeles para su estudio.

En primer lugar, seleccionamos una imagen hiperespectral desde el buscador del portal de AVIRIS [32] (ver figura 7.1) y descargamos el directorio comprimido asociado. Este directorio contiene toda la información acerca de la imagen en varios archivos de distinto formato, así como instrucciones para su manejo.

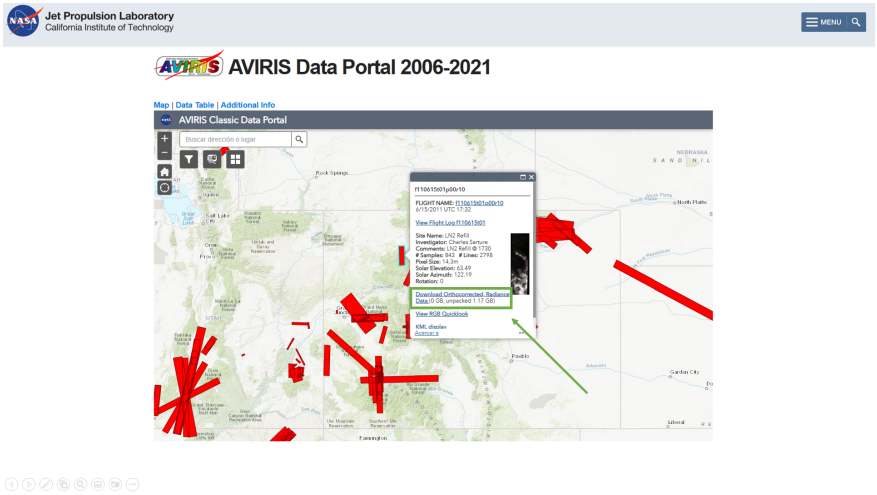
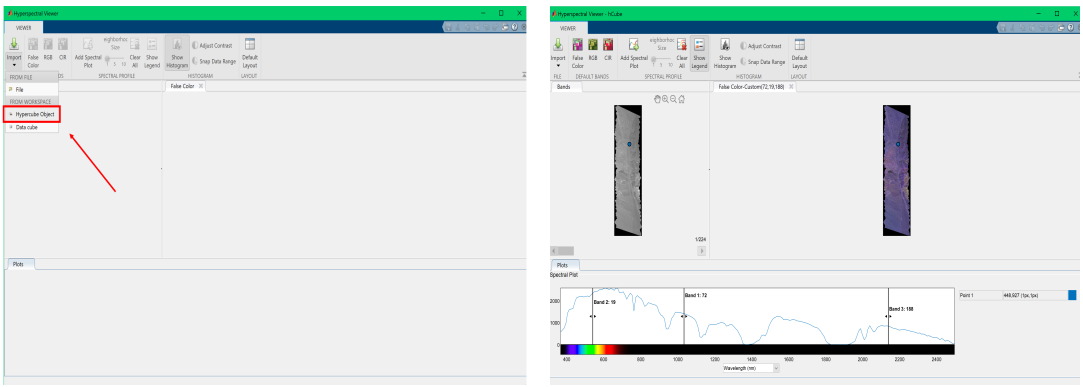


Figura 7.1: Portal de la base de datos de imágenes hiperespectrales del sensor AVIRIS.

Para visualizar la imagen podemos utilizar la biblioteca de Matlab [41] cuyo nombre es *Hyperspectral Imaging Library de Image Processing Toolbox* [42] junto con el módulo *Hyperspectral Viewer* [43]. De esta forma, basta con utilizar el *script* que se muestra a continuación para crear un hipercubo con la información de la imagen y posteriormente visualizarlo directamente importándolo desde *Hyperspectral Viewer* como se muestra en la figura 7.2.

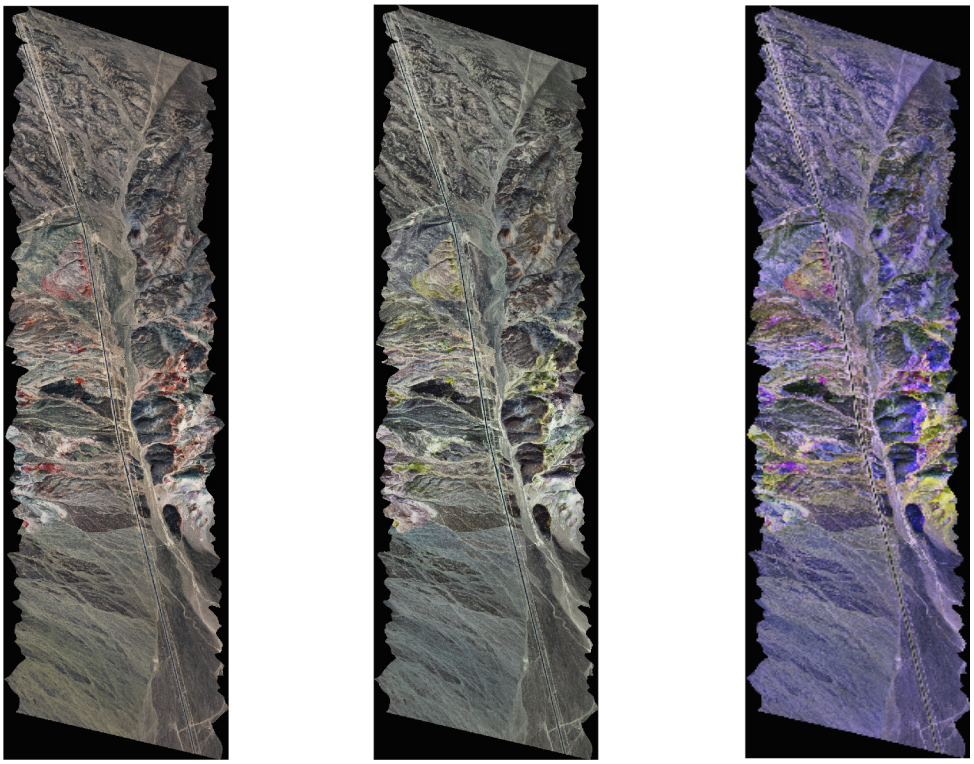
```
% Leer la imagen hiperespectral de AVIRIS (directorio/<nombre_archivo>.hdr)
hCube = hypercube('f080920t01p00r05/f080920t01p00r05rdn_c_sc01_ort_img.hdr');
% Obtener la imagen RGB, CIR, y de Falso Color
rgbImg = colorize(hCube, 'method', 'rgb', 'ContrastStretching', true);
cirImg = colorize(hCube, 'method', 'cir', 'ContrastStretching', true);
fcImg = colorize(hCube, 'method', 'falsecolored', 'ContrastStretching', true);
% Visualizar resultados
figure
tiledlayout(1, 3)
nexttile
imagesc(rgbImg)
axis image off
title('RGB image')
nexttile
imagesc(cirImg)
axis image off
title('CIR image')
nexttile
imagesc(fcImg)
axis image off
title('False-colored image')
```



(a) (b)

Figura 7.2: Visualizar una imagen hiperespectral usando el módulo *Hyperspectral Viewer* de Matlab.

Para la ejecución sobre hardware del algoritmo SUnSAL vamos a utilizar la imagen hiperespectral *f080920t01p00r05* [31], que se muestra en la figura 7.3. Esta imagen captura una zona de la cadena montañosa estadounidense *Cuprite* en el Condado de Esmeralda del estado de Nevada. Según el análisis realizado en [37] acerca de los minerales presentes en este área, podemos encontrar grandes cantidades de alunita, kaolinita, muscovita, calcita y diversos tipos de silicatos, entre otros. Esta información que se ha obtenido con una investigación previa al procesamiento de la imagen hiperespectral es clave para que podamos seleccionar más adelante los *endmembers* que incluiremos en la librería espectral, puesto que, como ya se explicó en el capítulo 5.5, el algoritmo SUnSAL supone construida esta librería espectral para poder operar. Todos los minerales que no incluyamos en la librería no se tendrán en cuenta durante el desmezclado espectral, por lo que buscaremos incluir todos aquellos materiales que queramos determinar si están o no presentes en el área de detección del píxel de la imagen que estemos considerando. Para la prueba de ejecución que vamos a realizar, seleccionamos concretamente el píxel de la posición $(448, 927)$.



(a) RGB (b) CIR (c) Falso Color

Figura 7.3: Visualización de la imagen hiperespectral *f080920t01p00r05* del sensor AVIRIS.

7.2. Librería espectral USGS. Selección y preprocesamiento de los *endmembers* a utilizar

Una vez tenemos seleccionados y analizados la imagen hiperespectral y el píxel concreto que utilizaremos para la ejecución del algoritmo SUnSAL, tenemos que construir la librería espectral seleccionando los *endmembers* adecuados. Recordemos que la librería tendrá forma de matriz $A \in \mathbb{R}^{k \times n}$ donde k será el número de bandas espectrales (que para la imagen hiperespectral seleccionada en la sección anterior toma el valor $k = 224$) y n el número de *endmembers*.

Para ello, nos serviremos de la librería espectral de uso abierto USGS *Spectral Library Version 7* [61] de la agencia gubernamental estadounidense USGS (*United States Geological Survey*) [60]. Contiene una extensa colección de firmas espectrales discretizadas de una gran cantidad de materiales que pueden filtrarse para seleccionar fácilmente aquellos que necesitemos. En base a [37] y a los comentarios de la sección anterior, seleccionamos las firmas espectrales de los 22 siguientes minerales:

1. Albita
2. Alunita
3. Andesita
4. Buddingtonita
5. Calcita
6. Dickita
7. Diópsido
8. Hectorita
9. Hornblenda
10. Illita
11. Jarosita
12. Kaolinita
13. Kieserita
14. Labradorita
15. Malachita
16. Microclina
17. Monacita
18. Muscovita
19. Pirofilita
20. Pirrotina
21. Richterita
22. Serpentina

Por supuesto, los valores de longitud de onda en los que se han hecho las mediciones de la reflectancia para cada uno de *endmembers* anteriores son diferentes, por lo que los datos obtenidos de la librería espectral USGS deben ser analizados y procesados previa la ejecución del algoritmo. Para afrontar esto, vamos a considerar el vector $u \in \mathbb{R}^{224}$ que contiene las longitudes de onda de las 224 bandas espectrales de la imagen hiperespectral *f080920t01p00r05* del sensor AVIRIS y lo vamos a comparar con el vector $v \in \mathbb{R}^s$ que indica las $s \in \mathbb{N}$ longitudes de onda en las que se ha medido la reflectancia de un *endmember* concreto de la librería espectral USGS. Asociado al vector v, tendremos también otro vector $w \in \mathbb{R}^s$ con los valores de la reflectancia del *endmember* para cada uno de los valores de longitud de onda indicados en v.

El objetivo principal será extraer del vector w un vector $z \in \mathbb{R}^{224}$ con los valores de reflectancia correspondientes a los valores de longitud de onda del vector v más próximos a cada uno de los 224 valores de longitud de onda del vector u. A priori, suponemos que la longitud de v es mayor que la de u, que el rango de longitudes de onda de v es más amplio que el de u, y que la partición del rango espectral de v es suficientemente fina. En caso contrario, no nos queda otra que descartar el *endmember* considerado o buscar otra fuente de datos.

Para ello, consideramos el siguiente *script* de Python, que compara los valores de la longitud de onda de los vectores u y v, asumiendo que están ordenados de menor a mayor. A medida que examina los vectores, se va quedando con aquellos valores que están más cerca entre sí hasta completar las 224 bandas espectrales. De esta forma, genera un vector z extrayendo de w los valores correspondientes a las 224 longitudes de onda elegidas en v. Antes de aplicarlo, es imprescindible asegurarse que la longitud de onda se mide en la misma escala en u y en v, pues es común que a veces se utilicen los micrómetros (um) en vez de los nanómetros (nm) a la hora de medir longitudes de onda.

```
z = []
j = 0
for i in range(0,224):
    found = False
    while not found:
        if (abs(v[j] - u[i])  < abs(v[j + 1] - u[i])):
            found = True
            z.append(w[j])
        j = j + 1
```

Aplicando este *script* a cada uno de los 22 *endmembers* seleccionados, conseguimos unificar los valores de longitud de onda de las mediciones de reflectancia de todos los *endmembers* y el píxel extraído de la imagen hiperespectral. Finalmente, basta con formar la matriz $A \in \mathbb{R}^{224\times 22}$ que tiene por columnas a cada uno de los vectores $z \in \mathbb{R}^{244}$ extraídos de los 22 *endmembers*.

7.3. Ejecución sobre hardware del algoritmo SUnSAL con la tarjeta de aceleración *Xilinx* Alveo U250. Análisis de los resultados obtenidos

Para ejecutar sobre una tarjeta de aceleración *Xilinx* Alveo U250 el algoritmo SUnSAL para el problema FCLS acelerado por hardware desde *Xilinx Vitis Unified Software Platform* vamos a seguir los pasos que se detallan a continuación, partiendo de lo desarrollado a lo largo del capítulo 6:

1. Incluir en el código del archivo Host los datos preprocesados ($A \in \mathbb{R}^{224\times 22}$, $y \in \mathbb{R}^{224}$) de las secciones 7.1 y 7.2, tal y como se explica en el apéndice 9.2. Además, se deben fijar adecuadamente los valores $K = 224$ (número de bandas espectrales), $N = 22$ (número de *endmembers* seleccionados) y $M = 100$ (número de iteraciones del algoritmo SUnSAL) tanto en el archivo Kernel como en el Host, teniendo cuidado de que sean compatibles entre sí.

2. Compilar en modo hardware desde *Xilinx Vitis Unified Software Platform* la aplicación con el algoritmo SUnSAL. Esto generará el archivo binario *.xclbin* y un ejecutable *.exe* para poder lanzar el código compilado sobre la tarjeta de aceleración. La compilación suele tardar entre 2 y 3 horas, aunque posteriormente se pueden volver a compilar rápidamente ciertos bloques de código, si es que resulta necesario hacer alguna modificación adicional.

3. Conectar una tarjeta de aceleración *Xilinx* Alveo U250 a la máquina Host. En el caso de este trabajo, se han podido utilizar unos servidores para computación de alto rendimiento de la universidad de ETH Zürich que forman parte del programa HACC que se explica en el apéndice 9.3.

4. Lanzar el ejecutable generado tras la compilación en modo hardware con el siguiente comando:

```
$ ./<nombre_del_ejecutable> <nombre_del_xclbin>.xclbin
```

   Si queremos que al terminar la ejecución sobre la tarjeta de aceleración se generen varios archivos con datos acerca de latencia, consumo de hardware, secuencia de llamadas al API de XRT, etcétera, debemos incluir un archivo con nombre *xrt.ini* en el directorio desde el cual lancemos el ejecutable, con el siguiente contenido:

xrt.ini

```
[Debug]
native_xrt_trace=true
```

El principal archivo que nos proporciona los resultados de ejecución sobre la tarjeta de aceleración es el *xrt.run_summary*, que puede visualizarse con el módulo *Vitis Analyzer* usando el comando:

```
$ vitis_analyzer xrt.run_summary
```

Al hacerlo, lo primero que veremos será una pantalla con un resumen de la ejecución que incluye la fecha en la que se realizó y el tiempo que duró, que en este caso es de 279 ms. Esto no quiere decir que la latencia del algoritmo sea de 279 ms, ya que el tiempo de ejecución incluye pasos como la detección del dispositivo hardware o la carga del *.xclbin*.

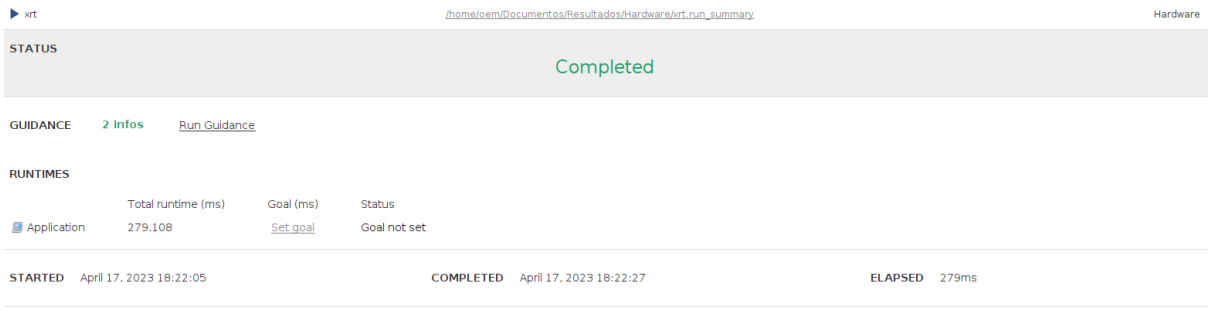


Figura 7.4: Pantalla principal de *Vitis Analyzer*.

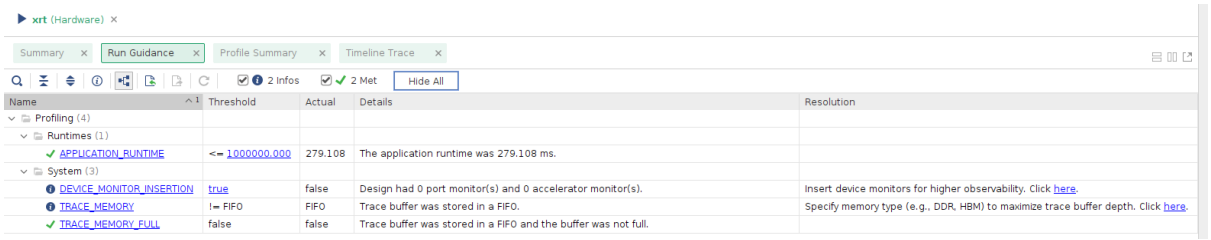


Figura 7.5: Pantalla con el tiempo de ejecución total.

Con *Vitis Analyzer* también podemos ver un diagrama de tiempo que muestra los pasos seguidos durante la ejecución, que coinciden con las llamadas al API de la librería XRT que se hacen en el código Host. Por ejemplo, en la figura 7.6 podemos ver los pasos iniciales de detección del dispositivo hardware usando *xrt::device::device* y la carga del *.xclbin* bajo *xrt::device::load_xclbin*, los cuales llevan más de 200 ms. Además, en la figura 7.7 se muestra el momento en que se lanza a ejecución sobre la tarjeta de aceleración el Kernel con el algoritmo, a los 262 ms.

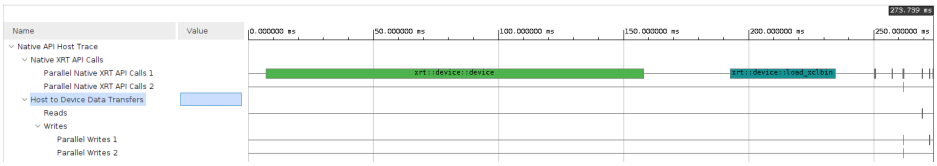


Figura 7.6: Llamadas iniciales al API de XRT en el código Host para detectar el dispositivo hardware y cargar el binario *.xclbin*.

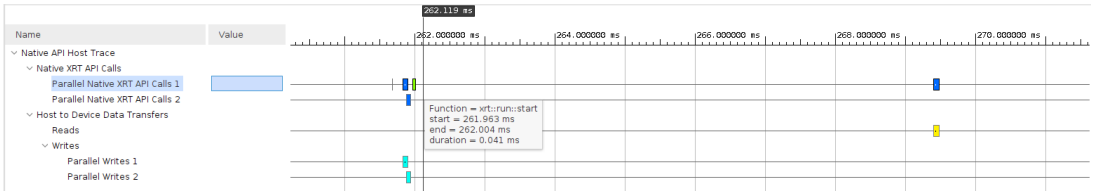


Figura 7.7: Inicio de la ejecución del código Kernel en el dispositivo hardware.

Por último, también se muestran datos relativos a las transferencias de lectura y escritura que hace la máquina Host sobre memoria principal del dispositivo hardware (ver figura 7.8), así como una recopilación de todas las llamadas al API de XRT junto con sus latencias asociadas (ver figura 7.9).

Host Data Transfers

Host Reads from Global Memory

Number of Reads	Maximum Buffer Size (KB)	Minimum Buffer Size (KB)	Average Buffer Size (KB)	Transfer Rate (MB/s)	Average Bandwidth Utilization (%)	Maximum Time (ms)	Minimum Time (ms)	Total Time (ms)	Average Time (ms)
1	32.768	32.768	32.768	383.979	∞	0.085	0.085	0.085	0.085

Host Writes to Global Memory

Number of Writes	Maximum Buffer Size (KB)	Minimum Buffer Size (KB)	Average Buffer Size (KB)	Transfer Rate (MB/s)	Average Bandwidth Utilization (%)	Maximum Time (ms)	Minimum Time (ms)	Total Time (ms)	Average Time (ms)
4	32.768	32.768	32.768	555.752	∞	0.073	0.039	0.236	0.059

Top Memory Reads: Host from Global Memory

Start Time (ms)	Buffer Size (KB)	Duration (ms)	Reading Rate (MB/s)
269.387	32.768	0.085	383.979

Top Memory Writes: Host to Global Memory

Start Time (ms)	Buffer Size (KB)	Duration (ms)	Writing Rate (MB/s)
261.821	32.768	0.073	446.729
261.869	32.768	0.063	517.009
272.285	32.768	0.060	543.624
272.386	32.768	0.039	843.710

Figura 7.8: Pantalla con las transferencias de datos entre el dispositivo hardware y la máquina Host.

Native API Calls

API Name	Number Of Calls	Total Time (ms)	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
xrt::bo::address	3	0.003	0.000	0.001	0.002
xrt::bo::bo	3	0.084	0.024	0.028	0.034
xrt::bo::map	6	0.015	0.001	0.003	0.006
xrt::bo::sync	5	0.317	0.038	0.063	0.085
xrt::device::device	1	150.926	150.926	150.926	150.926
xrt::device::load_xclbin	1	42.064	42.064	42.064	42.064
xrt::device::reset	1	0.028	0.028	0.028	0.028
xrt::kernel::kernel	1	0.037	0.037	0.037	0.037
xrt::run::run	1	0.042	0.042	0.042	0.042
xrt::run::start	1	0.044	0.044	0.044	0.044

Figura 7.9: Pantalla con las llamadas al API de XRT de *Vitis Analyzer*.

Los datos de latencia final del Kernel con la implementación acelerada por hardware del algoritmo SUnSAL ejecutado sobre la tarjeta de aceleración *Xilinx* Alveo U250 nos indican que, con una tolerancia de 0.0001 y tras 10 ejecuciones con los datos obtenidos en las secciones 7.1 y 7.2, el algoritmo tarda 6.59 ± 0.3 ms (ver figura 7.10), que entra dentro de la estimación obtenida al realizar la emulación por hardware previa, como se muestra en el siguiente reporte de síntesis.

```
================================================================
Kernel Name:                krnl_SUNSAL
Module Name:                krnl_SUNSAL
Target Frequency:           300.3 MHz
Estimated Frequency:        376.93 MHz
Best (cycles):              1081289
Worst (cycles):             3476491
Best (absolute):            3.604 ms
Worst (absolute):           11.587 ms
FF:                         70469
LUT:                        116750
BRAM:                       223
================================================================
```

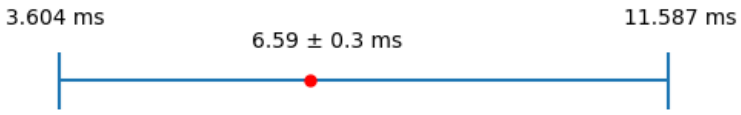


Figura 7.10: Latencia final de la ejecución del código Kernel con el algoritmo SUnSAL acelerado por hardware para $K = 224$ y $N = 22$. Los valores de los extremos indican la latencia mínima y máxima de la estimación dada al emular por hardware.

En cuanto a los resultados numéricos del vector de abundancias $x \in \mathbb{R}^{22}$, obtenemos altos porcentajes de kaolinita, dickita y calcita, con algunas trazas de alunita, lo que se adecua a la investigación previa cuyos resultados se explicaron en la sección 7.2. Unos resultados experimentales más detallados sobre la zona de la cadena montañosa de *Cuprite* usando el algoritmo SUnSAL y otros métodos de desmezclado espectral puede encontrarse en [14].

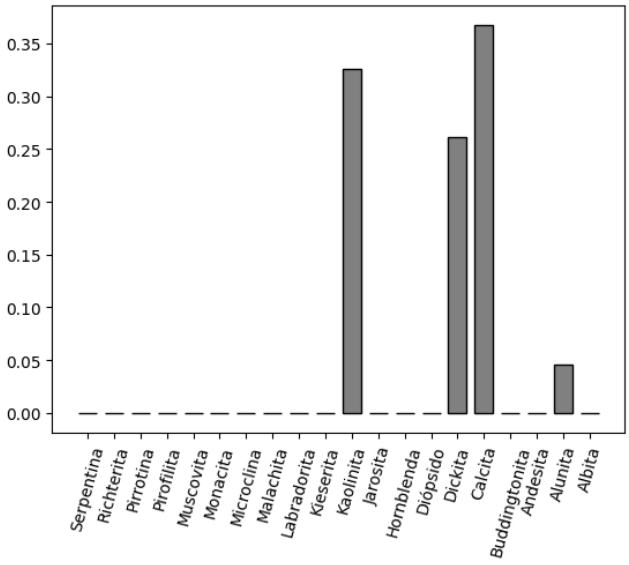


Figura 7.11: Abundancias relativas ($x \in \mathbb{R}^{22}$) obtenidas tras la ejecución del algoritmo SUnSAL.

Debido a la naturaleza iterativa del algoritmo SUnSAL derivada tanto del bucle principal como de la implementación del cálculo de la descomposición en valores singulares de *Vitis Libraries*, que internamente utiliza un método iterativo de Jacobi, no es posible indicar un factor de mejora del rendimiento del algoritmo SUnSAL acelerado en *Vitis Unified Software Platform* respecto de la implementación en Python, debido a que la librería de Python para la descomposición en valores singulares es diferente de la que posee *Vitis Unified Software Platform*, y por ende se obtienen resultados de latencia muy dispares según los *inputs* que se proporcionen. Esto dificulta también realizar comparativas entre SUnSAL y otros algoritmos de desmezclado espectral, para lo cual sería necesario un estudio más extenso y detallado del comportamiento del algoritmo ante un amplio rango de *inputs* y el uso de métricas específicas según la comparativa que se quiera realizar. Sin embargo, sí que es posible analizar el rendimiento de SUnSAL para imágenes hiperespectrales concretas con una misma librería espectral.

En el caso del ejemplo construido a lo largo de este capítulo, los resultados de latencia muestran una reducción de latencia de 13.154 ± 1.4 ms para el algoritmo SUnSAL en Python ejecutado en una CPU *Intel Core i7-1255U* de 10 núcleos con 16 GB de memoria RAM, hasta los 6.59 ± 0.3 ms del algoritmo acelerado en *Vitis Unified Software Platform*. Esto supone que el algoritmo acelerado con HLS es 2 veces más rápido que el original en Python para los datos concretos con los que hemos construido la matriz de *endmembers* y el vector de reflectancia del píxel seleccionado en la sección 7.1. Sin embargo, con modificar ligeramente la matriz de *endmembers*, la latencia del cálculo de la descomposición en valores singulares puede ser muy diferente en Python con respecto a la versión de SUnSAL acelerada en *Vitis Unified Software Platform*, obteniéndose grandes variaciones en cuanto a la latencia comparativa entre el algoritmo implementado en Python y el acelerado con HLS.

Teniendo en cuenta las apreciaciones anteriores, y con ánimo de realizar ejecuciones adicionales del algoritmo SUnSAL acelerado en *Vitis Unified Software Platform* para asentar los resultados de latencia, seleccionamos un conjunto de 20 píxeles cercanos de la misma imagen hiperespectral utilizada hasta ahora y, manteniendo también la misma librería espectral con los 22 *endmembers* seleccionados en la sección 7.2, ejecutamos el algoritmo en cada uno de los píxeles por separado. Como la velocidad de convergencia del algoritmo SUnSAL varía significativamente según el vector de reflectancia $y \in \mathbb{R}^{224}$ utilizado, obtenemos unos datos de latencia distintos para cada ejecución, que por supuesto se mantienen dentro de las estimaciones de la figura 7.10, pero que además son proporcionales a la latencia obtenida ejecutando en Python, debido a que la descomposición en valores singulares, que es la única parte que no es equivalente en Python y en el algoritmo diseñado para HLS por la ya comentada diferencia de librerías, solo depende de la matriz de *endmembers*, que en este caso se mantiene fija en todas las ejecuciones.

Figura 7.12: Latencia final de la ejecución en Python (amarillo) y con aceleración por hardware en la tarjeta de aceleración *Xilinx* Alveo U250 (rojo) del algoritmo SUnSAL con 20 vectores de reflectancia diferentes para $K = 224$ y $N = 22$. Los valores de los extremos indican la latencia mínima y máxima de la estimación dada al emular por hardware en *Vitis Unified Software Platform*. El rango de latencia en Python es de 11.869 ms hasta 14.321 ms, y acelerado con HLS de 4.234 ms hasta 9.123 ms.

Como comentarios finales, si este proceso lo realizamos para todos y cada uno de los píxeles de la imagen hiperespectral seleccionada en la sección 7.1, obtendríamos un mapa de abundancia de las sustancias puras presentes en el área de detección con el que poder analizar la composición del suelo correspondiente a la región de la Tierra fotografiada. Tal y como se ha implementado el algoritmo SUnSAl, esto requeriría ejecutarlo un número muy elevado de veces, para lo cual se puede proceder rediseñando el algoritmo SUnSAL para poder aplicar el desmezclado espectral a todos los píxeles de la imagen a la vez, lo cual no es viable en la gran mayoría de casos puesto que necesitaríamos tener en memoria la imagen hiperespectral completa, o construyendo un sistema de manejo de memoria e integración continua que se encargara de ejecutar el algoritmo con los datos apropiados todas las veces que hiciera falta.

Capítulo 8

Conclusiones

El análisis hiperespectral es un campo de investigación en auge que cada vez juega un papel más importante en ámbitos como la medicina, la monitorización de la superficie terrestre, la seguridad o la autenticación de obras artísticas, entre otros. Nos proporciona información acerca de la reflectancia de las sustancias a lo largo de todo el espectro electromagnético, lo que permite estudiar remotamente las condiciones y distribución de los materiales en una cierta zona de detección.

Por su lado, el hardware reconfigurable da la posibilidad de adaptar los componentes hardware de un dispositivo como una FPGA para implementar sobre una misma placa todo tipo de algoritmos sin más que indicar en un lenguaje HDL su comportamiento ante diversas entradas. Sumado a esto, debido al aumento de la complejidad de los algoritmos y con ánimo de abstraer aún más el proceso de diseño sobre hardware reconfigurable, surge la síntesis de alto nivel (HLS), que permite generar un diseño RTL en base a código en alto nivel escrito, por ejemplo, en lenguajes como C++ o Java.

Como herramienta HLS de referencia hoy en día encontramos *Xilinx Vitis Unified Software Platform*, sobre la cual se ha implementado un algoritmo de desmezclado espectral lineal basado en técnicas de regularización conocido como SUnSAL (*Spectral Unmixing by Variable Splitting and Augmented Lagrangian*). Durante todo el proceso de diseño se ha estudiado el funcionamiento del compilador que realiza la síntesis de alto nivel en *Xilinx Vitis Unified Software Platform* y se han puesto de relieve tanto las limitaciones como el potencial de esta nueva tecnología.

Más allá de una implementación básica del algoritmo SUnSAL, se han aplicado directivas de aceleración por hardware al código para optimizar el diseño y mejorar la latencia. Teniendo en cuenta siempre que se trabaja bajo las restricciones que impone *Xilinx Vitis Unified Software Platform* a la hora de dirigir el compilador HLS, se han podido ver en detalle los efectos de múltiples directivas de aceleración y se han explicado las razones de su inclusión y el porqué de su efecto beneficioso sobre el diseño del algoritmo. Ejemplo de ello es la aceleración por hardware realizada sobre el producto de matrices, con una mejora de más de 1000 veces la latencia original para matrices de tamaño 150 x 150 en adelante gracias al buen manejo de los recursos de memoria y la paralelización de los bucles involucrados. Los resultados mostrados a lo largo del trabajo validan la eficacia de este método de implementación hardware, que no está exento de problemas como el gran aumento de consumo de recursos hardware, la pérdida de precisión por la abstracción de las herramientas HLS y la gran barrera de entrada que existe por la actual complejidad de los sistemas HLS.

Los resultados obtenidos en cuanto a latencia y consumo de recursos hardware del algoritmo SUnSAL tras la aceleración son positivos y resisten bien ante tamaños grandes de matrices y vectores, como se ha justificado con datos de los reportes de síntesis de numerosas pruebas de emulación que muestran mejoras de hasta 10 veces la latencia original para matrices de tamaño 250 x 250 y de hasta 100 veces para las de tamaño 500 x 500. Teniendo en cuenta que el número de bandas espectrales de las imágenes captadas por la mayoría de sensores hiperespectrales actuales suele ser de entre 200 y 400, estos datos aseguran una mejora considerable de la latencia total del algoritmo en todos los casos. Sin embargo, esta mejora de latencia trae consigo un aumento considerable de consumo de recursos hardware, de hasta 20 veces el número de LUTs y FFs utilizados en la implementación inicial.

Por último, se han extraído datos de imágenes hiperespectrales tomadas por el sensor AVIRIS y se han seleccionado *endmembers* de la librería espectral USGS para simular un caso de ejecución real, para el cual se ha utilizado una tarjeta de aceleración *Xilinx* Alveo U250, con resultados que muestran una reducción de la latencia a la mitad respecto de la implementación del algoritmo SUnSAL en Python, ejecutado sobre una CPU convencional. Durante todo el proceso, se ha aprendido a procesar adecuadamente los datos hiperespectrales utilizados, prestando especial atención a la compatibilidad de los vectores de reflectancia de los *endmembers* con los valores de longitud de onda de las bandas espectrales de la imagen hiperespectral original.

Respecto a posibles avances futuros que expandan y completen lo realizado en este trabajo de fin de grado, hay multitud de líneas de investigación que pueden seguirse, desde un estudio más profundo y detallado de los compiladores HLS y la aceleración por hardware con directivas embebidas en el código, hasta la implementación y aceleración de otros algoritmos de desmezclado espectral lineal en *Xilinx Vitis Unified Software Platform* y su posterior comparación en términos de latencia y consumo de hardware con SUnSAL, pasando por la aplicación del propio algoritmo SUnSAL a imágenes hiperespectrales completas para la obtención de mapas de abundancia que involucren a todos los píxeles y análisis del rendimiento sobre conjunto de datos de mayor tamaño.

En conclusión, el potencial de la síntesis de alto nivel aplicada al análisis hiperespectral es muy grande, aún con las desventajas que surgen de las restricciones que actualmente imponen los compiladores HLS y la pérdida de precisión a la hora de generar un diseño RTL óptimo debido a la abstracción que se requiere a la hora de implementar un algoritmo en hardware reconfigurable directamente desde un lenguaje de programación de alto nivel como C++. En particular, la implementación y ejecución de algoritmos complejos y costosos de análisis hiperespectral se vuelve una tarea mucho más sencilla, como bien prueba la implementación del algoritmo de desmezclado espectral SUnSAL realizada, que no hubiera sido posible sobre un lenguaje HDL de descripción del hardware en un tiempo tan corto. Por tanto, esta nueva tecnología puede suponer una verdadera revolución en el ámbito del hardware reconfigurable en un futuro próximo, incluso facilitando la implementación de algoritmos sobre dispositivos basados en FPGAs a personas con un menor conocimiento de los entresijos del hardware reconfigurable actual.

Conclusions

Hyperspectral analysis is a growing field of research that is playing an increasingly important role in areas such as medicine, monitoring of the earth's surface, security or authentication of artistic works, among others. It provides us with information about the reflectance of substances across the entire electromagnetic spectrum, allowing us to remotely study the conditions and distribution of materials in a certain detection zone.

Meanwhile, reconfigurable hardware gives us the possibility to adapt the hardware components of a device such as an FPGA to implement all types of algorithms on the same board by just indicating in an HDL language their behavior in the face of different inputs. In addition to this, due to the increased complexity of algorithms and in order to further abstract the design process on reconfigurable hardware, high-level synthesis (HLS) has emerged, which allows generating an RTL design based on high-level code written, for example, in languages such as C++ or Java.

As a modern and technologically advanced reference HLS tool we find *Xilinx Vitis Unified Software Platform*, on which a linear spectral unmixing algorithm based on regularization techniques known as SUnSAL (*Spectral Unmixing by Variable Splitting and Augmented Lagrangian*) has been implemented. Throughout the design process, the performance of the compiler that performs the high-level synthesis in *Xilinx Vitis Unified Software Platform* has been studied and both the limitations and the potential of this new technology have been highlighted.

Beyond a basic implementation of the SUnSAL algorithm, hardware acceleration directives have been applied to the code to optimize the design and improve latency. Always keeping in mind that we are working under the constraints imposed by *Xilinx Vitis Unified Software Platform* when driving the HLS compiler, we have been able to see in detail the effects of multiple acceleration directives and have explained the reasons for their inclusion and why they have a beneficial effect on the design of the algorithm. Case in point is the hardware acceleration performed on the product of matrices, with an improvement of more than 1000 times the original latency for matrices of size 150 x 150 and upwards thanks to the good management of memory resources and the parallelization of the loops involved. The results shown throughout the work validate the effectiveness of this hardware implementation method, which is not without problems such as the large increase in hardware resource consumption, the loss of accuracy due to the abstraction of HLS tools, and the high entry barrier due to the current complexity of HLS systems.

The results obtained in terms of latency and hardware resource consumption of the SUnSAL algorithm after acceleration are positive and hold up well to large array and vector sizes, as substantiated by data from synthesis reports of numerous emulation tests showing improvements of up to 10 times the original latency for arrays of size 250 x 250 and up to 100 times for those of size 500 x 500. Considering that the number of spectral bands in images captured by most current hyperspectral sensors is typically between 200 and 400, these data ensure a considerable improvement in the overall latency of the algorithm in all cases. However, this latency improvement brings with it a considerable increase in hardware resource consumption, up to 20 times the number of LUTs and FFs used in the initial implementation.

Finally, hyperspectral image data taken by the AVIRIS sensor have been extracted and *endmembers* from the USGS spectral library have been selected to simulate a real execution case, for which a *Xilinx* Alveo U250 acceleration card has been used, with results showing a reduction of latency by half with respect to the implementation of the SUnSAL algorithm in Python, executed on a conventional CPU. Throughout the process, we have learned how to properly process the hyperspectral data used, paying special attention to the compatibility of the reflectance vectors of the *endmembers* with the wavelength values of the spectral bands of the original hyperspectral image.

Regarding possible future developments that expand and complete what has been done in this bachelor's thesis, there are many lines of research that can be pursued, from a deeper and more detailed study of HLS compilers and hardware acceleration with embedded directives in the code, to the implementation and acceleration of other linear spectral unmixing algorithms in *Xilinx Vitis Unified Software Platform* and their subsequent comparison in terms of latency and hardware consumption with SUnSAL, to the application of the SUnSAL algorithm itself to full hyperspectral images for obtaining abundance maps involving all pixels and performance analysis on larger datasets.

In conclusion, the potential of high-level synthesis applied to hyperspectral analysis is very large, even with the disadvantages arising from the restrictions currently imposed by HLS compilers and the loss of accuracy in generating an optimal RTL design due to the abstraction required when implementing an algorithm in reconfigurable hardware directly from a high-level programming language such as C++. In particular, the implementation and execution of complex and expensive hyperspectral analysis algorithms becomes a much simpler task, as well proven by the implementation of the SUnSAL spectral unmixing algorithm performed, which would not have been possible on a hardware description HDL language in such a short time. Therefore, this new technology can be a real revolution in the field of reconfigurable hardware in the near future, even facilitating the implementation of algorithms on FPGA-based devices to people with less knowledge of the intricacies of current reconfigurable hardware.

Capítulo 9

Apéndices

9.1. Instalación y configuración básica de *Xilinx Vitis Unified Software Platform* 2022.1 en Ubuntu 20.04.4 LTS

Se describe a continuación el proceso que se debe llevar a cabo para la instalación y configuración de todo el entorno *Xilinx Vitis Unified Software Platform* 2022.1 para desarrollo HLS en Alveo U250 sobre el sistema operativo Ubuntu 20.04.4 LTS. Debido a que es una plataforma moderna que se encuentra en desarrollo activo en *Xilinx*, hay que prestar especial atención a las especificaciones y compatibilidades que se indican en los manuales, y proceder acordemente. Como se indica en [71], los requerimientos mínimos para la instalación son los mostrados en la tabla 9.1.

Componente	Requerimiento Mínimo
Sistema operativo	Linux, 64-bit, Ubuntu 20.04.4 LTS
Memoria del Sistema Host	80 GB (Mínimo 64 GB)
Conexión a Internet	Sí
Espacio de Disco Duro	100 GB

Tabla 9.1: Requerimientos mínimos para la instalación de *Xilinx Vitis Unified Software Platform* 2022.1.

Asímismo, *Xilinx Vitis Unified Software Platform* 2022.1 también está disponible en los siguientes sistemas operativos Linux, 64-bit: Red Hat Enterprise Workstation/Server 7/CentOS: 7.8, 7.9, Red Hat Enterprise Workstation/Server: 8.1, 8.2, 8.3, 8.4, 8.5, Ubuntu 18.04.4 LTS, 18.04.5 LTS, 20.04 LTS, 20.04.2 LTS, 20.04.3 LTS y Amazon Linux 2 AL2 LTS. Sin embargo, para tener compatibilidades con el mayor número posible de tarjetas de aceleración (incluyendo la Alveo U250) y funciones de *Vitis HLS*, recomendamos instalar Ubuntu 20.04.4 LTS si se va a realizar un uso extensivo de la plataforma.

Lo primero es instalar los paquetes necesarios. Algunos de ellos son requerimientos de *Xilinx Vitis Unified Software Platform*, y otros nos serán de utilidad a la hora de manejar los archivos de configuración.

```
$ apt-get install ocl-icd-opencl-dev libboost-dev libboost-filesystem-dev uuid-dev
dkms libprotoc-dev protobuf-compiler libncurses5-dev lsb-release libxml2-dev libyaml
-dev
$ apt-get install ocl-icd-libopencl1 opencl-headers ocl-icd-opencl-dev linux-libc-
dev g++ gcc gdb make libopencv-core-dev libopencv-core4.2  libjpeg-dev libpng-dev
python3 git dmidecode lsb unzip linux-headers-$(uname -r)
$ apt-get install libncurses5-dev libstdc++6 libgtk2.0-0 dpkg-dev libtinfo5
libncurses5 libncursesw5-dev libtinfo-dev libstdc++6:i386 libgtk2.0-0:i386 libc6-dev
-i386
```

Ahora, entramos en la página web de *Xilinx* a través del siguiente enlace: `https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis/2022-1.html`. Teniendo cuidado de estar en la pestaña de *Vitis (SW Developer)*, como se muestra en la figura 9.1, descargamos el .bin que se encuentra bajo el título Xilinx Unified Installer 2022.1: Linux Self Extracting Web Installer de la figura 9.2.

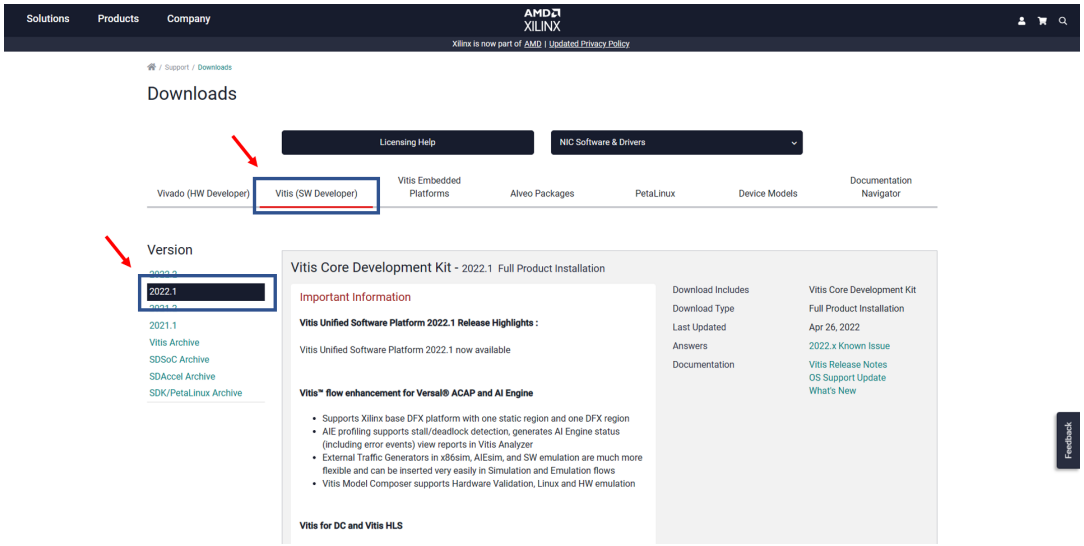


Figura 9.1: Pestaña para la descarga del instalador de *Xilinx Vitis Unified Software Platform* 2022.1.

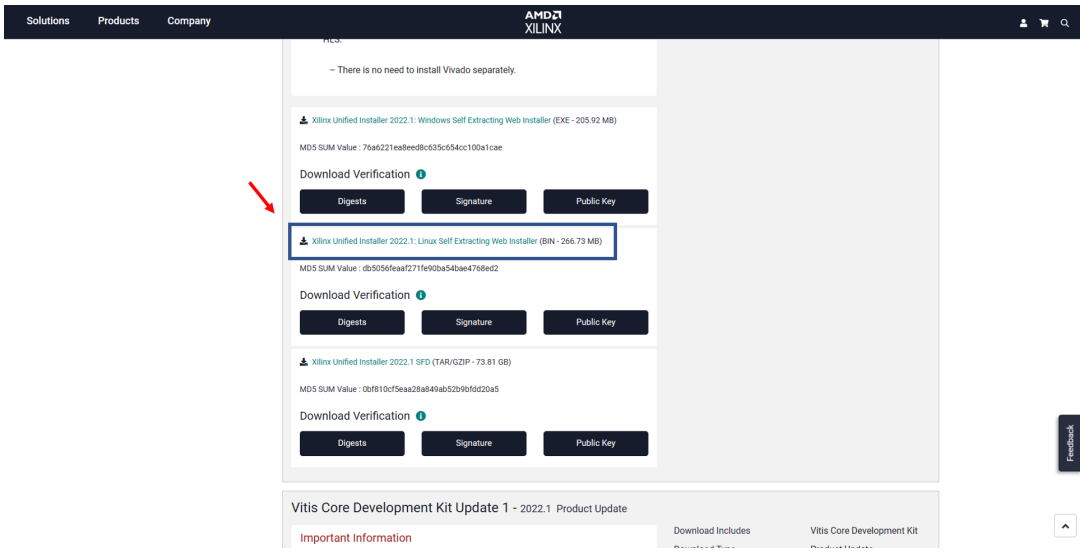


Figura 9.2: Archivo .bin para Xilinx Unified Installer 2022.1: Linux Self Extracting Web Installer.

Desde el directorio en donde se descargue, le damos permisos de ejecución con

```
$ chmod +x Xilinx_Unified_2022.1_0420_0327_Lin64.bin
```

y lanzamos el instalador con el comando

```
$ ./Xilinx_Unified_2022.1_0420_0327_Lin64.bin
```

siguiendo hasta el final todos los pasos que se nos vayan indicando.

En primer lugar, veremos la pantalla de inicio del instalador que se muestra en la figura 9.3. Como se indica, para evitar que la instalación dure un tiempo excesivo, se recomienda deshabilitar cualquier antivirus que se tenga instalado en la máquina Host.

En segundo lugar, nos pedirá que introduzcamos una cuenta de *Xilinx*, que debemos crear desde la página web `https://www.xilinx.com/`, y seleccionaremos, como en la figura 9.4, la instalación de *Vitis*.

Como último paso antes de iniciarse la instalación, marcaremos y desmarcaremos con cuidado todas las opciones que se muestran en la figura 9.5. Tras hacer esto, comenzará la instalación que procederá de manera automática a lo largo de todo el proceso, avisándonos de los errores que se produzcan.

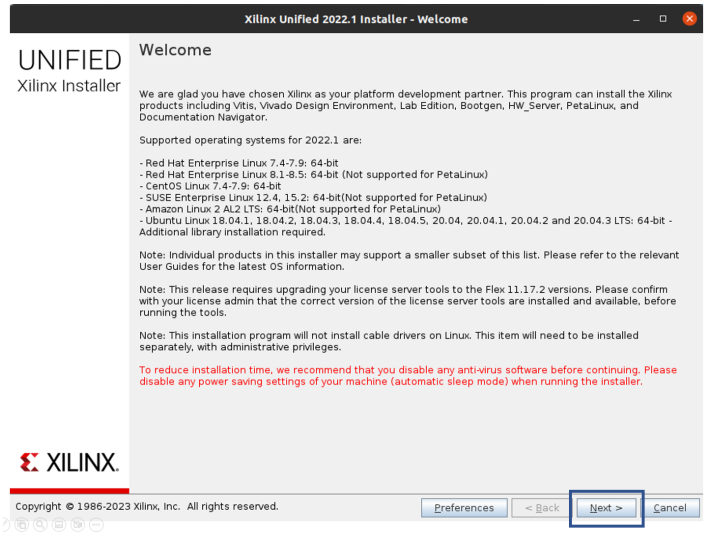


Figura 9.3: Pestaña inicial para la instalación de *Xilinx Vitis Unified Software Platform*.

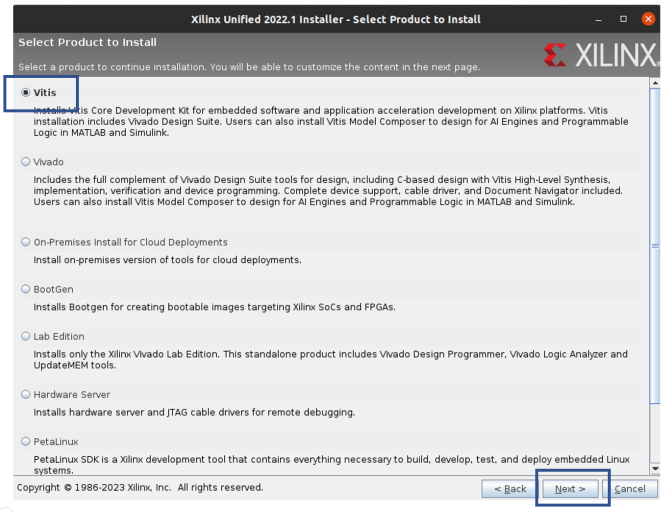


Figura 9.4: Pestaña para la selección de *Xilinx Vitis Unified Software Platform*.

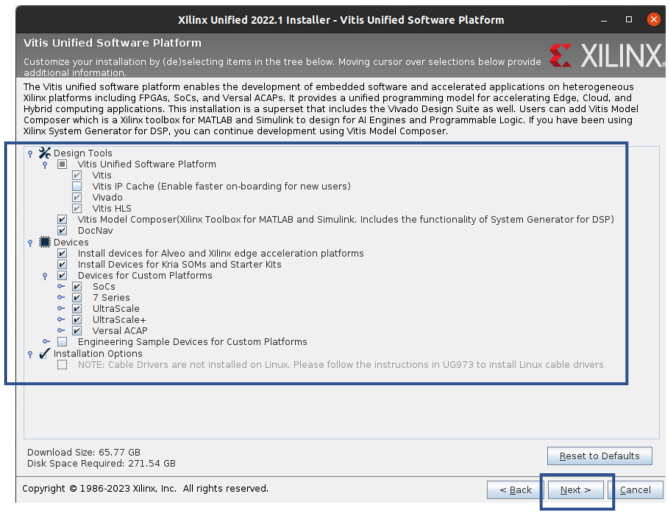


Figura 9.5: Pestaña para selecciones de los paquetes a instalar.

Aunque el tiempo de instalación varíe según las características de la máquina que estemos usando para ello, suele tardar entre 3 y 5 horas debido a la alta cantidad de archivos. Una vez terminado, salimos del instalador y regresamos a la página web de *Xilinx* para Alveo U250 a través del siguiente enlace: `https://www.xilinx.com/products/boards-and-kits/alveo/u250.html#gettingStarted`, desde donde debemos seleccionar 2022.1 -> ubuntu -> 20.04 como se muestra en la figura 9.6.

Figura 9.6: Pestaña para la descarga de los paquetes de Alveo U250.

Al hacerlo nos aparecerán los enlaces de las descargas para la librería XRT, la plataforma de deployment y la plataforma de desarrollo (ver figura 9.7), que deberemos pulsar uno a uno.

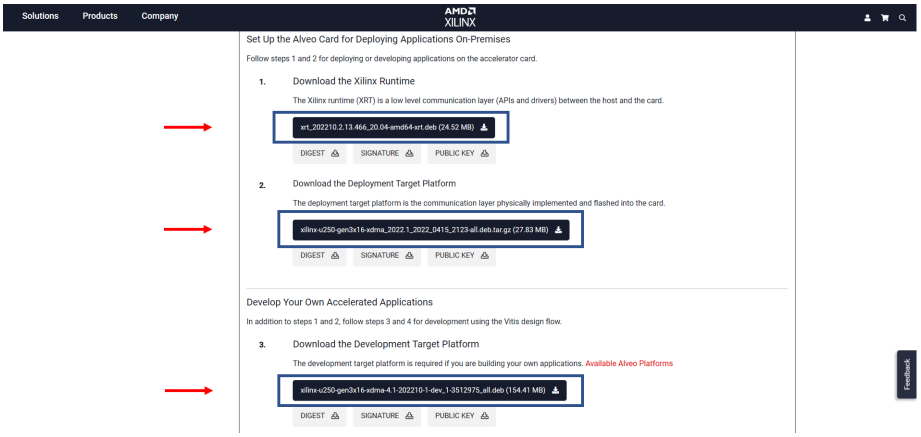


Figura 9.7: Enlaces para descargar la librería XRT, la plataforma de deployment y la plataforma de desarrollo para Alveo U250.

Ahora, y desde los respectivos directorios en donde se hayan descargado, ejecutamos los siguientes comandos para instalar todos los archivos .deb:

- Para XRT:

```
$ apt install ./xrt_202210.2.13.466_20.04-amd64-xrt.deb
```

- Para DTP deployment:

```
$ tar -xf xilinx-u250-gen3x16-xdma_2022.1_2022_0415_2123-all.deb.tar.gz
$ apt install ./xilinx-cmc-u200-u250_1.2.23-3395909_all.deb
$ apt install ./xilinx-sc-fw-u200-u250_4.6.20-1.28f0c61_all.deb
$ apt install ./xilinx-u250-gen3x16-base_4-3494623_all.deb
$ apt install ./xilinx-u250-gen3x16-xdma-validate_4.1-3512975_all.deb
$ apt install ./xilinx-u250-gen3x16-xdma-shell_4.1-3494623_all.deb
```

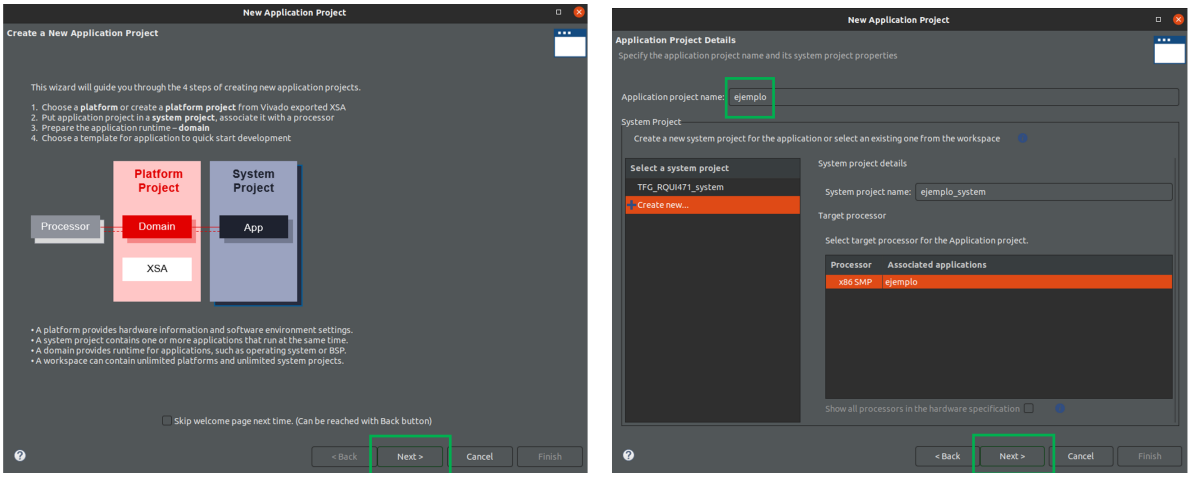
- Para DTP development:

```
$ apt install ./xilinx-u250-gen3x16-xdma-4.1-202210-1-dev_1-3512975_all.deb
```

Con esto habremos terminado el proceso de instalación de *Xilinx Vitis Unified Software Platform*, y podemos lanzar la plataforma de la siguiente forma:

```
$ cd /tools/Xilinx_2022.1/Vitis/2022.1
$ source settings64.sh
$ vitis
```

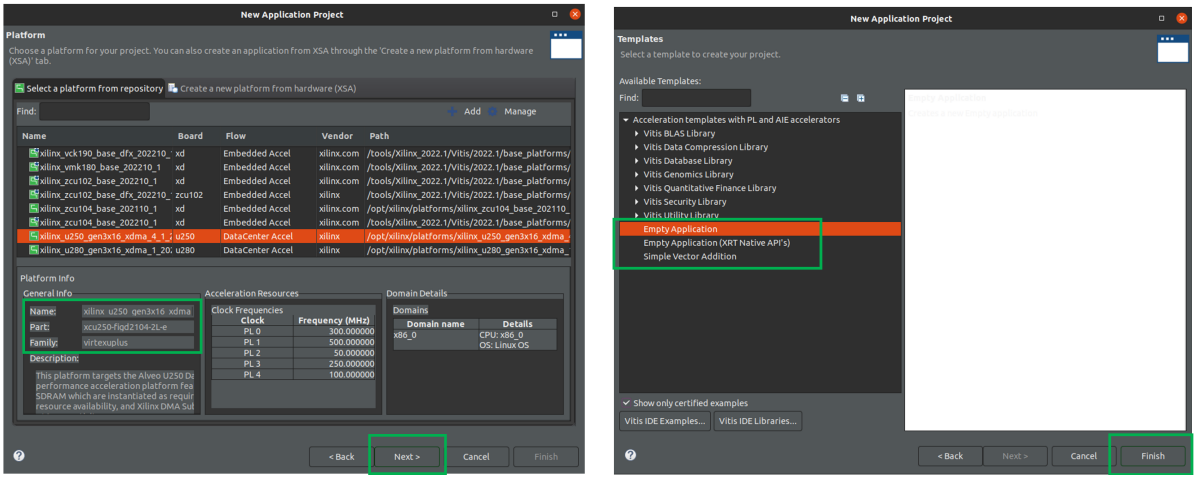
lo que hará que se muestre una ventana semejante a la mostrada en la figura 9.8a. Para crear un proyecto de HLS debemos seguir las indicaciones que nos vayan apareciendo, de la manera que se indica a continuación. Primero, pulsamos *Next* en la pantalla de inicio. En segundo lugar, indicamos un nombre para el proyecto en la pantalla que se muestra en la figura 9.8b. De esta forma, se generarán todos los directorios y archivos del proyecto bajo un nombre común.



(a) Pantalla de inicio de creación de un proyecto HLS. (b) Pantalla de introducción del nombre del proyecto.

Figura 9.8: Creación de un nuevo proyecto en *Xilinx Vitis Unified Software Platform*.

En tercer lugar, seleccionamos en la pantalla de la figura 9.9a un dispositivo hardware para nuestro proyecto, que puede ser una tarjeta de aceleración u otro tipo de dispositivo *Xilinx* compatible. Por último, y en caso de querer una plantilla base para nuestro proyecto, podemos indicarla en la pantalla de la figura 9.9b. Por defecto, *Xilinx Vitis Unified Software Platform* incluye una plantilla vacía, un ejemplo de suma de vectores como el visto en la sección 4.2.1 y ejemplos de uso para *Vitis Libraries*.



(a) Pantalla de selección de un dispositivo hardware. (b) Pantalla de selección de una plantilla base.

Figura 9.9: Opciones de configuración para un nuevo proyecto de *Xilinx Vitis Unified Software Platform*.

9.2. Código de archivo Host usando librerías de OpenCL

En la sección 4.2.1 vimos un ejemplo de archivo Host para *Xilinx Vitis Unified Software Platform* que utilizaba la librería XRT de *Xilinx* para realizar la detección del dispositivo hardware, la asignación de memoria en la máquina Host y la ejecución del código Kernel. Sin embargo, la manera óptima de diseñar el Host pasa por utilizar unas librerías *OpenCl C++ Bindings* [18] que permiten gestionar de manera más precisa los dispositivos hardware y realizar un control de errores más detallado. A continuación, se explica el código Host utilizado para la implementación del capítulo 6 con librerías de OpenCL.

En primer lugar, definimos una función que va a ayudarnos a lo largo de todo el código Host a detectar errores de ejecución, configuraciones erróneas de argumentos o fallos en las asignaciones de *buffers* tanto en la memoria de la máquina Host como en memoria principal del dispositivo hardware utilizado.

```
# define OCL_CHECK(error, call)
    call;
    if (error != CL_SUCCESS) {
        printf("%s:%d Error calling " #call ", error code is: %d\n",
                          __FILE__, __LINE__, error);
        exit(EXIT_FAILURE);
    }
```

El *main* tiene un único argumento de entrada que debe indicarse al ejecutar el Host. Este argumento será la ruta al archivo *.xclbin* introducido en la sección 4.2.1, que posteriormente se utilizará para lanzar el código Kernel al dispositivo hardware que esté disponible. Si no se indica ningún archivo *.xclbin*, la aplicación dará error al lanzarse y no podrá completar la ejecución del Kernel.

```
int main(int argc, char* argv[]) {

    if (argc != 2) {
        std::cout << "Usage: " << argv[0] << " <xclbin>" << std::endl;
        return EXIT_FAILURE;
    }

    std::string xclbinFilename = argv[1];

    ...
```

Ahora, buscamos un dispositivo hardware de *Xilinx* compatible. Para ello, recorremos todas las plataformas de dispositivos compatibles con OpenCL [79], y si encontramos una con nombre *Xilinx* buscamos en ella un dispositivo hardware como una tajeta de aceleración Alveo o una FPGA. Si se han seguido los pasos de instalación de la sección 9.1, se debería detectar directamente la tarjeta Alveo U250.

```
    ...

    std::vector<cl::Device> devices;
    std::vector<cl::Platform> platforms;
    bool found_device = false;

    // Recorrer todos los dispositivos hardware para encontrar uno disponible
    cl::Platform::get(&platforms);
    for (size_t i = 0; (i < platforms.size()) & (found_device == false); i++) {
        cl::Platform platform = platforms[i];
        std::string platformName = platform.getInfo<CL_PLATFORM_NAME>();
        if (platformName == "Xilinx") {
            devices.clear();
            platform.getDevices(CL_DEVICE_TYPE_ACCELERATOR, &devices);
            if (devices.size()) {
                found_device = true;
                break;
```

```
            }
        }
    }

    if (found_device == false) {
        std::cout << "Error: Unable to find Target Device " << std::endl;
        return EXIT_FAILURE;
    }

    ...
```

Si se ha conseguido encontrar un dispositivo hardware compatible, abrimos y cargamos el archivo *.xclbin* posicionando adecuadamente sobre él el puntero de lectura con ayuda de las funciones de C++ *seekg* y *tellg*. Después, leemos su contenido en un *buffer*, que posteriormente usamos para programar y configurar el dispositivo hardware bajo un contexto de ejecución que aúna la información obtenida del *.xclbin*, los datos de la máquina Host y la configuración del dispositivo hardware. Si no hay ningún error, creamos una cola de ejecución con *cl::CommandQueue* para gestionar múltiples archivos Kernel (en caso de haberlos), y cargamos los archivos Kernel que vamos a ejecutar en forma de objetos *cl::Kernel*.

```
    ...

    cl_int err;
    cl::Context context;
    cl::CommandQueue q;
    cl::Kernel krnl_SUNSAL;
    cl::Program program;

    std::cout << "INFO: Reading " << xclbinFilename << std::endl;
    FILE* fp;
    if ((fp = fopen(xclbinFilename.c_str(), "r")) == nullptr) {
        printf("ERROR: %s xclbin not available please build\n", xclbinFilename.c_str());
        exit(EXIT_FAILURE);
    }

    // Cargar el archivo .xclbin y posicionar los punteros de lectura
    std::cout << "Loading: '" << xclbinFilename << "'\n";
    std::ifstream bin_file(xclbinFilename, std::ifstream::binary);
    bin_file.seekg(0, bin_file.end);
    unsigned nb = bin_file.tellg();
    bin_file.seekg(0, bin_file.beg);
    char* buf = new char[nb];
    bin_file.read(buf, nb);

    // Programar el dispositivo hardware a partir del .xclbin
    cl::Program::Binaries bins;
    bins.push_back({buf, nb});
    bool valid_device = false;

    for (unsigned int i = 0; i < devices.size(); i++) {

        auto device = devices[i];

        // Creación del contexto de ejecución para el dispositivo

        OCL_CHECK(err, context = cl::Context(device, nullptr, nullptr, nullptr, &err));

        // Cola de ejecución de los códigos Kernel

        OCL_CHECK(err, q = cl::CommandQueue(context, device,
```

```
            CL_QUEUE_PROFILING_ENABLE, &err));

    std::cout << "Trying to program device[" << i << "]: " <<
            device.getInfo<CL_DEVICE_NAME>() << std::endl;
    cl::Program program(context, {device}, bins, nullptr, &err);

    if (err != CL_SUCCESS) {
        std::cout << "Failed to program device[" << i << "] with xclbin file!\n";
    } else {
        std::cout << "Device[" << i << "]: program successful!\n";
        // Objeto cl::Kernel para krnl_SUNSAL
        OCL_CHECK(err, krnl_SUNSAL = cl::Kernel(program, "krnl_SUNSAL", &err));
        valid_device = true;
        break; // hemos encontrado un dispositivo compatible
    }

}

if (!valid_device) {
    std::cout << "Failed to program any device found, exit!\n";
    exit(EXIT_FAILURE);
}

...
```

A partir de este punto, preparamos ya los argumentos del Kernel para lanzar su ejecución. Creamos los respectivos *buffers* de la misma forma que en el código de la sección 4.2.1, los asignamos a los argumentos del Kernel y por último generamos unos punteros que nos indiquen el inicio de estos *buffers* para poder rellenarlos con los datos concretos que tendrán los argumentos del Kernel.

```
...

// Crear los OpenCL buffers
OCL_CHECK(err, cl::Buffer buffer_A(context, CL_MEM_READ_ONLY,
        size_in_bytes, NULL, &err));
OCL_CHECK(err, cl::Buffer buffer_y(context, CL_MEM_READ_ONLY,
        size_in_bytes, NULL, &err));
OCL_CHECK(err, cl::Buffer buffer_x(context, CL_MEM_WRITE_ONLY,
        size_in_bytes, NULL, &err));

// Argumentos del Kernel
int narg = 0;
OCL_CHECK(err, err = krnl_SUNSAL.setArg(narg++, buffer_A));
OCL_CHECK(err, err = krnl_SUNSAL.setArg(narg++, buffer_y));
OCL_CHECK(err, err = krnl_SUNSAL.setArg(narg++, buffer_x));

// Mapear los OpenCL buffers a punteros
float* ptr_A;
float* ptr_y;
float* ptr_x;
OCL_CHECK(err, ptr_A = (float*)q.enqueueMapBuffer(buffer_A, CL_TRUE,
        CL_MAP_WRITE, 0, size_in_bytes, NULL, NULL, &err));
OCL_CHECK(err, ptr_y = (float*)q.enqueueMapBuffer(buffer_y, CL_TRUE,
        CL_MAP_WRITE, 0, size_in_bytes, NULL, NULL, &err));
OCL_CHECK(err, ptr_x = (float*)q.enqueueMapBuffer(buffer_x, CL_TRUE,
        CL_MAP_READ, 0, size_in_bytes, NULL, NULL, &err));

...
```

Utilizando los punteros obtenidos en el anterior fragmento de código, indicamos los valores de los argumentos del Kernel y migramos los *buffers* a memoria principal del dispositivo hardware desde el Host. Si esto se ha podido llevar a cabo sin errores, ahora sí se lanza a ejecutar el código Kernel en el dispositivo hardware (notar que puede haber múltiples Kernel, en cuyo caso se gestionan como una cola FIFO) y se recupera el resultado de la ejecución así como los datos generados, que pueden ser leídos justo después. Si ocurre algún error, la función *OCL_CHECK* que escribimos al principio nos ayudará a detectar su origen y a determinar la causa del fallo.

```
    ...

    // ESCRIBIR INPUTS con ptr_A y ptr_y

    // Migrar datos a espacio del Kernel
    OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_A, buffer_y},
             0 /* 0 quiere decir desde el Host*/));

    // Ejecutar el Kernel
    std::cout << "Launching the Kernel" << "\n";
    OCL_CHECK(err, err = q.enqueueTask(krnl_SUNSAL));

    // Los resultados de la ejecución del Kernel deben ser recuperados
    OCL_CHECK(err, q.enqueueMigrateMemObjects({buffer_x}, CL_MIGRATE_MEM_OBJECT_HOST));

    OCL_CHECK(err, q.finish());

    // OBTENER OUTPUTS con ptr_x

    ...
```

Por último, se deshace la migración de los *buffers* al dispositivo hardware y se cierra la cola de ejecución de los archivos Kernel. Con esto, habríamos acabado con la ejecución de nuestro algoritmo en el dispositivo hardware que tengamos, y se habrían generado el reporte de síntesis y otros datos de ejecución en los directorios correspondientes del proyecto.

```
    ...

    OCL_CHECK(err, err = q.enqueueUnmapMemObject(buffer_A, ptr_A));
    OCL_CHECK(err, err = q.enqueueUnmapMemObject(bufferXilinx_y, ptr_y));
    OCL_CHECK(err, err = q.enqueueUnmapMemObject(buffer_x, ptr_x));
    OCL_CHECK(err, err = q.finish());

    return EXIT_SUCCESS;

}
```

Aunque es cierto que es un código complejo, hay un punto importante a tener en cuenta al pensar acerca del código Host en *Xilinx Vitis Unified Software Platform*: es completamente reutilizable. Basta con cambiar el nombre del archivo Kernel a utilizar y ajustar los *buffers* según los argumentos de éste para tener un código Host compatible con otro Kernel distinto. Por tanto, pensando en facilitar el uso de *Xilinx Vitis Unified Software Platform* de cara al futuro, nos hacemos la siguiente pregunta: ¿dejará el archivo Host de ser considerado como algo que un usuario tiene que escribir para pasar a ser parte de las opciones de configuración de *Xilinx Vitis Unified Software Platform* en un futuro próximo? Después de entender el funcionamiento completo del código Host anterior, no parece descabellado pensarlo.

9.3. Programa HACC de ETH Zürich para *high performance computing* (HPC). Servidores de tarjetas de aceleración *Xilinx* Alveo

Para realizar las ejecuciones del algoritmo de desmezclado espectral SUnSAL implementado en el capítulo 6 aplicado al problema FCLS introducido en la sección 5.3.2 sobre las tarjetas de aceleración *Xilinx* Alveo U250 detalladas en la sección 4.3, se han utilizado unos clusters de servidores remotos de la universidad ETH Zürich [21] con diversas tarjetas Alveo preinstaladas, que son accesibles a través del programa HACC (*Heterogeneous Accelerated Compute Clusters*) [22] [82] [23].

En el marco del AMD XUC (*AMD Xilinx University Program*) [80], *Heterogeneous Accelerated Compute Clusters* (HACCs) es una iniciativa para apoyar la investigación novedosa en aceleración computacional adaptativa para computación de alto rendimiento (HPC). El alcance del programa abarca sistemas, arquitectura, herramientas y aplicaciones. Los HACC están equipados con las últimas tecnologías de hardware y software de AMD *Xilinx* para la investigación en aceleración computacional adaptativa. En concreto, las tarjetas disponibles en los clusters de ETH Zürich se muestran en la figura 9.10.

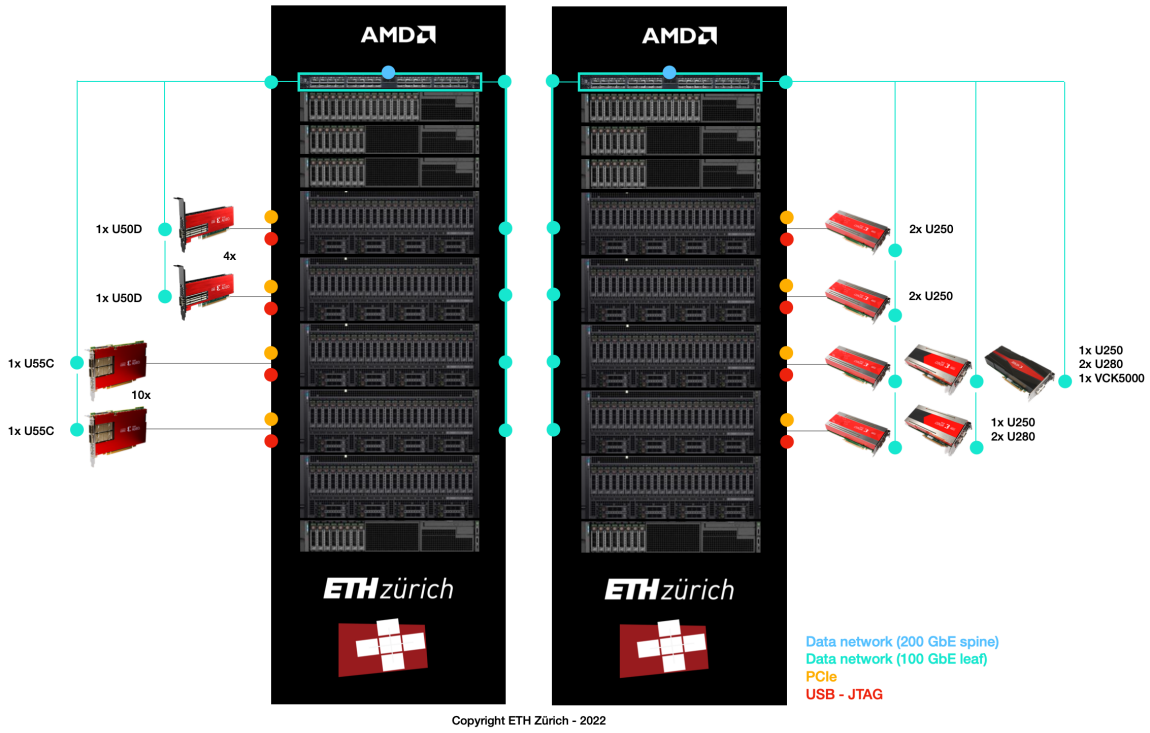


Figura 9.10: Esquema de las tarjetas *Xilinx* Alveo disponibles en los servidores de ETH Zürich para el programa HACC (*Heterogeneous Accelerated Compute Clusters*). [22]

Para obtener acceso a estos servidores, es necesario tener un trabajo concreto de investigación que requiera de HPC y basta con seguir las instrucciones que se detallan en [81] y [48]. Una vez obtenidas las credenciales para los servidores remotos, se debe proceder de la siguiente manera:

1. Primero, hay que realizar una reserva del servidor remoto que se vaya a utilizar desde la opción correspondiente de [22]. Desde la universidad de ETH Zürich, se pide no realizar reservas superiores a las 5 horas y hacerlas con suficiente antelación para no molestar a otros usuarios.

2. En segundo lugar, hay que incluir en la carpeta *HOME* del usuario de nuestra máquina Host que vaya a acceder a los servidores remotos el archivo *.ssh/config* con el siguiente contenido, que permite atravesar el PROXY de los servidores con nombre *jumphost.ethz.ch*:

Código a incluir en el archivo $HOME/.ssh/config de la máquina Host.

```
# Remote Access by Secure Shell SSH - ETHZ

        ServerAliveInterval 300
        ServerAliveCountMax 12

        Host jumphost.inf.ethz.ch
            User <user_name>

        Host             User <user_name>
            ProxyJump jumphost.inf.ethz.ch
```

3. Por último, hacer

```
$  ssh <user_name>@<nombre_del_servidor>.ethz.ch
```

Al entrar al servidor, pedirá la contraseña del *jumphost* y la contraseña de ETH para el servidor concreto, haciendo el redireccionamiento directamente. Para subir un proyecto que se quiera ejecutar, hacemos un *scp* incluyendo el *.exe* generado en el Build en Hardware. Desde este punto, ya podemos manejar el proyecto en el servidor a nuestro gusto.

Cluster	# instances	Booking	Name	Example
Build	1	No	alveo-build-01	ssh USERNAME@alveo-build-01.ethz.ch
U250	6	Yes	alveo-u250-[01:06]	ssh USERNAME@alveo-u250-01.ethz.ch
U280	4	Yes	alveo-u280-[01:04]	ssh USERNAME@alveo-u280-01.ethz.ch
U50D	4	Yes	alveo-u50d-[01:04]	ssh USERNAME@alveo-u50d-01.ethz.ch
U55C	10	Yes	alveo-u55c-[01:10]	ssh USERNAME@alveo-u55c-01.ethz.ch
Versal	1	Yes	versal-vck5000-01	ssh USERNAME@versal-vck5000.ethz.ch

Figura 9.11: Ejemplos de servidores remotos con sus respectivas tarjetas de aceleración del cluster de ETH Zürich en el programa HACC (*Heterogeneous Accelerated Compute Clusters*). [23]

Cluster	High-end servers				Xilinx accelerator card			FPGA/ACAP				
	Family	Memory	CPU cores	SSD	Family	DDR	FPGA/ACAP	LUTs	Registers	DSPs	RAM	HBM2
Build	PowerEdge	394 GB	80	3 TB	-	-	-	-	-	-	-	-
U250	PowerEdge	128 GB	16	200/300 GB	Alveo U250	64 GB	VU13P	1'728 K	3'456 K	12'288	UltraRAM: 368.0 Mb SRAM: 54.0 MB	-
U280	PowerEdge	128 GB	16	200/300 GB	Alveo U280	32 GB	VU37P	1'304 K	2'607 K	9'024	Distributed RAM: 36.7 Mb BRAM: 70.9 Mb UltraRAM: 270.0 Mb SRAM: 41.0 MB	8 GB
U50D	AMD EPYC	64 GB	32	480 GB	Alveo U50D	-	VU35P	872 K	1'743 K	5'952	Distributed RAM: 24.6 Mb BRAM: 47.3 Mb UltraRAM: 180.0 Mb SRAM: 28.0 MB	8 GB
U55C	AMD EPYC	64 GB	32	1.2 TB	Alveo U55C	-	VU47P	1'304 K	2'607 K	9'024	Distributed RAM: 36.7 Mb BRAM: 70.9 Mb UltraRAM: 270.0 Mb SRAM: 43.0 MB	16 GB
Versal	PowerEdge	128 GB	16	200 GB	Versal VCK5000	16 GB	VC1902	899'840	-	-	SRAM: 23.9 MB	-

Figura 9.12: Información detallada de los servidores remotos con sus respectivas tarjetas de aceleración del cluster de ETH Zürich en el programa HACC (*Heterogeneous Accelerated Compute Clusters*). [23]

9.4. Códigos y reportes de síntesis completos

En este apéndice incluimos el código completo utilizado en la implementación realizada en el capítulo 6 del algoritmo de desmezclado espectral SUnSAL aplicado al problema FCLS introducido en la sección 5.3.2. Además, mostramos también los diversos reportes de síntesis completos obtenidos, que también pueden consultarse en [26] en sus formatos de archivo originales.

9.4.1. Código Python del algoritmo SUnSAL para el problema FCLS

```
import sys
import scipy as sp
import numpy as np
import scipy.linalg as splin
from numpy import linalg as LA

def sunsal(A, y, iters = 100, tol = 1e-4, x0 = None):

    # PARSEO DE LOS ARGUMENTOS

    [k,n] = A.shape
    [l] = y.shape
    if k != l:
        sys.exit('matrix A and data vector y are inconsistent')

    if (iters < 0 ):
        sys.exit('iters must a positive integer')

    if x0 is not None:
        if x0.shape != k:
            sys.exit('initial x0 is not consistent with matrix A')

    # NORMALIZACION

    # Norma de A
    norm_A = splin.norm(A)*(25 + n) / float(n)

    # Reescalar A e y
    A = A / norm_A
    y = y / norm_A

    # PREPROCESADO (IB, Aux, IB1, yy)

    mu = 0.01

    # Descomposicion en valores singulares
    [U,S] = splin.svd(sp.dot(A.T,A))[:2]

    # Inversa de B
    IB = sp.dot( sp.dot(U,sp.diag(1./(S+mu))) , U.T )

    # C
    C = (1./IB.sum()) * sp.sum(IB,axis=1,keepdims=True)

    # Aux
    Aux = sp.sum(C,axis=1,keepdims=True)

    # Matrix auxiliar IB1
    IB1 = IB - sp.dot(Aux,sp.sum(IB,axis=0,keepdims=True))

    # Traspuesta de A por y
    yy = sp.dot(A.T,y)

```

```
54      # VALORES INICIALES
55
56      if x0 is None:
57          x = sp.dot( sp.dot(IB,A.T) , y)
58      else:
59          x = x0
60
61      u = x
62      d  = 0
63
64      # ITERACIONES
65
66      # Tolerancia de error
67      tol = sp.sqrt(k * n) * tol
68
69      # Indice de las iteraciones
70      i=1
71
72      # Residuos
73      res_p = sp.inf
74      res_d = sp.inf
75
76      mu_changed = 0
77
78      # BUCLE PRINCIPAL
79
80      while (i <= iters) and ((abs(res_p) > tol) or (abs(res_d) > tol)):
81
82          # Guardar u_{k} para calcular posteriormente los residuos
83          if (i%10) == 1:
84              u0 = u
85
86          # u_{k+1}
87
88          u = sp.maximum(x - d,0)
89
90          # x_{k+1}
91
92          x = sp.dot(IB1,yy + mu*(u+d)) + Aux
93
94          # d_{k+1}
95
96          d = d - (x - u)
97
98          # Actualizar mu para mantener los residuos bajo un factor de 10
99          if (i%10) == 1:
100
101             # Residuo primal
102             res_p = splin.norm(x - u)
103
104             # Residuo dual
105             res_d = mu * splin.norm(u - u0)
106
107             # Actualizar mu
108             if res_p > 10*res_d:
109                 mu = mu*2
110                 d = d/2
111                 mu_changed = True
112             elif res_d > 10*res_p:
113                 mu = mu/2
114                 d = d*2
115                 mu_changed = True
116
```

```
            # Ajustar IB, Aux, IB1 al nuevo mu
            if  mu_changed:
                # Actualizar IB e IB1
                IB = sp.dot( sp.dot(U,sp.diag(1./(S+mu))) , U.T )
                C = (1./IB.sum()) * sp.sum(IB,axis=1,keepdims=True)
                Aux = sp.sum(C,axis=1,keepdims=True)
                IB1 = IB - sp.dot(Aux,sp.sum(IB,axis=0,keepdims=True))
                mu_changed = False

        i = i + 1

    return x, res_p, res_d, i
```

9.4.2. Kernel C++ acelerado en *Vitis HLS* del algoritmo SUnSAL FCLS

```
// Vitis Data Structure (hls::stream)
#include <hls_stream.h>
// Vitis Libraries Solver (svd)
#include <hw/svd.hpp>

#define K 500
#define N 500
#define M 10

const unsigned int c_M = M;

static void load_input_A(float* A, hls::stream<float>& A_stream) {
  load_input_A_loop: for (int i = 0; i < K*N; i++)
    A_stream << A[i];
}

static void load_input_y(float* y, hls::stream<float>& y_stream) {
  load_input_y_loop: for (int i = 0; i < K; i++)
    y_stream << y[i];
}

static void store_result_x(float* x, hls::stream<float>& x_stream) {
  store_result_x_loop: for (int i = 0; i < N + 5*N*N; i++)
    x[i] = x_stream.read();
}

static void traspuesta_KporN(float A[K][N], float R[N][K]) {
  traspuesta_KporN_outer_loop: for(int i = 0; i < N; i++)
    traspuesta_KpoN_inner_loop: for(int j = 0; j < K; j++)
      R[i][j] = A[j][i];
}

static void traspuesta_NporN(float U[N][N], float R[N][N]) {
  traspuesta_NporN_outer_loop: for(int i = 0; i < N; i++)
    traspuesta_NporN_inner_loop: for(int j = 0; j < N; j++)
      R[i][j] = U[j][i];
}

static void producto_vector_NporK(float A[N][K], float y[K], float R[N]) {
  producto_vector_NporK_outer_loop: for(int i = 0; i < N; i++) {
    float Ri = 0;
    producto_vector_NporK_inner_loop: for(int j = 0; j < K; j++) {
      Ri += A[i][j] * y[j];
    }
    R[i] = Ri;
  }
}
```

```
48 static void producto_NporKporN(float A[N][K], float B[K][N], float R[N][N]) {
49   // #pragma HLS array_reshape variable=A complete dim=2
50   // #pragma HLS array_reshape variable=B complete dim=1
51   producto_NporKporN_outer_loop: for (int i = 0; i < N; i++) {
52     producto_NporKporN_inner_loop: for (int j = 0; j < N; j++) {
53       // #pragma HLS pipeline II=1
54       float Rij = 0;
55       producto_NporKporN_innermost_loop: for (int k = 0; k < K; k++) {
56         Rij += A[i][k] * B[k][j];
57       }
58       R[i][j] = Rij;
59     }
60   }
61 }
62
63 static void producto_NporNporK(float A[N][N], float B[N][K], float R[N][K]) {
64   // #pragma HLS array_reshape variable=A complete dim=2
65   // #pragma HLS array_reshape variable=B complete dim=1
66   producto_NporNporK_outer_loop: for (int i = 0; i < N; i++) {
67     producto_NporNporK_inner_loop: for (int j = 0; j < K; j++) {
68       // #pragma HLS pipeline II=1
69       float Rij = 0;
70       producto_NporNporK_innermost_loop: for (int k = 0; k < N; k++) {
71         Rij += A[i][k] * B[k][j];
72       }
73       R[i][j] = Rij;
74     }
75   }
76 }
77
78 static void producto_NporNporN(float A[N][N], float B[N][N], float R[N][N]) {
79   #pragma HLS array_reshape variable=A complete dim=2
80   #pragma HLS array_reshape variable=B complete dim=1
81   producto_NporNporN_outer_loop: for (int i = 0; i < N; i++) {
82     producto_NporNporN_inner_loop: for (int j = 0; j < N; j++) {
83       #pragma HLS pipeline II=1
84       float Rij = 0;
85       producto_NporNporN_innermost_loop: for (int k = 0; k < N; k++) {
86         Rij += A[i][k] * B[k][j];
87       }
88       R[i][j] = Rij;
89     }
90   }
91 }
92
93 static void reescalado(float A[K][N], float y[K]) {
94   float norm_f = 0;
95   norma_frobenius_outer_loop: for(int i = 0; i < K; i++) {
96     norma_frobenius_inner_loop: for(int j = 0; j < N; j++) {
97       float Aij = A[i][j]; norm_f += Aij * Aij;
98     }
99   }
100  norm_f = hls::sqrt(norm_f);
101  norm_f = norm_f * (25 + N) / N;
102  reajuste_A_outer_loop: for(int i = 0; i < K; i++) {
103    reajuste_A_inner_loop: for(int j = 0; j < N; j++)
104      A[i][j] = A[i][j] / norm_f;
105  }
106  reajuste_y_loop: for(int i = 0; i < K; i++) {
107    y[i] = y[i] / norm_f;
108  }
109 }
110
```

```
111 static void calcular_IB1(float U[N][N], float Ut[N][N], float S_0[N][N],
112       float mu, float C[N], float IB[N][N], float IB1[N][N]) {
113   // #pragma HLS expression_balance
114   float S[N][N] = {{}}, SUt[N][N] = {{}}, Cspc[N][N] = {{}};
115   float suma_por_filas[N] = {}, suma_por_columnas[N] = {};
116   float suma_total_IB = 0;
117   calculo_S_loop: for(int i = 0; i < N; i++)
118     S[i][i] = 1 / (S_0[i][i] + mu);
119   producto_NporNporN(S, Ut, SUt);
120   producto_NporNporN(U, SUt, IB);
121   sumas_outer_loop: for(int i = 0; i < N; i++) {
122     sumas_inner_loop: for(int j = 0; j < N; j++) {
123       suma_por_filas[i] += IB[i][j];
124       suma_por_columnas[i] += IB[j][i];
125     }
126   }
127   suma_total_loop: for(int i = 0; i < N; i++)
128     suma_total_IB += suma_por_filas[i];
129   vector_C_loop: for(int i = 0; i < N; i++)
130     C[i] = (1/suma_total_IB) * suma_por_filas[i];
131   Cspc_outer_loop: for(int i = 0; i < N; i++)
132     Cspc_inner_loop: for(int j = 0; j < N; j++)
133       Cspc[i][j] = C[i] * suma_por_columnas[j];
134   IB1_outer_loop: for(int i = 0; i < N; i++)
135     IB1_inner_loop: for(int j = 0; j < N; j++)
136       IB1[i][j] = IB[i][j] - Cspc[i][j];
137 }
138
139 static void compute_SUNSAL(hls::stream<float>& A_stream,
140       hls::stream<float>& y_stream, hls::stream<float>& x_stream) {
141
142   float A[K][N];
143   float y[K];
144   float x_sol[N];
145
146   // LECTURA
147
148   lectura_A_outer_loop: for(int i = 0; i < K; i++)
149     lectura_A_inner_loop: for(int j = 0; j < N; j++)
150       A[i][j] = A_stream.read();
151
152   lectura_y_loop: for(int i = 0; i < K; i++)
153     y[i] = y_stream.read();
154
155   // REESCALADO
156
157   reescalado(A,y);
158
159   // CALCULO
160
161       float mu = 0.01;
162
163       bool mu_changed = false;
164
165       float const tol = hls::sqrt(N) * 0.0001;
166       float res_p = 1000;
167       float res_d = 1000;
168
169       // INICIALIZACION
170
171       float At[N][K] = {{}}, IBAt[N][K] = {{}}, AtA[N][N] = {{}};
172
173       traspuesta_KporN(A, At);
```

```
174
175       for(int i = 0; i < N; i++)
176         for(int j = 0; j < N; j++)
177           x_stream << A[i][j];
178
179       float At_aux[N*K], A_aux[K*N], AtA_aux[N*N];
180
181       producto_NporKporN(At, A, AtA);
182
183       float U[N][N], S_0[N][N], Utr[N][N], IB[N][N], IB1[N][N];
184       float C[N] = {}, Aty[N] = {};
185
186       producto_vector_NporK(At, y, Aty);
187
188       float v[N][N] = {{}};
189       xf::solver::svdTop<N, N, xf::solver::svdTraits<N, N, float, float>,
190             float, float>(AtA, S_0, U, v);
191
192       for(int i = 0; i < N; i++)
193         for(int j = 0; j < N; j++)
194           x_stream << U[i][j];
195
196       for(int i = 0; i < N; i++)
197         for(int j = 0; j < N; j++)
198           x_stream << S_0[i][j];
199
200       traspuesta_NporN(U,Utr);
201
202       for(int i = 0; i < N; i++)
203         for(int j = 0; j < N; j++)
204           x_stream << Utr[i][j];
205
206       calcular_IB1(U, Utr, S_0, mu, C, IB, IB1);
207
208       for(int i = 0; i < N; i++)
209         for(int j = 0; j < N; j++)
210           x_stream << IB[i][j];
211
212       // x
213
214       producto_NporNporK(IB, At, IBAt);
215       producto_vector_NporK(IBAt, y, x_sol);
216
217       // u, d
218
219       float u[N], d[N] = {};
220       inicializar_x_loop: for(int i = 0; i < N; i++) {
221         u[i] = x_sol[i];
222       }
223
224       float W[N] = {}, u_0[N] = {};
225
226       // ITERACIONES
227
228       int iters = 1;
229
230       main_loop: for (int iters = 1; iters < M; iters++) {
231         #pragma HLS loop_tripcount min = c_M max = c_M
232
233         if (!((res_d > tol) || (res_d < 0 - tol)
234               || (res_p > tol) || (res_p < 0 - tol))) {
235           break;
236         }
```

```
237
238         if ((iters % 10) == 1) {
239             residuo_u_loop: for(int i = 0; i < N; i++) {
240               u_0[i] = u[i];
241             }
242         }
243
244         // u = max(x - d, 0)
245
246         actualizar_u_loop: for (int i = 0; i < N; i++) {
247           float r = x_sol[i] - d[i];
248           if (r > 0)
249             u[i] = r;
250           else
251             u[i] = 0;
252         }
253
254         // x
255
256         w_loop: for(int i = 0; i < N; i++) {
257           W[i] = Aty[i] + mu * (u[i] + d[i]);
258         }
259         actualizar_x_outer_loop: for(int i = 0; i < N; i++) {
260           x_sol[i] = C[i];
261           actualizar_x_inner_loop: for(int j = 0; j < N; j++) {
262             x_sol[i] += IB1[i][j] * W[j];
263           }
264         }
265
266         // d = d - (x - u)
267
268         actualizar_d_loop: for (int i = 0; i < N; i++) {
269           d[i] = d[i] - (x_sol[i] - u[i]);
270         }
271
272         // control de los residuos (res_p y res_d)
273
274         if ((iters % 10) == 1) {
275
276           residuos: {
277             #pragma HLS loop_merge
278
279             // residuo primal (x - z)
280
281             res_p = 0;
282
283             res_p_loop: for(int i = 0; i < N; i++) {
284               res_p += (x_sol[i] - u[i]) * (x_sol[i] - u[i]);
285             }
286
287             res_p = hls::sqrt(res_p);
288
289             // residuo dual (u - u_0)
290
291             res_d = 0;
292
293             res_d_loop: for(int i = 0; i < N; i++) {
294               res_d += (u[i] - u_0[i]) * (u[i] - u_0[i]);
295             }
296
297             res_d = mu * hls::sqrt(res_d);
298
299           }
```

```
300
301           // actualizar mu
302
303           if (res_p > 10 * res_d) {
304             mu = mu * 2;
305             d_res_p_loop: for(int i = 0; i < N; i++) {
306               d[i] = d[i] / 2;
307             }
308             mu_changed = true;
309           }
310           else if (res_d > 10 * res_p) {
311             mu = mu / 2;
312             d_res_d_loop: for(int i = 0; i < N; i++) {
313               d[i] = d[i] * 2;
314             }
315             mu_changed = true;
316           }
317
318           if (mu_changed) {
319             calcular_IB1(U, Utr, S_0, mu, C, IB, IB1);
320             mu_changed = false;
321           }
322
323         }
324
325       }
326
327   // ESCRITURA
328
329   escritura_x_loop: for(int i = 0; i < N; i++)
330     x_stream << x_sol[i];
331
332 }
333
334 extern "C" {
335
336   void krnl_SUNSAL(float* A, float* y, float* x) {
337
338     #pragma HLS INTERFACE m_axi port = A bundle = gmem0
339     #pragma HLS INTERFACE m_axi port = y bundle = gmem1
340     #pragma HLS INTERFACE m_axi port = x bundle = gmem2
341
342     static hls::stream<float> A_stream("input_A");
343     static hls::stream<float> y_stream("input_y");
344     static hls::stream<float> x_stream("output_x");
345
346     #pragma HLS dataflow
347
348     load_input_A(A, A_stream);
349
350     load_input_y(y, y_stream);
351
352     compute_SUNSAL(A_stream, y_stream, x_stream);
353
354     store_result_x(x, x_stream);
355
356   }
357
358 }
```

9.4.3. Reportes de síntesis

Reporte de síntesis inicial para K = 500, N = 500 y M = 100 sin ningún tipo de aceleración por hardware con directivas de aceleración.

```
================================================================
Design Name:             krnl_SUNSAL
Target Device:           xilinx:u250:gen3x16_xdma_4_1:202210.1
Target Clock:            300.000000MHz
Total number of kernels: 1
================================================================

Kernel Summary
Kernel Name  Type  Target              OpenCL Library  Compute Units
-----------  ----  ------------------  --------------  -------------
krnl_SUNSAL  c     fpga0:OCL_REGION_0  krnl_SUNSAL     1

================================================================
OpenCL Binary:     krnl_SUNSAL
Kernels mapped to: clc_region

Timing Information MHz
Compute Unit   Kernel Name  Module Name                                                                    Target Frequency  Estimated Frequency
-------------  -----------  -----------------------------------------------------------------------------  ----------------  -------------------
krnl_SUNSAL_1  krnl_SUNSAL  krnl_SUNSAL_Pipeline_load_input_A_loop                                          300.300293        411.015198
krnl_SUNSAL_1  krnl_SUNSAL  krnl_SUNSAL_Pipeline_load_input_y_loop                                          300.300293        411.015198
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_lectura_A_outer_loop_lectura_A_inner_loop             300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_lectura_y_loop                                         300.300293        414.421875
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_norma_frobenius_outer_loop_norma_frobenius_inner_loop  300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_reajuste_A_outer_loop_reajuste_A_inner_loop           300.300293        429.737885
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_reajuste_y_loop                                        300.300293        429.737885
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_6                                                    300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_7                                                    300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_8                                                    300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_traspuesta_KporN_outer_loop_traspuesta_KpoN_inner_loop  300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_VITIS_LOOP_166_1_VITIS_LOOP_167_2                    300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_producto_NporKporN_outer_loop_producto_NporKporN_inner_l  300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_13                                                   300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i_1  300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_15                                                   300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  calc_angle_float_float_s                                                     300.300293        382.701874
krnl_SUNSAL_1  krnl_SUNSAL  svdBasic_500_500_svdTraits_float_float_Pipeline_off_col                      300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  svdBasic_500_500_svdTraits_float_float_Pipeline_off_row                      300.300293        376.931763
krnl_SUNSAL_1  krnl_SUNSAL  svdBasic_500_500_svdTraits_500_500_float_float_float_float_s                 300.300293        376.931763
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_VITIS_LOOP_183_3_VITIS_LOOP_184_4                    300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_VITIS_LOOP_187_5_VITIS_LOOP_188_6                    300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_traspuesta_NporN_outer_loop_traspuesta_NporN_inner_loop  300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_VITIS_LOOP_193_7_VITIS_LOOP_194_8                    300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_1                                                       300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_2                                                       300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_3                                                       300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_4                                                       300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_5                                                       300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_calculo_S_loop                                          300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_producto_NporNporN_outer_loop_producto_NporNporN_inner_loo_1  300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_producto_NporNporN_outer_loop_producto_NporNporN_inner_loo  300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_sumas_inner_loop                                        300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_suma_total_loop                                        300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_vector_C_loop                                          300.300293        430.663208
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_Cspc_outer_loop_Cspc_inner_loop                        300.300293        430.663208
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_IB1_outer_loop_IB1_inner_loop                          300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1                                                                 300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_VITIS_LOOP_199_9_VITIS_LOOP_200_10                   300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_producto_NporNporK_outer_loop_producto_NporNporK_inner_l  300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i  300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_23                                                   300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_24                                                   300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_25                                                   300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_inicializar_x_loop                                   300.300293        418.060211
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_residuo_u_loop                                       300.300293        418.060211
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_actualizar_u_loop                                    300.300293        426.985474
```

krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_w_loop	300.300293	426.985474
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_x_inner_loop	300.300293	426.985474
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_d_loop	300.300293	426.985474
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_res_p_loop	300.300293	426.985474
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_res_d_loop	300.300293	426.985474
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_d_loop	300.300293	430.663208
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_p_loop	300.300293	430.663208
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_escritura_x_loop	300.300293	414.421875
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL	300.300293	376.931763
krnl_SUNSAL_1	krnl_SUNSAL	krnl_SUNSAL_Pipeline_store_result_x_loop	300.300293	411.015198
krnl_SUNSAL_1	krnl_SUNSAL	krnl_SUNSAL	300.300293	376.931763

Latency Information

Compute Unit	Kernel Name	Module Name	Start Interval	Best cycles	Avg cycles	Worst cycles	Best absolute	Avg absolute	Worst absolute
krnl_SUNSAL_1	krnl_SUNSAL	krnl_SUNSAL_Pipeline_load_input_A_loop	250003	250003	250003	250003	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	krnl_SUNSAL_Pipeline_load_input_y_loop	503	503	503	503	1.676 us	1.676 us	1.676 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_lectura_A_outer_loop_lectura_A_inner_loop	250004	250004	250004	250004	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_lectura_y_loop	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_norma_frobenius_outer_loop_norma_frobenius_inner_loop	1750011	1750011	1750011	1750011	5.833 ms	5.833 ms	5.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_reajuste_A_outer_loop_reajuste_A_inner_loop	250018	250018	250018	250018	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_reajuste_y_loop	515	515	515	515	1.716 us	1.716 us	1.716 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_6	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_7	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_8	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_traspuesta_KporN_outer_loop_traspuesta_KpoN_inner_loop	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_166_1_VITIS_LOOP_167_2	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_NporKporN_outer_loop_producto_NporKporN_inner_l	1000000010	1000000010	1000000010	1000000010	3.333 sec	3.333 sec	3.333 sec
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_12	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_13	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i_1	2000010	2000010	2000010	2000010	6.666 ms	6.666 ms	6.666 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_15	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calc_angle_float_float_s	2 ~ 99	2	67	99	6.666 ns	0.223 us	0.330 us
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_float_float_Pipeline_off_col	4011	4011	4011	4011	13.369 us	13.369 us	13.369 us
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_float_float_Pipeline_off_row	4012	4012	4012	4012	13.372 us	13.372 us	13.372 us
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_500_500_float_float_float_float_s	40414011 ~ 20398191861	40414011	undef	20398191861	0.135 sec	undef	67.987 sec
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_183_3_VITIS_LOOP_184_4	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_187_5_VITIS_LOOP_188_6	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_traspuesta_NporN_outer_loop_traspuesta_NporN_inner_loop	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_193_7_VITIS_LOOP_194_8	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_1	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_2	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_3	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_4	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_5	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_calculo_S_loop	522	522	522	522	1.740 us	1.740 us	1.740 us
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_producto_NporNporN_outer_loop_producto_NporNporN_inner_loo_1	1000000010	1000000010	1000000010	1000000010	3.333 sec	3.333 sec	3.333 sec
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_producto_NporNporN_outer_loop_producto_NporNporN_inner_loo	1000000010	1000000010	1000000010	1000000010	3.333 sec	3.333 sec	3.333 sec
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_sumas_inner_loop	3505	3505	3505	3505	11.682 us	11.682 us	11.682 us
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_suma_total_loop	3504	3504	3504	3504	11.679 us	11.679 us	11.679 us
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_vector_C_loop	507	507	507	507	1.690 us	1.690 us	1.690 us
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_Cspc_outer_loop_Cspc_inner_loop	250007	250007	250007	250007	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_IB1_outer_loop_IB1_inner_loop	250013	250013	250013	250013	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1	2002509101	2002509101	2002509101	2002509101	6.674 sec	6.674 sec	6.674 sec
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_199_9_VITIS_LOOP_200_10	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_NporNporK_outer_loop_producto_NporNporK_inner_l	1000000010	1000000010	1000000010	1000000010	3.333 sec	3.333 sec	3.333 sec
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i	2000010	2000010	2000010	2000010	6.666 ms	6.666 ms	6.666 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_23	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_24	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_25	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_inicializar_x_loop	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_residuo_u_loop	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_u_loop	512	512	512	512	1.706 us	1.706 us	1.706 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_w_loop	521	521	521	521	1.736 us	1.736 us	1.736 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_x_inner_loop	3508	3508	3508	3508	11.692 us	11.692 us	11.692 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_d_loop	517	517	517	517	1.723 us	1.723 us	1.723 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_res_p_loop	3515	3515	3515	3515	11.715 us	11.715 us	11.715 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_res_d_loop	3515	3515	3515	3515	11.715 us	11.715 us	11.715 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_d_loop	507	507	507	507	1.690 us	1.690 us	1.690 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_p_loop	507	507	507	507	1.690 us	1.690 us	1.690 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_escritura_x_loop	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL	4224681788 ~ 224834177140	4224681788	undef	224834177140	14.081 sec	undef	749.372 sec
krnl_SUNSAL_1	krnl_SUNSAL	krnl_SUNSAL_Pipeline_store_result_x_loop	1250503	1250503	1250503	1250503	4.168 ms	4.168 ms	4.168 ms
krnl_SUNSAL_1	krnl_SUNSAL	krnl_SUNSAL	4226183014 ~ 224835678366	4226183013	undef	224835678365	14.086 sec	undef	749.377 sec

Area Information

Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM	URAM
krnl_SUNSAL_1	krnl_SUNSAL	krnl_SUNSAL_Pipeline_load_input_A_loop	1019	140	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	krnl_SUNSAL_Pipeline_load_input_y_loop	242	128	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_lectura_A_outer_loop_lectura_A_inner_loop	94	172	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_lectura_y_loop	21	82	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_norma_frobenius_outer_loop_norma_frobenius_inner_loop	187	240	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_reajuste_A_outer_loop_reajuste_A_inner_loop	236	193	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_reajuste_y_loop	175	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_6	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_7	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_8	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_traspuesta_KporN_outer_loop_traspuesta_KpoN_inner_loop	152	161	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_166_1_VITIS_LOOP_167_2	98	172	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_NporKporN_outer_loop_producto_NporKporN_inner_l	418	528	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_12	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_13	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i_1	272	289	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_15	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calc_angle_float_float_s	1405	2164	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_float_float_Pipeline_off_col	504	476	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_float_float_Pipeline_off_row	1410	971	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_500_500_float_float_float_float_s	9258	10072	0	48	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_183_3_VITIS_LOOP_184_4	98	172	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_187_5_VITIS_LOOP_188_6	98	172	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_traspuesta_NporN_outer_loop_traspuesta_NporN_inner_loop	152	161	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_193_7_VITIS_LOOP_194_8	98	172	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_1	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_2	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_3	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_4	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_5	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_calculo_S_loop	248	146	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_producto_NporNporN_outer_loop_producto_NporNporN_inner_loo_1	418	528	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_producto_NporNporN_outer_loop_producto_NporNporN_inner_loo	418	528	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_sumas_inner_loop	263	300	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_suma_total_loop	116	150	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_vector_C_loop	159	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_Cspc_outer_loop_Cspc_inner_loop	246	193	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_IB1_outer_loop_IB1_inner_loop	258	193	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1	2862	3705	0	50	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_199_9_VITIS_LOOP_200_10	98	172	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_NporNporK_outer_loop_producto_NporNporK_inner_l	418	528	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i	272	289	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_23	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_24	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_25	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_inicializar_x_loop	21	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_residuo_u_loop	21	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_u_loop	265	166	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_w_loop	315	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_x_inner_loop	209	177	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_d_loop	275	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_res_p_loop	183	183	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_res_d_loop	183	183	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_d_loop	159	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_p_loop	159	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_escritura_x_loop	12	82	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL	18550	23259	0	266	0
krnl_SUNSAL_1	krnl_SUNSAL	krnl_SUNSAL_Pipeline_store_result_x_loop	157	222	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	krnl_SUNSAL	28281	38957	0	266	0

Reporte de síntesis intermedio para K = 500, N = 500 y M = 100 con una única directiva de aceleración *dataflow*.

```
--------------------------------------------------------------------------------
Design Name:             krnl_SUNSAL
Target Device:           xilinx:u250:gen3x16_xdma_4_1:202210.1
Target Clock:            300.000000MHz
Total number of kernels: 1
--------------------------------------------------------------------------------

Kernel Summary
Kernel Name  Type  Target              OpenCL Library  Compute Units
-----------  ----  ------------------  --------------  -------------
krnl_SUNSAL  c     fpga0:OCL_REGION_0  krnl_SUNSAL     1

--------------------------------------------------------------------------------
OpenCL Binary:     krnl_SUNSAL
Kernels mapped to: clc_region

Timing Information MHz
Compute Unit   Kernel Name  Module Name                                                                                  Target Frequency  Estimated Frequency
-------------  -----------  -------------------------------------------------------------------------------------------  ----------------  -------------------
krnl_SUNSAL_1  krnl_SUNSAL  entry_proc                                                                                    300.300293        821.692688
krnl_SUNSAL_1  krnl_SUNSAL  load_input_A_Pipeline_load_input_A_loop                                                      300.300293        411.015198
krnl_SUNSAL_1  krnl_SUNSAL  load_input_A                                                                                 300.300293        411.015198
krnl_SUNSAL_1  krnl_SUNSAL  load_input_y_Pipeline_load_input_y_loop                                                      300.300293        411.015198
krnl_SUNSAL_1  krnl_SUNSAL  load_input_y                                                                                 300.300293        411.015198
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_lectura_A_outer_loop_lectura_A_inner_loop                           300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_lectura_y_loop                                                      300.300293        414.421875
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_norma_frobenius_outer_loop_norma_frobenius_inner_loop              300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_reajuste_A_outer_loop_reajuste_A_inner_loop                         300.300293        429.737885
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_reajuste_y_loop                                                      300.300293        429.737885
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_6                                                                   300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_7                                                                   300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_8                                                                   300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_traspuesta_KporN_outer_loop_traspuesta_KpoN_inner_loop             300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_VITIS_LOOP_166_1_VITIS_LOOP_167_2                                  300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_producto_NporKporN_outer_loop_producto_NporKporN_inner_l           300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_12                                                                  300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_13                                                                  300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i_1        300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_15                                                                  300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  calc_angle_float_float_s                                                                    300.300293        382.701874
krnl_SUNSAL_1  krnl_SUNSAL  svdBasic_500_500_svdTraits_float_float_Pipeline_off_col                                     300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  svdBasic_500_500_svdTraits_float_float_Pipeline_off_row                                     300.300293        376.931763
krnl_SUNSAL_1  krnl_SUNSAL  svdBasic_500_500_svdTraits_500_500_float_float_float_float_s                               300.300293        376.931763
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_VITIS_LOOP_183_3_VITIS_LOOP_184_4                                  300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_VITIS_LOOP_187_5_VITIS_LOOP_188_6                                  300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_traspuesta_NporN_outer_loop_traspuesta_NporN_inner_loop           300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_VITIS_LOOP_193_7_VITIS_LOOP_194_8                                  300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_1                                                                     300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_2                                                                     300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_3                                                                     300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_4                                                                     300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_5                                                                     300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_calculo_S_loop                                                        300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_producto_NporNporN_outer_loop_producto_NporNporN_inner_loo_1        300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_producto_NporNporN_outer_loop_producto_NporNporN_inner_loo          300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_sumas_inner_loop                                                      300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_suma_total_loop                                                       300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_vector_C_loop                                                         300.300293        430.663208
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_Cspc_outer_loop_Cspc_inner_loop                                       300.300293        430.663208
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_IB1_outer_loop_IB1_inner_loop                                         300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1                                                                                300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_VITIS_LOOP_199_9_VITIS_LOOP_200_10                                 300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_producto_NporNporK_outer_loop_producto_NporNporK_inner_l          300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i          300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_23                                                                  300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_24                                                                  300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_25                                                                  300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_inicializar_x_loop                                                  300.300293        418.060211
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_residuo_u_loop                                                      300.300293        418.060211
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_actualizar_u_loop                                                   300.300293        426.985474
```

krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_w_loop	300.300293	426.985474
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_x_inner_loop	300.300293	426.985474
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_d_loop	300.300293	426.985474
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_res_p_loop	300.300293	426.985474
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_res_d_loop	300.300293	426.985474
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_d_loop	300.300293	430.663208
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_p_loop	300.300293	430.663208
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_escritura_x_loop	300.300293	414.421875
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL	300.300293	376.931763
krnl_SUNSAL_1	krnl_SUNSAL	store_result_x_Pipeline_store_result_x_loop	300.300293	411.015198
krnl_SUNSAL_1	krnl_SUNSAL	store_result_x	300.300293	411.015198
krnl_SUNSAL_1	krnl_SUNSAL	krnl_SUNSAL	300.300293	376.931763

Latency Information

Compute Unit	Kernel Name	Module Name	Start Interval	Best cycles	Avg cycles	Worst cycles	Best absolute	Avg absolute	Worst absolute
krnl_SUNSAL_1	krnl_SUNSAL	entry_proc	0	0	0	0	0 ns	0 ns	0 ns
krnl_SUNSAL_1	krnl_SUNSAL	load_input_A_Pipeline_load_input_A_loop	250003	250003	250003	250003	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	load_input_A	250074	250074	250074	250074	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	load_input_y_Pipeline_load_input_y_loop	503	503	503	503	1.676 us	1.676 us	1.676 us
krnl_SUNSAL_1	krnl_SUNSAL	load_input_y	574	574	574	574	1.913 us	1.913 us	1.913 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_lectura_A_outer_loop_lectura_A_inner_loop	250004	250004	250004	250004	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_lectura_y_loop	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_norma_frobenius_outer_loop_norma_frobenius_inner_loop	1750011	1750011	1750011	1750011	5.833 ms	5.833 ms	5.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_reajuste_A_outer_loop_reajuste_A_inner_loop	250018	250018	250018	250018	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_reajuste_y_loop	515	515	515	515	1.716 us	1.716 us	1.716 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_6	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_7	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_8	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_traspuesta_KporN_outer_loop_traspuesta_KpoN_inner_loop	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_166_1_VITIS_LOOP_167_2	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_NporKporN_outer_loop_producto_NporKporN_inner_l	1000000010	1000000010	1000000010	1000000010	3.333 sec	3.333 sec	3.333 sec
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_12	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_13	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i_1	2000010	2000010	2000010	2000010	6.666 ms	6.666 ms	6.666 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_15	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calc_angle_float_float_s	2 ~ 99	2	67	99	6.666 ns	0.223 us	0.330 us
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_float_float_Pipeline_off_col	4011	4011	4011	4011	13.369 us	13.369 us	13.369 us
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_float_float_Pipeline_off_row	4012	4012	4012	4012	13.372 us	13.372 us	13.372 us
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_500_500_float_float_float_float_s	40414011 ~ 20398191861	40414011	undef	20398191861	0.135 sec	undef	67.987 sec
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_183_3_VITIS_LOOP_184_4	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_187_5_VITIS_LOOP_188_6	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_traspuesta_NporN_outer_loop_traspuesta_NporN_inner_loop	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_193_7_VITIS_LOOP_194_8	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_calculo_S_loop	522	522	522	522	1.740 us	1.740 us	1.740 us
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_producto_NporNporN_outer_loop_producto_NporNporN_inner_loo_1	1000000010	1000000010	1000000010	1000000010	3.333 sec	3.333 sec	3.333 sec
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_producto_NporNporN_outer_loop_producto_NporNporN_inner_loo	1000000010	1000000010	1000000010	1000000010	3.333 sec	3.333 sec	3.333 sec
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_sumas_inner_loop	3505	3505	3505	3505	11.682 us	11.682 us	11.682 us
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_suma_total_loop	3504	3504	3504	3504	11.679 us	11.679 us	11.679 us
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_vector_C_loop	507	507	507	507	1.690 us	1.690 us	1.690 us
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_Cspc_outer_loop_Cspc_inner_loop	250007	250007	250007	250007	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_IB1_outer_loop_IB1_inner_loop	250013	250013	250013	250013	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1	2002509101	2002509101	2002509101	2002509101	6.674 sec	6.674 sec	6.674 sec
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_199_9_VITIS_LOOP_200_10	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_NporNporK_outer_loop_producto_NporNporK_inner_l	1000000010	1000000010	1000000010	1000000010	3.333 sec	3.333 sec	3.333 sec
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i	2000010	2000010	2000010	2000010	6.666 ms	6.666 ms	6.666 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_23	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_inicializar_x_loop	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_residuo_u_loop	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_u_loop	512	512	512	512	1.706 us	1.706 us	1.706 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_w_loop	521	521	521	521	1.736 us	1.736 us	1.736 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_x_inner_loop	3508	3508	3508	3508	11.692 us	11.692 us	11.692 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_d_loop	517	517	517	517	1.723 us	1.723 us	1.723 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_res_p_loop	3515	3515	3515	3515	11.715 us	11.715 us	11.715 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_res_d_loop	3515	3515	3515	3515	11.715 us	11.715 us	11.715 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_d_loop	507	507	507	507	1.690 us	1.690 us	1.690 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_p_loop	507	507	507	507	1.690 us	1.690 us	1.690 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_escritura_x_loop	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL	4224681788 ~ 224834177140	4224681788	undef	224834177140	14.081 sec	undef	749.372 sec
krnl_SUNSAL_1	krnl_SUNSAL	store_result_x_Pipeline_store_result_x_loop	1250503	1250503	1250503	1250503	4.168 ms	4.168 ms	4.168 ms
krnl_SUNSAL_1	krnl_SUNSAL	store_result_x	1250574	1250574	1250574	1250574	4.168 ms	4.168 ms	4.168 ms
krnl_SUNSAL_1	krnl_SUNSAL	krnl_SUNSAL	4224681789 ~ 224834177141	1250709	undef	1495877819	4.169 ms	undef	4.986 sec

Area Information

Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM	URAM
krnl_SUNSAL_1	krnl_SUNSAL	entry_proc	3	37	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	load_input_A_Pipeline_load_input_A_loop	1019	140	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	load_input_A	1152	649	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	load_input_y_Pipeline_load_input_y_loop	242	128	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	load_input_y	376	628	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_lectura_A_outer_loop_lectura_A_inner_loop	94	172	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_lectura_y_loop	21	82	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_norma_frobenius_outer_loop_norma_frobenius_inner_loop	187	240	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_reajuste_A_outer_loop_reajuste_A_inner_loop	236	193	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_reajuste_y_loop	175	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_6	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_7	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_8	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_traspuesta_KporN_outer_loop_traspuesta_KpoN_inner_loop	152	161	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_166_1_VITIS_LOOP_167_2	98	172	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_NporKporN_outer_loop_producto_NporKporN_inner_l	418	528	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_12	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_13	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i_1	272	289	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_15	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calc_angle_float_float_s	1405	2164	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_float_float_Pipeline_off_col	504	476	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_float_float_Pipeline_off_row	1410	971	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_500_500_float_float_float_float_s	9290	10072	0	48	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_183_3_VITIS_LOOP_184_4	98	172	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_187_5_VITIS_LOOP_188_6	98	172	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_traspuesta_NporN_outer_loop_traspuesta_NporN_inner_loop	152	161	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_193_7_VITIS_LOOP_194_8	98	172	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_1	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_2	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_3	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_4	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_5	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_calculo_S_loop	248	146	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_producto_NporNporN_outer_loop_producto_NporNporN_inner_loo_1	418	528	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_producto_NporNporN_outer_loop_producto_NporNporN_inner_loo	418	528	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_sumas_inner_loop	263	300	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_suma_total_loop	116	150	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_vector_C_loop	159	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_Cspc_outer_loop_Cspc_inner_loop	246	193	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_IB1_outer_loop_IB1_inner_loop	258	193	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1	2862	3705	0	50	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_199_9_VITIS_LOOP_200_10	98	172	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_NporNporK_outer_loop_producto_NporNporK_inner_l	418	528	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i	272	289	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_23	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_24	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_25	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_inicializar_x_loop	21	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_residuo_u_loop	21	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_u_loop	265	166	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_w_loop	315	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_x_inner_loop	209	177	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_d_loop	275	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_res_p_loop	183	183	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_res_d_loop	183	183	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_d_loop	159	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_p_loop	159	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_escritura_x_loop	12	82	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL	18583	23270	0	266	0
krnl_SUNSAL_1	krnl_SUNSAL	store_result_x_Pipeline_store_result_x_loop	157	222	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	store_result_x	291	756	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	krnl_SUNSAL	28326	39212	0	266	0

Reporte de síntesis final para K = 500, N = 500 y M = 100 con aceleración por hardware para el producto de matrices, técnicas para evitar paradas de *pipeline* y uso de la directiva de aceleración *dataflow*.

```
--------------------------------------------------------------------------------
Design Name:             krnl_SUNSAL
Target Device:           xilinx:u250:gen3x16_xdma_4_1:202210.1
Target Clock:            300.000000MHz
Total number of kernels: 1
--------------------------------------------------------------------------------

Kernel Summary
Kernel Name  Type  Target              OpenCL Library  Compute Units
-----------  ----  ------------------  --------------  -------------
krnl_SUNSAL  c     fpga0:OCL_REGION_0  krnl_SUNSAL     1

--------------------------------------------------------------------------------
OpenCL Binary:     krnl_SUNSAL
Kernels mapped to: clc_region

Timing Information MHz
Compute Unit   Kernel Name  Module Name                                                                                  Target Frequency  Estimated Frequency
-------------  -----------  -------------------------------------------------------------------------------------------  ----------------  -------------------
krnl_SUNSAL_1  krnl_SUNSAL  entry_proc                                                                                    300.300293        821.692688
krnl_SUNSAL_1  krnl_SUNSAL  load_input_A_Pipeline_load_input_A_loop                                                      300.300293        411.015198
krnl_SUNSAL_1  krnl_SUNSAL  load_input_A                                                                                 300.300293        411.015198
krnl_SUNSAL_1  krnl_SUNSAL  load_input_y_Pipeline_load_input_y_loop                                                      300.300293        411.015198
krnl_SUNSAL_1  krnl_SUNSAL  load_input_y                                                                                 300.300293        411.015198
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_lectura_A_outer_loop_lectura_A_inner_loop                           300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_lectura_y_loop                                                      300.300293        414.421875
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_norma_frobenius_outer_loop_norma_frobenius_inner_loop             300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_reajuste_A_outer_loop_reajuste_A_inner_loop                       300.300293        429.737885
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_reajuste_y_loop                                                    300.300293        429.737885
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_6                                                                300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_7                                                                300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_8                                                                300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_traspuesta_KporN_outer_loop_traspuesta_KpoN_inner_loop           300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_VITIS_LOOP_166_1_VITIS_LOOP_167_2                                 300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_producto_NporKporN_outer_loop_producto_NporKporN_inner_l          300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_12                                                               300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_13                                                               300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i_1      300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_15                                                               300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  calc_angle_float_float_s                                                                 300.300293        382.701874
krnl_SUNSAL_1  krnl_SUNSAL  svdBasic_500_500_svdTraits_float_float_Pipeline_off_col                                  300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  svdBasic_500_500_svdTraits_float_float_Pipeline_off_row                                  300.300293        376.931763
krnl_SUNSAL_1  krnl_SUNSAL  svdBasic_500_500_svdTraits_500_500_float_float_float_float_s                             300.300293        376.931763
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_VITIS_LOOP_183_3_VITIS_LOOP_184_4                                300.300293        450.856628
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_VITIS_LOOP_187_5_VITIS_LOOP_188_6                                300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_traspuesta_NporN_outer_loop_traspuesta_NporN_inner_loop         300.300293        477.099243
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_VITIS_LOOP_193_7_VITIS_LOOP_194_8                                300.300293        477.099243
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_1                                                                  300.300293        459.558807
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_2                                                                  300.300293        459.558807
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_3                                                                  300.300293        513.610657
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_4                                                                  300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_5                                                                  300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_calculo_S_loop                                                     300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  producto_NporNporN                                                                       300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  producto_NporNporN_1                                                                     300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_sumas_inner_loop                                                   300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_suma_total_loop                                                    300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_vector_C_loop                                                      300.300293        430.663208
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_Cspc_outer_loop_Cspc_inner_loop                                    300.300293        430.663208
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1_Pipeline_IB1_outer_loop_IB1_inner_loop                                      300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  calcular_IB1                                                                             300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_VITIS_LOOP_199_9_VITIS_LOOP_200_10                              300.300293        496.524353
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_producto_NporNporK_outer_loop_producto_NporNporK_inner_l       300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i       300.300293        426.985474
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_23                                                               300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_24                                                               300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_25                                                               300.300293        558.971497
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_inicializar_x_loop                                               300.300293        418.060211
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_residuo_u_loop                                                   300.300293        418.060211
krnl_SUNSAL_1  krnl_SUNSAL  compute_SUNSAL_Pipeline_actualizar_u_loop                                                300.300293        426.985474
```

krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_w_loop	300.300293	426.985474
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_x_inner_loop	300.300293	426.985474
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_d_loop	300.300293	426.985474
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_res_d_loop	300.300293	426.985474
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_d_loop	300.300293	430.663208
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_p_loop	300.300293	430.663208
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_escritura_x_loop	300.300293	414.421875
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL	300.300293	376.931763
krnl_SUNSAL_1	krnl_SUNSAL	store_result_x_Pipeline_store_result_x_loop	300.300293	411.015198
krnl_SUNSAL_1	krnl_SUNSAL	store_result_x	300.300293	411.015198
krnl_SUNSAL_1	krnl_SUNSAL	krnl_SUNSAL	300.300293	376.931763

Latency Information

Compute Unit	Kernel Name	Module Name	Start Interval	Best cycles	Avg cycles	Worst cycles	Best absolute	Avg absolute	Worst absolute
krnl_SUNSAL_1	krnl_SUNSAL	entry_proc	0	0	0	0	0 ns	0 ns	0 ns
krnl_SUNSAL_1	krnl_SUNSAL	load_input_A_Pipeline_load_input_A_loop	250003	250003	250003	250003	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	load_input_A	250074	250074	250074	250074	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	load_input_y_Pipeline_load_input_y_loop	503	503	503	503	1.676 us	1.676 us	1.676 us
krnl_SUNSAL_1	krnl_SUNSAL	load_input_y	574	574	574	574	1.913 us	1.913 us	1.913 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_lectura_A_outer_loop_lectura_A_inner_loop	250004	250004	250004	250004	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_lectura_y_loop	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_norma_frobenius_outer_loop_norma_frobenius_inner_loop	1750011	1750011	1750011	1750011	5.833 ms	5.833 ms	5.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_reajuste_A_outer_loop_reajuste_A_inner_loop	250018	250018	250018	250018	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_reajuste_y_loop	515	515	515	515	1.716 us	1.716 us	1.716 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_6	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_7	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_8	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_traspuesta_KporN_outer_loop_traspuesta_KpoN_inner_loop	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_166_1_VITIS_LOOP_167_2	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_NporKporN_outer_loop_producto_NporKporN_inner_l	1000000010	1000000010	1000000010	1000000010	3.333 sec	3.333 sec	3.333 sec
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_12	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_13	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i_1	2000010	2000010	2000010	2000010	6.666 ms	6.666 ms	6.666 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_15	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calc_angle_float_float_s	2 ~ 99	2	67	99	6.666 ns	0.223 us	0.330 us
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_float_float_Pipeline_off_col	4011	4011	4011	4011	13.369 us	13.369 us	13.369 us
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_float_float_Pipeline_off_row	4012	4012	4012	4012	13.372 us	13.372 us	13.372 us
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_500_500_float_float_float_float_s	40414011 ~ 20400681871	40414011	undef	20400681871	0.135 sec	undef	67.995 sec
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_183_3_VITIS_LOOP_184_4	250004	250004	250004	250004	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_187_5_VITIS_LOOP_188_6	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_traspuesta_NporN_outer_loop_traspuesta_NporN_inner_loop	250004	250004	250004	250004	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_193_7_VITIS_LOOP_194_8	250004	250004	250004	250004	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_1	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_2	250002	250002	250002	250002	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_5	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_calculo_S_loop	522	522	522	522	1.740 us	1.740 us	1.740 us
krnl_SUNSAL_1	krnl_SUNSAL	producto_NporNporN	253507	253507	253507	253507	0.845 ms	0.845 ms	0.845 ms
krnl_SUNSAL_1	krnl_SUNSAL	producto_NporNporN_1	253507	253507	253507	253507	0.845 ms	0.845 ms	0.845 ms
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_sumas_inner_loop	3505	3505	3505	3505	11.682 us	11.682 us	11.682 us
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_suma_total_loop	3504	3504	3504	3504	11.679 us	11.679 us	11.679 us
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_vector_C_loop	507	507	507	507	1.690 us	1.690 us	1.690 us
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_Cspc_outer_loop_Cspc_inner_loop	250007	250007	250007	250007	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_IB1_outer_loop_IB1_inner_loop	250013	250013	250013	250013	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1	3016095	3016095	3016095	3016095	10.053 ms	10.053 ms	10.053 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_199_9_VITIS_LOOP_200_10	250006	250006	250006	250006	0.833 ms	0.833 ms	0.833 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_NporNporK_outer_loop_producto_NporNporK_inner_l	1000000010	1000000010	1000000010	1000000010	3.333 sec	3.333 sec	3.333 sec
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i	2000010	2000010	2000010	2000010	6.666 ms	6.666 ms	6.666 ms
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_inicializar_x_loop	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_residuo_u_loop	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_u_loop	512	512	512	512	1.706 us	1.706 us	1.706 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_w_loop	521	521	521	521	1.736 us	1.736 us	1.736 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_x_inner_loop	3508	3508	3508	3508	11.692 us	11.692 us	11.692 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_d_loop	517	517	517	517	1.723 us	1.723 us	1.723 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_res_d_loop	3516	3516	3516	3516	11.719 us	11.719 us	11.719 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_d_loop	507	507	507	507	1.690 us	1.690 us	1.690 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_p_loop	507	507	507	507	1.690 us	1.690 us	1.690 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_escritura_x_loop	502	502	502	502	1.673 us	1.673 us	1.673 us
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL	2225188776 ~ 22887521938	2225188776	undef	22887521938	7.417 sec	undef	76.284 sec
krnl_SUNSAL_1	krnl_SUNSAL	store_result_x_Pipeline_store_result_x_loop	1250503	1250503	1250503	1250503	4.168 ms	4.168 ms	4.168 ms
krnl_SUNSAL_1	krnl_SUNSAL	store_result_x	1250574	1250574	1250574	1250574	4.168 ms	4.168 ms	4.168 ms
krnl_SUNSAL_1	krnl_SUNSAL	krnl_SUNSAL	2225188777 ~ 22887521939	1250709	undef	1412685529	4.169 ms	undef	4.708 sec

Area Information

Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM	URAM
krnl_SUNSAL_1	krnl_SUNSAL	entry_proc	3	37	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	load_input_A_Pipeline_load_input_A_loop	1019	140	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	load_input_A	1152	649	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	load_input_y_Pipeline_load_input_y_loop	242	128	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	load_input_y	376	628	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_lectura_A_outer_loop_lectura_A_inner_loop	94	172	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_lectura_y_loop	21	82	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_norma_frobenius_outer_loop_norma_frobenius_inner_loop	187	240	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_reajuste_A_outer_loop_reajuste_A_inner_loop	236	193	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_reajuste_y_loop	175	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_6	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_7	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_8	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_traspuesta_KporN_outer_loop_traspuesta_KpoN_inner_loop	152	161	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_166_1_VITIS_LOOP_167_2	98	172	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_NporKporN_outer_loop_producto_NporKporN_inner_l	418	528	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_12	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_13	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i_1	272	289	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_15	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calc_angle_float_float_s	1405	2164	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_float_float_Pipeline_off_col	504	476	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_float_float_Pipeline_off_row	33379	34231	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	svdBasic_500_500_svdTraits_500_500_float_float_float_float_s	73287	109825	0	48	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_183_3_VITIS_LOOP_184_4	16094	2343	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_187_5_VITIS_LOOP_188_6	98	172	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_traspuesta_NporN_outer_loop_traspuesta_NporN_inner_loop	32135	6724	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_193_7_VITIS_LOOP_194_8	16085	2352	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_1	75	2370	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_2	75	2370	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_3	20	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_4	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_5	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_calculo_S_loop	239	4497	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	producto_NporNporN	103134	36544	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	producto_NporNporN_1	103134	32193	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_sumas_inner_loop	263	300	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_suma_total_loop	116	150	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_vector_C_loop	159	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_Cspc_outer_loop_Cspc_inner_loop	246	193	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1_Pipeline_IB1_outer_loop_IB1_inner_loop	258	193	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	calcular_IB1	438434	259984	0	34	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_VITIS_LOOP_199_9_VITIS_LOOP_200_10	98	172	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_NporNporK_outer_loop_producto_NporNporK_inner_l	418	528	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_producto_vector_NporK_outer_loop_producto_vector_NporK_i	272	289	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_23	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_24	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_25	11	60	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_inicializar_x_loop	21	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_residuo_u_loop	21	71	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_u_loop	265	166	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_w_loop	315	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_x_inner_loop	209	177	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_actualizar_d_loop	275	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_res_d_loop	313	280	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_d_loop	159	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_d_res_p_loop	159	103	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL_Pipeline_escritura_x_loop	12	82	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	compute_SUNSAL	582062	390080	0	234	0
krnl_SUNSAL_1	krnl_SUNSAL	store_result_x_Pipeline_store_result_x_loop	157	222	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	store_result_x	291	756	0	0	0
krnl_SUNSAL_1	krnl_SUNSAL	krnl_SUNSAL	591805	406022	0	234	0

Bibliografía

[1] Ahmad, M., Shabbir, S., Roy, S. K., Hong, D., Wu, X., Yao, J., Mazzara, M., Distefano, S. y Chanussot, J. *Hyperspectral Image Classification - Traditional to Deep Models: A Survey for Future Prospects*, publicado en la revista *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 15, 2022, https://arxiv.org/pdf/2101.06116.pdf.

[2] Aisa Systems, *Specim. Aisa, Airborne Imaging Spectrometer for Applications*, 2020. Disponible en: https://www.specim.com/aisa/ (Accedido: 3 de Diciembre de 2022).

[3] Altavilla, D. (Autor) *Xilinx Unveils Vitis, Breakthrough Open-Source Design Software For Adaptable Processing Engines*, Forbes, 2019. Disponible en: https://www.forbes.com/sites/davealtavilla/2019/10/01/xilinx-unveils-vitis-disruptive-open-source-design-software-tools-for-adaptable-processing-engines/?sh=1fa99fc475ff (Accedido: 3 de Febrero de 2023).

[4] Alterini, T. *Hyperspectral imaging system for the fast recording of the ocular fundus*, tesis doctoral publicada por *Universidad Politécnica de Cataluña*, págs. 26-58, 2021, https://www.tesisenred.net/handle/10803/672138.

[5] Anigbogu, O. y Olanloye, O. D., *Processing of Hyperspectral Data using Wavelet Transform*, publicada como parte de *FUOYE Journal of Engineering and Technology*, vol. 3, nº 1, 2018, https://doi.org/10.46792/fuoyejet.v3i1.144.

[6] Basedow, R. W. y Zalewski, L. *Characteristics of the HYDICE sensor*, publicado en *JPL, Summaries of the Fifth Annual JPL Airborne Earth Science*, vol. 1, 1995, https://ntrs.nasa.gov/citations/19950027312.

[7] Behnood, R., Scheunders, P., Ghamisi, P., Licciardi, G. y Chanussot, J. *Noise Reduction in Hyperspectral Imagery: Overview and Application*, publicado como parte de *Data Restoration and Denoising of Remote Sensing Data*, vol. 10, nº 3, 2018, https://doi.org/10.3390/rs10030482.

[8] Bioucas-Dias, J. M. y Figueiredo, M. A. T. *Alternating direction algorithms for constrained sparse regression: Application to hyperspectral unmixing*, publicado como parte de *2nd Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing*, 2010, págs. 1-4, http://www.lx.it.pt/~bioucas/files/whispers10_c_sunsal.pdf.

[9] Boyd, S., Parikh, N., Chu, E., Peleato, B. y Eckstein, E. *Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers*, vol. 3, págs. 1-122, 2010, https://stanford.edu/~boyd/papers/pdf/admm_distr_stats.pdf.

[10] Budd, C. J., Freitag, M. A., y Nichols, N. K. *Regularization techniques for ill-posed inverse problems in data assimilation*, en la serie *Computers & Fluids*, vol. 46, nº 1, págs. 168-7930, 2011, https://doi.org/10.1016/j.compfluid.2010.10.002.

[11] Budruk, R., Don, A. y Shanley, T. *PCl Express System Architecture*, edit. *Addison-Wesley Developers Press*, 9ª ed., 2003, ISBN:978-0-321-15630-7.

[12] Cadence Design Systems, *Cadence to Enhance High-Level Synthesis Offering with Acquisition of Forte Design Systems*, 2014. Disponible en: https://www.cadence.com/en_US/home/search.html?k=Cynthesizer%20Solution (Accedido: 1 de Febrero de 2023).

[13] Chan, Y., Frankovich, R., Hartmann, R., McCarthy, C. y Wong, D. *Altera EP300 Design & Development Oral History Panel*, *Computer History Museum*, 2009, http://archive.computerhistory.org/resources/access/text/2012/10/102702147-05-01-acc.pdf.

[14] Chen, C. H. y Li, F. *Low-Rank and Spectral-Spatial Sparse Unmixing for Hyperspectral Remote Sensing Imagery*, publicado en la revista *Wireless Communications and Mobile Computing*, 2021, https://doi.org/10.1155/2021/9374908.

[15] Chu, P. P. *RTL Hardware Design using VHDL. Coding for Efficiency, Portability, and Scalability*, edit. *John Wiley and Sons*, 1ª ed., 2006, ISBN: 978-0-471-72092-8.

[16] Datta, D., Mallick, P. K., Bhoi, A. K. y Ijaz, M. F. *Hyperspectral Image Classification: Potentials, Challenges, and Future Directions*, publicado en la revista *Computational Intelligence and Neuroscience*, 2022, https://www.hindawi.com/journals/cin/2022/3854635/.

[17] Deshpande, G. *Thermal Management Techniques for Field Programmable Gate Arrays*, 2017, https://utd-ir.tdl.org/bitstream/handle/10735.1/5655/ETD-5608-7473.49.pdf?sequence=5&isAllowed=y.

[18] Doxygen, *OpenCL C++ Bindings*, 2022, https://github.khronos.org/OpenCL-CLHPP/ (Accedido: 5 de Marzo de 2023).

[19] Dunbar, B. (Oficial) y Massengill, D. (Editora) *NASA*, NASA, 2023. Disponible en: https://www.nasa.gov/ (Accedido: 3 de Abril de 2023).

[20] Elecnor Deimos, *COASTNET*, Deimos, 2017. Disponible en: https://elecnor-deimos.com/es/project/coastnet-es/ (Accedido: 7 de Noviembre de 2022).

[21] ETH Zürich, *ETH Zürich*, Eidgenössische Technische Hochschule Zürich, 2023. Disponible en: https://ethz.ch/en.html (Accedido 20 de Marzo de 2023).

[22] ETH Zürich, *Heterogeneous Accelerated Compute Cluster*, Eidgenössische Technische Hochschule Zürich, 2023. Disponible en: https://systems.ethz.ch/research/data-processing-on-modern-hardware/hacc.html (Accedido 20 de Marzo de 2023).

[23] ETH Zürich, *fpga_systems: hacc*, Eidgenössische Technische Hochschule Zürich [Computer software], 2022. https://github.com/fpgasystems/hacc.

[24] FS, *100GBASE-PSM4 QSFP28 1310nm 500m DOM Transceiver*, FS.COM Intelligent Quality Control Systems, 2022, https://img-en.fs.com/file/datasheet/100g-qsfp28-psm4.pdf.

[25] Ghamisi, P., Yokoya, N., Liao, W., Liu, S., Plaza, J., Rasti, B., y Plaza, A. *Advances in Hyperspectral Image and Signal Processing: A Comprehensive Overview of the State of the Art*, publicado en la revista *IEEE Geoscience and Remote Sensing Magazine*, vol. 5, nº 4, págs. 37-78, 2017, https://ieeexplore.ieee.org/document/8113122.

[26] Gisbert, E. R. *TFG 2022-2023 UCM FDI*, version 1.0.0, [Computer software], 2023. https://github.com/enrey471/TFG_2022_2023.git.

[27] Green, R. O., Eastwood, M. L., Sarture, C. M., Chiren, T. G., Aronsson, M., Chippendale, B. J., Faust, J. A., Pavri, B. E., Chovit, C. J., Solis, M., Olah, M. R. y Williams, O. *Imaging Spectroscopy and the Airborne Visible/Infrared Imaging Spectrometer (AVIRIS)*, publicado en *Remote Sensing Environ*, págs. 227-248, 1998, https://folk.ntnu.no/sivertba/master_thesis/NTNU-Master-Thesis-2018/appendix/HSI_specifications/AVIRIS_doc.pdf.

[28] Griffiths, D. J. y College, R. *Introduction to Electrodynamics*, edit. *Prentice Hall*, 2ª ed., págs. 443-469, 1999, ISBN: 978-1108420419.

[29] Iordache, M. D., Bioucas-Dias, J. M., Plaza, A. *Total Variation Spatial Regularization for Sparse Hyperspectral Unmixing*, publicado en la revista *IEEE Transactions on geoscience and remote sensing*, 2012, http://www.lx.it.pt/~bioucas/files/ieee_tgrs_12_sparse_unmix_tv.pdf.

[30] Jet Propulsion Laboratory, *Jet Propulsion Laboratory*, *California Institute of Technology*, NASA, 2023. Disponible en: https://aviris.jpl.nasa.gov/data/image_cube.html (Accedido: 23 de abril de 2023).

[31] Jet Propulsion Laboratory, *AVIRIS Flight: f080920t01*, *California Institute of Technology*, NASA, 2023. Disponible en: https://aviris.jpl.nasa.gov/cgi/flights_08.cgi?step=view_flightlog&flight_id=f080920t01.

[32] Jet Propulsion Laboratory, *AVIRIS Data Portal 2006-2021*, NASA, 2023. Disponible en: `https://aviris.jpl.nasa.gov/dataportal/` (Accedido: 4 de Abril de 2023).

[33] Kaipio, Jari P. y Somersalo, E. *Statistical and Computational Inverse Problems*, en la serie *Applied Mathematical Sciences*, edit. Springer New York, vol. 160, 1ª ed., 2005, ISBN: 978-0-387-22073-4, `https://doi.org/10.1007/b138659`.

[34] Kastner, R., Matai, J. y Neuendorffer, S. *Parallel Programming for FPGAs*, publicado en la revista *ArXiv e-prints*, 2018, `https://kastner.ucsd.edu/hlsbook/`.

[35] Keller, Joseph B. *Inverse Problems*, publicado en la revista *The American Mathematical Monthly*, vol. 83, nº 2, págs. 107–118, 1976, `https://doi.org/10.2307/2976988`.

[36] Khan, M. J., Khan, H. S., Yousaf, A., Khurshid, K. y Abbas, A. *Modern Trends in Hyperspectral Image Analysis: A Review*, en *IEEE Access*, vol. 6, págs. 14118-14129, 2018, `https://ieeexplore.ieee.org/document/8314827`.

[37] Kruse, F. A. *Comparison between AVIRIS and Hyperion for Hyperspectral Mineral Mapping*, 2002, `https://aviris.jpl.nasa.gov/proceedings/workshops/02_docs/2002_Kruse_Hyperion_web.pdf`.

[38] Li, Q., He, X., Wang, Y., Liu, H., Xu, D. y Guo, F. *Review of spectral imaging technology in biomedical engineering: achievements and challenges*, publicado en la revista *Journal of Biomedical Optics*, vol. 18, nº 10, 2013, `https://doi.org/10.1117/1.JBO.18.10.100901`.

[39] Lukasiak, L., Jakubowski, A. y Pioro, Z. *Silicon microelectronics: where we have come from and where we are heading*, publicado en la revista *Journal of Telecommunications and Information Technology*, 2004, `https://oa.mg/work/1495738953`.

[40] Lundeen, S. (Manager) *Aviris - Airborne Visible / Infrared Imaging spectrometer*, NASA, 2022. Disponible en: `https://aviris.jpl.nasa.gov/index.html` (Accedido: 13 de Diciembre de 2022).

[41] Matlab, *Matlab. Matemáticas. Gráficas. Programación.*, *MathWorks*, 2023. Disponible en: `https://es.mathworks.com/products/matlab.html` (Accedido: 3 de Abril de 2023).

[42] Matlab, *Procesamiento de imágenes hiperespectrales*, *MathWorks*, 2023. Disponible en: `https://es.mathworks.com/help/images/hyperspectral-image-processing.html` (Accedido: 3 de Abril de 2023).

[43] Matlab, *Hyperspectral Viewer*, *MathWorks*, 2023. Disponible en: `https://es.mathworks.com/help/images/ref/hyperspectralviewer-app.html` (Accedido: 3 de Abril de 2023).

[44] Mead, C. A. y Conway, L. A. *Introduction to VLSI Systems*, 2ª ed., 1978, `https://ai.eecs.umich.edu/people/conway/VLSI/VLSIText/PP-V2/V2.pdf`.

[45] Meredith, M. *High-Level Synthesis*, edit. *Springer*, 1ª ed., 2008, ISBN: 978-1-4020-8587-1.

[46] Moorby, P. R. y Thomas, D. E. *The Verilog Hardware Description Language*, edit. *Kluwer*, 5ª ed., 2002, ISBN: 1-4020-7089-6.

[47] Motorola Semiconductor Products Inc., *TTL Integrated Circuits Data Book*, 1ª Ed., 1971.

[48] Moya, J. *Welcome to the ETH Alveo Cluster - Heterogeneous Accelerated Compute Clusters (HACC) Program*, Eidgenössische Technische Hochschule Zürich, 2022. Disponible en: `https://public.3.basecamp.com/p/DfoiDaU2d3ZFFph6BzW5mCbj` (Accedido: 20 de Marzo de 2023).

[49] Onera, *Onera, the French Aerospace Lab*, *Office National d'Etudes et de Recherches Aérospatiales*, 2022. Disponible en: `https://www.onera.fr/en` (Accedido: 23 de Diciembre de 2022).

[50] Patel, N. y Soni, H. *Hyperspectral Change Detection using Multi-temporal Hyperspectral images and Sparse Unmixing Algorithm*, publicado en la revista *International Journal of Applied Engineering Research*, vol. 13, nº 2, págs. 16072-16076, 2018, `https://www.ripublication.com/ijaer18/ijaerv13n22_86.pdf`.

[51] Perry, D. L. *VHDL. Programming by example*, edit. *McGraw-Hill*, 4ª ed., 2002, ISBN: 978-0071400701.

[52] Plaza, A., Bioucas-Dias, J. M., Dobigeon, L., Parente, M., Du, Q., Gader, P. y Chanussot, J. *Hyperspectral Unmixing Overview: Geometrical, Statistical, and Sparse Regression-Based Approaches*, publicado en la revista *IEEE Journal of selected topics in applied earth observations and remote sensing*, vol. 5, nº 2, 2012, `https://ieeexplore.ieee.org/document/6200362`.

[53] Sentinel, *Sentinel Online - ESA - Sentinel Online*, Agencia Espacial Europea (ESA), 2021. Disponible en: `https://sentinels.copernicus.eu/web/sentinel/home` (Accedido: 2 de Diciembre de 2022).

[54] Société Française de Photogrammétrie et Télédétection, *Spectral Unmixing*, Gipsa-lab, 2017, `https://www.sfpt.fr/hyperspectral/wp-content/uploads/2013/01/cours_Licciardi.pdf`.

[55] Sveinsson, J. R., Behnood, R., Ulfarsson, M. O. y Sigurdsson, J. *First order roughness penalty for hyperspectral image denoising*, publicado como parte de *5th Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing*, 2013, `https://ieeexplore.ieee.org/document/8080623`.

[56] Taylor, M. (Manager) y Rocchio, L. (Autora) *Landsat science*, NASA, 2022. Disponible en: `https://landsat.gsfc.nasa.gov/` (Accedido: 4 de Noviembre de 2022).

[57] Texas Instruments Inc., *Integrated Circuits Catalog. 1967-8.*, 1ª ed., 1969.

[58] Trimberger, S. *A Reprogrammable Gate Array and Applications*, publicado en la revista *Proceeding of the IEEE*, vol. 81, nº 7, 1993, `http://arantxa.ii.uam.es/~die/[Lectura%20FPGA%20Architecture]%20A%20reprogrammable%20gate%20array%20-Trimberger.pdf`

[59] Ulfarsson, M. O., Rasti, B. y Ghamisi, P. *Automatic Hyperspectral Image Restoration Using Sparse and Low-Rank Modeling*, publicado en la serie *IEEE Geoscience and Remote Sensing Letters*, vol. 14, nº 12, págs. 2335-2339, 2017, `https://ieeexplore.ieee.org/document/8098642`.

[60] USGS, *USGS. Science for a changing world*, Departamento del Interior de Estados Unidos, 2023. Disponible en: `https://www.usgs.gov/` (Accedido: 4 de Abril de 2023).

[61] USGS, *USGS Spectral Library Version 7*, Departamento del Interior de Estados Unidos, 2023. Disponible en: `https://pubs.er.usgs.gov/publication/ds1035` (Accedido: 4 de Abril de 2023).

[62] Vahid, F. *Digital Design with RTL Design, VHDL, and Verilog*, edit. *John Wiley and Sons*, 2ª ed., 2010, ISBN: 978-0-470-53108-2.

[63] Wang, Y., Shao, Z. *A modified SUnSAL-TV algorithm for hyperspectral unmixing based on spatial homogeneity analysis*, publicado como parte de *IOP Conference Series: Earth and Environmental Science*, vol. 17, 2014, `https://www.researchgate.net/publication/260910300_A_modified_SUnSAL-TV_algorithm_for_hyperspectral_unmixing_based_on_spatial_homogeneity_analysis`.

[64] Weste, N. y Harris, D. *CMOS VLSI Design: A Circuits and Systems Perspective*, edit. *Pearson*, 4ª ed., 2010, ISBN: 978-0321547743.

[65] Xilinx, *Vitis Unified Software Platform*, Advanced Micro Devices, 2023. Disponible en: `https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html` (Accedido: 4 de Febrero de 2023).

[66] Xilinx, *Vitis Unified Software Platform Documentation*, Advanced Micro Devices, 2023. Disponible en: `https://docs.xilinx.com/v/u/2022.1-English/ug1416-vitis-documentation` (Accedido: 4 de Febrero de 2023).

[67] Xilinx, *Introduction to Vitis HLS*, Advanced Micro Devices, 2023. Disponible en: `https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Introduction-to-Vitis-HLS` (Accedido: 4 de Febrero de 2023).

[68] Xilinx, *Vivado 2020.1 - High-Level Synthesis (C based)*, Advanced Micro Devices, 2023. Disponible en: `https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0012-vivado-high-level-synthesis-hub.html` (Accedido: 4 de Febrero de 2023).

[69] Xilinx, *Vivado*, Advanced Micro Devices, 2023. Disponible en: `https://www.xilinx.com/products/design-tools/vivado.html` (Accedido: 4 de Febrero de 2023).

[70] Xilinx, *AMD Xilinx*, Advanced Micro Devices, 2023. Disponible en: `https://www.xilinx.com/` (Accedido: 4 de Febrero de 2023).

[71] Xilinx, *Vitis Unified Software Platform 2022.1 Documentation: Application Acceleration Development. Installation Requirements*, Advanced Micro Devices, 2023. Disponible en: `https://docs.xilinx.com/r/2022.1-English/ug1393-vitis-application-acceleration/Installation-Requirements` (Accedido: 4 de Febrero de 2023).

[72] Xilinx, *Alveo U250 Data Center Accelerator Card*, Advanced Micro Devices, 2023. Disponible en: `https://www.xilinx.com/products/boards-and-kits/alveo/u250.html#documentation` (Accedido: 11 de Febrero de 2023).

[73] Xilinx, *HLS Pragmas*, Advanced Micro Devices, 2023. Disponible en: `https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas` (Accedido: 15 de Febrero de 2023).

[74] Xilinx, *UltraScale Architecture and Product Data Sheet: Overview*, Advanced Micro Devices, 2022, `https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds890-ultrascale-overview.pdf`.

[75] Xilinx, *Xilinx Alveo: Product Selection Guide. Data Center Acceleration Cards*, Advanced Micro Devices, 2023. Disponible en: `https://docs.xilinx.com/v/u/en-US/alveo-product-selection-guide` (Accedido: 19 de Febrero de 2023).

[76] Xilinx, *HLS Programming Guide*, Advanced Micro Devices, 2023. Disponible en: `https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Programmers-Guide` (Accedido: 26 de Febrero de 2023).

[77] Xilinx, *Vitis Libraries*, Advanced Micro Devices, 2023. Disponible en: `https://www.xilinx.com/products/design-tools/vitis/vitis-libraries.html` (Accedido: 4 de Marzo de 2023).

[78] Xilinx, *Vitis Solver Library*, Advanced Micro Devices, 2023. Disponible en: `https://www.xilinx.com/products/design-tools/vitis/vitis-libraries/vitis-solver.html` (Accedido: 4 de Marzo de 2023).

[79] Xilinx, *OpenCL Platform Model*, Advanced Micro Devices, 2023. Disponible en: `https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/fpn1504034296297.html` (Accedido: 10 de Marzo de 2023).

[80] Xilinx, *AMD University Program*, Advanced Micro Devices, 2023. Disponible en: `https://www.xilinx.com/support/university.html` (Accedido: 20 de Marzo de 2023).

[81] Xilinx, *Heterogeneous Accelerated Compute Clusters (HACC)*, Advanced Micro Devices, 2023. Disponible en: `https://www.xilinx.com/member/xup_research_clusters.html` (Accedido: 20 de Marzo de 2023).

[82] Xilinx, *Heterogeneous Accelerated Compute Clusters*, Advanced Micro Devices, 2023. Disponible en: `https://www.amd-haccs.io/` (Accedido: 20 de Marzo de 2023).

[83] Zhang, H., Zhang, L., Shen, H. y He, W. *Hyperspectral Image Denoising via Noise-Adjusted Iterative Low-Rank Matrix Approximation*, publicado en *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 8, nº 6, págs. 3050-3061, 2015, `https://ieeexplore.ieee.org/document/7062902`.