Optimización del procesado de imágenes hiperespectrales mediante programación con GPUs y aceleradores

Optimizing Hyperspectral Image Processing with GPUs and Accelerators

Adrián Real del Noval y Óscar Ruiz de Pedro

GRADO EN INGENIERÍA DE COMPUTADORES. FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo de Fin de Grado en Ingeniería de Computadores

Curso 2021-2022

Directores:

Carlos García Sánchez

Sergio Bernabé García

Índice general

Índice de figuras

Índice de tablas

Resumen

Durante siglos se han creado diversas teorías sobre el planeta en el que vivimos y con el desarrollo de las tecnologías se han podido conocer muchas de sus características, se han ido mejorando las técnicas que se utilizan hasta dar con las que se encuentran ahora, donde se analiza a través de satélites obteniendo imágenes hiperespectrales para un análisis píxel a píxel de los materiales que contiene.

El análisis de imágenes hiperespectrales es una tarea ardua, al captar el material de el que se compone la imagen a través de un solo píxel se dificulta cuando ese píxel tiene más de un material, llamado *el problema de la mezcla espectral*, por ello se hace un desmezclado espectral a través de una cadena de procesamiento. Dependiendo de la imagen, algoritmos seleccionados para la cadena de desmezclado espectral y el factor tecnológico puede dar diversos rendimientos.

La cadena de desmezcaldo espectral tiene tres fases, la primera fase se trata de obtener el numero de materiales o *endmembers* que tiene la imagen hiperespectral, la segunda fase se extrae los diferentes materiales que componen la imagen hiperespectral y la tercera fase se saca un mapa de abundancia de cada material. Se han seleccionado en el mismo orden los algoritmos VD, VCA e ISRA para completar la cadena de desmezclado espectral.

En este proyecto se han implementado todas las fases de forma paralela, contribuyendo a la optimización de estos algoritmos de procesado de imágenes hiperespectrales en distintos paradigmas de programación paralela, como son: OpenACC, OpenMP y SYCL (oneAPI). Se utilizan este tipo de paradigmas ya que otro de los objetivos de este trabajo es poder ejecutar todos los algoritmos en sistemas heterogéneos, con todos los resultados obtenidos se hace una comparativa de rendimiento buscando la mejor combinación entre estos.

Palabras clave

Imágenes hiperespectrales, desmezclado espectral, computación paralela, sistemas heterogéneos, OpenACC, OpenMP, SYCL.

Abstract

For centuries, various theories have been created about the planet we live in, and with the development of technologies, many of its characteristics have been discovered. The techniques used for this matter have been improved and perfected, making satellites able to take hyperspectral images for a pixel-by-pixel analysis of the materials they contain.

The analysis of hyperspectral images is an arduous task, capturing the material from which an image is composed through a single pixel becomes difficult when that pixel has more than one material, this is known as *the problem of spectral mixing*. For this reason, spectral unmixing is done through a chain of processing. Depending on the image, the selected algorithms for the spectral unmixing chain and the technological factor can lead to different performance results.

The spectral unmixing chain has three phases: the first phase obtains the number of materials (or *endmembers*) present in the hyperspectral image, the second phase extracts what different materials that make up the hyperspectral image, and the third phase generates an abundance map of each material in the hyperspectral image. The VD, VCA and ISRA algorithms have been selected in that order to create the spectral unmixing chain.

In this project, all the phases have been implemented using parallel computing, contributing to the optimization of these hyperspectral image processing algorithms in different parallel programming paradigms, such as: OpenACC, OpenMP and SYCL (oneAPI). This type of paradigm is used since one of the objectives of this work is to be able to execute all the algorithms in heterogeneous systems. With all the results obtained, a performance comparison is made looking for the best combination between them.

Keywords

Hyperspectral imaging, spectral unmixing, parallel computing, heterogeneous systems, OpenACC, OpenMP, SYCL.

Capítulo 1

Introducción

1.1. Motivación

Este trabajo tiene como motivación principal la contribución a la optimización de los algoritmos de procesado de imágenes hiperespectrales usando paralelismo a través de paradigmas que permitan la ejecución en sistemas heterogéneos. Una imagen hiperespectral (Figura 1.1[1]) es una imagen que contiene muchas más capas de información que una imagen digital RGB. Estas capas adicionales corresponden a diferentes rangos de longitud de onda (que pueden no ser visibles para el ojo humano), y son utilizadas en una gran cantidad de aplicaciones, entre ellas: control de calidad, análisis climático, análisis de polución, análisis de terreno para diferentes ámbitos como la agricultura, visión por computador, exploración extraterrestre, análisis de yacimientos geológicos, aplicaciones del sector de la defensa, y muchas más.

Las imágenes hiperespectrales son tomadas por dispositivos especiales situados en satélites, drones, UAVs, o plataformas aerotransportadas, operados principalmente por organismos internacionales como NASA o la Agencia Europea del Espacio (ESA). En la actualidad se lanzan una gran cantidad de satélites para todas las aplicaciones mencionadas anteriormente. Al igual que durante los últimos tiempos ha aumentado la calidad de los satélites que

[1] https://aviris.jpl.nasa.gov/data/image_cube.html

contienen los dispositivos que capturan las imágenes hiperespectrales, también han mejorado estos dispositivos, pudiendo capturar más bandas espectrales.

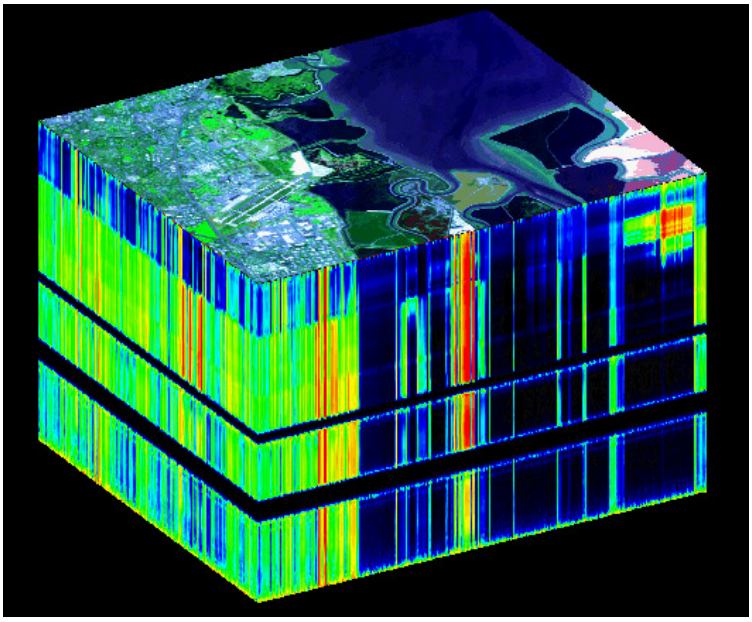

Figura 1.1: *Imagen tomada por el sensor AVIRIS en 1992.*

La extracción de información relevante de estas imágenes y su posterior procesamiento conlleva un esfuerzo computacional extraordinario, principalmente por la cantidad de información espectral que contienen. A su vez, es necesario el uso de algoritmos especializados debido al problema de los píxeles mezcla [1]. Si además se tiene en cuenta que muchas aplicaciones exigen un análisis en tiempo real, la necesidad de optimizar el procesamiento de estas imágenes se hace indispensable.

El enfoque elegido en este trabajo para optimizar estos algoritmos es la computación paralela. A pesar de la enorme utilidad y resultados obtenidos con este tipo de computación, ha tenido un problema clave a lo largo de toda su historia: la dependencia de la arquitectura subyacente, que conlleva poca (o nula) portabilidad con otros sistemas y arquitecturas. Los paradigmas de programación escogidos revolucionan este aspecto de la computación paralela, siendo en gran medida independientes de la arquitectura subyacente, lo que permite ejecutarlos en multitud de dispositivos (tanto CPU como GPU).

La computación heterogénea se refiere a sistemas que cuentan con más de un tipo de procesador. Esta forma de computación supone múltiples aspectos revolucionarios como la posibilidad de ser desarrollada, compilada y ejecutada en todo tipo de equipos (clústers, portátiles, computadores de sobremesa, etc). A su vez, permite obtener más rendimiento en la ejecución de muchas tareas variadas, precisamente por la heterogeneidad de los componentes que la conforman. Las tendencias actuales muestran que este tipo de computación puede no solo mejorar el rendimiento sino al mismo tiempo ofrecer un consumo energético.

En resumen, este trabajo tiene dos motivaciones principales: la primera es contribuir a la optimización de los algoritmos de procesado de imágenes hiperespectrales a través del paralelismo, para el uso eficiente de estos algoritmos; y la segunda es hacer uso de los paradigmas de programación que permiten una ejecución heterogénea, aumentando en enorme medida la portabilidad en diferentes sistemas, resultando más interesante que una implementación dependiente de una arquitectura exclusiva.

1.2. Objetivos

El objetivo principal de este trabajo es optimizar una cadena completa de procesamiento para el desmezclado espectral en imágenes hiperespectrales a través de la computación paralela, haciendo uso de modernos paradigmas de programación que permiten una ejecución de los algoritmos en sistemas heterogéneos. Los resultados obtenidos se analizarán y compararán con la versión secuencial (inicial, no paralela), y entre el resto de paradigmas, y se obtendrán conclusiones sobre las mejoras obtenidas. Para este cometido se realizarán los siguientes pasos o sub-objetivos para cada etapa de la cadena de procesamiento:

- Análisis del grado de paralelismo potencial de los algoritmos elegidos.
- Optimización de los algoritmos en los paradigmas elegidos.
- Análisis de resultados y comparación de rendimiento entre los paradigmas.

1.3. Plan de trabajo

El plan de trabajo seguido a lo largo del año se ilustra de forma simplificada con un diagrama de Gantt en la Figura 1.2.

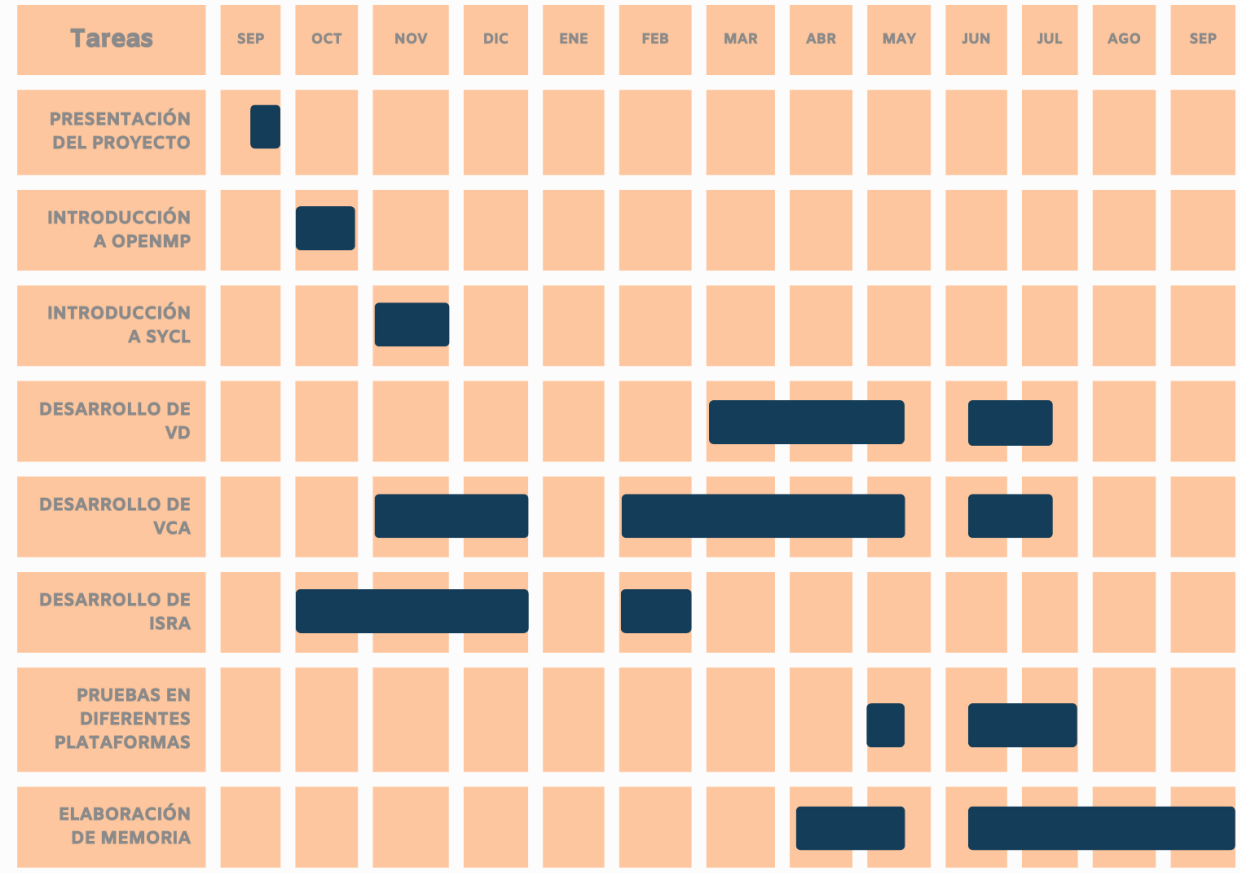

Figura 1.2: *Diagrama de Gantt sobre la optimización del procesado de imágenes hiperespectrales mediante programación con GPUs y aceleradores 2021-2022.*

1.4. Organización de esta memoria

Teniendo presentes los anteriores objetivos concretos, se procede a describir la organización del resto de esta memoria, estructurada en una serie de capítulos cuyos contenidos se describen a continuación:

- **Análisis hiperespectral**: se definen las imágenes hiperespectrales y se nombran algunos sensores hiperespectrales (AVIRIS y EO-1 Hyperion), los modelos de mezcla espectral, y se justifica la necesidad de paralelización de los algoritmos que trabajan con estas imágenes.

- **Paralelismo y paradigmas**: se define qué es el paralelismo, conceptos básicos y tipos asociados a este, nomenclatura, y los paradigmas de programación paralela más importantes y utilizados en este trabajo.

- **Implementaciones**: se comienza detallando el enfoque escogido en este trabajo relacionando los conceptos de paralelismo con las imágenes hiperespectrales. Después, se analiza cada algoritmo seleccionado, y se justifica su grado de paralelismo y se explican las diferentes implementaciones y diferencias entre estas.

- **Análisis de resultados**: se presentan los resultados de todas las implementaciones de cada algoritmo para las imágenes hiperespectrales y los sistemas elegidos.

- **Conclusiones y trabajo futuro**: se presentan las conclusiones obtenidas a raíz de los resultados y posibles líneas de trabajo futuro que continúen y expandan el alcance de este trabajo.

Capítulo 2

Análisis hiperespectral

2.1. Imágenes hiperespectrales

Las imágenes hiperespectreales [2] son imágenes que por cada píxel contienen información espectral llamada vector que corresponde a los diferentes tipos de longitudes de onda presentes en dicho píxel [3]. Las longitudes de onda pueden ser muy variables, tanto visibles como otras no captables para el ser humano como el ultravioleta o el infrarrojo.

Cuando la imagen contiene varias bandas espectrales (decenas, aproximadamente), nos encontramos con las imágenes multiespectrales [4]. La principal diferencia con las hiperespectrales es que tienen menor cantidad de bandas espectrales, a su vez, estas bandas espectrales no necesitan ser contiguas unas con otras, mientras que en una imagen hiperespectral se obtiene el espectro continuo o también llamado *firma espectral* [5] del objeto (Figura 2.1).

Las imágenes hiperespectrales se representan mediante un cubo llamado "cubo hiperespectral" o "hipercubo", cada eje de este hipercubo representa información diferente sobre la imagen hiperespectral: el eje X son las líneas (*lines*), el eje Y son las muestras (*samples*) y el eje Z representa las bandas espectrales (*bands*). Estos datos son recogidos por un sensor hiperespectral para que puedan ser más tarde interpretadas con el hipercubo (Figura 2.2).

En las imágenes hiperespectrales existen dos tipos de píxeles [6], esta distinción depende

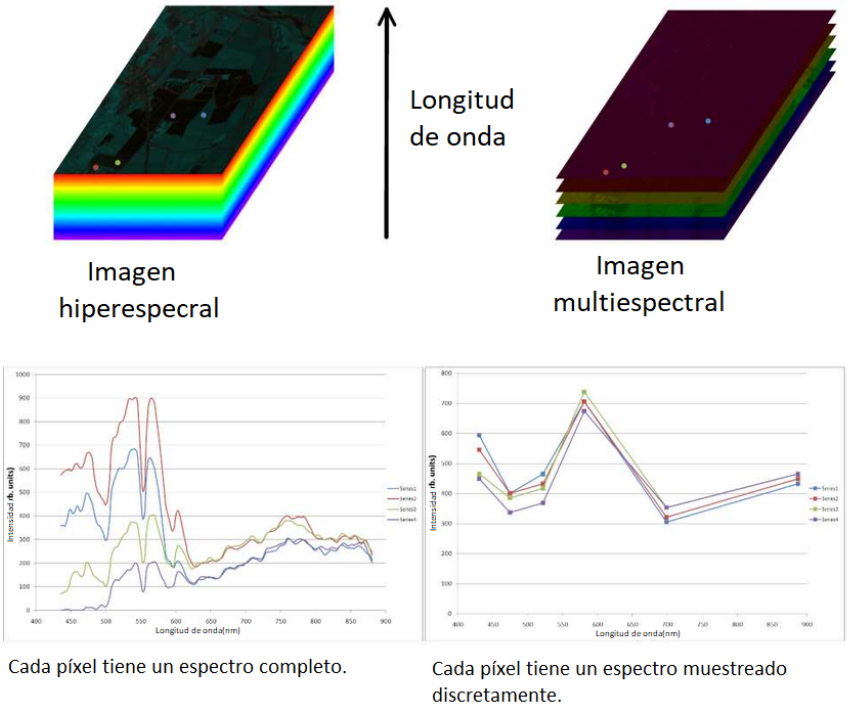

Figura 2.1: *Diferencia entre imagen multiespectral e imagen hiperespectral.*

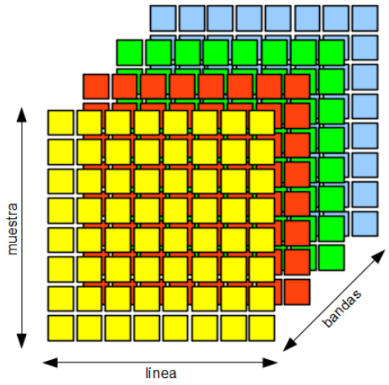

Figura 2.2: *Ejemplo de cubo hiperespectral.*

de su firma espectral: aquellos cuya firma contiene solamente un material reciben el nombre de píxeles puros (*endmembers*) [7], todos aquellos cuya firma espectral esté compuesta por más de un material se les llama píxeles mezcla [1][4], son los más abundantes en las imágenes hiperespectrales, ya que es es muy difícil encontrar en un solo píxel un único material. En la Figura 2.3 se observa un ejemplo claro de un píxel mezcla que está compuesto por vegetación y suelo, también se encuentra un píxel puro conformado por agua.

La resolución espectral es otro concepto a tener en cuenta en este tipo de imágenes, se define como la separación que hay entre las longitudes de onda, si hay una separación pequeña se obtendrá una resolución mayor, esta depende de la tecnología del sensor que se use y habrá un mayor costo computacional en caso de contar con un sensor de altas prestaciones; en cambio, se obtendrán mejores resultados en el análisis de la imagen.

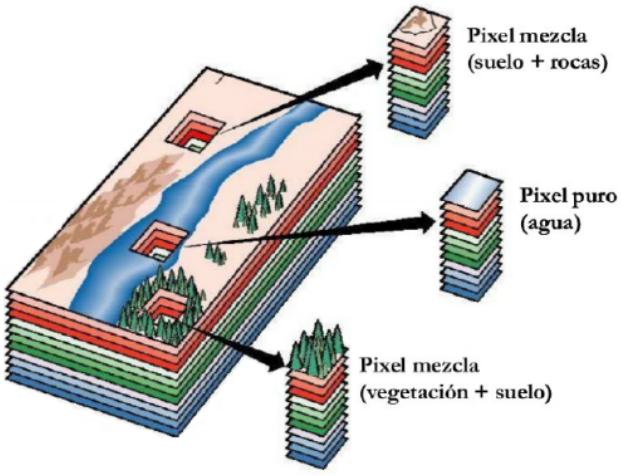

Figura 2.3: *Ejemplo de píxeles puros y píxeles mezcla.*

2.2. Sensores hiperespectrales

Los sensores hiperespectrales han sido uno de los inventos más revolucionarios en lo que respecta a la observación de la Tierra, antes de esta invención se utilizaban cámaras adheridas a globos dirigibles o el uso de satélites, pero no conseguían la precisión que tienen los sensores actuales. Estos sensores se basan en la espectroscopia para la obtención de la

imágenes hiperespectrales, esta disciplina estudia la luz reflejada o emitida de los materiales. También gracias a estos sensores se consigue la variación de energía con la longitud de onda. Los sensores que se encargan de este trabajo utilizan los denominados espectrómetros de imágenes que se encargan de obtener mediciones de bandas con una resolución espectral alta.

2.2.1. Sensor AVIRIS

El sensor AVIRIS[1] es uno de los sensores capacitados para la obtención de imágenes hiperespectrales, se transporta por vía aérea para poder analizar zonas tanto del espectro visible como del no visible, tales como las infrarrojas. Este sensor es capaz de obtener 224 bandas con una longitud de onda de entre 400 y 2500 nm y también puede obtener una resolución espectral de 10 nm (ancho que hay entre las bandas). El objetivo de este sensor es el análisis tanto de la superficie como de la atmósfera terrestre, se centra en investigar el progreso del medio ambiente y el cambio climático.

2.2.2. Sensor EO-1 Hyperion

El sensor EO-1 Hyperion[2] es otro sensor capaz de obtener imágenes hiperespectrales con una capacidad de 220 bandas espectrales que cubre un rango espectral de 400 a 2500 nm, su ancho de bandas espectrales es de 10 nm y es capaz de tener un tamaño de los píxeles superficiales de 30m.

2.3. Problema principal: mezcla espectral

Al analizar una imagen hiperespectral se persigue determinar los distintos materiales que la conforman, así como la cantidad de material por píxel. Cuando se trata de píxeles puros esta tarea es sencilla, el problema aparece con los píxeles mezcla ya que en estos se

[1] https://aviris.jpl.nasa.gov/

[2] http://eo1.usgs.gov/hyperion.phpes

combinan varios materiales en un mismo píxel. A este problema se le llama *problema de la mezcla espectral*.

El tratamiento de este problema se hace a través del desmezclado espectral [8], consiste en una serie de estrategias sobre la imagen con el fin de encontrar todos los píxeles puros de esta, llegando a encontrar sus respectivas firmas espectrales de los endmembers, y la cantidad de estos en la imagen, denominada "vector de abundancia" [9]. Al utilizar esta estrategia se consigue una cadena de desmezclado que permite la extracción de información sobre la imagen a nivel de sub-píxel.

Dependiendo del análisis de la imagen y de lo que se quiera detallar se puede cambiar la cantidad de *endmembers* que hay. Por ejemplo, si hay una gran variedad de árboles en la imagen pero lo que importa es únicamente la cantidad de árboles, en vez de utilizar una gran cantidad de *endmembers* para referenciar los distintos tipos de árboles, puede ser conveniente usar solo uno; dependerá de la investigación que se esté haciendo.

Respecto a la abundancia, su cálculo se basa en una función lineal donde se toman en cuenta todos los materiales que no reflejan la luz del mismo modo [10]. Hay que tener en cuenta la humedad del ambiente ya que un material húmedo no refleja la luz igual que el mismo material seco; también se han de tener en cuenta las condiciones climáticas, ya que una meteorología adversa puede condicionar de forma directa los resultados. En el último caso de que en un píxel predomine un material que no refleja la luz y en el mismo píxel se encuentre un material que sí refleje la luz, la estimación de la cantidad de ambos materiales será errónea.

Hay numerosos algoritmos para la realización del desmezclado espectral, pero hay dos modelos principales: los modelos lineales y los no lineales [11]. Los modelos lineales no toman en cuenta los reflejos secundarios o los efectos de dispersión de la luz, y los modelos no lineales [12] sí lo tienen presente, dando como resultado una caracterización del espectro de mezcla. Los modelos lineales dan como resultado una combinación lineal de firmas espectrales puras.

En la Figura 2.4 se ve la diferencia que hay entre ambos modelos, en el lineal, nada más tocar la superficie los rayos de luz son reflejados, y en el modelo no lineal se observa como antes de ser reflejados se dispersan por la superficie.

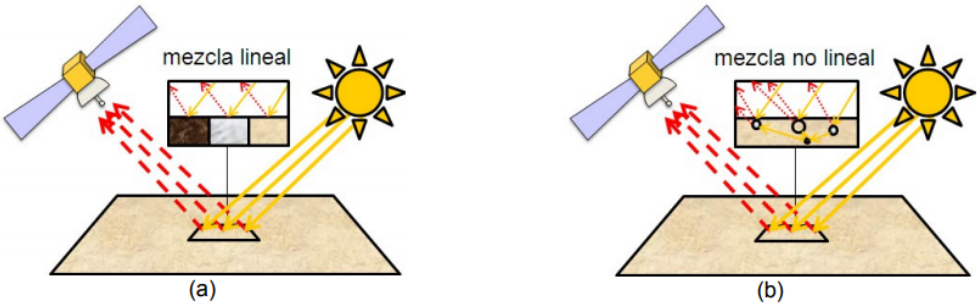

Figura 2.4: *Comparativa entre modelos de desmezclado espectral.*

2.4. Necesidad de paralelización

Uno de los problemas de las imágenes hiperespectrales es la gran cantidad de información que contienen, ya que dificultan tanto su almacenamiento como su procesado. Las constituciones como la NASA, que almacenan Terabytes de información cada día con el fin de poder ser procesadas en un futuro, se encuentran con el problema de que la mayoría de las imágenes no se pueden procesar en grandes cantidades debido al coste de ser almacenadas, procesadas y luego distribuidas.

Otro de los grandes problemas que tienen las imágenes hiperespectrales en cuestión al coste computacional son las técnicas de análisis que se utilizan, debido a que los píxeles mezcla recogidos por los sensores tienen varios materiales que están a un nivel de sub-píxel, es necesario un diseño algorítmico que pueda adaptarse a ello, alzando enormemente el costo computacional.

En cuanto al problema de la alta cantidad de información contenida en las imágenes hiperespectrales, existen varias soluciones relacionadas con la selección de bandas espectrales dependiendo de la zona y la investigación, escogiendo aquellos materiales que interesen como, por ejemplo, rocas en zonas mineras. Existen otro tipo de soluciones relacionadas con la compresión de datos, desde una compresión sin pérdida que tiene una menor compresión pero contiene mayor cantidad de la información original, a una compresión con pérdida que tiene una compresión más alta pero en la cual se pierde parte de la información original.

Por todos estos problemas anteriores es necesaria una solución eficiente que pueda llevar

a cabo un procesado de alta dimensionalidad en tiempo real, por ello en la actualidad se utilizan arquitecturas paralelas para las imágenes hiperespectrales. En la tendencia actual existen y se utilizan múltiples maneras diferentes de implementar estas soluciones, como FPGAs, aceleradores, procesadores multinúcleo o GPUs. Estas dos últimas opciones son en la que se centra el trabajo, paralelizando las fases de este análisis, consiguiendo ejecuciones eficientes tanto en procesadores multinúcleo como en GPUs, haciendo que su procesado sea menos costoso.

2.5. Cadena de desmezclado propuesta

En el proyecto se ha paralelizado con varios paradigmas de programación diferentes una posible cadena de desmezclado formada por tres fases. Una vez obtenida la imagen hiperespectral a través de los sensores explicados en secciones anteriores, empieza la primera fase que consiste en la estimación del número de *endmembers* que tiene la imagen hiperespectral, para realizar esta fase se ha optado por el algoritmo *Virtual Dimensionality* (VD) [13]. A continuación, comienza la segunda fase donde se hace una extracción de los diferentes *endmembers* que la componen, para esto, se ha seleccionado el algoritmo *Vertex Component Analysis* (VCA) [14]. Por último, será necesario realizar la estimación de abundancias de la imagen hiperespectral para cada uno de los *endmembers* obtenidos en la fase anterior, para ello, se hará uso del algoritmo *Image Space Reconstruction Algorithm* (ISRA) [15].

La Figura 2.5 es una imagen original en donde se puede ver el proceso de desmezclado completo de una imagen hiperespectral. Cuando se tiene la imagen hiperespectral con todos los datos correspondientes, lo primero que se hace es la estimación del número de *endmembers* para saber el contenido de la muestra. Tras esta fase se puede hacer un paso opcional donde se realiza una reducción de la imagen hiperespectral, este paso previo puede ayudar a la siguiente fase. La fase siguiente es la extracción, donde se identifica las firmas de los diferentes *endmembers* que componen la imagen. Por último, se obtiene un mapa de abundancia por cada uno.

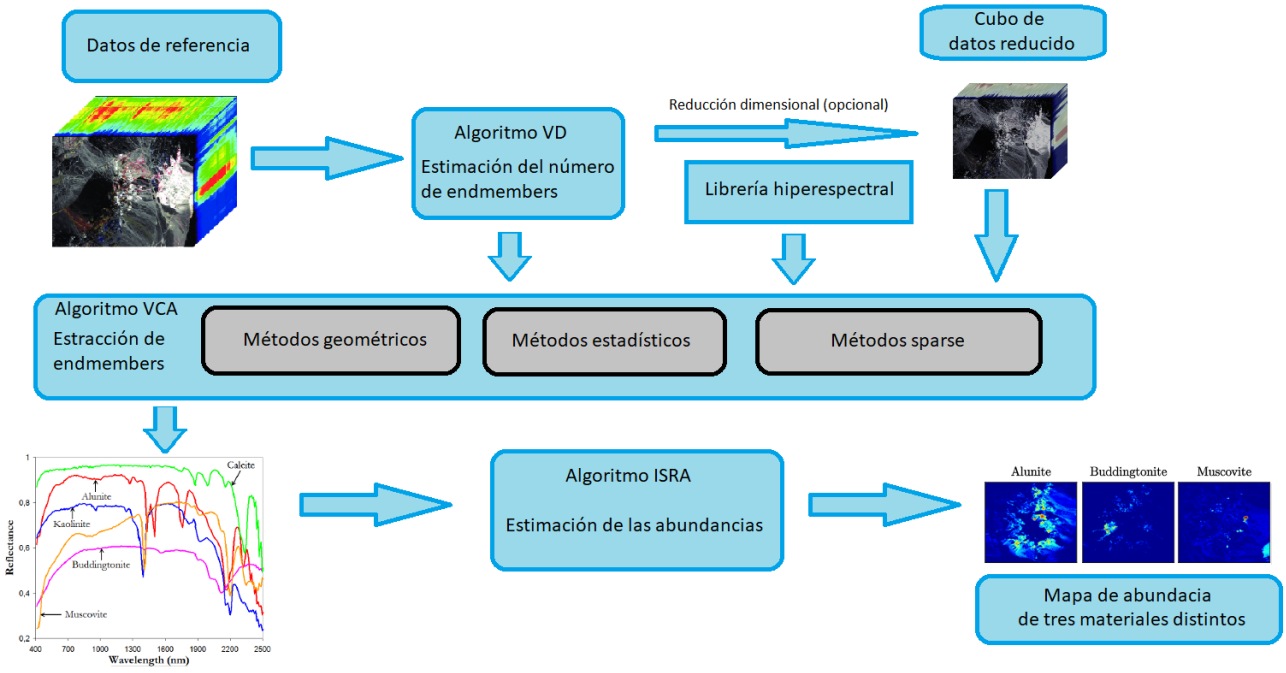

Figura 2.5: *Proceso de desmezclado para la imagen hiperespectral Cuprite.*

Capítulo 3

Paralelismo y paradigmas

3.1. Introducción

El paralelismo es una forma de cómputo en la que varias operaciones pertenecientes a un programa se ejecutan simultáneamente haciendo uso de múltiples recursos computacionales, en pos de conseguir una mayor eficiencia. Dada la dificultad en los últimos años para mejorar la frecuencia de las CPUs (*Central Processing Units*) de un único núcleo (*single-core*), la computación paralela se ha convertido en un estándar tanto en la informática de altas prestaciones como en la particular, principalmente en forma de CPUs multi-núcleo (*multicore*) y GPUs (*Graphics Processing Units*), aunque existen muchos otros tipos de dispositivos y aceleradores.

Otro factor a tener en cuenta cuando se programa de forma paralela es la posible necesidad de sincronización entre los múltiples hilos en ejecución; de forma que solo se puede empezar una parte de un programa cuando se ha terminado la anterior en caso de ser dependientes, evitando dependencias de datos.

Aunque hay varios tipos de paralelismo, el usado en este trabajo es del tipo "una instrucción, múltiples datos" (del inglés *Single Instruction, Multiple Data*, o SIMD). En este tipo de paralelismo se lanza la misma instrucción (*single instruction*) a las diferentes unidades de cómputo, pero cada una de estas realiza la instrucción con diferentes datos (*multiple*

data). Normalmente esta técnica se usa cuando la cantidad de datos es alta, como es el caso del procesamiento de imágenes hiperespectrales, ya que independiza el procesado sobre los datos.

En nomenclatura paralela, se llama *host* o anfitrión al elemento computacional que ejecuta inicialmente el código y que invocará al elemento de cómputo paralelo cuando sea necesario, asignándole el cómputo de un conjunto de instrucciones, estas llamadas al dispositivo se denominan (*kernels*); de la misma forma, se llama *device* o dispositivo al elemento de cómputo que realiza los cálculos paralelos. Con los paradigmas de programación paralela escogidos para este trabajo, el *device* no está limitado a un modelo específico, sino que es genérico, pudiendo ejecutarse en múltiples GPUs o CPUs, dependiendo del paradigma.

3.2. CUDA

CUDA[1] es uno de los modelos de programación paralela más importantes. Fue introducido por NVIDIA en 2006, por lo que sirve únicamente para sus GPUs. Hay una funcionalidad que cabe destacar de CUDA (y de los paradigmas basados en CUDA), y es que, en compilación, el código del *host* y el del *device* se independizan, de forma que las llamadas originales al *device* en el código del *host* se sustituyen por referencias al código del *device* (llamado *PTX*). Cuando se va a ejecutar el código, hay una etapa de compilación denominada "compilación en el momento" (*just-in-time compilation*) en donde el código PTX es compilado por el controlador del dispositivo, esto tiene dos ventajas:

- 1ª. El programa se beneficia de cualquier mejora que haya sido desarrollada en las nuevas versiones del controlador.
- 2ª. Permite adaptar la ejecución de los *kernels* al dispositivo en concreto, de forma que cualquier código *PTX* puede ser ejecutado en varias GPUs distintas de NVIDIA. Por lo que, dentro del universo de NVIDIA, un código PTX cuenta con un grado de heterogeneidad muy alto a la hora de ejecución.

[1] https://docs.nvidia.com/cuda/

A continuación se ilustra un ejemplo con las partes más relevantes de un código CUDA para la suma de vectores. Primero, se crea el *kernel* CUDA:

```
// Kernel de CUDA
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    if (id < n)
        c[id] = a[id] + b[id];
}
```

Después, ese *kernel* se invoca en el programa como una función estándar:

```
// Invocacion de kernel CUDA
int main() {
    // Declaracion e inicializacion de los vectores en el host
    ...

    // Reserva de memoria en GPU
    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
    cudaMalloc(&d_c, bytes);

    // Copia de vectores del host al device
    cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

    int blockSize, gridSize;
```

```
    // Numero de hilos en cada bloque
    blockSize = 1024;

    // Numero de bloques en cada grid
    gridSize = (int)ceil((float)n/blockSize);

    // Ejecucion del kernel
    vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

    // Liberacion de memoria
    ...
}
```

CUDA es uno de los paradigmas que ofrece al programador mayor control sobre los *kernels* y el código paralelo generado, pero a cambio es más lento de desarrollar y requiere una atención al detalle muy superior. En busca de un modelo de programación más sencillo que compita en eficiencia con CUDA, han surgido varias alternativas.

3.3. OpenACC

OpenACC[2] es una interfaz de programación de aplicaciones (API) cuyo diseño está orientado a simplificar la programación paralela heterogénea en GPUs y CPUs. Es multilenguaje y declarativo, donde se programa a través de pragmas, en donde se indica qué comportamiento se quiere realizar en cierta zona de código. OpenACC ofrece distintos niveles de abstracción al programador:

- Una de las principales sentencias de OpenACC es `#pragma acc data`, la cual indica los movimientos de memoria entre *host* y *device*. Dependiendo de la dirección de la operación de memoria (*host -> device* o *device -> host*), el pragma anterior se combina

[2]https://www.openacc.org/

con `copyin`, `copyout` o `copy`. Véase la línea 5 del Algoritmo 4 para un ejemplo del uso de este pragma.

- Las sentencias `#pragma acc seq` o `#pragma acc parallel` antes de un bloque de código indican que ese código ha de ser ejecutado de forma secuencial o paralela en el dispositivo, respectivamente. En un bucle, se debe de utilizar añadiendo `loop`. Si dentro de un bucle hay otro bucle se añade `loop` pero no es necesario añadir `parallel` si en el bucle externo ya se ha utilizado esa sentencia. En la línea 8 del Algoritmo 4 se puede ver un ejemplo de este pragma en uso.

- La sentencia `#pragma acc kernels` antes de un bloque de código delega al compilador la tarea de detectar qué zonas del código son paralelizables y cuáles no. También se puede utilizar en conjunto con las siguientes sentencias para indicar que el código contenido en el `#pragma` debe ser ejecutado en el *device* o dispositivo.

Algorithm 1 Uso de Kernel en OpenACC

```
1: #pragma acc kernels
2: **Codigo**
```

- En el caso de haber varios bucles anidados en donde todos son paralelizables, se puede utilizar la sentencia `#pragma acc parallel loop collapse (nº de bucles)`, como se muestra en el Algoritmo 2.

Algorithm 2 Uso de collapse en OpenACC

```
1: #pragma acc parallel loop collapse(2)
2: for i = 0 to N do
3:     for j = 0 to M do
4:         **Codigo**
5:     end for
6: end for
```

- Cuando se invoca a una función de una librería externa (por ejemplo, `cublasDgemm` de cuBLAS) y hay que especificar variables presentes en el dispositivo necesarias para ha-

cer la llamada se utiliza la sentencia `#pragma acc host_data use_device(variables)`, representado en el Algoritmo 3.

Algorithm 3 Llamadas a librerias externas en OpenACC (cublasDgemm)

```
1: #pragma acc host_data use_device(A,B,C)
2: {
3: cublasDgemm(handle, transa, transb, m, n, k, alpha, A, lda, B, ldb, beta, C, ldc)
4: }
```

En este trabajo siempre se han utilizado las sentencias `seq` y `parallel`, excepto cuando es necesario usar una sentencia `kernels` para indicar que un bloque de código se ejecute en el *device*, pero nunca se usa para delegar la decisión de paralelización al compilador. A continuación se muestra la parte más relevante de un código de suma de vectores en OpenACC para ilustrar algunos de los pragmas anteriormente mencionados:

Algorithm 4 Ejemplo de suma de vectores en OpenACC

```
1: int main() {
2:     // Declaracion e inicializacion de los vectores
3:     ...
4:     // Mapeo de variables haciendo uso de data
5:     #pragma acc data copyin(a[0:n],b[0:n]), copyout(c[0:n])
6:     {
7:         // Expresion de paralelismo
8:         #pragma acc parallel loop
9:         for(i=0; i<n; i++) {
10:            c[i] = a[i] + b[i];
11:        }
12:    }
13:    // Liberacion de memoria
14:    ...
15: }
```

3.4. OpenMP

OpenMP[3] es una interfaz de programación de aplicaciones (API) portable y flexible. Definida por un gran conjunto de compañías hardware y software, su objetivo es ser compatible con un amplio conjunto de dispositivos heterogéneos. Al igual que OpenACC, es multilenguaje y se basa en directivas, aunque no ofrece al programador distintos niveles de abstracción.

En cuanto a las sentencias de las que dispone el programador, son similares a OpenACC, pero con pequeñas diferencias de formato como el uso de `#pragma omp` en vez de `#pragma acc` o `for` en lugar de `loop`. Algunos de los cambios más significativos son los siguientes:

- Una de las muchas formas de realizar movimientos de memoria se hace con `#pragma omp target data map(...)`. Dependiendo de la dirección de la operación de memoria, map se usa en combinación con `to`, `from` o `tofrom`. Este pragma asocia una variable presente en el *host* a su copia en el *device* de cara a otras funcionalidades de OpenMP mientras se permanezca dentro de la región abarcada por el pragma. La línea 5 del Algoritmo 8 contiene un ejemplo de uso de este pragma.

- En OpenMP se utilizan las sentencias `#pragma omp single` y `#pragma omp parallel` para especificar secciones de cómputo a realizar de forma secuencial o paralela, respectivamente. Es importante destacar que el pragma `targets` es necesario para indicar el *offloading* al dispositivo. Otros pragmas como `teams distribute` potencial el paralelizado de la sección abarcada por los primeros pragmas. En caso de bucles anidados en donde todos son paralelizables, en los internos solo es necesario poner `#pragma omp for`. En la línea 8 del Algoritmo 8 se puede ver un ejemplo de este pragma.

- La función que tiene `#pragma acc kernels` en OpenACC se puede conseguir en OpenMP con la sentencia de `#pragma omp target`. En la línea 8 del Algoritmo 8 se puede observar un ejemplo de uso de este pragma.

[3] https://www.openmp.org/

- En OpenMP también se puede poner `collapse` en caso de que haya varios bucles anidados con la sentencia `#pragma omp target teams distribute parallel for collapse(nº de bucles)`, mostrada en el Algoritmo 5.

Algorithm 5 Uso de collapse en OpenMP

```
#pragma omp target teams distribute parallel for collapse(2)
for i = 0 to N do
    for j = 0 to M do
        **Codigo**
    end for
end for
```

- Para las llamadas a librerías externas (por ejemplo, `cublasDgemm` de cuBLAS) y comunicar al compilador el uso de variables en el dispositivo se pueden utilizar las sentencias `#pragma omp target variant dispatch use_device_ptr()` o `#pragma omp target data use_device_ptr()`, dependiendo de la función a invocar, representada en el Algoritmo 6 .

Algorithm 6 Llamadas a librerias externas en OpenMP (cublasDgemm)

```
#pragma omp target variant dispatch use_device_ptr(A,B,C)
{
cublasDgemm(handle, transa, transb, m, n, k, alpha, A, lda, B, ldb, beta, C, ldc)
}
```

- En caso de querer copiar una variable declarada dinámicamente haciendo uso de punteros, se utiliza la función de librería `omp_target_memcpy()`, que permite hacer operaciones de memoria en ambas direcciones entre *host* y *device*. Para copiar variables unitarias (como enteros o doble precisión) o arrays de tamaño fijo, se puede usar la sentencia `#pragma omp target update [dirección] (variables)`, en donde `[dirección]` puede ser `to` o `from`, ilustrada en el Algoritmo 7.

Algorithm 7 Uso de sentencia *update* en OpenMP

```
1: // Actualiza la variable var en el host con el valor del dispositivo
2: #pragma omp target update from(var)
```

A continuación se muestra la sección más relevante de un código de suma de vectores en OpenMP para ilustrar algunos de los pragmas anteriormente mencionados:

Algorithm 8 Ejemplo de suma de vectores en OpenMP

```
1: int main() {
2:     // Declaracion e inicializacion de los vectores
3:     ...
4:     // Mapeo de variables haciendo uso de target data
5:     #pragma omp target data to(a[0:n],b[0:n]), from(c[0:n])
6:     {
7:         // Expresion de paralelismo
8:         #pragma omp target teams distribute parallel for
9:         for(i=0; i<n; i++) {
10:            c[i] = a[i] + b[i];
11:        }
12:    }
13:    // Liberacion de memoria
14:    ...
15: }
```

OpenMP Offload es una funcionalidad de oneAPI que permite descargar (*offloading*) cómputo usando OpenMP en dispositivos heterogéneos, como puede ser una GPU de Intel. Algunos de los cambios que conlleva realizar esta implementación incluyen cambios en el *makefile*, el uso de pragmas propios para la descarga de funciones de librerías en los dispositivos, o el uso de una librería concreta en el archivo fuente, entre otros. En este trabajo se han desarrollado versiones para todos los algoritmos que hacen uso de OpenMP Offloading, tanto para GPU como en CPU (principalmente para comparativa de rendimiento).

3.5. SYCL

SYCL[4] es un estándar de programación paralela heterogénea desarrollado por Khronos Group. Pretende converger en gran medida con ISO C++, facilitando así la programación paralela en este lenguaje, incluyendo en el mismo archivo de código el programa principal y los *kernels*. El código se programa a un nivel de abstracción más alto y luego se compila como código OpenCL, obteniendo la sencillez en la programación pero mayor eficiencia en la ejecución, al igual que mayor portabilidad. A pesar de esta simplificación del modelo de programación, los desarrolladores siempre pueden acceder a una versión de bajo nivel del código a través de diferentes *frameworks* o librerías de C/C++.

En este trabajo se ha usado *oneAPI* (y su compilador *DPC++* [16]), que es la implementación de SYCL de Intel. Esta implementación funciona con CPUs y GPUs de Intel, al igual que con GPUs de Nvidia en algunos casos (a través de un *toolchain*[5]). *oneAPI* se compone de los siguientes elementos, diseñados para crear aplicaciones paralelas:

- **Data Parallel C++ (DPC++)**: fue lanzado en 2019, e integra los estándares SYCL y OpenCL con extensiones adicionales.
- **oneCCL**: primitivas de comunicación orientadas al desarrollo de *deep learning* en múltiples dispositivos.
- **oneDAL**: algoritmos centrados en ciencias de datos.
- **oneDNN**: primitivas de alto rendimiento orientadas al desarrollo de *deep learning*.
- **oneDPL**: un complemento al compilador oneAPI DPC++/C++ para programar haciendo uso de las librerías estándar C++, Parallel STL, y extensiones.
- **oneMKL**: librería de funciones matemáticas de alto rendimiento para fines científicos, de ingeniería, y de finanzas.

[4] https://www.khronos.org/sycl/

[5] https://intel.github.io/llvm-docs/GetStartedGuide.html#build-dpc-toolchain-with-support-for-nvidia-cuda

- **oneTBB**: librería especializada en paralelismo en multiprocesadores.
- **oneVPL**: algoritmos optimizados para procesado de vídeo.

Es relevante añadir que en el ecosistema de *oneAPI* existe una herramienta de conversión de código conocida con el nombre de DCPT (Data Parallel Compatibility Tool) que facilita la tarea de portabilidad de códigos CUDA a SYCL. Existen algunos trabajos que evalúan dicha herramienta tomando como caso de uso aplicaciones de campos de conocimiento dispares como la suite de bechmarks de Rodinia [17], alineamiento de secuencias biológicas [18], del aprendizaje automático usando la conocida SVM [19] o al aplicación *easyWave* para simulación de tsunamis [20].

DPC++ es el núcleo de oneAPI. Integra programas escritos en ISO C++ y SYCL que distribuyen la computación entre múltiples dispositivos. La estructura de un programa SYCL es como sigue:

- Al principio del programa, se declara una cola donde se selecciona el tipo de *device* que va a utilizar el programa, con la sentencia `queue my_queue{default_selector{}}`. `default_selector` elegirá el dispositivo disponible más potente; si se quiere elegir un tipo de *device* concreto, se puede cambiar el selector por `cpu_selector` o `gpu_selector`.
- En la declaración de las variables se especifica de forma explícita si la variable declarada se almacenará en el *host* o en el *device*, con las sentencias `malloc_host<tipo>(tamanyo en memoria, my_queue)` y `malloc_device<tipo>(tamanyo en memoria, my_queue)` respectivamente.
- Para realizar cualquier operación relacionada con el dispositivo, se hace uso de la cola declarada al principio, a través de la sentencia `my_queue.submit([&] (handler& h){}`.
- Para realizar copias de variables entre *host* y *device*, se hace uso de la sentencia `h.memcpy(variableDestino, variableOriginal, tamaño en memoria)`, como se observa en el Algortimo 9. Al acabar, se puede de hacer uso de `wait()` para esperar a que

se complete la operación antes de seguir la ejecución del código. La copia de variables es muy relevante a la hora de invocar a funciones de librería que hacen uso de variables almacenadas en el *host* y en el *device* simultáneamente, para mantener la coherencia.

Algorithm 9 Copia de variable SYCL

```
1: my_queue.submit([&] (sycl::handler& h) {
2: h.memcpy(variableDestino, variableOriginal, tamaño en memoria)
3: }).wait();
```

- Para paralelizar secciones de código se hace uso de `parallel_for()` dentro de una sección de cola, representada en el Algoritmo 10. En esta sentencia se puede especificar la dimensionalidad de la paralelización a través del rango y los identificadores.

Algorithm 10 Uso de sentencias paralelizables en SYCL

```
1: my_queue.submit([&](auto &h) {
2: h.parallel_for(sycl::range(rango), [=](auto i) {
3: **Codigo**
4: }
5: }).wait();
```

Otras de las características de DPC++ son:

- **Excepciones asíncronas**: los *kernels* pueden ser ejecutados en diferentes pilas de forma que los errores en una pila no se propaguen al resto del programa. Para este objetivo, las colas de SYCL proporcionan funciones de manejo de errores:

```
static auto exception_handler = [ ](cl::sycl::exception_list eList) {
    for (std::exception_ptr const &e : eList) {
        try {
            std::rethrow_exception(e);
        }
        catch (std::exception const &e) {
```

```
            std::terminate();
        }
    }
};
...
try {
    sycl::queue q(d_selector, exception_handler);
    ...
}
catch (sycl::exception const &e) {
    ...
}
```

- **Buffers y Accessors**: Cuando se usan *buffers*, los datos declarados en el *host* se pasan implícitamente al dispositivo. Para modificar estos datos, se utilizan los *accessors*, en los que se indica el modo de acceso (*read*, *write*, o *read_ write*).

```
sycl::buffer a_buf(a_array);
sycl::buffer b_buf(b_array);
sycl::buffer sum_buf(sum_parallel.data(), num_items);
...
q.submit([&](sycl::handler &h) {
    sycl::accessor a(a_buf, h, read_only);
    sycl::accessor b(b_buf, h, read_only);
    sycl::accessor sum(sum_buf, h, write_only);

    ...
});
```

- **Unified Shared Memory**: es una alternativa a los *buffers* para manejar y acceder a memoria tanto del *host* como del *device*. Los datos se pueden almacenar en el *host*, *device* o en ambos (*shared*).

```
int* a_device = sycl::malloc_device<int> num_items);
int* b_device = sycl::malloc_device<int> num_items);
int* sum_device = sycl::malloc_device<int> num_items);

q.memcpy(a_device, a_array, sizeof(int) * num_items).wait();
q.memcpy(b_device, b_array, sizeof(int) * num_items).wait();
...
// Ejecucion de kernel
...
q.memcpy(sum_array, sum_device, sizeof(int) * num_items).wait();
```

Iconos de OpenACC, OpenMP, y SYCL.

3.6. Librerías

Para el procesamiento de imágenes hiperespectrales se usa un núcleo de procedimientos matemáticos concretos de forma extensa. Estas operaciones matemáticas están contenidas en dos librerías: *BLAS*[6] (del inglés *Basic Linear Algebra Subprograms*, en español "Subprogramas Básicos de Álgebra Lineal") y *LAPACK*[7] (del inglés *Linear Algebra Package*, en español, "Paquete de Álgebra Lineal").

[6]https://netlib.org/blas/
[7]https://netlib.org/lapack/

La librería BLAS es de dominio público y contiene las operaciones de álgebra lineal más comunes como suma de vectores, multiplicación de matrices, o producto escalar. Está optimizada para *hardware* que cuenta con una arquitectura que permita operaciones en punto flotante. A raíz de BLAS surgieron muchas librerías que eran compatibles y compartían su interfaz. **La librería LAPACK** contiene rutinas que resuelven sistemas de ecuaciones lineales, descomposición de valores singulares, y factorizaciones matriciales, entre otros. Esta librería permite tratamiento de matrices tanto en precisión simple como doble. Se diseñó para hacer uso eficazmente de *hardware* basado en memoria cache.

Ambas cuentan con versiones optimizadas paralelas, que han sido usadas en este trabajo en conjunto con los paradigmas de programación paralela.

Las librerías son: cuBLAS y cuSOLVER de Nvidia (parte de CUDA Toolkit), usadas en las versiones OpenACC y OpenMP; y oneMKL de Intel (usada en las versiones SYCL). Estas librerías encajan perfectamente con la filosofía heterogénea de este trabajo, siendo genéricas para cualquier GPU de Nvidia (en el caso de cuBLAS y cuSOLVER) y para cualquier CPU y GPU de Intel (en el caso de oneMKL).

Las librerías de Nvidia requieren una GPU para funcionar, por lo que si se pretende ejecutar los algoritmos paralelos en GPU funcionarán, pero no en una CPU multinúcleo. Para esto se puede usar oneMKL, que funciona con CPUs y GPUs de Intel, y más recientemente, con GPUs de Nvidia (a través de un *toolchain* avanzado)[8]. En este trabajo se ha optado por utilizar ambas alternativas, pudiendo así cuantificar las diferencias entre ellas.

[8]https://oneapi-src.github.io/oneMKL/building_the_project.html

Capítulo 4

Implementaciones

4.1. Paralelismo con imágenes hiperespectrales

4.1.1. Decisiones de diseño

Con los conceptos explicados anteriormente, ya es posible detallar las diferentes implementaciones del trabajo. En este proyecto se han seleccionado tres algoritmos, cada uno perteneciente a una de las etapas del desmezclado espectral en imágenes hiperespectrales, y se han paralelizado en tres paradigmas diferentes: OpenACC, OpenMP y SYCL. Cada uno de estos paradigmas cuenta con varias implementaciones, dependiendo de las librerías externas que se hayan usado (por ejemplo, cuBLAS para optimizar la ejecución en las GPUs de Nvidia).

Estas implementaciones se han desarrollado de la manera más genérica posible, de forma que pueden ser ejecutadas en una variedad de dispositivos heterogéneos. En las pruebas se detallarán los diferentes entornos en los que los códigos han sido ejecutados y los resultados de cada uno, comparándolos con la versión original (secuencial) del código correspondiente y entre las diferentes implementaciones paralelas.

Siguiendo el orden de la Figura 2.5, se presentarán las implementaciones de los algoritmos en orden de las etapas, es decir: primero el algoritmo VD, luego el algoritmo VCA y por último el algoritmo ISRA. A modo de resumen se muestra en la Tabla 4.1 todas las

implementaciones que se han llevado a cabo por cada algoritmo. Además, se puede encontrar los códigos de las implementaciones en el repositorio de GitHub[1]:

Versión	Modelo de programación	Compilador	Target	Librería
Base	C	GCC	CPU	oneMKL
	OpenACC	PGCC	CPU-Multicore	BLAS
	OpenACC	NVC	GPU-NVIDIA	CUBLAS
	OpenMP	NVC	GPU-NVIDIA	CUBLAS
	OpenMP	ICX(Intel)	CPU-Multicore	oneMKL
	OpenMP(Offload)	ICX(Intel)	GPU-Intel	oneMKL
	SYCL	ICX(Intel)	CPU-Multicore/GPU-Intel	oneMKL
	SYCL	ICX(Intel)	GPU-NVIDIA	oneMKL

Tabla 4.1: *Implementaciones llevadas a cabo para cada algoritmo.*

4.2. Etapa 1

4.2.1. Algoritmo VD

VD (del inglés *Virtual Dimensionality*) es un concepto creado para referirse al número de firmas o componentes espectrales distintos en una imagen hiperespectral. Este algoritmo usa el principio de Dirichlet, que establece que dados m huecos y n objetos, si hay más objetos que huecos ($n > m$) entonces habrá que reutilizar uno de los huecos para almacenar varios objetos.

En el ámbito hiperespectral, el algoritmo asume, usando este principio, que toda firma o componente espectral necesita una dimensión propia para su almacenamiento, es decir, que dos firmas espectrales distintas no pueden pertenecer a la misma banda espectral.

Se empezó a desarrollar en 1993, aunque su máximo potencial no fue alcanzado hasta 2003, año en el que se adoptó el nombre de VD. Desde 2004, es usado en muchas aplicaciones

[1]https://github.com/OscarRui/TFG_2122

diferentes [13]. A continuación se ilustra el pseudocódigo utilizado en este proyecto [21]:

Algorithm 11 Pseudocódigo de VD

1: INPUT: Y = [y^1, y^2, ..., y^N], P_{fa}
2: $CM = \frac{(Y^T \cdot Y)}{N-1}$;
3: $VM = \frac{(Y-\overline{Y})^T \cdot (Y-\overline{Y}))}{N}$; $\{\overline{Y} = mean(Y)\}$
4: $\lambda^{CM} = eig(CM)$; {calcular los valores propios de CM}
5: $\lambda^{VM} = eig(VM)$; {calcular los valores propios de VM}
6: dim = 0; {inicializar el número de endmembers}
7: **for** $i = 1$ to L **do**
8: $\sigma_i \cong \sqrt{\frac{2}{N}(\lambda_i^{CM})^2 + \frac{2}{N}(\lambda_i^{VM})^2}$;
9: resolver $P_{fa} = \frac{1}{\sigma_i\sqrt{2\pi}} \int_{-\infty}^{x} e^{\frac{z_i 2}{2\sigma_i^2}} dz_i$ para encontrar x;
10: $diff = \lambda_i^{CM} - \lambda_i^{VM}$;
11: **if** $diff > x$ **then**
12: dim = dim + 1;
13: **end if**
14: **end for**
15: OUTPUT: dim

4.2.2. Análisis del grado de paralelismo aplicable a VD

VD contiene dos bucles que suponen las mayores cargas computacionales. El primer bucle (para la obtención de $\overline{Y}$, en la línea 3 del algoritmo) edita la imagen original para hacer una versión de media cero (*zero-mean image*), representada en el pseudocódigo por $\overline{Y}$. Para ello suma todos los píxeles de cada banda. Tiene dos problemas que originalmente impedían la paralelización: el primero, el uso de una variable *mean* (presente en la obtención de $\overline{Y}$) que no puede ser compartida entre hilos, ya que cada uno asigna valores distintos, produciéndose una condición de carrera que afecta al valor de *mean*, provocando una ejecución impredecible (y por tanto no válida); el segundo, una reducción sobre *mean* dentro del primer bucle interior.

Esto se solucionó convirtiendo *mean* en un array de tamaño *bands* (tantos como i ite-

raciones), de forma que cada hilo puede modificar su propia variable de forma local. La reducción no se puede ejecutar en paralelo de ninguna manera, por lo que ha de permanecer en formato secuencial. Tras estas modificaciones, la creación de $\overline{Y}$ queda de la siguiente manera (se usa la versión OpenACC para ilustrarlo):

Algorithm 12 Primer conjunto de bucles de VD modificado para aumentar la paralelización

```
1: #pragma acc parallel loop
2: for i = 0 to bands do
3:     mean[i] = 0;
4:     #pragma acc loop seq
5:     for j = 0 to N do
6:         mean[i] += (image[(i * N) + j]);
7:     end for
8:     mean[i] /= N;
9:     meanSpect[i] = mean[i];
10:    #pragma acc loop
11:    for j = 0 to N do
12:        image[(i * N) + j] = image[(i * N) + j] - mean[i];
13:    end for
14: end for
```

Se ha conseguido un grado de paralelismo elevado, paralelizando el bucle principal y el último bucle interior para cada i iteración. El segundo bucle, representado en la línea 7 del pseudocódigo, tiene dos problemas: el primero es de nuevo el uso de las variables únicas *sigmaSquareTest* y *sigmaTest*, representadas en la línea 8 del pseudocódigo por la variable σ_i; el segundo es la condición de carrera sobre la variable *count* (en el pseudocódigo, *dim*).

A diferencia del bucle anterior, aunque las variables compartidas se pueden transformar en arrays y solucionar el problema, la condición de carrera no tiene solución paralela. En el bucle anterior, cada hilo i escribe y lee sobre un rango distinto de *image*, por lo que en ningún caso ocurrirá una dependencia de datos entre hilos; sin embargo, en este bucle todos los hilos trabajan sobre los mismos valores de *count*, de forma que sería necesaria una sincronización

total entre todos los hilos para garantizar la coherencia de *count*, haciendo imposible una solución paralela. El bucle (usando la versión OpenACC como ejemplo ilustrativo) queda de esta manera, representando las líneas del 7-14 del Algoritmo 11:

Algorithm 13 Segundo conjunto de bucles de VD con pragmas de paralelización

```
1: #pragma acc kernels
2: #pragma acc loop seq
3: for i = 0 to bands do
4:     sigmaSquareTest = (CovEigVal[i] * CovEigVal[i] + CorrEigVal[i] * CorrEigVal[i]) * 2 / sam-
       ples / lines;
5:     sigmaTest = sqrt(sigmaSquareTest);
6:     #pragma acc loop seq
7:     for j = 1 to FPS do
8:         . . .
9:         // Condicion de carrera para count[]
10:        if (CorrEigVal[i] − CovEigVal[i]) > TaoTest then
11:            count[j - 1]++;
12:        end if
13:    end for
14: end for
```

El cómputo de la matriz de correlación y la matriz de covarianza (representado en las líneas 2 y 3 en el pseudocódigo) y de los vectores propios (en inglés, *eigenvalues*) de estas matrices (líneas 4 y 5 del pseudocódigo) son paralelizables mediante funciones de las librerías de CUDA y oneMKL o mediante expresión directa de paralelismo sobre los bucles oportunos. En el Algoritmo 11 se han coloreado las líneas que representan las zonas paralelizadas. Por lo tanto, exceptuando el bucle anterior, el grado de paralelismo potencial de este algoritmo es alto.

4.2.3. Paralelización y optimización de VD usando OpenACC y cuBLAS

Primero se declaran e inicializan las variables necesarias para usar cuBLAS, aunque en este caso es importante mencionar que también es necesario inicializar variables exclusivas necesarias para usar cuSOLVER, la librería que contiene las llamadas a la función `dgesvd()`.

Los movimientos de memoria se realizan antes de comenzar la ejecución paralela, haciendo uso de los pragmas `copyin` y `copyout`, siguiendo el esquema de ACC en el Algoritmo 4 de la sección anterior con las variables correspondientes.

El primer conjunto de bucles queda como se especificó en la sección anterior, obteniendo un buen grado de paralelismo.De forma seguida, se encuentra la primera invocación a la librería cuBLAS, que queda señalizada con de la misma manera que en el Algoritmo 3 de ACC llamando a la función externa `cublasDgemm`, representada en la línea 3 del Algoritmo 11.

Tras esta invocación se encuentra un doble bucle anidado que puede ser totalmente paralelizado donde representa la línea 2 del pseudocódigo. Aunque se podría paralelizar cada uno de los bucles por separado (con pragmas independientes) se sigue el esquema del Algoritmo 2 haciendo uso del `collapse`.

Seguido de lo anterior se encuentra la primera invocación a la librería cuSOLVER, para la cuál es necesario invocar previamente a otra función (de la misma librería) que se encarga de reservar memoria en el dispositivo (en este caso, GPU) para la ejecución. Es decir, para hacer uso de la versión paralela de la función `dgesvd()` de cuSOLVER es necesario invocar previamente a la función `cusolverDnDgesvd_bufferSize()`, que reserva espacio en el dispositivo para la ejecución, antes de llamar a la función `dgesvd()` se sigue el esquema del Algoritmo 3, seguidamente se encuentra la misma llamada, estas dos llamadas a `dgesvd()` representan en el pseudocódigo las líneas 4 y 5. El segundo conjunto de bucles queda como se ilustró en la sección anterior. Por último, es necesario liberar los *handlers* de cuBLAS y cuSOLVER.

4.2.4. Paralelización y optimización de VD usando OpenMP y cuBLAS

La declaración e inicialización de las variables relacionadas con CUDA es idéntica. El primer cambio se observa en las operaciones de memoria iniciales. En OpenMP se realizan los mismos movimientos con los pragmas `to` y `from`, equivalentes a `copyin` y `copyout` siguiendo el esquema del Algoritmo 8 con las variables correspondientes.

El primer conjunto de bucles queda paralelizado de igual forma pero usando los pragmas propios de OpenMP, teniendo una estructura parecida a la que se vio en el Algoritmo 8 usando tanto pragmas secuenciales como paralelos.

Para invocar a la función `cublasDgemm()` solo es necesario cambiar el pragma principal por el del Algoritmo 6. Para paralelizar el bucle anidado se usa la versión del pragma `collapse` de OpenMP como se muestra en el Algoritmo 5.

De la misma forma, la invocación a las funciones de cuSOLVER es idéntica salvo por el cambio de los pragmas que envuelven a las invocaciones por `#pragma omp target data use_device_ptr`, como se ha visto anteriormente.

Por último, el segundo conjunto de bucles,mostrado en las líneas 7 al 14 del Algoritmo 11, queda paralelizado haciendo uso del pragma `single` dentro del pragma `target`, lo que indica que, dentro del dispositivo (*target*) la ejecución de ese bloque de código será realizada por un único hilo (*single*).

4.2.5. Paralelización y optimización de VD usando OpenMP y oneMKL

El *toolkit* de Intel oneAPI (y por tanto oneMKL) es compatible con OpenMP, y dependiendo del *target* (CPU o GPU), se han elaborado dos versiones:

- **OpenMP Offload y oneMKL**: es la versión para GPU. OpenMP Offload consiste en la capacidad de utilizar pragmas de OpenMP para descargar trabajo en el acelerador (en el caso de este trabajo una GPU de Intel). Los cambios que se han realizado en esta implementación son la adición del pragma `target` en los *kernels* paralelos, la sustitu-

ción de las funciones de las librerías de Nvidia por las de oneMKL, la modificación del *Makefile* con un nuevo compilador, ICX (el compilador usado en las implementaciones de oneAPI y oneMKL) y con varios *flags* nuevos, y la adición de un *include* exclusivo que indica el uso de la funcionalidad de OpenMP Offloading.

- **OpenMP y oneMKL**: para la versión que se ejecuta en CPU solo es necesario modificar el *Makefile* anterior (de ICX) eliminando algunos de los nuevos *flags* añadidos innecesarios. Los cambios de la versión Offload se mantienen excepto la adición del pragma `target` en los kernels, no necesario al no haber GPU.

4.2.6. Paralelización y optimización de VD usando SYCL y oneMKL

En SYCL las variables se almacenan explícitamente en memoria del *host* o del *device*. En su declaración, hay que indicar la cola que se usará para manejarlas con la sentencia `sycl::malloc_device<tipo>(tamanyo en memoria, my_queue)`. Hay que tener en cuenta sobre qué variables se trabaja en cada momento para mantener la coherencia de los datos.

En el caso del primer conjunto de bucles, SYCL no permite de forma manual la traducción directa de dos bucles anidados paralelizables habiendo más código de por medio (en caso de haber dos bucles anidados únicamente, sí se puede paralelizar como se vio anteriormente con el pragma `collapse`). Por esta razón, se pierde la paralelización del bucle interior. Todos los demás bucles del algoritmo que han sido paralelizados siguen el formato del Algoritmo 10 visto anteriormente.

Como SYCL ha sido desarrollado para ser lo más compatible y parecido posible a C++, la invocación a las funciones de oneMKL (que sustituye a las librerías de CUDA) se realizan dentro del mecanismo tradicional de control de excepciones *try-catch-finally*. La sincronización se lleva a cabo mediante el uso de `wait()`.

La paralelización de los dos bucles directamente anidados que en los otros paradigmas fue realizada con el pragma `collapse` se programa en SYCL siguiendo el Algoritmo 10, cambiando el rango a dos con sus correspondientes variables.

Para la invocación de `gesvd()` es necesario invocar previamente a la función que calcula el

espacio necesario en memoria para la ejecución. En SYCL, esto se lleva a cabo con la función `gesvd_scratchpad_size()`. Después de hacer la llamada anterior, ya se puede llamar a la función `gesvd()` correctamente usando el mismo mecanismo *try-catch-finally*.

Por último, para el segundo conjunto de bucles, como es totalmente secuencial se ha optado por transferir las variables con las que se ha trabajado en el dispositivo al *host*, y ejecutar el bloque de código en el host. De esta manera, el resultado se almacena directamente en una variable presente desde el primer momento en el *host* y no es necesario realizar transacciones de memoria adicionales. Para terminar, se libera la memoria de todas las variables relacionadas con SYCL.

4.2.7. Paralelización y optimización de VD usando OpenACC y BLAS

Para obtener una versión ejecutable en CPU multinúcleo, se eliminan todas las variables relacionadas con CUDA. Las invocaciones a cuBLAS y cuSOLVER se sustituyen por las versiones del código original (de las librerías BLAS y LAPACK respectivamente). Los pragmas se mantienen intactos. Por último, se cambia el *flag* del compilador que indica que el código se ejecutará en CPU:

`-ta=nvidia` se sustituye por `-ta=multicore` en OpenACC

4.3. Etapa 2

4.3.1. Algoritmo VCA

VCA (del inglés Vertex Component Analysis) es un método para la extracción de cada *endmember* a partir de los datos de la imagen hiperespectral y sabiendo la cantidad aproximada de *endmembers*, obtenida en la etapa anterior. Con el número de componentes puros, VCA estima tanto los espectros puros como los mapas de concentración de los componentes encontrados [22] mostrado en el Algoritmo 14.

Algorithm 14 Pseudocódigo de VCA

INPUT: R = [r_1, r_2, ..., r_N]

SNR$_{th}$ = 15 + 10log$_{10}$(p) **dB**

if $SNR > SNR_{th}$ **then**

  d:= p;

  X := U_d^TR; {U_d obtenida por SVD}

  u := mean(X); {u es 1 x d vector}

  $[Y]_{:,j} := [X]_{:,j}^T / ([X]_{:,j}u)$; {proyeccion proyectiva}

else

  d:= p - 1;

  $[X]_{:,j} := U_d^T$ ($[R]_{:,j}$ $\bar{r}$); {U_d obtenida por PCA}

  c := arg max$_{j=1,...,N}$||$[X]_{:,j}$||

  C := [c|c| ... |c]; {C es 1 x N vector}

  $Y := \begin{bmatrix} X \\ C \end{bmatrix}$;

end if

A := [e_u|0| ... |0]; { e_u := [0,...,0,1]^T y A es p x p matriz}

for i := 1 to p **do**

  w := rand(0,I_p); {w es vector de covarianza Gaussiano de media cero I_p }

  f := $\frac{(I-(AA^*)w)}{(||(I-AA^*)w||}$; {f es un vector ortonormal al subespacio generado por $[A]_{:,1:i.}$}

  v = f^TY;

  k:= arg max$_{j=1,...,N}$ ||$[v]_{:,j}$||; {encuentra el extremo de la proyeccion}

  $[A]_{:,i} := [Y]_{:,k}$;

  $[indice]_i$:= k; {almacena el indice de píxeles}

end for

if $SNR > SNR_{th}$ **then**

  $\hat{M}$:= U_d[X]$_{:,indice}$; {$\hat{M}$ es L x p matriz de mezcla estimada}

else

  $\hat{M}$:= U_d[X]$_{indice}$ + $\bar{r}$; {$\hat{M}$ es L x p matriz de mezcla estimada}

end if

4.3.2. Análisis del grado de paralelismo aplicable a VCA

La implementación de VCA escogida en este trabajo contiene nueve conjuntos de bucles que suponen las mayores cargas computacionales. El primer conjunto (representado en la línea 5 del Algoritmo 14) contiene un problema en el primer bucle con la variable r_m (no presente en el pseudocódigo), en donde se da una condición de carrera, en este caso no tiene solución paralela, ya que no se garantiza la coherencia de la variable.

El segundo conjunto de bucles (que toma lugar entre las líneas 5 y 6 del pseudocódigo) contiene un bucle donde varias variables se van actualizando en forma de reducción. Es necesario usar los pragmas especializados de cada lenguaje para tratar las reducciones y mantener la coherencia de los resultados. Las variables afectadas son *sum1*, *sum2* y *mult* (variables auxiliares presentes en la implementación). El tercer conjunto de bucles (línea 6 del pseudocódigo) puede ser paralelizado. El cuarto conjunto de bucles (representado en la línea 11 del pseudocódigo) también es paralelizable.

El quinto conjunto de bucles (que forma parte de la línea 18 del pseudocódigo) se encuentra ante otra condición de carrera en este caso con la variable *maxi*, ya que al estar realizando una asignación la misma variable unitaria puede dar lugar a un comportamiento impredecible, no alcanzando el resultado adecuado al finalizar el bucle, lo cual hace imposible la paralelización. Tanto el sexto conjunto de bucles con la variable *rank* como el séptimo conjunto de bucles (ambos pertenecientes a la línea 18 del pseudocódigo) con la variable *sum1*, tienen el mismo problema que los anteriores.

El octavo conjunto de bucles (donde también pertenece a la línea 18 del Algoritmo 14) se encuentra con el problema de que se utiliza *sqrt* para la obtención del resultado en la variable *f*, ya que a la hora de paralelizarlo el compilador no lo soporta, por lo cual es obligatorio dejarlo de forma secuencial.

El último conjunto de bucles (línea 20 del pseudocódigo) tiene dos problemas: el primero es la existencia de una condición de carrera, pero esta vez por la variable unitaria *sum2* (variable auxiliar en la implementación), el otro problema ocurre en una zona del código en donde se edita la variable *sumxu* (en el pseudocódigo, *v*) con sucesivas multiplicaciones, y el

compilador no soporta una reducción de este tipo de operador. Estos problemas hacen que sea obligatorio dejar el conjunto de forma secuencial.

Aunque parezca que muchos conjuntos de bucles no se pueden paralelizar, VCA es un programa de gran tamaño y la gran mayoría de las regiones de código y funciones (incluyendo todas las aquí no nombradas, como las regiones representadas por las líneas 7, 10, 11, 13, 17, 19, 21 y 22 del Algoritmo 14) sí se pueden paralelizar al completo.

4.3.3. Paralelización y optimización de VCA usando OpenACC y cuBLAS

Para empezar este paralelismo se debe declarar e inicializar las variables para el uso de cuBLAS, como se utiliza la función `dgesvd()` también se debe inicializar las variables necesarias para poder usar cuSOLVER.

Para que las variables se puedan usar en el device se deben pasar antes de empezar todas aquellas variables necesarias para que se puedan paralelizar utilizando pragmas de `copyin` y `copy` como se vio en el Algoritmo 4 con sus respectivas variables.

El primer conjunto que se encuentra ya se especificó en la sección anterior, con ello se obtiene un buen grado de paralelismo, siguiendo el esquema del Algoritmo 4 donde parte de ella si que es paralelizable pero otra no. El siguiente es la primera llamada a cuBLAS, se muestra en el Algoritmo 3 donde se llama dentro de la sentencia a la función `cublasDgemm`. Todos los bucles que pueden ser paralelizados siguen la estructura del Algoritmo 4.

A continuación se encuentra la primera invocación de la librería de cuSOLVER de la implementación, perteneciente a la línea 5 del Algoritmo 14; antes de la llamada es necesario invocar previamente a otra función de la misma librería que se encarga de comprobar la memoria necesaria en el dispositivo, `cusolverDnDgesvd_bufferSize()`, después de ello se puede llamar a la función `cusolverDnDgesvd`, para todas las llamadas a estas dos funciones se ha tenido que seguir la misma estructura de pragmas mencionada anteriormente.

Se han paralelizado los bucles anidados utilizando el pragma `collapse`, siguiendo la estructura del Algoritmo 2. En los bucles que no se han podido paralelizar, vistos en la

sección anterior, se sigue el esquema del Algoritmo 1 y la parte secuencial del Algoritmo 4.

En la siguiente llamada a la función de la librería cuSOLVER de la implementación (que forma parte de la línea 10 del pseudocódigo), no es necesario invocar de nuevo a la función `cusolverDnDgesvd_bufferSize()`, ya que el tamaño necesario en la memoria del dispositivo será el mismo, lo que hace que las siguientes llamadas sean más eficientes en comparación a la primera.

Se encuentra un bucle en el pseudocódigo, más concretamente la línea 16, que no se ha paralelizado ya que dentro hay llamadas a las librerías tanto de cuSOLVER como de cuBLAS haciendo imposible la paralelización, tampoco se va a poder aplicar ningún pragma en ese bucle, ya que si se ponen, no se puede llamar a las librerías siguiendo los esquemas anteriores.

Ya dentro del bucle, se encuentran dos bucles separados pero ambos se pueden paralelizar sin ningún problema (uno perteneciente a la línea 17 y el otro bucle pertenece a la línea 18 del pseudocódigo), en el primer bucle, el valor que tiene la variable w es determinada por la función `rand()` la cual da un número aleatorio; para hacer la experimentación con un número fijo se ha optado por sustituir la llamada a la función `rand()` por el número 16.000 para que todas las pruebas sean iguales, como se ve en el Algoritmo 15.

Algorithm 15 Uso de pragmas de OpenACC en VCA

```
1: #pragma acc parallel loop
2: for j = 0 to targets do
3:     // Se sustituye rand() por un valor fijo, 16000
4:     w[j] = 16000 % lmax;
5:     w[j] /= lmax;
6: end for
7:
8: #pragma acc parallel loop
9: for j = 0 to targets * targets do
10:    A2[j] = A[j];
11: end for
```

Después de estos dos bucles se encuentra una llamada a la librería cuSOLVER seguida

de dos bucles, después, otro par de bucles anidados, y una llamada a la librería cuBLAS. El primer y segundo bucle es el explicado en la sección anterior, concretamente el quinto y sexto conjunto, respectivamente, todos estos bucles pertenecen a la línea 18 del pseudocódigo.

Al final, se encuentran las últimas llamadas a la librería cuBLAS de la implementación (representada en la línea 19 del pseudocódigo), las cuales después de cada invocación, como en todos los demás casos, llevan a cabo una sincronía para asegurar un correcto funcionamiento, al igual que los últimos conjuntos de bucles donde son las líneas 21 y 22 respectivamente del Algoritmo 14. Antes de finalizar el programa hay que liberar los *handlers* tanto de la librería cuBLAS como cuSOLVER.

4.3.4. Paralelización y optimización de VCA usando OpenMP y cuBLAS

Al empezar este programa hay que declarar las variables de CUDA como se hizo en OpenACC ya que se van a utilizar las mismas librerías que en la sección anterior, respecto a la declaración de memoria, esta vez se debe de utilizar los pragmas de `to` y `tofrom`, que son equivalentes en OpenACC a `copyin` y `copy` siguiendo la estructura del Algoritmo 8 pero con sus respectivas variables.

El primer conjunto de bucles es igual a como se vio en las secciones anteriores que forma parte de la línea 5 del pseudocódigo, pero esta vez se hace con pragmas de OpenMP, adaptándose al esquema que se sigue en el Algoritmo 8. Después de este conjunto se encuentra la primera llamada a la librería de cuBLAS donde solo cambia el pragma principal que envuelve toda la llamada, visto en el Algoritmo 6.

Siguiendo con la implementación cuando todo es paralelizable y no está anidado en OpenMP se hace como en el Algoritmo 8. Para aquellos que tengan bucles anidados se sigue el Algoritmo 5, haciendo este que sea más óptimo.

En la parte de las librerías de cuSOLVER lo único que cambia respecto a OpenACC es el principal pragma que hay que se sustituye por `pragma omp target data use_device_ptr (variables involucrados)` como se ha visto con la librería de cuBLAS anteriormente. A

partir de este momento solo se comentará aquellos bucles que no se han podido paralelizar al completo y se explicará el cambio entre OpenACC a openMP ya que los demás son iguales a la sección anterior pero con los cambios de pragmas explicados.

El segundo conjunto de bucles que toma lugar entre las líneas 5 y 6 del Algoritmo 14, no es completamente paralelizable (que involucra a las variables sum1, sum2 y mult) sigue el esquema del Algoritmo 8, donde se hace uso del pragma `single`.

4.3.5. Paralelización y optimización de VCA usando OpenMP y oneMKL

Al utilizar OpenMP con la librería de oneMKL para CPU multinúcleo, el intercambio de datos entre la memoria del host y el device desaparece, lo que provoca que el rendimiento mejore ya que no se va a invertir tiempo en la copia de variables en el dispositivo o en actualizar las variables del *host*.

Los pragmas utilizados son única y exclusivamente para el uso de paralelismo, si algún conjunto de bucles no puede ser paralelizado no se pone ningún tipo de pragma especificando que no es posible su paralelización, sino que se deja como su versión serie. En caso de usar OpenMP con la librería oneMKL para GPU, se habla de "OpenMP Offloading", cuyas principales diferencias son las siguientes:

- Al hacer uso de una GPU sí hay que tener en cuenta los traspasos de datos entre *host* y *device*, esto afectará negativamente al rendimiento, por lo que es importante tener las variables actualizadas en la memoria correspondiente (especialmente en este algoritmo, que tiene una cantidad elevada de variables).

- Otra diferencia a destacar es la llamada a las funciones oneMKL, que se hacen con la sentencia `#pragma omp target variant dispatch use_device_ptr(...)` como se ve en el Algoritmo 6.

- Los últimos cambios relevantes corresponden al cambio de varios flags en el *Makefile*, que indican la presencia de OpenMP como también el uso de otros *includes* diferentes

para especificar el uso de OpenMP Offload.

4.3.6. Paralelización y optimización de VCA usando SYCL y oneMKL

En esta sección las variables se tienen que hacer igual que en la etapa anterior, es decir, las variables se almacenan explícitamente en memoria del *host* o *device*. Es importante declarar la cola que se va a usar para que el manejo de las operaciones de memoria, al igual que ser cuidadosos en la coherencia de las variables en ambas memorias en todo momento, como se ve en el Algoritmo 9. Algunas de las declaraciones de las variables se muestran en el Algoritmo 16.

Algorithm 16 Declaración de algunas variables de VCA en SYCL

```
1: // Memoria en el device
2: double* endmembers = sycl::malloc_device<double>(targets*bands, my_queue);
3: double* svdMat = sycl::malloc_device<double>(bands*bands, my_queue);
4: double* sumxu = sycl::malloc_device<double>(lines_samples, my_queue);
5: double* Utranstmp = sycl::malloc_device<double>(targets*targets, my_queue);
```

El primer bucle que se encuentra en la implementación sigue el esquema que se ve en el Algoritmo 10. En aquellos puntos donde no es posible paralelizar de forma idéntica a otro paradigma se ha optado por la opción más parecida y que ofrezca el mejor rendimiento, como en el caso del primer bucle, en donde se paraleliza la parte que supone el mayor coste computacional (el bucle principal) mientras que el interior queda secuencial.

En SYCL las llamadas a las librerías tanto cuBLAS como cuSOLVER se sustituyen por las librerías de oneMKL, en este caso a la función de `gemm()`, las sincronizaciones se hacen mediante el uso de `wait()`, y hay control de errores con el mecanismo *try-catch-finally*.

Como se hizo en las versiones anteriores para la invocación de la función `gesvd()`, es necesario el cálculo de memoria que se necesita para la ejecución, después de ello, se puede llamar a la función de forma muy parecida a la invocación de la función `gemm()`. Al igual que en todas las demás llamadas, hay un control de errores *try-catch-finally*.

Para aquellos bucles que están anidados se hace `collapse` como se ha visto en OpenMP

pero en SYCL se hace siguiendo el esquema del Algoritmo 10 (en la parte del rango se debe poner que son dos con sus variables correspondientes y dimensiones del bucle), después de ello con la variable *index[]* se referencia a las variables elegidas dentro del bucle. Por último antes de finalizar el programa hay que liberar la memoria de todas aquellas variables que tengan que ver con SYCL.

4.3.7. Paralelización y optimización de VCA usando OpenACC y BLAS

Para conseguir un ejecutable del algoritmo VCA en CPU multinúcleo, se sustituyen las llamadas de Nvidia (que requieren de una GPU) por las equivalentes de las librerías de BLAS y LAPACK, para que puedan ser ejecutadas en CPU. Todas aquellas variables relacionadas con CUDA se eliminan y los pragmas se dejan intactos.

Por último, se cambia el *flag* del compilador que indica que el código se ejecutará en CPU:

`-ta=nvidia` se sustituye por `-ta=multicore` en OpenACC

4.4. Etapa 3

4.4.1. Algoritmo ISRA

ISRA (del inglés *Image Space Reconstruction Algorithm*) es un algoritmo para la de convolución de imágenes astronómicas. La de convolución consiste en realizar operaciones matemáticas para recuperar o restaurar datos que se han visto degradados. ISRA es uno de los algoritmos más utilizados para estimar abundancia en el estudio de imágenes hiperespectrales [23]. Su pseudocódigo se muestra en el Algoritmo 17 (junto a la Ecuación 4.1, que representa los términos "numerador" y "denominador" utilizados en el pseudocódigo).

$$\widehat{\Phi}^{k+1} = \widehat{\Phi}^{k} * (\frac{\mathbf{E}^T * x}{\mathbf{E}^T * \mathbf{E} * \widehat{\Phi}^{k}}) \tag{4.1}$$

Algorithm 17 Pseudocódigo de ISRA para desmezclar un vector de píxeles **x** usando un conjunto de **E** *endmembers*

```
1: // Para cierto numero de iteraciones
2: for k = 0 to iters do
3:    // Para todos los endmembers
4:    for j = 0 to p do
5:       // Para todas las bandas espectrales
6:       for i = 0 to n do
7:          numerator = numerator + E[i][j] * x[i];
8:          for s = 0 to p do
9:             dot += E[i][s] * Φ̂[s];
10:         end for
11:         denominator += dot * E[i][j];
12:         dot = 0;
13:      end for
14:      // Calcular la nueva Φ̂ (vector de abundancia)
15:      Φ̂[j] *= (numerator/denominator);
16:      numerator = 0;
17:      denominator = 0;
18:   end for
19: end for
```

4.4.2. Análisis del grado de paralelismo aplicable a ISRA

La carga computacional de ISRA reside en su mayoría en las múltiples multiplicaciones de matrices que contiene (principalmente en forma de invocaciones a las funciones `dgemm`). Estas multiplicaciones paralelizables están representadas en las líneas 7, 9, 11 y 15, coloreadas en el pseudocódigo. Estas funciones son sustituibles por su alternativa optimizada (tanto en Intel como Nvidia).

ISRA también cuenta con dos bucles: el bucle exterior o principal (en el pseudocódigo, la

línea 2) no puede ser paralelizado, ya que realiza siempre las mismas operaciones matriciales sobre los mismos datos, de forma que se necesita mantener el orden de ejecución original para evitar incoherencias, ya que la iteración *k+1* utiliza los datos de la iteración *k*, como se puede ver en la Ecuación 4.1. El bucle interior (en el pseudocódigo representado en la línea 6) sí puede ser paralelizado. Aunque ISRA no contenga tantas regiones de código explícitamente paralelizables como los otros algoritmos, la optimización potencial que ofrece el uso de librerías externas paralelas de CUDA y oneMKL es elevada.

4.4.3. Paralelización y optimización de ISRA usando OpenACC y cuBLAS

Lo primero es declarar e inicializar las variables necesarias para las invocaciones a cuBLAS. Para el uso de cuBLAS es necesario crear una *stream*, que es la secuencia de operaciones que se realizan en la GPU o dispositivo. Después, se crean uno o varios *handler*, que son punteros que contienen el contexto de la librería. Estos *handlers* deben ser pasados como parámetro a todas las funciones de la librería y ser borrados al final del programa. También se declara una variable para control de errores.

Antes de empezar la región potencialmente paralela, se realizan las transacciones de memoria entre *host* y *device* correspondientes. Para esto se usa la sentencia `#pragma acc data` seguida de `create`, `copy`, `copyin`, o `copyout`, dependiendo de si se quiere solamente reservar espacio en el dispositivo, copiar la variable con su valor original en el *host* a la memoria del dispositivo al principio de la región paralela y luego traerla de vuelta al *host* al finalizar la región, solamente copiar la variable con su valor original en la memoria del dispositivo al comienzo de la región paralela, o reservar el espacio en el dispositivo y luego copiar la variable al *host* con el valor que tenga en el dispositivo al finalizar la región paralela, respectivamente.

A la hora de invocar a las funciones de cuBLAS es necesario indicar al compilador que use las variables almacenadas en memoria del *device*, para evitar incoherencias. Esto se lleva a cabo con la sentencia `#pragma acc host_data use_device(lista_de_variables)`, en

donde `lista_de_variables` contiene las variables que están dentro del dispositivo, dentro de esa sentencia se llama a la función `cublasDgemm` que sigue un esquema parecido al Algoritmo 3. Las tres llamadas a la librería cuBLAS corresponden respectivamente a las líneas 7, 9 y 11 del Algoritmo 17. Al final de cada llamada a la librería, es conveniente sincronizar todos los hilos generados, usando la función `cublasGetStream()` y `cudaStreamSynchronize()`.

El bucle interior del código (ilustrado en la línea 15 del Algoritmo 17), puede ser paralelizado con una de las sentencias básicas de OpenACC, `#pragma acc parallel loop`, que indica de forma explícita que el bloque de código abarcado por el pragma sea ejecutado de forma paralela en el *device*. Nótese que cuando se usan pragmas de este tipo no es necesario indicar al compilador que se quiere trabajar con las variables del dispositivo (como en el caso de las invocaciones a cuBLAS), ya que este entiende que, ya que todo el código abarcado por el pragma se ejecutará en el dispositivo, también se usarán las variables almacenadas en este. Por último, se libera la memoria de las variables cuBLAS a través de `cublasDestroy()`.

4.4.4. Paralelización y optimización de ISRA usando OpenMP y cuBLAS

Lo primero es declarar e inicializar las variables necesarias para cuBLAS, de manera idéntica a OpenACC. Luego se realizan las operaciones de memoria, en donde hay que utilizar los pragmas específicos de OpenMP `#pragma omp target data` (en donde `target` indica que se quiere descargar el bloque de código sobre el dispositivo o *device*) seguida de `alloc`, `tofrom`, `to`, o `from`; cuya funcionalidad es similar a los pragmas de OpenACC `create`, `copy`, `copyin`, y `copyout` respectivamente.

A la hora de invocar las llamadas a cuBLAS, el procedimiento es muy similar a OpenACC, usando el pragma `#pragma omp target data use_device_ptr(variables)`, en donde `variables` indica las variables que están en el dispositivo. De igual manera que en OpenACC, al final de cada llamada a la librería cuBLAS se invoca a las funciones de sincronización `cublasGetStream()` y `cudaStreamSynchronize()`.

Por último, el bucle interior del código es paralelizado con el pragma `#pragma omp`

`target teams distribute parallel for`, en donde `teams` genera múltiples hilos para paralelizar la región, `distribute` distribuye esos hilos en la región de código que abarca, y `parallel for` indica que se va a trabajar sobre un bucle. Todos estos pragmas, si son usados juntos, suponen la mayor eficiencia que OpenMP puede ofrecer al paralelizar bucles.

4.4.5. Paralelización y optimización de ISRA usando OpenMP y oneMKL

Como se mencionó en los algoritmos anteriores, oneAPI y oneMKL son compatibles con OpenMP, y dependiendo del *target*, se han elaborado dos versiones:

- **OpenMP Offload y oneMKL**: es la versión para GPU. OpenMP Offload consiste en la capacidad de utilizar pragmas de OpenMP para descargar trabajo en el acelerador (en el caso de este trabajo una GPU de Intel). Los cambios que se han realizado en esta implementación son la adición del pragma `target` en los *kernels* paralelos, la sustitución de las funciones de las librerías de Nvidia por las de oneMKL, la modificación del *Makefile* con un nuevo compilador, ICX (el compilador usado en las implementaciones de oneAPI y oneMKL) y con varios *flags* nuevos, y la adición de un *include* exclusivo que indica el uso de la funcionalidad de OpenMP Offloading.

- **OpenMP y oneMKL**: para la versión que se ejecuta en CPU solo es necesario modificar el *Makefile* anterior (de ICX) eliminando algunos de los nuevos *flags* añadidos innecesarios. Los cambios de la versión Offload se mantienen excepto la adición del pragma `target` en los kernels, no necesario al no haber GPU.

4.4.6. Paralelización y optimización de ISRA usando SYCL y oneMKL

La arquitectura de SYCL es diferente al resto de APIs. En la implementación con SYCL y oneMKL, se eliminan todas las variables y procedimientos relacionados con cuBLAS y se sustituyen por los equivalentes en oneMKL. Además, el primer cambio notable es la creación de una cola que conecte *host* y *device* con `queue my_queue{default_selector{}}`.

`sycl::default_selector{}` seleccionará uno de los múltiples dispositivos disponibles como *device*; dependiendo de los flags de compilación, se seleccionará una CPU o GPU de Intel o una GPU de Nvidia.

Las variables son declaradas especificando si se almacenan en memoria del *host* o del dispositivo (usando `malloc_host` o `malloc_device` respectivamente). La memoria almacenada en el *host* es accesible por ambos, pero la memoria almacenada en el dispositivo no es accesible por el *host*, y requiere una transacción de memoria explícita.

Antes de comenzar la región paralela, se realizan las transacciones de memoria correspondientes del *host* al *device* haciendo uso de `memcpy()`. Tras cada invocación a una función de oneMKL se usa `my_queue.wait()` para sincronizar, de forma equivalente a `cublasGetStream()` y `cudaStreamSynchronize()`.

Por último, para la paralelización del bucle se usa `my_queue.parallel_for(range<1>(lines_samples*targets), [=] (sycl::id<1> j){});`, en donde `range` indica el número de dimensiones de paralelismo, como se vio en el Algoritmo 10. La liberación de memoria se hace a través de la funcion oneMKL `free(variable,cola)`, indicando la variable a liberar y la cola o el contexto al que pertenece.

4.4.7. Paralelización y optimización de ISRA usando OpenACC y BLAS

Esta implementación es similar a las versiones para GPU, pero se han sustituido las llamadas a las librerías de Nvidia (que necesitan una GPU) por las llamadas a las funciones equivalentes en la librería BLAS, que se ejecutan en CPU.

De cara al paradigma, solo es necesario cambiar un *flag* del compilador para indicar que el código será ejecutado en un procesador multinúcleo:

`-ta=nvidia` se sustituye por `-ta=multicore` en OpenACC

Capítulo 5

Analisis de resultados

5.1. Imágenes hiperespectrales sintéticas vs reales

Las imágenes hiperespectrales contienen datos recogidos por los sensores hiperespectrales (explicados en el Capítulo 2), que utilizan espectrómetros de imágenes que son encargados de la obtención de mediciones de bandas espectrales con una resolución espectral alta, algunos ejemplos de estos sensores pueden ser el sensor *AVIRIS* o el sensor *EO-1 Hyperion*. En cambio, las imágenes hiperespectrales sintéticas comprenden datos creados a partir de un algoritmo que parte de firmas espectrales reales, a partir de estos elementos, por cada píxel se crea un mapa de abundancia, al poder crear este tipo de imágenes se da la posibilidad de elegir el número de *endmembers* de la imagen.

En este proyecto se han utilizado tres imágenes reales mostradas en la Tabla 5.1, se indica en diferentes columnas las dimensiones ($samples \times lines \times bands$), el tamaño que ocupa en MB y el tiempo real de procesamiento de cada imagen.

Una de las imágenes usadas es AVIRIS Cuprite, esta imagen mostrada en la Figura 5.1 fue captada en 1997, contiene 224 bandas espectrales donde los espectros tienen entre 400 a 2500 nanómetros, algunas de estas bandas espectrales han sido eliminadas para llevar a cabo un buen análisis de los resultado, en concreto las bandas espectrales 1-3, 105-115 y 150-170 ya que estas contienen baja SNR (signal-to-noise ratio). Para la obtención de resultados

Características de cada imagen				
Nombre	Dimensiones	Tamaño	Num. endmembers	Real-time
Hydice	64×64×169	5,4MB	7	0,066s
Cuprite	350×350×188	43,9MB	19	1,986s
SubsetWTC	512×614×224	134MB	30	5,09s

Tabla 5.1: *Características de cada imagen hiperespectral.*

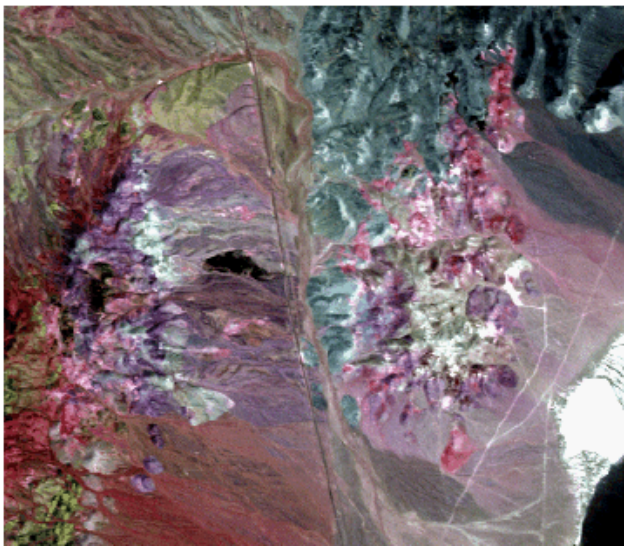

Figura 5.1: *Distrito minero de Cuprite en Nevada obtenido por el sensor AVIRIS.*

se ha optado por coger una fracción de la imagen de 350×350 píxeles con un peso de 50 MBytes. Como se ve en la Figura 5.2, las firmas espectrales obtenidas de esta imagen de la biblioteca U.S. Geological Survey (USGS) suponen un gran interés de cara a los minerales que expuestos como son la calcita, alunita, caolinita, moscovita y buddingtonita.

La segunda imagen usada para hacer pruebas se llama SubsetWTC, fue captada por el sensor AVIRIS como se ve en la Figura 5.3 unos días después de lo sucedido. Esta imagen tiene el objetivo de obtener los daños ocasionados del fuego por el atentado de *World Trade Center*.

En la Figura 5.3 se muestran dos imágenes: a la izquierda, la imagen hiperespectral de

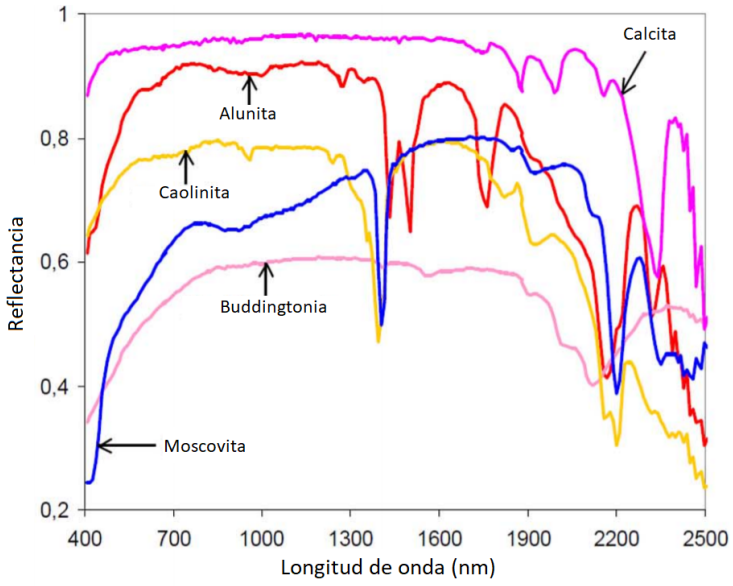

Figura 5.2: *Firmas espectrales de interés de la imagen hiperespectral de Cuprite.*

SubsetWTC, el recuadro rojo marcado en la imagen es donde paso los atentados, la imagen de la derecha representa las zonas cercanas que han sido afectadas por el fuego.

La última imagen hiperespectral se llama Hydice [3], se trata de una imagen que representa un subconjunto de datos de una radiancia forestal. Como se ve en la Figura 5.4(a), esta imagen consta de 64×64 píxeles con 169 bandas espectrales y 15 paneles. En la Figura 5.4(b) se observan 15 destellos blancos, que simbolizan la ubicación espacial de los paneles. La imagen fue adquirida con 210 bandas espectrales y con una cobertura espectral de 0,4 a 2,5 µm. Las bandas 1–3, 101–112, 137–153 y 202–210 se eliminaron antes del análisis debido a sus imperfecciones, ocasionadas por una baja relación señal-ruido (SNR signal-to-noise ratio) o absorción del agua.

Tras ver los resultados que se han obtenido con la imagen Hydice, se ha llegado a la conclusión de que no merece la pena utilizarla en las pruebas de los códigos paralelizados, ya que el ser tan pequeña en tamaño provoca que el tiempo obtenido en serie sea mejor a los obtenidos en *paralelo*, esto se debe al traspaso de datos de *host* al *device* y viceversa, entre otros. Por ejemplo, en el algoritmo VD, usando la imagen Hydice en serie se obtuvo un tiempo de 0,0391 segundos y en paralelo, con OpenACC, se obtuvo 0,0551 segundos, por lo

Figura 5.3: *Imagen hiperespectral de SubsetWTC (izquierda), Ubicación de los incendios (derecha).*

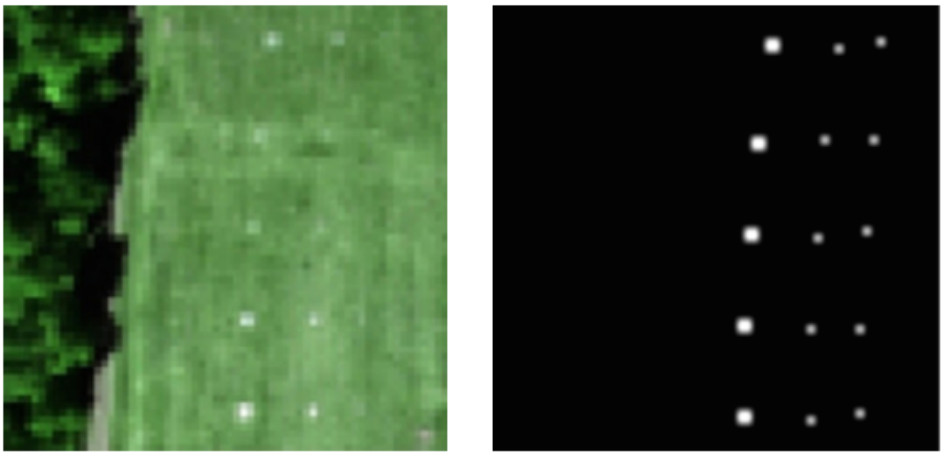

Figura 5.4: *(a) Representación en falso color de la escena hiperespectral HYDICE, (b) Información veraz sobre el terreno asociada.*

explicado con anterioridad, se observa un incremento del tiempo de un 40,92 % del original. Viendo que el *overhead* y el *offloading* consumen más tiempo que ejecutar el código serie para esta imagen, se ha optado por no usarla en las pruebas.

5.2. Sistemas utilizados para las pruebas

La máquina "Volta1", que cuenta con una CPU Intel Xeon Oro 6138, una GPU NVIDIA GeForce RTX 3090 y otra GPU NVIDIA Tesla V100.

- El **Intel Xeon Gold 6138** forma parte de la serie *Skylake*. Cuenta con 20 núcleos a 2 GHz (hasta 3,7 GHz) y permite una cantidad de hasta 40 subprocesos, al contar con la tecnología *Hyper-Threading*. Tiene 27,5 MB de Cache L3.
- La **NVIDIA GeForce RTX 3090** cuenta con tecnología *Ampere* y arquitectura RTX de segunda generación. Posee 10496 nodos CUDA (en inglés, *CUDA Cores*) que operan a una frecuencia de 1,4 hasta 1,7 GHz y cuenta con 24 GB de memoria GDDR6X.
- La **NVIDIA Tesla V100** es una de las GPUs más potentes de NVIDIA, diseñada para *deep learning*, *machine learning*, y computación de altas prestaciones. Su arquitectura es Volta y y cuenta con 5120 CUDA Cores y 640 Tensor Cores.

El DevCloud de Intel ofrece acceso a un cluster remoto que contiene múltiples arquitecturas con las que el programador puede interactuar de forma profunda y simple. Para las pruebas, se han utilizado tres nodos del DevCloud que cuentan con las siguientes configuraciones:

- 1º. **CPU Intel Xeon Gold 6128**: forma parte de la serie *Skylake*. A diferencia del Xeon Oro 6138 de la Volta1, cuenta con 6 núcleos a 3,4 GHz (hasta 3,7 GHz) y permite una cantidad de hasta 12 subprocesos, al contar con la tecnología *Hyper-Threading*. Tiene 19,25 MB de Cache L3.

- 2ª. **CPU Intel Core i9-10920X serie X**: forma parte de la serie *Cascade Lake*. Cuenta con 12 núcleos a 3,5 GHz (hasta 4,6 GHz, aunque puede ser aumentada hasta 4,8 GHz gracias a la tecnología Intel Turbo Boost Max 3.0) y permite una cantidad de hasta 24 subprocesos, al contar con la tecnología *Hyper-Threading*. Tiene 19,25 MB de Intel Smart Cache.

- 3ª. **CPU Intel Xeon E-2176G y GPU UHD Intel P630**: El Xeon E-2176G forma parte de la familia *Coffee Lake*. Al igual que el i9 del nodo anterior, cuenta con 6 núcleos a 3,7 GHz (que puede ser aumentado hasta 4,7 GHz gracias a la tecnología Intel Turbo Boost 2.0) y permite hasta 12 subprocesos (mediante la tecnología *Hyper-Threading*). Cuenta con 12 MB de Intel Smart Cache. Este procesador cuenta con la GPU UHD Intel P630, que cuenta con una frecuencia base de 350 MHz, que puede ser aumentada hasta 1,2 GHz haciendo uso de la característica de frecuencia dinámica.

5.3. Métricas

5.3.1. Calidad

Cada algoritmo tiene una forma específica de medir cuantitativamente su calidad, dependiendo del algoritmo, se utilizan unas medidas u otras. Para la medida de la calidad se ha utilizado la imagen Cuprite.

Siguiendo el orden de las etapas, el objetivo que tiene el algoritmo VD es la obtención del número de *endmembers* que tiene la imagen. Para comprobar su calidad, para cada banda prueba el número de veces que la condición dada por la Ecuación 5.1 falla para una "probabilidad de falsa alarma" dada (Pfa). El autor de la implementación del algoritmo usada ha fijado esta probabilidad, por lo que no es posible llevar a cabo las métricas de calidad.

$$\lambda_i^{CM} - \lambda_i^{VM} \geq 0 \tag{5.1}$$

	Alunita	Buddingtonia	Calcita	Caolinita	Moscovita	Media
VCA	11,99^o	7,17^o	11,04^o	14,01^o	5,45^o	9,93^o

Tabla 5.2: *Valores SAD obtenidos para la imagen real Cuprite.*

La medida de calidad del algoritmo VCA es algo más compleja, ya que se tienen en cuenta los ángulos que forman dos firmas espectrales, cada uno recibe el nombre de *Distancia de Ángulo Espectral* (*Spectral Angle Distance - SAD*). Los valores posibles que tiene este cálculo son de 0^o a 90^o, cuanto más alto sea el valor peor es la firma espectral en comparación a la muestra. Para este cálculo se debe de tener un píxel X en la posición i y j, con la firma espectral que se compara S, el cálculo se realiza como se observa en la Ecuación 5.2. En la Tabla 5.2 se muestra la calidad para la imagen Cuprite del algoritmo VCA.

$$SAD[X(i,j), S_i] = cos^{-1} \frac{X(i,j) \cdot S_i}{|X(i,j)| \cdot |S_i|} \tag{5.2}$$

Para el cálculo de calidad del último algoritmo, ISRA, se puede utilizar el *Error Cuadrático Medio Normalizado* (*Normalized Mean Squared Error* - NMSE) [24]. Cuando ya se tienen los mapas de abundancia de los *endmembers*, se comparan con los mapas de abundancia de las imágenes reales, siendo x la real y $\hat{x}$ la obtenida. Donde $||.||_F$ es la norma de Frobenius, mostrada en la Ecuación 5.3. El valor global que sale para la imagen Cuprite es de 0,0048, en la Figura 5.5 se observa la evolución de los mapas de NMSE para cada uno de los píxeles de la imagen a medida que se aumenta el número de iteraciones en el algoritmo ISRA.

$$NMSE = ||\hat{x} - x||_F^2 / ||x||_F^2 \tag{5.3}$$

5.3.2. Rendimiento

La medida utilizada para cuantificar el rendimiento es el *speedup*, que consiste en comparar dos ejecuciones de la misma tarea (en este caso los algoritmos) ejecutadas sobre la

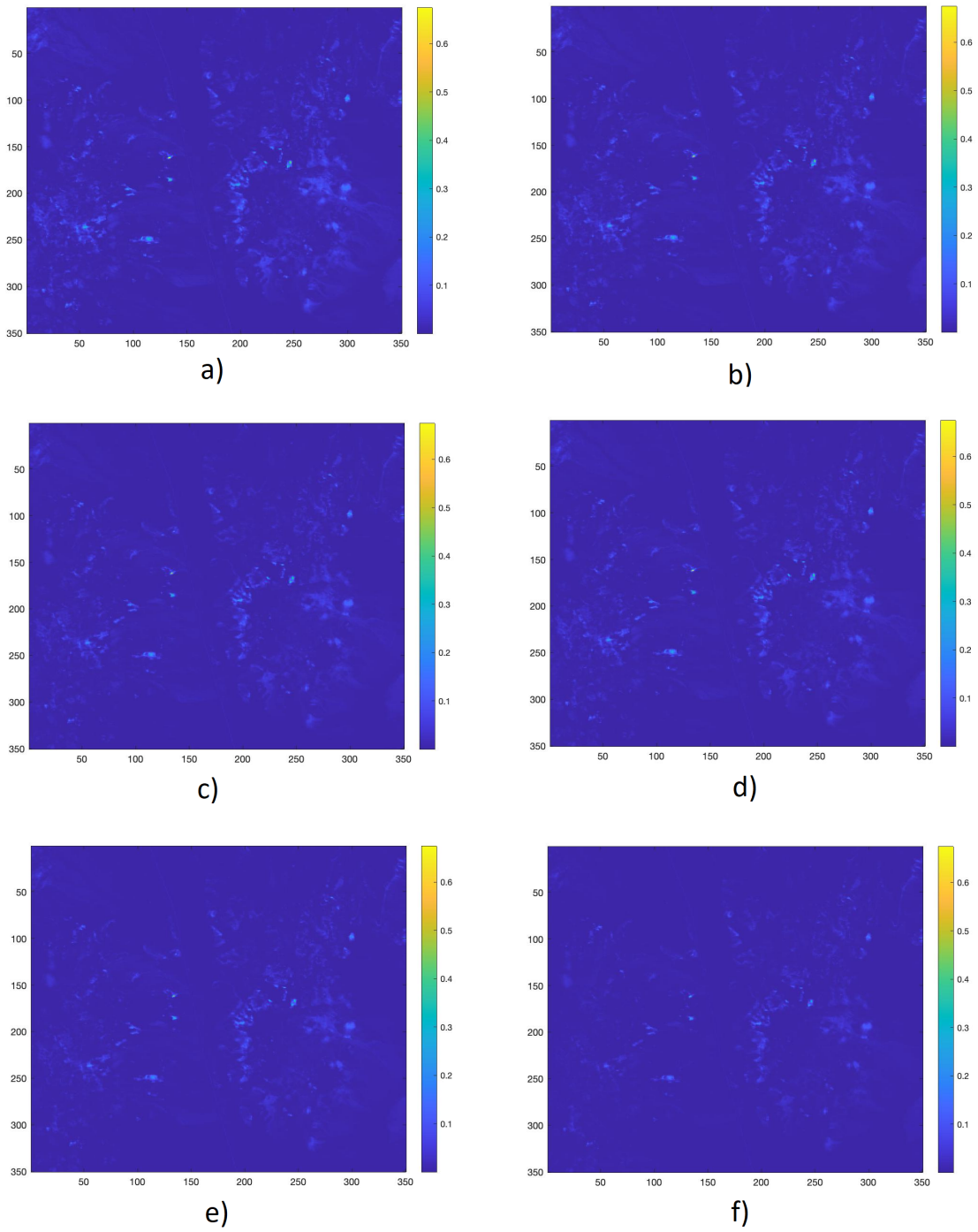

Figura 5.5: *Evolución de la imagen hiperespectral Cuprite con los valores de NMSE por cada píxel. Con un valor global (vg) e iteraciones (i) diferentes.(a)vg = 0,0104 e i = 10, (b)vg = 0,0086 e i = 100, (c)vg = 0,0078 e i = 150, (d)vg = 0,0072 e i = 200, (e)vg = 0,0063 e i = 300, (f)vg = 0,0048 e i = 600.*

misma arquitectura, cada una con diferentes recursos (las distintas implementaciones). El *speedup* se representa mediante la siguiente fórmula:

$$speedup = \frac{tiempo_{serie}}{tiempo_{paralelo}} \tag{5.4}$$

En el caso de este trabajo, se realiza el cociente entre el tiempo original / serie de los algoritmos y el tiempo paralelo de las distintas implementaciones. Para garantizar unas comparativas justas, se ha usado la misma versión de todos los compiladores para todas las implementaciones, las cuales se listan a continuación:

Compiladores y sus versiones	
Compilador	Versión
GCC	8.3.0-6
NVC / PGCC	21.9-0
ICX	2022.0.0.20211123

Tabla 5.3: *Compiladores y sus versiones.*

Las relaciones compilador-implementación se pueden consultar en la Tabla 4.1 expuesta anteriormente. De la misma manera, se han utilizado los mismos flags y opciones de compilación entre todas las implementaciones que usan el mismo compilador.

5.4. Análisis de rendimiento de VD

5.4.1. CPU

En la Tabla 5.4 se ilustran los *speedups* de las diferentes implementaciones de VD en la imagen Cuprite, todas las ejecuciones cuentan exactamente con los mismos parámetros. En la imagen Cuprite se consiguen unos resultados más eficientes en todos los paradigmas utilizados, alcanzando un mejor rendimiento en todas las CPUs. El menor *speedup* obtenido es de 1,49 y el mayor de 2,50, obtenido en la CPU *Intel Core i9-10920X* con el paradigma

VD en máquina CPU (Cuprite)				
Versión	Librería	Target	Speedup	Tiempo (s)
Serie	oneMKL	CPU (Gold 6138)	1,00	0,38
OpenACC	BLAS	CPU (Gold 6138)	2,01	0,19
OpenMP	oneMKL	CPU (Gold 6138)	1,49	0,26
SYCL	oneMKL	CPU (Gold 6138)	1,98	0,19
OpenMP	oneMKL	CPU (Gold 6128)	1,95	0,19
SYCL	oneMKL	CPU (Gold 6128)	1,88	0,20
OpenMP	oneMKL	CPU (i9-10920X)	2,50	0,15
SYCL	oneMKL	CPU (i9-10920X)	2,35	0,16
OpenMP	oneMKL	CPU (Xeon E-2176G)	2,20	0,17
SYCL	oneMKL	CPU (Xeon E-2176G)	2,17	0,17

Tabla 5.4: *Ejecuciones de VD para Cuprite en CPU.*

OpenMP. Se observa que el uso del paradigma OpenMP con la imagen Cuprite, obtiene resultados ligeramente mejores frente a otros paradigmas. Al ser una imagen pequeña, el factor tecnológico no supone una clara diferencia entre unos sistemas y otros.

En la Tabla 5.5 se ilustran los *speedups* de las diferentes implementaciones de VD para la imagen SubsetWTC, todas las ejecuciones cuentan exactamente con los mismos parámetros. Se alcanza un mejor rendimiento en todas las versiones y CPUs. El menor *speedup* obtenido es 1,81 y el mayor 4,64, haciendo uso de la CPU *Intel Xeon Gold 6138* con el paradigma SYCL y oneMKL.

La gran diferencia que ocurre respecto a la anterior imagen, es que el paradigma SYCL en todas las ejecuciones y en diferentes plataformas mejora el rendimiento proporcionalmente al tamaño de la imagen. Esto es debido al uso de la librería oneMKL, que obtiene mejores resultados que la librería BLAS con el paradigma OpenACC, al estar más optimizada.

VD en CPU (SubsetWTC)				
Versión	Librería	Target	Speedup	Tiempo (s)
Serie	oneMKL	CPU (Gold 6138)	1,00	1,25
OpenACC	BLAS	CPU (Gold 6138)	2,24	0,56
OpenMP	oneMKL	CPU (Gold 6138)	1,81	0,69
SYCL	oneMKL	CPU (Gold 6138)	4,67	0,27
OpenMP	oneMKL	CPU (Gold 6128)	2,48	0,50
SYCL	oneMKL	CPU (Gold 6128)	3,71	0,34
OpenMP	oneMKL	CPU (i9-10920X)	2,90	0,43
SYCL	oneMKL	CPU (i9-10920X)	3,75	0,33
OpenMP	oneMKL	CPU (Xeon E-2176G)	2,61	0,48
SYCL	oneMKL	CPU (Xeon E-2176G)	3,28	0,38

Tabla 5.5: *Ejecuciones de VD para SubsetWTC en CPU.*

También, el uso de oneMKL[1] y de modernas plataformas Intel[2] facilita que SYCL alcance un mejor rendimiento.

Para concluir la decisión del paradigma a utilizar, la opción más eficiente para ambas imágenes es el paradigma SYCL y oneMKL, ya que aunque el paradigma OpenMP alcance unos *speedups* ligeramente mejores con la imagen Cuprite, son muy similares a los del paradigma SYCL. Además, es relevante destacar que con la imagen SubsetWTC (más grande) el paradigma SYCL obtiene una gran diferencia de rendimiento frente a los otros dos paradigmas.

[1]https://www.alcf.anl.gov/support-center/training-assets/overview-new-intel-oneapi-math-kernel-library-onemkl-0

[2]https://www.intel.es/content/www/es/es/developer/tools/frameworks/overview.html

VD en máquina GPU (Cuprite)				
Versión	Librería	Target	Speedup	Tiempo (s)
Serie	oneMKL	CPU (Gold 6138)	1,00	0,38
OpenACC	CUDA	GPU (RTX 3090)	2,95	0,13
OpenMP	CUDA	GPU (RTX 3090)	2,34	0,16
OpenACC	CUDA	GPU (Tesla V100)	3,32	0,11
OpenMP	CUDA	GPU (Tesla V100)	2,89	0,13
OpenMP (Offloading)	oneMKL	GPU (G9 UHD P630)	0,17	2,26
SYCL	oneMKL	GPU (G9 UHD P630)	0,12	3,20

Tabla 5.6: *Ejecuciones de VD para Cuprite en GPU.*

5.4.2. GPU

En la Tabla 5.6 se ilustran los *speedups* de las diferentes implementaciones de VD en la imagen Cuprite, todas las ejecuciones cuentan exactamente con los mismos parámetros. En la imagen Cuprite se observan unos resultados eficientes en todos los paradigmas utilizados, alcanzando una mejora de *speedup* en todas las GPUs menos en la GPU integrada de Intel. Esta GPU supone el menor *speedup*, de 0,12; el mayor *speedup* es 3,32, en la GPU dedicada *NVIDIA Tesla V100* con el paradigma OpenACC y librerías CUDA.

Se observa como el uso del paradigma OpenACC, al igual que OpenMP, alcanza rendimientos notables en ambas GPUs dedicadas de la máquina Volta. El uso de la GPU integrada del DevCloud de Intel empeora el rendimiento de la versión secuencial (*speedup* de 0,12). Este resultado es totalmente esperable, y se debe tanto a su factor tecnológico (incomparable al de una GPU dedicada moderna) como a la necesidad de las operaciones de memoria entre CPU y GPU, las cuales no solo empeoran el rendimiento por definición, sino que a su vez se ven acentuadas por el bajo factor tecnológico.

En la Tabla 5.7 se ilustran los *speedups* de las diferentes implementaciones de VD en la imagen SubsetWTC, todas las ejecuciones cuentan exactamente con los mismos parámetros. En la imagen SubsetWTC se observan unos resultados notables en todos los paradigmas

utilizados, alcanzando una mejora de rendimiento en todas las GPUs dedicadas menos en la GPU integrada de Intel, obteniendo el peor *speedup* de 0,33. El mayor *speedup* es de 5,58, ejecutada en la GPU de Volta *NVIDIA Tesla V100* con el paradigma OpenACC.

Usando la imagen SubsetWTC, al igual que en el resto de ejecuciones, se acentúan los resultados anteriores de forma congruente, dejando clara la poca conveniencia de usar la GPU integrada. También, el uso de la GPU *NVIDIA Tesla V100* en cualquier paradigma consigue los mejores rendimientos debido a su gran factor tecnológico. El mejor paradigma a utilizar tanto en la imagen Cuprite como en SubsetWTC, es OpenACC y las librerías de CUDA.

VD en GPU (SubsetWTC)				
Versión	Librería	Target	Speedup	Tiempo (s)
Serie	oneMKL	CPU (Gold 6138)	1,00	1,25
OpenACC	CUDA	GPU (RTX 3090)	4,32	0,29
OpenMP	CUDA	GPU (RTX 3090)	3,29	0,38
OpenACC	CUDA	GPU (Tesla V100)	5,58	0,22
OpenMP	CUDA	GPU (Tesla V100)	4,23	0,29
OpenMP (Offloading)	oneMKL	GPU (G9 UHD P630)	0,39	3,23
SYCL	oneMKL	GPU (G9 UHD P630)	0,33	3,82

Tabla 5.7: *Ejecuciones de VD para SubsetWTC en GPU.*

5.5. Análisis de rendimiento de VCA

5.5.1. CPU

En la Tabla 5.8 se ilustran los tiempos de ejecución de las diferentes implementaciones de VCA en CPU para la imagen Cuprite, todas las ejecuciones cuentan exactamente con los mismos parámetros. VCA cuenta con gran cantidad de variables, al igual que con muchas operaciones de memoria, lo que lo hace el algoritmo menos eficiente cuando se utilizan imá-

VCA en CPU (Cuprite)				
Versión	Librería	Target	Speedup	Tiempo (s)
Serie	oneMKL	CPU (Gold 6138)	1,00	0,91
OpenACC	BLAS	CPU (Gold 6138)	1,75	0,52
OpenMP	oneMKL	CPU (Gold 6138)	2,22	0,41
SYCL	oneMKL	CPU (Gold 6138)	1,07	0,85
OpenMP	oneMKL	CPU (Gold 6128)	1,94	0,47
SYCL	oneMKL	CPU (Gold 6128)	1,44	0,63
OpenMP	oneMKL	CPU (i9-10920X)	2,76	0,33
SYCL	oneMKL	CPU (i9-10920X)	1,65	0,55
OpenMP	oneMKL	CPU (Xeon E-2176G)	2,84	0,32
SYCL	oneMKL	CPU (Xeon E-2176G)	1,82	0,50

Tabla 5.8: *Ejecuciones de VCA para Cuprite en CPU.*

genes de poco tamaño (como Cuprite), ya que el *overhead* de estas operaciones se hace más notorio. Aun así, todas las implementaciones consiguen mejorar el rendimiento secuencial, aunque sea ligeramente, siendo OpenMP el paradigma más rápido. Aun así, cabe destacar que el rendimiento obtenido en todos los paradigmas con la CPU Intel Gold 6138 es menor al de los nodos del DevCloud de Intel.

La Tabla 5.8 muestra los tiempos de ejecución de las diferentes implementaciones de VCA en CPU para la imagen SubsetWTC. Como es observable en el resto de algoritmos, el rendimiento de las versiones paralelas respecto a la versión secuencial mejora con el uso de la imagen SubsetWTC. SYCL es el paradigma que mayor beneficio obtiene a partir de esta imagen, aumentando en gran medida el *speedup* obtenido con la imagen Cuprite, incluso superando al de OpenMP en el nodo del DevCloud de Intel equipado con la CPU *Gold 6128*. De nuevo, es puede ver cómo el rendimiento del Intel Gold 6138 es más similar al de los nodos del DevCloud que en el resto de algoritmos. El mejor paradigma es, para ambas imágenes, OpenMP y oneMKL.

VCA en CPU (SubsetWTC)				
Versión	Librería	Target	Speedup	Tiempo (s)
Serie	oneMKL	CPU (Gold 6138)	1,00	3,34
OpenACC	BLAS	CPU (Gold 6138)	2,49	1,34
OpenMP	oneMKL	CPU (Gold 6138)	4,51	0,74
SYCL	oneMKL	CPU (Gold 6138)	3,48	0,96
OpenMP	oneMKL	CPU (Gold 6128)	2,47	1,35
SYCL	oneMKL	CPU (Gold 6128)	2,57	1,30
OpenMP	oneMKL	CPU (i9-10920X)	3,67	0,91
SYCL	oneMKL	CPU (i9-10920X)	2,98	1,12
OpenMP	oneMKL	CPU (Xeon E-2176G)	3,18	1,05
SYCL	oneMKL	CPU (Xeon E-2176G)	2,93	1,14

Tabla 5.9: *Ejecuciones de VCA para SubsetWTC en CPU.*

5.5.2. GPU

En la Tabla 5.10 se ilustran los *speedups* de las diferentes implementaciones de VCA en GPU, todas las ejecuciones cuentan exactamente con los mismos parámetros. Se observan unos resultados eficientes en todos los paradigmas utilizados, obteniendo una mejora de *speedup* en todas las GPUs menos en la GPU integrada de Intel. Esta GPU supone el menor *speedup*, de 0,28; el mayor *speedup* es de 2,33, en la GPU dedicada *NVIDIA Tesla V100* con el paradigma OpenMP y librerías CUDA.

El uso de la GPU integrada del DevCloud de Intel empeora el rendimiento de la versión secuencial (*speedup* de 0,28). Este resultado es esperable, por los mismos motivos que los vistos en el algoritmo VD. En cambio, el uso de las GPUs dedicadas en la máquina Volta alcanzan unos rendimientos notables en cualquiera de los dos paradigmas, OpenACC y OpenMP.

En la Tabla 5.11 se observan los *speedups* de cada implementación de VCA en GPU para la imagen SubsetWTC, todas las ejecuciones cuentan exactamente con los mismos paráme-

VCA en GPU (Cuprite)				
Versión	Librería	Target	Speedup	Tiempo (s)
Serie	oneMKL	CPU (Gold 6138)	1,00	0,91
OpenACC	CUDA	GPU (RTX 3090)	1,38	0,66
OpenMP	CUDA	GPU (RTX 3090)	1,63	0,56
OpenACC	CUDA	GPU (Tesla V100)	1,72	0,53
OpenMP	CUDA	GPU (Tesla V100)	2,33	0,39
OpenMP (Offloading)	oneMKL	GPU (G9 UHD P630)	0,28	3,27
SYCL	oneMKL	GPU (G9 UHD P630)	0,29	3,11

Tabla 5.10: *Ejecuciones de VCA para Cuprite en GPU.*

tros. Se observan unos resultados notables en todos los paradigmas utilizados, alcanzando una mejora de rendimiento en todas las GPUs dedicadas menos en la GPU integrada de Intel, obteniendo el peor *speedup* de 0,40. El mayor *speedup* es de 3,31, ejecutado en la GPU dedicada de Volta *NVIDIA Tesla V100* con el paradigma OpenMP.

Al igual que en la anterior imagen cabe destacar que el uso de GPU integrada no conviene para este tipo de ejecuciones. La mejora de rendimiento en las GPUs dedicadas se ven beneficiadas proporcionalmente al tamaño de la imagen. El mejor paradigma a escoger es OpenMP con la librería de CUDA.

5.6. Análisis de rendimiento de ISRA

5.6.1. CPU

Como se observa en la Tabla 5.12, todas las implementaciones obtienen unos tiempos de ejecución que mejoran notablemente el rendimiento original de ISRA para la imagen Cuprite. Las versiones OpenACC y OpenMP obtienen rendimientos casi idénticos, mientras que la versión SYCL que hace uso de oneMKL obtiene un rendimiento ligeramente superior.

VCA en GPU (SubsetWTC)				
Versión	Librería	Target	Speedup	Tiempo (s)
Serie	oneMKL	CPU (Gold 6138)	1,00	3,34
OpenACC	CUDA	GPU (RTX 3090)	1,38	2,42
OpenMP	CUDA	GPU (RTX 3090)	2,18	1,53
OpenACC	CUDA	GPU (Tesla V100)	1,82	1,84
OpenMP	CUDA	GPU (Tesla V100)	3,31	1,01
OpenMP (Offloading)	oneMKL	GPU (G9 UHD P630)	0,40	8,43
SYCL	oneMKL	GPU (G9 UHD P630)	0,72	4,66

Tabla 5.11: *Ejecuciones de VCA para SubsetWTC en GPU.*

Como el cómputo del algoritmo reside principalmente en las funciones, los resultados de las versiones OpenACC y OpenMP prueban que los pragmas utlizados en ambas para expresar paralelismo garantizan una eficiencia muy similar. En el resto de sistemas se obtienen unos rendimientos similares, coherentes con el factor tecnológico de cada CPU.

En la Tabla 5.13 se pueden observar los resultados para la imagen SubsetWTC, todas las ejecuciones cuentan exactamente con los mismos parámetros. De nuevo, todas las versiones obtienen sustanciales mejoras en todos los sistemas. Cabe destacar la eficiencia de la versión SYCL y oneMKL en la CPU *Gold 6138*, en donde, como se ha comentado anteriormente, DPC++ aprovecha mejor que cualquier otra versión el gran factor tecnológico de esta CPU Intel. En el resto de sistemas los rendimientos obtenidos son similares. La implementación SYCL y oneMKL ha obtenido el mejor resultado. Aunque cabe aclarar que, excepto en la ejecución en la CPU *Gold 6138* mencionada anteriormente, los resultados de esta versión son semejantes a los otros paradigmas.

5.6.2. GPU

La Tabla 5.14 muestra las ejecuciones de ISRA en GPU para la imagen Cuprite. Todas las ejecuciones alcanzan *speedups* muy similares, siendo muy notable la enorme mejora que

ISRA en CPU (Cuprite)				
Versión	Librería	Target	Speedup	Tiempo (s)
Serie	oneMKL	CPU (Gold 6138)	1,00	11,51
OpenACC	BLAS	CPU (Gold 6138)	7,41	1,55
OpenMP	oneMKL	CPU (Gold 6138)	7,48	1,54
SYCL	oneMKL	CPU (Gold 6138)	8,65	1,33
OpenMP	oneMKL	CPU (Gold 6128)	3,67	3,14
SYCL	oneMKL	CPU (Gold 6128)	4,08	2,82
OpenMP	oneMKL	CPU (i9-10920X)	3,42	3,37
SYCL	oneMKL	CPU (i9-10920X)	3,39	3,39
OpenMP	oneMKL	CPU (Xeon E-2176G)	3,94	2,92
SYCL	oneMKL	CPU (Xeon E-2176G)	3,71	3,10

Tabla 5.12: *Ejecuciones de ISRA para Cuprite en CPU.*

ISRA en CPU (SubsetWTC)				
Versión	Librería	Target	Speedup	Tiempo (s)
Serie	oneMKL	CPU (Gold 6138)	1,00	39,59
OpenACC	BLAS	CPU (Gold 6138)	8,67	4,56
OpenMP	oneMKL	CPU (Gold 6138)	8,23	4,81
SYCL	oneMKL	CPU (Gold 6138)	13,55	2,92
OpenMP	oneMKL	CPU (Oro 6128)	3,57	11,08
SYCL	oneMKL	CPU (Oro 6128)	4,25	9,32
OpenMP	oneMKL	CPU (i9-10920X)	3,91	10,14
SYCL	oneMKL	CPU (i9-10920X)	3,73	10,60
OpenMP	oneMKL	CPU (Xeon E-2176G)	3,73	10,61
SYCL	oneMKL	CPU (Xeon E-2176G)	3,61	10,96

Tabla 5.13: *Ejecuciones de ISRA para SubsetWTC en CPU.*

ISRA en GPU (Cuprite)				
Versión	Librería	Target	Speedup	Tiempo (s)
Serie	oneMKL	CPU (Gold 6138)	1,00	11,51
OpenACC	CUDA	GPU (RTX 3090)	6,19	1,86
OpenMP	CUDA	GPU (RTX 3090)	6,17	1,86
SYCL	oneMKL	GPU (RTX 3090)	7,02	1,64
OpenACC	CUDA	GPU (Tesla V100)	12,47	0,92
OpenMP	CUDA	GPU (Tesla V100)	12,17	0,95
SYCL	oneMKL	GPU (Tesla V100)	14,82	0,78
OpenMP (Offloading)	oneMKL	GPU (G9 UHD P630)	2,43	4,74
SYCL	oneMKL	GPU (G9 UHD P630)	2,81	4,09

Tabla 5.14: *Ejecuciones de ISRA para Cuprite en GPU.*

deriva del uso de la GPU *Tesla V100*. En el caso de la GPU integrada *G9 UHD 9630* los rendimientos obtenidos, aun siendo el doble de eficientes que la versión original secuencial, no son comparables al resto de GPUs, por la enorme diferencia en factor tecnológico.

En la Tabla 5.15 se ilustran los resultados para la imagen SubsetWTC, en donde los rendimientos obtenidos son coherentes con los de la imagen anterior. Es destacable como a mayor tamaño de imagen mejor rendimiento obtienen las implementaciones paralelas, principalmente debido al deficiente rendimiento que se obtiene con la versión secuencial al usar grandes imágenes. La versión SYCL y oneMKL, aunque ligeramente, ha obtenido mejores resultados que el resto de paradigmas, por lo que vuelve a ser la opción más eficiente.

ISRA en GPU (SubsetWTC)				
Versión	Librería	Target	Speedup	Tiempo (s)
Serie	oneMKL	CPU (Gold 6138)	1,00	39,59
OpenACC	CUDA	GPU (RTX 3090)	12,16	3,26
OpenMP	CUDA	GPU (RTX 3090)	12,54	3,16
SYCL	oneMKL	GPU (RTX 3090)	12,76	3,10
OpenACC	CUDA	GPU (Tesla V100)	31,88	1,24
OpenMP	CUDA	GPU (Tesla V100)	31,84	1,24
SYCL	oneMKL	GPU (Tesla V100)	35,67	1,11
OpenMP (Offloading)	oneMKL	GPU (G9 UHD P630)	2,93	13,51
SYCL	oneMKL	GPU (G9 UHD P630)	3,12	12,70

Tabla 5.15: *Ejecuciones de ISRA para SubsetWTC en GPU.*

5.7. Cadenas de desmezclado espectral completas

El principal enfoque de este trabajo es comparar los diferentes paradigmas en los algoritmos seleccionados, ya que, aunque técnicamente funcionan de manera similar, pueden tener diferencias entre sí. Para concluir, se presentan los paradigmas más eficientes para toda la cadena de desmezclado espectral:

- En **CPU** la cadena estaría compuesta de VD SYCL con oneMKL, VCA OpenMP con oneMKL, e ISRA SYCL con oneMKL.
- En **GPU** la cadena estaría compuesta de VD OpenACC con CUDA, VCA OpenMP con CUDA, e ISRA SYCL con oneMKL.
- Utilizando tanto **CPU** como **GPU**, la mejor combinación es VD OpenACC con CUDA, VCA OpenMP con oneMKL, e ISRA SYCL con oneMKL.

El hecho de que la cadena más eficiente contenga todos los paradigmas (OpenACC,

OpenMP, y SYCL), es prueba de que todos pueden garantizar un aumento de eficiencia sobresaliente y ninguno queda eclipsado. Además de las mejores cadenas por tipo de dispositivo, a continuación se enumeran las mejores cadenas en cada paradigma con sus diferentes librerías:

- **OpenACC**: la cadena estaría compuesta de VD con CUDA, VCA con BLAS, e ISRA con CUDA.
- **OpenMP**: la cadena estaría compuesta de VD con CUDA, VCA con oneMKL, e ISRA con CUDA.
- **SYCL**: la mejor combinación es VD, VCA, e ISRA con oneMKL, ya que SYCL cuenta con una única implementación, compatible con todos los tipos de *targets* u objetivos.

En la Tabla 5.16 se comparan los tiempos de ejecución totales de las cadenas completas de desmezclado espectral anteriormente propuestas con los tiempos reales de las imágenes utilizadas en este trabajo, mostrados en la Tabla 5.1:

Comparativa de tiempos de ejecución y tiempos reales (s)				
Cadena	T_{total} Cuprite	T_{total} SubsetWTC	$< T_{real}$ Cuprite	$< T_{real}$ SubsetWTC
CPU	1,81	3,93	✓	✓
GPU	1,28	2,34	✓	✓
CPU/GPU	1,21	2,07	✓	✓
OpenACC	1,55	2,80	✓	✓
OpenMP	1,40	2,27	✓	✓
SYCL	1,44	2,34	✓	✓

Tabla 5.16: *Comparativa de tiempos de ejecución y tiempos reales para las cadenas completas propuestas.*

A partir de los resultados anteriores, se puede observar que tanto las CPUs multinúcleo como las GPUs pueden presentar grandes ventajas a la hora de paralelizar los algoritmos.

Aunque los tiempos obtenidos con la cadena exclusiva de GPU son mejores que los obtenidos en la cadena de CPU, ambas alcanzan los tiempos reales de las imágenes trabajadas, lo que significa que un procesador multinúcleo de altas prestaciones, de mucho menor coste que una GPU de alta gama, es suficiente para llevar a cabo el procesamiento requerido en este trabajo. Esto presenta muchas ventajas a la hora utilizar este *target* en un sistema, como el bajo tamaño o la facilidad para instalar una refrigeración potente. Para obtener el mejor rendimiento posible es necesario el uso de ambos dispositivos, aunque esto supone un coste y espacio adicionales.

En caso de usar únicamente uno de los paradigmas vistos en este trabajo para la cadena completa, todas las cadenas obtenidas alcanzan los tiempos reales de ambas imágenes. Aunque las diferencias de tiempo son sutiles entre ellas, OpenMP representa la más rápida, seguida de SYCL, y OpenACC.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

Las imágenes hiperespectrales representan una enorme utilidad en multitud de disciplinas a la par que retos de computación, por lo que la necesidad de optimización en los algoritmos que procesan estas imágenes es cada vez más relevante. En este proyecto se ha llevado a cabo la optimización de una cadena completa de procesamiento para el desmezclado espectral en imágenes hiperespectrales con diferentes paradigmas de programación y diferentes plataformas. Tiene como objetivo mejorar el rendimiento con el uso del paralelismo y la ejecución de los algoritmos en sistemas heterogéneos. Los paradigmas de programación que se han utilizado han sido OpenACC, OpenMP y SYCL con el uso de las librerías externas de oneMKL, CUDA (cuBLAS y cuSOLVER) y BLAS. La fase de experimentación contempla sistemas heterogéneos con arquitecturas diversas, haciendo uso de varias CPUs y GPUs de Intel, y varias GPUs de Nvidia.

Se han seleccionado tres algoritmos (VD, VCA, e ISRA) para la realización de la cadena de dezmeclado completa, se han analizado en busca de regiones de código potencialmente paralelizables, se han optimizado haciendo uso de los paradigmas anteriormente mencionados, y se han ejecutado y extraído resultados en diferentes sistemas. El paralelismo ha demostrado ser una forma de cómputo adecuada para aumentar la eficiencia de los algo-

ritmos, obteniendo resultados con *speedups* notables. A su vez, el uso de plataformas con dispositivos heterogéneos ha manifestado la relevancia y oportunidades que presenta el usar este tipo de plataformas.

Respecto a los paradigmas utilizados, cada uno proporciona una serie de ventajas y desventajas, y dependiendo de la necesidad habrá uno más conveniente que otro. OpenACC es el paradigma que cuenta con los pragmas más simples, que más responsabilidad delega al compilador y, por tanto, menos exige al programador, por lo que es el paradigma que cuenta con la curva de aprendizaje más suave. Además, el cambio de dispositivo (o *target*) solo requiere de un cambio en el *makefile*, por lo que no es necesario modificar el código desarrollado. OpenMP es el paradigma que ofrece más opciones al programador de los utilizados en este trabajo; a cambio, la curva de aprendizaje es algo más abrupta. Los rendimientos obtenidos con OpenMP se ven más afectados por la habilidad del programador que en los otros paradigmas, aunque, como se ha demostrado, si bien optimizado puede suponer la opción más eficiente. Como desventaja, OpenMP sí requiere edición del código desarrollado para cambiar de *target*. SYCL (en el caso de este trabajo, oneAPI DPC++) es el paradigma más nuevo de todos, y por tanto la curva de aprendizaje y la resolución de desafíos que vayan surgiendo es más complicada. Aun así, no requiere de ningún cambio de formato para cambiar de *target*, pues precisamente una de sus funcionalidades clave es ser un estándar para todos los tipos de dispositivo. Como conclusión, todos los paradigmas ofrecen la posibilidad de mejorar notablemente el rendimiento, pero es la situación y necesidades del proyecto las que determinarán la conveniencia de uno en particular.

6.2. Trabajo futuro

Siguiendo con el enfoque de este trabajo, posibles trabajos de investigación futuros podrían abarcar el desarrollo o implementación de los algoritmos vistos en este trabajo para FPGAs y arquitecturas heterogéneas de última generación de bajo consumo, al igual que otros algoritmos de desmezclado espectral. Como se ha mencionado en este trabajo, la eficiencia es cada vez más relevante y cualquier aportación en este aspecto contribuirá en gran

medida a mejorar la capacidad de captación de los sensores hiperespectrales, gracias a una mayor velocidad de procesamiento.

Otra rama de trabajo podría ser la de desarrollar las implementaciones en paradigmas paralelos de más bajo nivel (como CUDA u OpenCL) de los algoritmos vistos en este trabajo, y posteriormente comparar la comodidad de programación, compatibilidad con diferentes sistemas, y rendimiento con los algoritmos aquí presentados.

Por último, como la labor más importante de paralelizar un código existente consiste en el análisis y la detección de las zonas paralelas, cualquier labor relacionada con un compilador capaz de detectar y completar un código paralelo o una inteligencia artificial (IA) que contribuya a este mismo objetivo pueden presentar oportunidades de investigación muy interesantes.

Bibliografía

[1] D.E. Sabol, J.B. Adams, and M.O. Smith. Predicting the spectral detectability of surface materials using spectral mixture analysis. In *10th Annual International Symposium on Geoscience and Remote Sensing*, pages 967–970, 1990.

[2] Chein-I Chang. An information-theoretic approach to spectral variability, similarity, and discrimination for hyperspectral image analysis. *IEEE Transactions on Information Theory*, 46(5):1927–1932, 2000.

[3] C.-I Chang. *Hyperspectral Imaging: Techniques for Spectral Detection and Classification*. Kluwer Academic/Plenum, 2003.

[4] D.A.Landgrebe. *Signal Theory Methods in Multispectral Remote Sensing*. ohn Wiley Sons:New York, 2003.

[5] D. Landgrebe. Hyperspectral image data analysis. *IEEE Signal Processing Magazine*, 19(1):17–28, 2002.

[6] John B. Adams, Milton O. Smith, and Paul E. Johnson. Spectral mixture modeling: A new analysis of rock and soil types at the viking lander 1 site. *Journal of Geophysical Research*, 91:8098–8112, 1986.

[7] Liaoying Zhao, Chein-I Chang, Shih-Yu Chen, Chao-Cheng Wu, and Mingyang Fan. Endmember-specified virtual dimensionality in hyperspectral imagery. In *2014 IEEE Geoscience and Remote Sensing Symposium*, pages 3466–3469, 2014.

[8] Mingming Xu, Bo Du, and Liangpei Zhang. Spatial-spectral information based abundance-constrained endmember extraction methods. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 7(6):2004–2015, 2014.

[9] J. E. Ball, L. M. Bruce, and N. H. Younan. Hyperspectral pixel unmixing via spectral band selection and dc-insensitive singular value decomposition. *IEEE Geoscience and Remote Sensing Letters*, 4(3):382–386, 2007.

[10] M. Petrou and P.G. Foschi. Confidence in linear spectral unmixing of single pixels. *IEEE Transactions on Geoscience and Remote Sensing*, 37(1):624–626, 1999.

[11] Rob Heylen, Paul Scheunders, Anand Rangarajan, and Paul Gader. Nonlinear unmixing by using different metrics in a linear unmixing chain. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 8(6):2655–2664, 2015.

[12] Yoann Altmann, Marcelo Pereyra, and Steve McLaughlin. Nonlinear spectral unmixing using residual component analysis and a gamma markov random field. In *2015 IEEE 6th International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP)*, pages 165–168, 2015.

[13] Chein-I Chang. A review of virtual dimensionality for hyperspectral imagery. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 11(4):1285–1305, 2018.

[14] J.M.P. Nascimento and J.M.B. Dias. Vertex component analysis: a fast algorithm to unmix hyperspectral data. *IEEE Transactions on Geoscience and Remote Sensing*, 43(4):898–910, 2005.

[15] Margaret E. Daube-Witherspoon and Gerd Muehllehner. An iterative image space reconstruction algorthm suitable for volume ect. *IEEE Transactions on Medical Imaging*, 5(2):61–66, 1986.

[16] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner aand John Pennycook, and Xinmin Tian. *Data Parallel C++, Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Apress Open, 2020.

[17] Germán Castaño, Youssef Faqir-Rhazoui, Carlos García, and Manuel Prieto-Matías. Evaluation of intel's dpc++ compatibility tool in heterogeneous computing. *Journal of Parallel and Distributed Computing*, 165:120–129, 2022.

[18] Manuel Costanzo, Enzo Rucci, Carlos García-Sánchez, Marcelo Naiouf, and Manuel Prieto-Matías. Migrating cuda to oneapi: A smith-waterman case study. In Ignacio Rojas, Olga Valenzuela, Fernando Rojas, Luis Javier Herrera, and Francisco Ortuño, editors, *Bioinformatics and Biomedical Engineering*, pages 103–116, Cham, 2022. Springer International Publishing.

[19] Marcel Breyer, Alexander Van Craen, and Dirk Pflüger. A comparison of sycl, opencl, cuda, and openmp for massively parallel support vector machine classification on multi-vendor hardware. In *International Workshop on OpenCL*, IWOCL'22, New York, NY, USA, 2022. Association for Computing Machinery.

[20] Steffen Christgau and Thomas Steinke. Porting a legacy cuda stencil code to oneapi. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 359–367, 2020.

[21] Emanuele Torti, Alessandro Fontanella, and Antonio Plaza. Parallel real-time virtual dimensionality estimation for hyperspectral images. *J Real-Time Image Proc*, 14:753–761, 2018.

[22] J.M.P. Nascimento and J.M.B. Dias. Vertex component analysis: a fast algorithm to unmix hyperspectral data. *IEEE Transactions on Geoscience and Remote Sensing*, 43(4):898–910, 2005.

[23] Carlos Gonzalez, Javier Resano, Antonio Plaza, and Daniel Mozos. Fpga implementation of abundance estimation for spectral unmixing of hyperspectral data using the image space reconstruction algorithm. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 5(1):248–261, 2012.

[24] Sergio Bernabé, Gabriel Martín, José M. P. Nascimento, José M. Bioucas-Dias, Antonio Plaza, and Vítor Silva. Parallel hyperspectral coded aperture for compressive sensing on gpus. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 9(2):932–944, 2016.

Apéndice A

Introduction

A.1. Motivation

The main motivation of this work is to contribute to the optimization of the hyperspectral images processing algorithms by using parallel computing through paradigms that allow the execution on heterogeneous systems. A hyperspectral image (Figure 1.1) is an image that contains many more layers of information than a common image. These additional layers correspond to different wavelength ranges (which may not be visible to the human eye), and are used in a large number of applications, including: quality control, climate analysis, pollution analysis, terrain analysis for different purposes such as agriculture, computer vision, outworld exploration, geological deposit analysis, defense sector applications, and many more.

Hyperspectral images are taken by special devices located on satellites, drones, UAVs, or airborne platforms, mainly operated by international organizations such as NASA or the European Space Agency (ESA). A large number of satellites are currently being used and launched for all the applications mentioned above. Just as in recent times the quality of the satellites that contain the devices that capture hyperspectral images has increased, these devices have also improved, being able to capture more spectral bands (or layers).

Having a lot of information, the processing of these images entails an extraordinary

computational effort, mainly due to the mixed pixel problem [1]. If is taken into account that many applications require real-time analysis, the need to optimize the processing of these images becomes essential.

The approach chosen in this work to optimize these algorithms is parallel computing. Despite the enormous utility and results obtained with this type of computing, it has had a key problem throughout its history: the dependence on the underlying architecture, which entails little (or no) compatibility with other systems and architectures. The programming paradigms chosen in this work revolutionize this aspect of parallel computing, being largely independent of the underlying architecture, which allows them to be executed on multiple devices (both CPU and GPU).

In summary, this work has two main motivations: the first one is to contribute to the optimization of hyperspectral image processing algorithms through parallelism, for the efficient use of these algorithms; and the second one is to make use of programming paradigms that allow heterogeneous execution, greatly increasing compatibility in different systems, resulting in more interest than an implementation dependent on an exclusive architecture.

A.2. Objectives

The main objective of this work is to optimize a complete processing chain for spectral unmixing in hyperspectral images through parallel computing, making use of modern programming paradigms that allow execution of algorithms in heterogeneous systems. The results obtained will be analyzed and compared with the serial version (initial, not parallel), and with the other paradigms, and conclusions will be drawn on the improvements obtained. For this purpose, the following steps or sub-objectives will be carried out for each stage of the processing chain:

- Choice and explanation of an algorithm for each processing stage.
- Analysis of the degree of potential parallelism of the chosen algorithms.
- Optimization of the algorithms in the chosen paradigms.

- Analysis of results and comparison of performance between the paradigms.

A.3. Organization of this memory

Bearing in mind the previous specific objectives, the organization of the rest of this report is described, structured in a series of chapters whose contents are described below:

- **Hyperspectral analysis**: hyperspectral images are defined and some hyperspectral sensors (AVIRIS and EO-1 Hyperion) are named, as well as the mixing models, and the need for parallelization of the algorithms that work with these images is justified.
- **Parallelism and paradigms**: parallelism, basic concepts and types associated with it are defined; also nomenclature, and the most important parallel programming paradigms used in this work.
- **Implementations**: the approach chosen in this work is detailed, relating the concepts of parallelism with hyperspectral images. Afterwards, each selected algorithm is analyzed, and its degree of parallelism is justified and the different implementations and differences between them are explained.
- **Result analysis**: the results of all the implementations of each algorithm for the hyperspectral images and the chosen systems are presented.
- **Conclusions and future work**: the conclusions obtained as a result of the results and possible lines of future work that continue and expand the scope of this work are presented.

Apéndice B

Conclusions and future work

B.1. Conclusions

Hyperspectral images represent an enormous utility in many disciplines as well as numerous computational challenges, so the need for optimization in the algorithms that process these images is more relevant every day. In this project, the optimization of a complete processing chain for spectral unmixing in hyperspectral images has been carried out with different programming paradigms and different platforms. It aims to improve performance with the use of parallelism and the execution of algorithms in heterogeneous systems. The programming paradigms that have been used have been OpenACC, OpenMP and SYCL with the use of the external libraries of oneMKL, CUDA (cuBLAS and cuSOLVER) and BLAS. The experimentation phase contemplates heterogeneous systems with diverse architectures, making use of several CPUs and GPUs from Intel, and several GPUs from Nvidia.

Three algorithms (VD, VCA, and ISRA) have been selected for the realization of the complete unmixing chain, they have been analyzed in search of potentially parallelizable code regions, then optimized using the aforementioned paradigms, and then have been executed and extracted results on different systems. Parallelism has proven to be a suitable way of computation to increase the efficiency of algorithms, obtaining results with remarkable *speedups*. In turn, the use of platforms with heterogeneous devices has shown the relevance

and opportunities presented by using different devices.

Regarding the paradigms used, each one provides a series of advantages and disadvantages, and depending on the need, one will be more convenient than another. OpenACC is the paradigm that has the simplest pragmas, that delegates more responsibility to the compiler and, therefore, requires less knowledge from the programmer, so it is the paradigm with the smoothest learning curve. Also, changing the device (or *target*) only requires a change in the *makefile*, so there is no need to modify the developed code. OpenMP is the paradigm that offers more options to the programmer; in return, the learning curve is steeper. The yields obtained with OpenMP are more affected by the ability of the programmer than in the other paradigms, although, as has been shown, if used to its maximum capabilities, it can be the most efficient option. On the downside, OpenMP does require editing of the developed code to change the *target*. SYCL (in the case of this work, oneAPI DPC++) is the newest paradigm of all, and therefore the learning curve and the resolution of challenges that arise is more complicated. Even so, it does not require any format changes to change the *target*, since precisely one of its key features is to be a standard for all types of devices. In conclusion, all paradigms offer the possibility of significantly improving performance, but it is the situation and needs of the project that will determine the convenience of one in particular.

B.2. Lines of future work

Continuing with the focus of this work, possible future research works could cover the development or implementation of the algorithms seen in this work for FPGAs and next-generation low-power heterogeneous architectures, as well as other spectral unmixing algorithms. As mentioned in this work, efficiency is becoming more and more relevant and any contribution in this regard will greatly contribute to improving the capture capacity of hyperspectral sensors, thanks to a higher processing speed.

Another branch of work could be to develop the implementations in parallel paradigms of lower level (such as CUDA or OpenCL) of the algorithms seen in this work, and later

compare the ease of programming, compatibility with different systems, and performance with the algorithms here presented.

At last, since the most important task of parallelizing existing code is the analysis and detection of parallel zones, any task related to a compiler capable of detecting and completing parallel code or artificial intelligence (AI) that contributes to the same matter presents very interesting research opportunities.

Apéndice C

Reparto de trabajo

C.1. Real del Noval, Adrián

Para el desarrollo de este trabajo se han utilizado en gran medida los conocimientos adquiridos en la asignatura "Programación de GPUs y Aceleradores", en donde se aprendió sobre el funcionamiento de paradigmas de programación paralela basados en directivas, por lo que, aunque se han ampliado muchos conocimientos durante el desarrollo del trabajo, se contaba con una buena preparación que ha servido como base. La incorporación al trabajo y aprendizaje de los nuevos paradigmas, OpenMP y SYCL, se llevo a cabo a través de la documentación proporcionada por los profesores, la bibliografía que se adjunta, y recursos oficiales de *Khronos Group*, Intel, Nvidia, y los otros desarrolladores de las tecnologías utilizadas.

Las primeras implementaciones paralelas desarrolladas fueron del código ISRA, donde el principal desafío consistió en la correcta implementación de las funciones que componen el código, sobre todo, en el aspecto de almacenamiento en memoria, en donde tomó mucha relevancia si las matrices con las que se trabaja están almacenadas de la forma *row major* o *column major*. Ambos participantes del grupo han aportado al desarrollo de las implementaciones de ISRA de forma similar.

Respecto al código VCA, el siguiente en empezar a ser paralelizado, la principal dificultad

fue el análisis del paralelismo, puesto que VCA es un código de tamaño superior a ISRA o VD, al igual que cuenta con un número de variables muy superior. Por estas razones, el análisis del grado de paralelismo y la detección de zonas potencialmente paralelas fue más costoso que en otros códigos. Ambos participantes aportaron al análisis de VCA, mientras que Óscar realizó casi todas las implementaciones, y Adrián las de SYCL y OpenMP con las librerías de cuBLAS y cuSOLVER.

El último código, VD, fue la principal aportación de Adrián. Realizando el análisis del código y todas las implementaciones. La principal dificultad de VD fue la detección de las zonas paralelas, no por el tamaño (como en VCA) sino por las dependencias de datos y la estructura del código.

Las pruebas han sido realizadas en la Volta por ambos miembros del equipo, en donde cada uno ha realizado las ejecuciones de sus respectivos códigos desarrollados como se indicó anteriormente. Las pruebas en el DevCloud de ISRA y VD las realizó Adrián mientras que Óscar realizó las de VCA.

Por último, ambos miembros han trabajado en proporciones similares en la elaboración de la memoria final del proyecto.

C.2. Ruiz de Pedro, Óscar

Este proyecto se empezó en el mes de Septiembre donde el trabajo más importante a realizar fue tanto de documentación sobre la parte de las imágenes hiperespectrales como de aprendizaje en OpenACC, OpenMP y SYCL (oneAPI), aunque se tuvo una asignatura dada por Carlos, uno de los directores de este trabajo, llamada *Programación de GPU y Aceleradores*, además se realizó en la misma asignatura un trabajo relacionado con oneAPI, lo que ayudó a una mejor preparación para este proyecto.

El primer algoritmo donde se trabajó para este proyecto fue ISRA, perteneciente a la última fase del desmezclado espectral, tanto Óscar como Adrián se encargaron del algoritmo, introduciendo a ambos al uso de los paradigmas para los futuros algoritmos. Se empezó con este algoritmo ya que es el más simple y puede dar pie a otros más complejos. La mayor dificultad en este algoritmo fue entender el formato de las funciones de las librerías externas, se solucionó programando un sencillo código de multiplicaciones de matrices donde se llamaba a estas funciones llegando a comprender el almacenamiento en memoria de las matrices, si se almacenaban por columna o por fila, al variar este dato puede dar una matriz diferente al que se quiere.

Al tener resultados de algunos paradigmas de ISRA se empezó con un nuevo algoritmo, se trataba del algoritmo de VCA perteneciente a la segunda fase de la cadena de desmezclado espectral. Este algoritmo es mucho más grande que ISRA, lo cual hizo la separación del trabajo, donde Adrián se encargó de acabar el algoritmo ISRA y Óscar se encargó de empezar el nuevo algoritmo VCA que se presentó.

A la par que se empezó a ver el algoritmo VCA, se tuvieron que documentar ambos de un nuevo paradigma, SYCL, aunque ya se trabajó ligeramente con ello (en el trabajo de oneAPI que se menciona anteriormente) pero no se puso en practica en su totalidad, aunque se haya dado más tarde, sigue siendo igual de importante que el resto de paradigmas ya que se aplica de igual manera en todos los algoritmos.

El algoritmo VCA en el momento en el que se estaba salían todos los paradigmas que se proponían, aunque la mayor dificultad que se encontraba era una nueva llamada de las

librerías que no estaba en ISRA, ya que se pensaba que se programaría igual que la otra llamada, pero que tiene ligeras diferencias, por ejemplo, que hay que reservar memoria en una de sus variables de una forma especifica para que pueda ser llamado de forma correcta. En ese momento se presentó el último algoritmo por dar para tener completo toda la cadena de desmezclado espectral, se trataba del algoritmo VD que pertenece a la primera fase, el código a programar era más grande que ISRA pero más pequeñas que VCA, juntando ISRA y VD hace más o menos el tamaño de VCA, lo cual se decidió que Adrián tomara este algoritmo y Óscar siguiese con VCA.

La dificultad más grande encontrada en el algoritmo VCA fue con el paradigma OpenMP, la cual ni en el compilador, ni en la ejecución daban ningún tipo de error pero los resultados no daban las correctas, esto hizo que VCA no saliese más rápido y retrasase las ejecuciones en sistemas heterogéneas, la solución a este problema fue que no se tuvo en cuenta la actualización de variables de una manera determinada, haciendo que la variable no se actualizase y también hizo que no saliese ningún tipo de error. Este error costó que Adrián se ocupase del paradigma SYCL de VCA.

Al tener la mayoría de algoritmos se empezó con la ejecución de los algoritmos con los diferentes paradigmas en diferentes plataformas, se empezó la fase de pruebas en la maquina Volta por dos motivos, la primera es que la mayor parte del tiempo se ha trabajado en esa maquina y la segunda, se tenía que comprender como funcionaba el DevCloud de Intel. En ese momento el trabajo actual a hacer era investigar sobre DevCloud, empezar la memoria, acabar los algoritmos y ejecutar los algoritmos completos en la maquina Volta. Para finalizar este trabajo se completaron las ejecuciones a la par que se completaba la memoria hasta su finalización.