UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica
Ingeniería en Informática

Proyecto Fin de Carrera

OpenCL frente a CUDA para análisis de imágenes hiperespectrales en GPU

José Manuel Rodríguez Alves

diciembre, 2011

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica
Ingeniería en Informática

Proyecto Fin de Carrera
OpenCL frente a CUDA para análisis de imágenes hiperespectrales en GPU

Autor: José Manuel Rodríguez Alves

Fdo.:

Director: Antonio Plaza Miguel

Fdo.:

Tribunal Calificador

Presidente: José Moreno del Pozo

Fdo.:

Secretario: Javier Plaza Miguel

Fdo.:

Vocal: Pablo Bustos García de Castro

Fdo.:

CALIFICACIÓN:

FECHA:

Índice

Índice de figuras ____ 5

Índice de tablas ____ 7

1. Motivaciones y Objetivos ____ 9

1.1. Motivaciones ____ 9

1.2. Objetivos ____ 10

2. Introducción ____ 12

2.1. Imagen hiperespectral ____ 12

2.2. Computación de Propósito General en unidades de procesamiento gráfico (GPGPU) ____ 14

2.3. Sensores de imágenes hiperespectrales considerados ____ 15

3. GPU's y lenguajes utilizados ____ 18

3.1. Introducción a las GPU's ____ 18

3.2. Funcionamiento de una GPU ____ 20

3.3. Arquitectura de una GPU haciendo uso científico ____ 21

3.3.1. Arquitectura de la GPU Tesla C1060 usada ____ 22

3.4. Arquitectura de CUDA en las nVidia ____ 24

3.4.1. Modelo de programación en CUDA ____ 26

3.4.2. Modelo de memoria ____ 28

3.5. Arquitectura de openCL ____ 29

3.5.1. Modelo de Plataforma ____ 30

3.5.2. Modelo de ejecución ____ 31

3.5.3. Modelo de memoria ____ 33

3.5.4. Modelo de programación ____ 35

4. Algoritmo PPI ____ 36

4.1. Técnicas para el procesamiento de imágenes. Algoritmo PPI ____ 36

4.2. Paralelización de PPI en GPU 40
4.3. PPI en CUDA vs PPI en openCL 42
5. Imágenes de entrada y resultados Obtenidos 50
5.1. Imágenes de entrada utilizadas 50
5.2. Resultados en la tesla del CETA-CIEMAT 56
5.2.1. Imágenes obtenidas como resultado de las ejecuciones 56
5.2.2. Comparativa de los tiempos de ejecución obtenidos 62
6. Conclusiones y trabajos futuros 72
7. Bibliografía 77
8. Apéndice 80
8.1. Lenguaje CUDA 80
8.2. Lenguaje openCL 85
8.3. Herramientas utilizadas 94
8.4. Problemas encontrados 96
8.5. Código fuente 99
8.5.1. Archivo ppi.h común a ambos lenguajes 99
8.5.2. Código en CUDA 99
8.5.3. Código en openCL 106

Índice de figuras

Figura 1.1: Ejemplo de aplicaciones a la computación paralela en datos. 9
Figura 2.1: Ejemplo de imagen hiperespectral 13
Figura 2.2: Imagen que muestra cuáles son los píxeles mezcla y cuáles son los píxeles puros. 14
Figura 2.3: La ilustración nos muestra cómo ha ido mejorando la SNR 17
Figura 3.1: Gráfica de evolución de las GPU vs CPU. 19
Figura 3.2: Foto de una de las GPU TESLA usadas. 20
Figura 3.3: Proceso de renderizado en una GPU 21
Figura 3.4: Comparación entre la arquitectura CPU y GPU 22
Figura 3.5: Arquitectura Fermi usada en las Tesla 23
Figura 3.6: Pila de CUDA 25
Figura 3.7: Operaciones de memoria Gather y Scatter (dispersión y reunión) 26
Figura 3.8: Organización de los threads en CUDA 27

Figura 3.9: Modelo de memoria en la arquitectura CUDA ____ *28*
Figura 3.10: Modelo de plataforma en OpenCL ____ *31*
Figura 3.11: Explicación del NDRange ____ *32*
Figura 3.12: Modelo de memoria en OpenCL ____ *34*
Figura 4.1: La ilustración nos muestra como se visualiza el modelo lineal de mezcla ____ *37*
Figura 4.2: Representación gráfica del algoritmo PPI. ____ *40*
Figura 4.3: Ilustración que muestra como son las estructuras de datos usadas. ____ *42*
Figura 4.4: Ilustración que muestra los ficheros que forman PPI en CUDA y en openCL ____ *43*
Figura 4.5: Comparación visual de los kernels implementados en CUDA y openCL ____ *45*
Figura 4.6: Ejemplo de host simplificado para ejecutar un programa en openCL ____ *47*
Figura 4.7: Ejemplo de host simplificado para ejecutar un programa en CUDA ____ *48*
Figura 5.1: Ilustración de cómo se ha formado la imagen sintética. ____ *51*
Figura 5.2: Representación de la imagen hiperespectral sintética con distintas bandas de color ____ *52*
Figura 5.3: Representación de la imagen hiperespectral cuprite con distintas bandas de color ____ *53*
Figura 5.4:Explicación de cómo se realiza el análisis de precisión ____ *55*
Figura 5.5: Escala de colores de los píxeles de la imagen. ____ *56*
Figura 5.6: Resultado de la ejecución en CUDA para 15360 iteraciones de sintetica.bsq ____ *57*
Figura 5.7: Resultado de la ejecución en CUDA para 15360 iteraciones de Cuprite.bsq ____ *58*
Figura 5.8: Resultado de la ejecución en openCL con 15360 iteraciones de sintetica.bsq ____ *60*
Figura 5.9: Resultado de la ejecución en openCL para 15360 iteraciones de Cuprite.bsq ____ *61*
Figura 5.10:Comparativa gráfica CUDA vs openCL, 1000 iteraciones, sintética (I) ____ *63*
Figura 5.11: Comparativa gráfica CUDA vs openCL, 1000 iteraciones, sintética (II) ____ *64*
Figura 5.12: Comparativa gráfica CUDA vs openCL, 1000 iteraciones, cuprite (I) ____ *65*
Figura 5.13: Comparativa gráfica CUDA vs openCL, 1000 iteraciones, cuprite (II) ____ *66*
Figura 5.14: Comparativa gráfica CUDA vs openCL, 15360 iteraciones, sintética (I) ____ *68*
Figura 5.15: Comparativa gráfica CUDA vs openCL, 15360 iteraciones, sintética (II) ____ *69*
Figura 5.16: Comparativa gráfica CUDA vs openCL, 15360 iteraciones, cuprite (I) ____ *70*
Figura 5.17: Comparativa gráfica CUDA vs openCL, 15360 iteraciones, cuprite (II) ____ *71*
Figura 8.1: a) Código serie en C b) Código CUDA en paralelo. Ejemplo de programas que suman arrays ____ *80*
Figura 8.2: Ejemplo de código fuente de un kernel que suma dos vectores ____ *86*
Figura 8.3: Host para el ejemplo de código fuente de un kernel que suma dos vectores ____ *90*

Índice de tablas

Tabla 3.1: Especificación de la GPU usada para el proyecto ____ *24*
Tabla 3.2: Tipo de acceso en el modelo de memoria de CUDA ____ *29*
Tabla 3.3: Tipo de acceso en el modelo de memoria de openCL ____ *34*
Tabla 5.1: Características de la imagen hiperespectral sintética utilizada ____ *50*
Tabla 5.2: Características de la imagen hiperespectral cuprite utilizada ____ *53*
Tabla 5.3: Datos sobre la ejecución de la imagen sintetica.bsq en CUDA ____ *56*
Tabla 5.4: Datos sobre la ejecución de la imagen Cuprite.bsq en CUDA ____ *58*
Tabla 5.5: Datos sobre la ejecución de la imagen sintetica.bsq en openCL ____ *59*
Tabla 5.6: Datos sobre la ejecución de la imagen Cuprite.bsq en openCL ____ *61*
Tabla 6.1: Comparación de tiempos de ejecución en ambos lenguajes ____ *73*
Tabla 6.2: Comparación CUDA vs OpenCL ____ *74*

1. Motivaciones y Objetivos

1.1.Motivaciones

Actualmente hay gran demanda de computación de datos en paralelo debido a campos de la ciencia como la bioinformática, la medicina, la investigación de nuevos materiales. También en la simulación aplicada a casi cualquier campo que nos podamos imaginar,...Pero no solo se limita al ámbito científico, sino que a campos tan diversos de la ingeniería hacen uso de la computación de alto rendimiento como puede ser: la industria aeroespacial y en la industria automovilística haciendo uso del CFD (Simulación Dinámica de Fluidos), en el cálculo de modelos climatológicos, la investigación de nuevos materiales, en competiciones deportivas de élite donde se requiere gran paralelismo de datos como puede ser la simulación en la Fórmula 1, la IndyCar,…

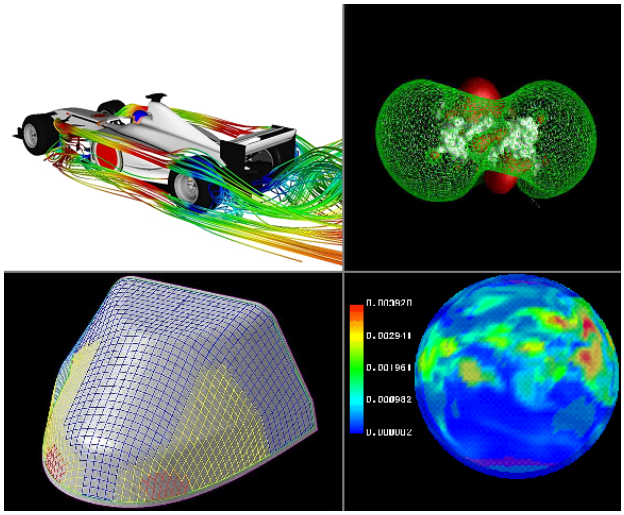

Figura 1.1: Ejemplo de aplicaciones a la computación paralela en datos.

Debido a la gran demanda de aplicaciones surge la necesidad de computación de gran cantidad de datos en paralelo a un bajo coste temporal y económico. Es en este punto donde entran en juego la computación paralela mediante GPU's (Graphics Processing Units).

Las GPU's son unidades de proceso gráfico que hasta hace muy poco tiempo habitualmente solo se usaban en los juegos 3d donde se requería y sigue requiriendo un alto rendimiento gráfico, pero en la actualidad, con el soporte que hace potencialmente su uso para la computación paralela debido a la gran cantidad de núcleos que pueden llegar a contener (en torno a 400 - 500) para la gama baja, y la bajada de precios, hacen atractivo su uso para este tipo de aplicaciones.

Con la creciente popularidad de CUDA, y con el inconveniente que solo se puede ejecutar en GPU's nVidia, surge la necesidad de un estándar que permita la ejecución multiplataforma. Es en este punto donde surge openCL. OpenCL es un estándar multiplataforma GPGPU (General-Purpose Computing on Graphics Processing Units) que ha sido desarrollado por Apple inicialmente y refinado por IBM, Intel, AMD y nVidia para posteriormente ser remitido al Grupo Khronos que es un consorcio de 36 empresas que se dedica a la creación de APIs estándares abiertas.

Así pues, una vez determinado los dos lenguajes GPGPU se manifiesta una necesidad de comparar cual de los lenguajes es más rápido en la ejecución de algoritmos. Obviamente las comparaciones deben ser realizadas sobre la misma arquitectura tanto software como hardware para obtener resultados fiables, por tanto, para las comparaciones se solo se pueden usar GPU's nVidia ya que es la única arquitectura que puede soportar CUDA.

1.2.Objetivos

En este contexto, el proyecto a desarrollar en el presente trabajo tratará el procesamiento de imágenes hiperespectrales: unas obtenidas desde satélite de una imagen de una zona minera de cobre en EEUU y otra generada sintéticamente. Dicho procesamiento se llevará a cabo con dos lenguajes de programación específicos de programación en GPU's, en los cuales se implementará el algoritmo PPI (Píxel Purity Index) en openCL a partir de la versión inicial existente de CUDA en el laboratorio del grupo HyperComp de la Universidad de Extremadura. Hecho esto,

se comprobaran que los resultados en la ejecución de los algoritmos son similares y finalmente se hará una comparativa de rendimiento de ambos lenguajes.

Las pruebas serán realizadas en el clúster GPU CETA-ciemat de Trujillo. En la comparativa se analizará los resultados para diferentes parámetros como son el número de iteraciones del algoritmo PPI, el número de hilos y el número de bloques para cada una de las imágenes señaladas con anterioridad.

El fin último de la comparativa es guiar al usuario interesado en desarrollo de aplicaciones GPU, para decidir cuál es el lenguaje que más adecúa a sus necesidades.

La estructura que seguiremos en este proyecto será la siguiente:

- En primer lugar se hará una introducción del procesamiento en GPU definiendo los distintos términos sobre las imágenes hiperespectrales y sobre el procesamiento en GPU.
- A continuación se hará una descripción de las GPU usadas indicando su capacidad de procesamiento y demás características, también se expondrá las arquitecturas CUDA y openCL para su mejor comprensión, y además se hará una presentación de las imágenes usadas.
- Seguidamente se analizarán comparativamente los resultados obtenidos en el CETA-ciemat
- Posteriormente se harán una serie de conclusiones de lo tratado en el punto anterior. Además se sugerirá una serie de extensiones que sería interesante para presente trabajo.
- A la postre se expondrá un apéndice donde se explicarán nociones sobre la programación tanto en CUDA como en openCL. Se expondrán todas las herramientas que han sido utilizadas para el desarrollo de este proyecto. También se hará entrega del código fuente y de los problemas encontrados para el desarrollo del PFC.
- Por último se hará presentación de la bibliografía utilizada.

2. Introducción

Para comprender mejor este proyecto fin de carrera será conveniente en primer lugar presentar una serie de definiciones que mostraremos a continuación. Dichas definiciones son la de imagen hiperespectral, la de Computación de propósito general en GPU (GPGU) que es aquello sobre lo que está basado el presente trabajo y finalmente se mencionará los sensores hiperespectrales usados en la obtención de las imágenes.

2.1.Imagen hiperespectral

Normalmente cuando observamos una foto normal y corriente, vemos representada una parte de la realidad, la luz visible, pero nos estamos perdiendo información como consecuencia de no poder visualizar el resto del espectro electromagnético (valiosa por ejemplo en observación terrestre). Ello se consigue con las imágenes hiperespectrales. En esencia, una imagen hiperespectral es aquella en lugar de representar únicamente la luz visible, se representa así mismo parte del espectro restante dividido en diferentes bandas. Esto quiere decir que dicha imagen está formada por un cubo de tres dimensiones, dos dimensiones para representar la dimensión espacial de la imagen, y la tercera dimensión para representar las distintas frecuencias del espectro electromagnético (bandas) para cada punto espacial. La ventaja de representar una mayor amplitud del espectro es la de obtener información que únicamente con la luz visible no se puede captar, por ejemplo, podemos supervisar los cultivos en satélites o aeronaves no tripuladas, control de calidad en producción de alimentos procesados, detección de materias extrañas que tengan una composición diferente a la de un producto inspeccionado, detección de material militar [1], incluso las imágenes hiperespectrales pueden ser usadas para buscar posibles textos ocultos en pergaminos antiguos.

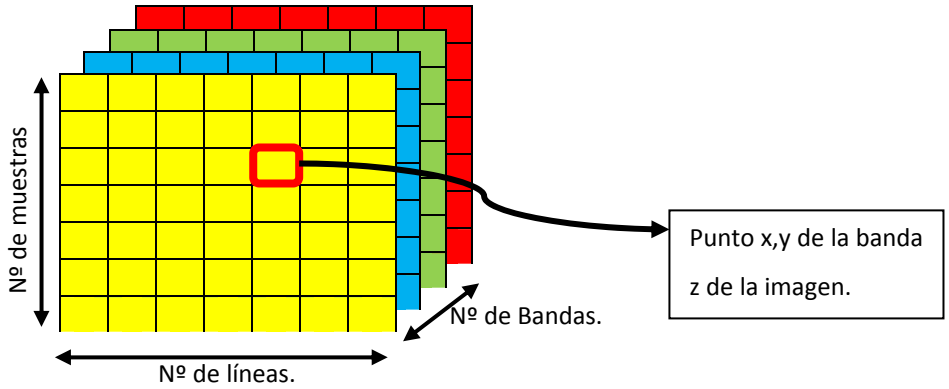

Figura 2.1: Ejemplo de imagen hiperespectral

Para comprender mejor la definición de imagen hiperespectral tenemos a la imagen anterior (figura 2.1), en ella se presentan una serie de bandas de imágenes la cuales cada una representa un conjunto de frecuencias. Para cada banda, cada punto en el espacio (x, y) está representado. Como consecuencia, un punto de una imagen podemos verlo en distintas frecuencias, lo que permite que podamos ver determinadas radiaciones que emitan determinado material que no se podían observar solamente con la luz visible.

Con todo lo dicho, cabe destacar que cuando se obtienen imágenes hiperespectrales, normalmente no se obtienen píxeles espectralmente puros, es decir, que es normal la existencia de mezcla a nivel subpíxel, y por tanto, hay a grandes rasgos hay dos tipos de píxeles en estas imágenes: pixeles puros y píxeles mezcla. Se dice que un píxel es píxel mezcla cuando en él hay varios tipos de materiales. En la mayoría de cualquier imagen obtenida hay píxeles mezcla como podemos ver en el ejemplo siguiente:

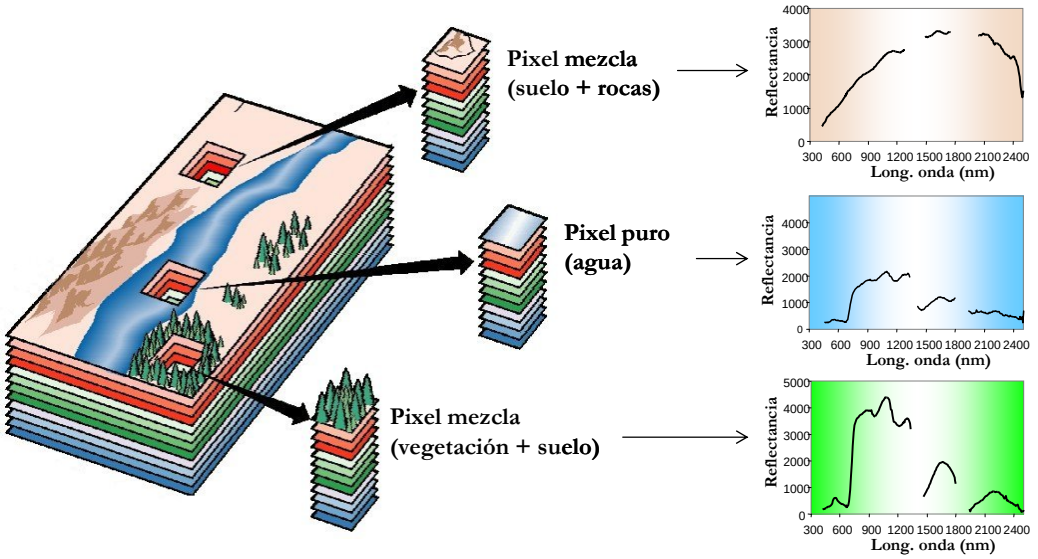

Figura 2.2: Imagen que muestra cuáles son los píxeles mezcla y cuáles son los píxeles puros.

En este PFC se trabajará con imágenes hiperespectrales obtenidas desde satélite utilizando para ello la capacidad de captar distintas bandas de una misma imagen, así, podemos tener una aproximación muy precisa de la superficie de nuestro planeta.

2.2.Computación de Propósito General en unidades de procesamiento gráfico (GPGPU)

La computación de propósito general en unidades de procesamiento gráfico consiste en el aprovechamiento de la capacidad de cómputo de una GPU para la ejecución de algoritmos, a diferencia de la ejecución CPU como convencionalmente se venía haciendo hasta ahora. La diferencia principal entre la computación CPU y GPU estriba en el modelo de computación y en el modelo de memoria usados. Así la computación CPU, grosso modo, consiste en generar un archivo binario con instrucciones ensamblador que se ejecutan en la CPU haciendo uso de la memoria principal, mientras que en la computación GPU se genera el binario para ejecutar en CPU, y una vez en ejecución en la CPU (parte de host), se lanza la ejecución de los cálculos que necesitan gran paralelismo en la GPU (parte de device), pudiendo hacer uso de la memoria principal y la memoria de procesamiento gráfico. Es decir,

que en la GPGU tiene una parte del programa se ejecuta en la CPU y otra parte en la GPU.

Para que la ejecución GPU se le pueda sacar rendimiento, los algoritmos codificados deberían ser candidatos a tener gran cantidad de paralelismo ya que habitualmente las GPU's actuales tienen una gran cantidad de núcleos. Ello no quiere decir que un algoritmo secuencial no se pueda ejecutar en una gráfica, sino que, no obtendremos el mismo rendimiento en la GPU si no optimizamos el código.

Una de las aplicaciones de GPGPU puede ser en procesamiento de imágenes hiperespectrales. Como ya hemos mencionado con anterioridad, las imágenes hiperespectrales están representadas en diferentes bandas, ello implica que su procesamiento sea computacionalmente muy costoso (a la hora de procesar información que nos pueda resultar útil). Además normalmente cada banda puede llegar a tener una resolución media-alta lo que incrementa notablemente la cantidad de datos que pueden ocupar dependiendo del nº de bandas y de la resolución en torno 40-200MB aproximadamente (incluso más), con lo cual son candidatos a su procesamiento paralelo. Es aquí donde surge la idea de procesamiento paralelo de datos. Si los algoritmos de procesamiento de las imágenes son potencialmente paralelos, puede resultarnos muy útil el procesamiento de datos GPGPU o procesamiento de computación general en tarjetas gráficas.

En este PFC concretamente se hará uso del algoritmo de procesamiento PPI. Este algoritmo es uno muy habitual en procesamiento de imágenes hiperespectrales, que en esencia trata de extraer los endmembers de una imagen dada, siendo el algoritmo un gran candidato al procesamiento paralelo.

2.3.Sensores de imágenes hiperespectrales considerados

Cuando hablamos de sensores de imágenes hiperespectrales podemos clasificarlos de muchas maneras, nosotros según por la forma en que son transportados, los podemos dividir en dos grupos: los que obtienen imágenes desde

satélite, y los que obtienen imágenes desde avión (aerotransportados). Algunos sensores hiperespectrales cuya plataforma de transporte son los satélites pueden ser MODIS, MERIS, CHRIS, HYPERION, además de los sensores usados en misiones extra-terrestre,… A pesar de la gran variedad de sensores que existen en los satélites, la mayoría de los sensores hiperespectrales son aerotransportados como pueden ser: AVISA, CASI, DAIS, AVIRIS,… [2-4] Nosotros en el presente trabajo nos centraremos en el sensor aerotransportado AVIRIS.

El sensor AVIRIS (Airbone Visible/Infra-Red Imaging Spectometer) es un sensor que obtiene imágenes hiperespectrales de alta resolución espacial con 224 bandas cubriendo las longitudes de onda comprendidas entre 400 y 2500 nanómetros de manera que el ancho de banda entre imágenes es bajo (10 nanómetros) [2] y con un ángulo de visión de 30º. Además, como hemos mencionado, AVIRIS es un sensor aerotransportado, que se puso por primera vez en un avión en el año 1989 y se ha dedicado a estar haciendo tomas de datos hasta estos momentos de América del norte, Europa, partes de América del Sur y más recientemente Argentina. Los aviones usados para ello han sido los siguientes [5-7]:

- NASA's ER-jet: vuela a 20 km sobre el nivel del mar a una velocidad de unos 730 km/h.
- El avión turbohélice Twin Otter International: vuela a 4 km sobre el nivel del mar a una velocidad de unos 130 km/h.
- Scaled Composites' Proteus.
- Por último el NASA's WB-57.

Las características más significativas respecto al diseño interno del sensor son las siguientes:

- La forma de obtención de las imágenes del sensor es mediante un explorador de barrido.
- La parte visible del espectro la realiza un espectrómetro EFOS-A. El infrarrojo con el EFOS-B, EFOS-C y EFOS-D.
- La señal medida por cada detector después de ser amplificada, se codifica usando 12 bits. Después, se almacena para posteriormente

ser sometida a una etapa de pre procesamiento, almacenándola en una cinta de alta densidad de 10,4GB.

- El sensor dispone de un sistema de calibrado con una lámpara halógena de cuarzo que proporciona una radiación de referencia para la comprobación de los distintos espectrómetros.
- Durante los últimos años, el sensor ha mejorado en cuanto a prestaciones de la Señal-Ruido como se puede observar en la figura 2.3.

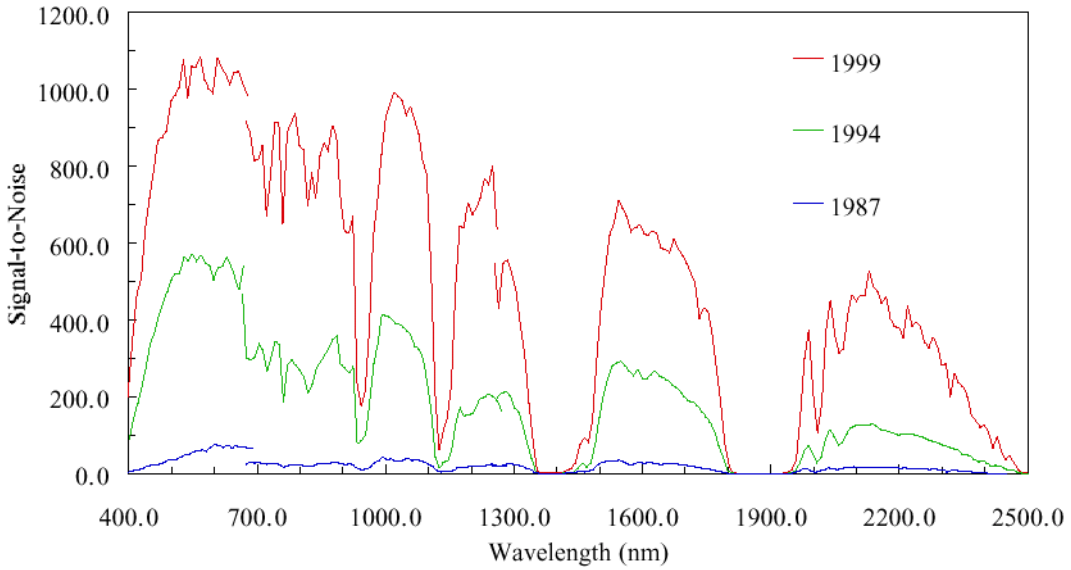

Figura 2.3: La ilustración nos muestra cómo ha ido mejorando la SNR

3. GPU's y lenguajes utilizados

En la realización del presente proyecto, ha sido necesario la utilización de determinadas GPU's. Concretamente, para poder realizar las distintas comparativas, han sido necesarias GPU's nVidia debido a que son las únicas GPU's que soportan tanto la arquitectura de CUDA como la arquitectura de openCL.

3.1.Introducción a las GPU's

Como sabemos GPU (Graphics Processing Unit) se trata de un procesador dedicado al procesamiento gráfico y/o operaciones de coma flotante para liberar la carga del procesador cuando se requieren una gran cantidad de procesamiento gráfico como por ejemplo en aplicaciones o juegos que hacen uso del renderizado 3D[8].

Con el aumento de la capacidad de procesamiento de las GPU's comparada con las CPU's (ver figura 3.1) y la bajada de coste, se abre la posibilidad de aprovechar la gran capacidad de procesamiento que tienen las GPU's, esto unido a que el procesamiento en GPU's es a través de mucho paralelismo, las GPU's se convierten en un candidato perfecto de bajo coste para la computación GPGPU. A partir de aquí surge la arquitectura CUDA para la computación a la que le ha seguido la arquitectura openCL.

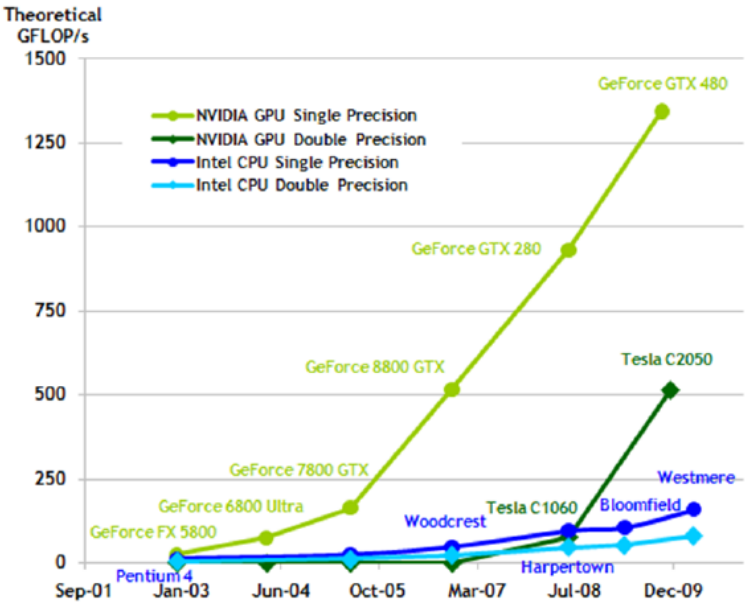

Figura 3.1: Gráfica de evolución de las GPU vs CPU.

El hardware usado en este PFC ha sido la GPU Tesla C1060 (ver figura 3.2). Esta GPU contiene 240 unidades de procesamiento a 1,296 GHz cada una, con un rendimiento de punto flotante de simple precisión de 933 GFLOPS y con rendimiento de punto flotante de doble precisión de 78 GFLOPS. La memoria dedicada que contiene es de 4GB con 512-bit GDDR3 y un ancho de memoria de 102 GB/s [9]. Cabe a mencionar que la GPU está conectada a una placa base ASUS P6T7 WS Supercomputer que contiene dos Quad Core Intel Xeon a 2.26GHz, 24 GB de SDRAM (1333MHz) y 500GB SATA.

Figura 3.2: Foto de una de las GPU TESLA usadas.

3.2.Funcionamiento de una GPU

En este apartado se explicará de manera más detallada el funcionamiento de una GPU genérica para poder exponer a continuación la arquitectura de una GPU, y posteriormente la arquitectura de CUDA y la de openCL.

La GPU contiene una memoria muy rápida que es útil para el almacenamiento de resultados intermedios. Otra de las características principales de la arquitectura de una GPU es que se encuentra muy segmentada, o sea, tiene muchas unidades funcionales. Dichas unidades funcionales están divididas en dos tipos: las unidades funcionales que procesan vértices (vertex shader) y las que procesan píxeles (pixel shader). Por un lado, los vertex shader son programas que son capaces de manipular (que no eliminar) los vértices de figuras 3D, como la deformación en tiempo real de olas, o expresiones corporales de personas o animales. Por otro lado, los píxel shaders son programas que a diferencia con el píxel shader, no definen la forma de las figuras, sino, se encarga de la rasterización de la escena, es decir, de qué color se pinta cada píxel, donde es necesario tener en cuenta la iluminación en tiempo real de la escena. Por lo general en una GPU hay mayor carga de trabajo en la ejecución de los píxel shader que en los vertex shader [8 y 10].

Con los conceptos mencionados en el párrafo anterior, podemos presentar el procesamiento básico que sigue una GPU. En general la GPU sigue el siguiente proceso. En primer lugar la CPU envía a la GPU los vértices. En GPU, el tratamiento que realiza en primer lugar, es aplicar el vertex shader a los modelos 3D. A partir de las transformaciones realizadas, se realiza un proceso por el cual se define las caras de las figuras que se van a ver (clipping). A continuación pasamos al denominado pixel. Posteriormente se almacenan los píxeles en caché y se les aplican distintos efectos. Por último, la unidad ROP (Raster Operations) obtiene los datos de la caché para ser visualizados en un monitor, proyector, etc. [11] (Ver figura 3.3).

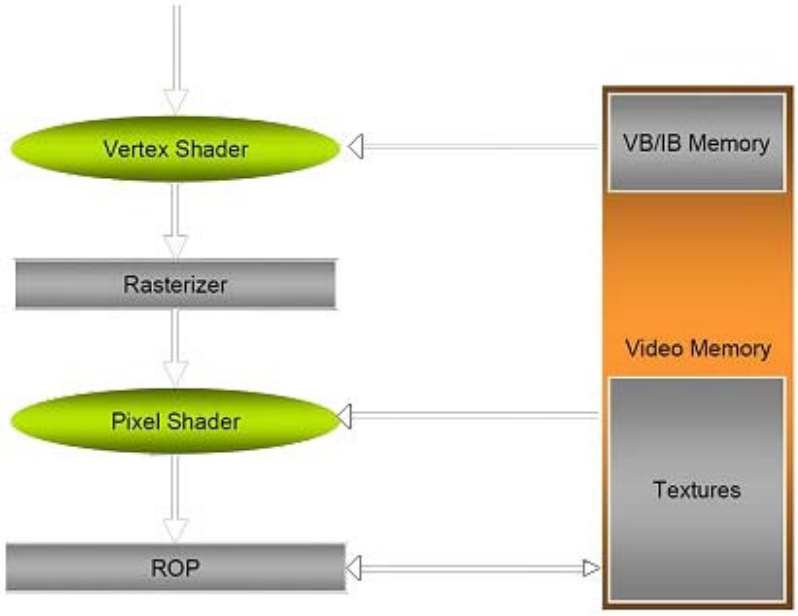

Figura 3.3: Proceso de renderizado en una GPU

3.3.Arquitectura de una GPU haciendo uso científico

Acabamos de ver el funcionamiento básico de una GPU, ahora pasemos a ver como es su arquitectura.

Los ordenadores personales actuales contienen una CPU con varios núcleos (4, 8,…) pero aunque tienen cierto grado de paralelismo al disponer de varios núcleos, no llegan al nivel de paralelismo que se dispone en las GPU actuales (ver figura 3.4). Así, la GPU está pensada para computación de operaciones aritméticas masivamente paralelas, como los que puede haber en los problemas que se plantean en el ámbito científico. El objetivo de la computación paralela, como su nombre indica, es la ejecución de muchas operaciones aritméticas en el mismo instante, y por tanto, es necesario que el ancho de banda con la memoria de datos sea suficientemente grande como para que pueda soportar la gran demanda de datos de las unidades de procesamiento, de lo contrario habrá una bajada de rendimiento en el procesamiento, desaprovechando en parte el gran paralelismo existente.

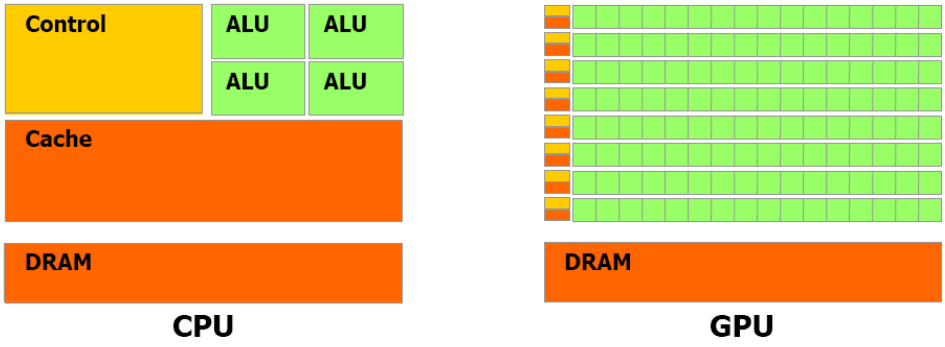

Figura 3.4: Comparación entre la arquitectura CPU y GPU

En la figura 3.4 podemos observar la enorme diferencia entre el número de núcleos que tiene una CPU (en este caso ocho) [12], y el número de núcleos que tiene una GPU (en este caso 128). Además hay que señalar también, que a pesar de que la GPU tiene más núcleos, éstos también se ejecutan a menor velocidad, por lo tanto en el caso que ejecutásemos un algoritmo que no sea lo suficientemente paralelo (no esté optimizado o no sea masivamente paralelo), podría ejecutarse en un tiempo superior al que se ejecutaría en una CPU, lo que nos indica un alto desperdicio de las capacidades de la GPU. Igualmente podemos ver como la GPU está formada por 8 multiprocesadores de 16 cores cada uno, cada multiprocesador tiene su banco de registros, memoria compartida (SRAM) y una caché para constantes y otra para texturas (las dos de solo lectura), también se observa la memoria global de vídeo (DRAM) que es el triple de rápida que la memoria de la CPU aunque 500 veces más lenta que la memoria compartida que contiene cada multiprocesador de la GPU.

En este apartado se ha explicado la arquitectura de una GPU genérica, a continuación se presenta la arquitectura de la GPU Tesla usada.

3.3.1. Arquitectura de la GPU Tesla C1060 usada

La GPU usada ha sido una nVidia Tesla C1060, 4GB a 800Mhz, GDDR3, PCI Express 2.0. Esta GPU está basada en la tercera generación de nVidia denominada arquitectura Fermi.

La primera GPU basada en Fermi fue implementada con 3,0 millones de transistores, cuenta con hasta 512 núcleos CUDA. Los núcleos de CUDA ejecutan una instrucción de punto flotante o entero por ciclo de reloj para un hilo. Los 512 núcleos CUDA están organizados en 16 SM de 32 núcleos cada uno. La GPU tiene seis particiones de memoria de 64 bits, para una interfaz de memoria de 384 bits, que soporta hasta un total de 6 GB de memoria GDDR5 DRAM. Una interfaz de host se conecta la CPU a la GPU a través de PCI-Express [13].

A continuación se muestra un gráfico explicativo de la arquitectura CUDA (figura 3.5):

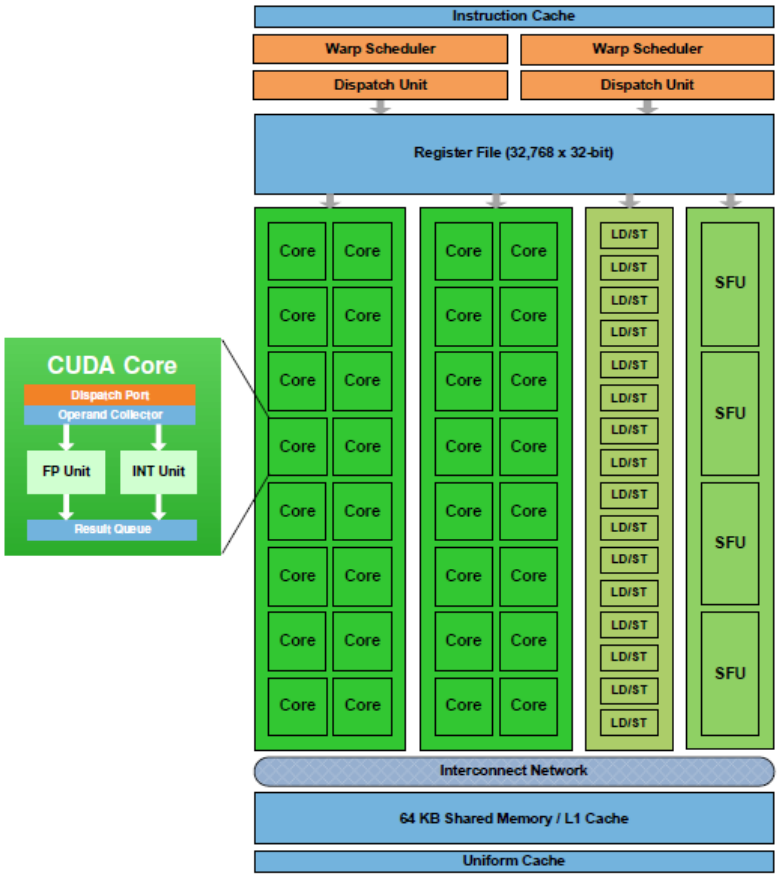

Figura 3.5: Arquitectura Fermi usada en las Tesla

La siguiente tabla muestra las especificaciones de la GPU usada [14]:

Especificaciones	Descripción
Generic SKU reference	900-20607-0000-000
Chip	Tesla T10 GPU
Package size: GPU	45.0 x 45.0 mm
Processor clock	1296 MHz

Memory clock	800 MHz
Number of multiprocessors	240
Memory size	4 GB
Memory I/O	512-bit GDDR3
Memory configuration	32 pcs 32M × 32 GDDR3 SDRAM
External connectors	None
Internal connectors and headers	8-pin PCI Express power connector 6-pin PCI Express power connector 4-pin fan connector
Board power	200 W maximum (160 W typical)
Thermal cooling solution	TM72 active fan sink

Tabla 3.1: Especificación de la GPU usada para el proyecto

3.4.Arquitectura de CUDA en las nVidia

CUDA (Compute Unified Device Architecture) se trata de una arquitectura tanto hardware como software que trata de aprovechar las GPU's haciendo uso de un nuevo lenguaje de programación similar a C, sin necesidad de tener que usar la API de gráficos, y que tiene la particularidad que se ejecuta en la gráfica. Esta arquitectura tiene la peculiaridad que solo se encuentra disponible para la gama de gráficas nVidia en los modelos GeForce 8100 mGPU, GeForce Mobile 8400M GS, Quadro Plex 2100 S4, Quadro NVS 130M, Tesla D870, o en GPU's más actuales. En GPU's anteriores y en GPU's de otras marcas no existe el soporte.

La pila de software de CUDA está compuesta de varias capas: como se muestra en la figura 3.6, un controlador de hardware (CUDA driver), una API (CUDA runtime) y dos librerías matemáticas de alto nivel (CUDA libraries). El hardware ha sido diseñado para soportar controladores ligeros y como consecuencia se consigue un alto rendimiento [12, 25].

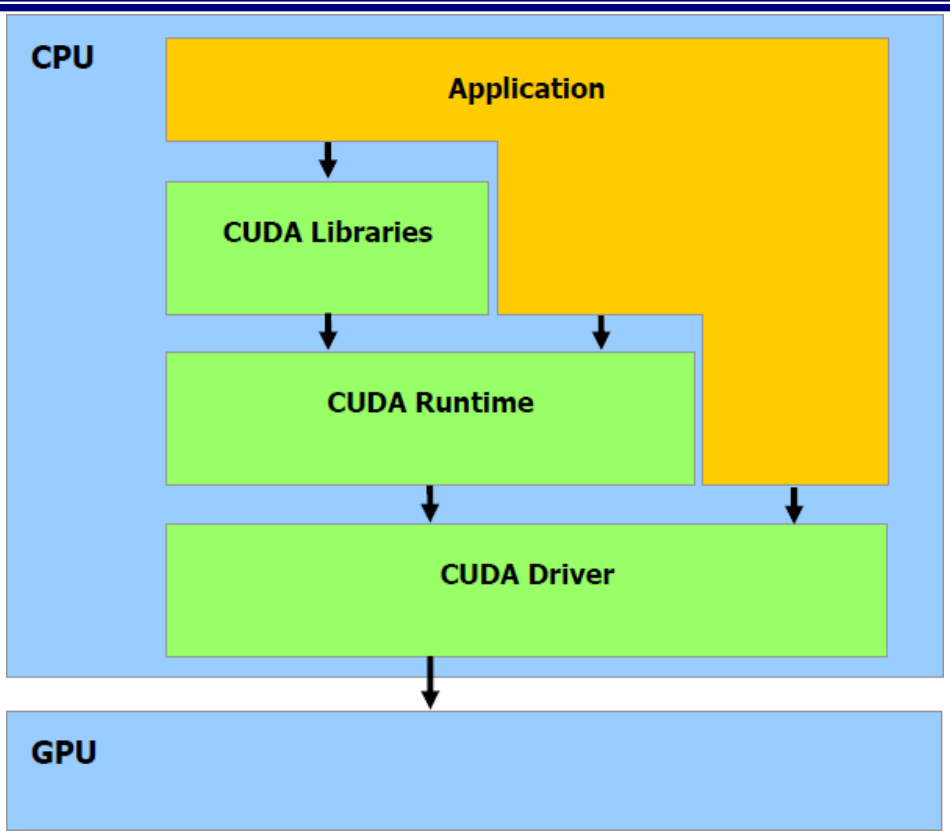

Figura 3.6: Pila de CUDA

Asimismo CUDA proporciona memoria DRAM para conseguir mayor flexibilidad en la programación al suministrar operaciones de memoria de dispersión y reunión. Desde el punto de vista de la programación, esto se traduce en la capacidad de leer y escribir datos en cualquier lugar de DRAM, de manera equivalente a la CPU (ver figura 3.7).

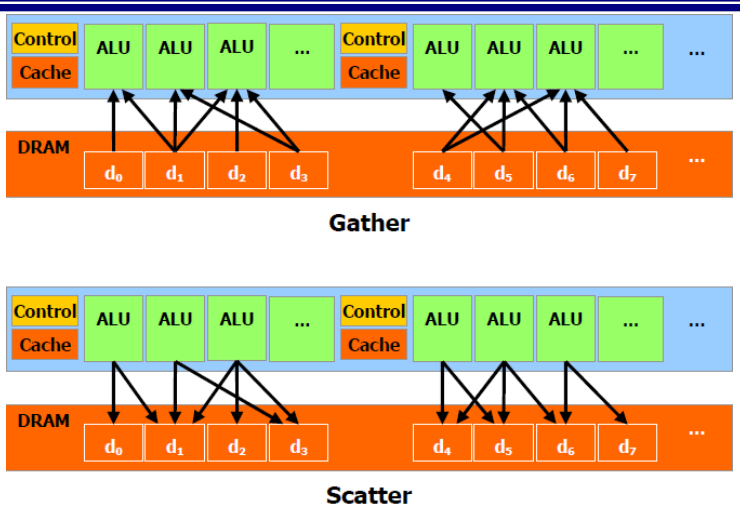

Figura 3.7: Operaciones de memoria Gather y Scatter (dispersión y reunión)

3.4.1. Modelo de programación en CUDA

La GPU (device) ofrece a la CPU (host) la visión de un coprocesador altamente ramificado en hilos el cual dispone de una memoria RAM propia donde cada hilo puede ejecutarse en paralelo sobre los distintos núcleos (cores o streams processors). Cabe destacar que los hilos CUDA son considerablemente ligeros, y como consecuencia, se crean en un tiempo muy breve y la conmutación de contexto es instantánea. Como resultado de esto, el programador de CUDA tiene como objetivo declarar los miles de hilos que la GPU necesita para lograr el rendimiento y la escalabilidad deseados. Teniendo en cuenta esto vamos a explicar cómo están estructurados los hilos en CUDA.

En la figura 3.8 se ilustra como en el host se envían para ejecutar en el device los diferentes kernels (programas que queremos ejecutar en paralelo) de manera secuencial al dispositivo [12].

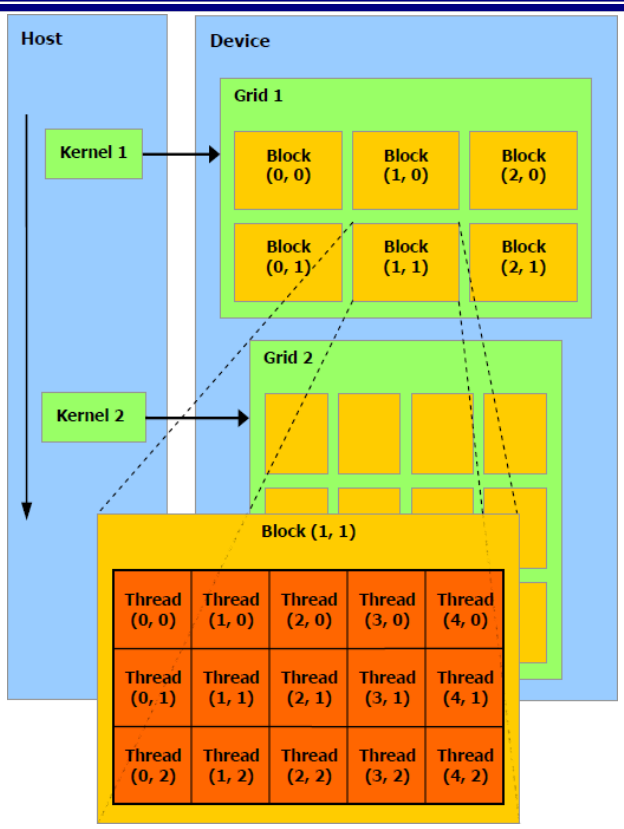

Figura 3.8: Organización de los threads en CUDA

En la figura 3.8 se también ilustra como en el modelo de programación, en el nivel más bajo se localizan los threads. Estos se pueden organizar en bloques para cooperar entre sí compartiendo datos a través de memorias rápidas, y sincronizando su ejecución para coordinar los accesos a memoria. Cada thread se identifica por su identificador de hilo, que es el número de hilo dentro del bloque (threadID), los cuales pueden ser definidos con dos o tres dimensiones para ayudar con los direccionamientos complejos basados en el thread ID.

En CUDA el número de threads por bloque que puede haber está limitado por un número máximo determinado (512 threads en CUDA 1.0-1.3, 1024 en Fermi). A pesar de ello, en CUDA los threads pueden ser agrupados en bloques de threads para que en una única ejecución del kernel puedan ejecutarse un número de threads mucho mayor. Como consecuencia de esto, se produce una reducción de en la cooperación de los distintos hilos, porque los hilos de los diferentes bloques no pueden comunicarse entre sí. El conjunto de bloques a su vez se le denomina grid

del kernel, y lo tenemos que definir antes de cada ejecución de cada kernel donde debemos especificar el número de threads y el número de bloques.

Con este modelo los kernels pueden ejecutarse sin necesidad de recompilar en varios dispositivos con diferentes capacidades de paralelismo: si tiene muy pocas capacidades de paralelismo, un dispositivo puede ejecutar todos los bloques de threads del grid de manera secuencial, sin embargo, si tiene mucha capacidad de paralelismo puede ejecutar todos los bloques de threads del grid en paralelo [12].

3.4.2. Modelo de memoria

Como podemos observar en la figura 3.9, en el modelo de memoria de CUDA podemos encontrar la memoria: constante, de textura, global, local, compartida y los registros.

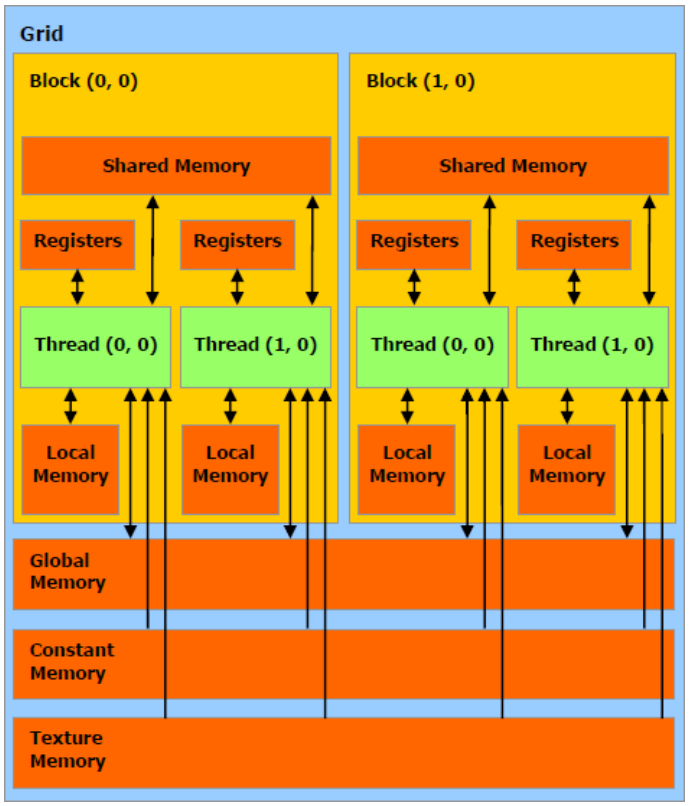

Figura 3.9: Modelo de memoria en la arquitectura CUDA

Cuando un thread se encuentra en ejecución en el dispositivo, se podrá acceder únicamente a la memoria DRAM y a la memoria del chip teniendo en cuenta que el tipo acceso a los espacios de memoria son los siguientes [12]:

	Registros de cada hilo	Memoria local de cada hilo	Memoria compartida de cada hilo	Memoria global de cada grid	Memoria constante de cada grid	Memoria textura de cada grid
Lectura	✓	✓	✓	✓	✓	✓
Escritura	✓	✓	✓	✓	✗	✗

Tabla 3.2: Tipo de acceso en el modelo de memoria de CUDA

La memoria global, constante, y la memoria de textura pueden ser leídas o escritas por el host (fragmento de código c que se ejecuta en CPU) y serán persistentes con todas las ejecuciones del kernels lanzados (fragmento de código CUDA que se ejecuta en GPU). (Ver figura 3.9)

3.5.Arquitectura de openCL

OpenCL (Open Computing Lenguaje) se trata de un estándar abierto para la programación de propósito general multiplataforma de una colección heterogénea de CPU's, GPU's y otros dispositivos informáticos organizado en una única plataforma. Es más que un lenguaje. OpenCL es un marco para la programación en paralelo e incluye un lenguaje de programación, API para la librería de control de plataformas, librerías y un sistema de ejecución para soportar el desarrollo de software. Usando OpenCL, por ejemplo, un programador puede escribir programas de uso general que se ejecutan en la GPU, sin necesidad de mapear sus algoritmos en un API de gráficos 3D como OpenGL o DirectX [15].

El objetivo de OpenCL es permitir a programadores expertos escribir código portable y eficiente. Esto incluye a desarrolladores de librerías. Por lo tanto OpenCL proporciona una abstracción de hardware de bajo nivel, además de un framework de apoyo a la programación.

Para describir las ideas básicas detrás de OpenCL, utilizaremos una jerarquía de modelos:

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

3.5.1. Modelo de Plataforma

El modelo de la Plataforma de OpenCL se ilustra en la figura 3.10. El modelo consiste en un host conectado a uno o más dispositivos de computación de OpenCL. Un dispositivo de computación de OpenCL está dividido en una o más unidades de computación (UC), que se dividen en uno o varios elementos de procesamiento (PE). Cálculos en un dispositivo de producirse dentro de los elementos de procesamiento.

Una aplicación OpenCL se ejecuta en una máquina de acuerdo a los modelos nativos de la plataforma de acogida. La aplicación OpenCL envía comandos desde el host (procesador principal del sistema que ejecuta el programa principal) para ejecutar cálculos en los elementos de procesamiento dentro de un dispositivo. Los elementos de procesamiento dentro de una unidad de procesamiento ejecutan un único flujo de instrucciones como unidades SIMD (ejecutar a la par de una única secuencia de instrucciones) o como unidades SPMD (cada elemento de procesamiento mantiene su propio contador de programa) [15].

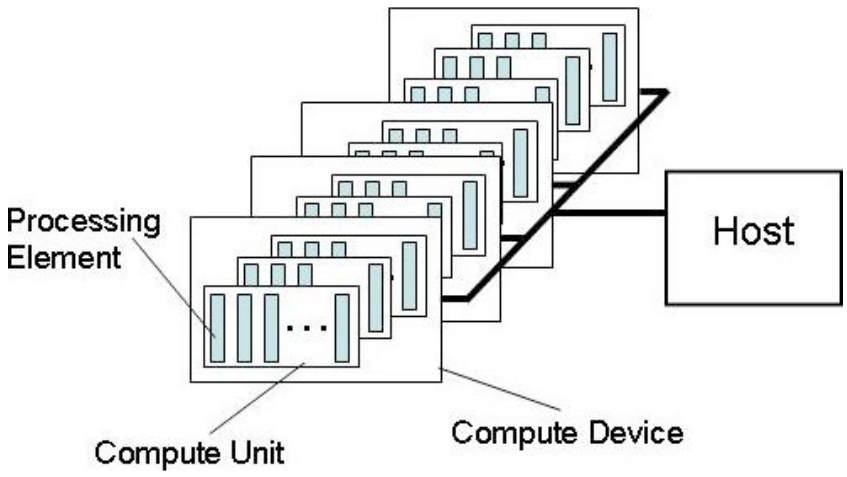

Figura 3.10: Modelo de plataforma en OpenCL

3.5.2. Modelo de ejecución

La ejecución de un programa de OpenCL se lleva a cavo en dos partes: los kernels que se ejecutan en uno o más dispositivos de OpenCL y el host que ejecuta el programa principal (igual que en CUDA). El host define el contexto para los kernels y gestiona su ejecución [15].

Cuando el host envía un kernel al device para su ejecución, se define un espacio de índice. Una instancia del kernel se ejecuta para cada punto en este espacio de índice. Esta instancia del kernel se llama work-item (más o menos equivale a un thread de CUDA) y se identifica por su punto en el espacio de índice, que le proporciona un identificador global (global ID) para cada work-item. Cada work-item ejecuta el mismo código, pero cada uno ejecuta un camino de ejecución específico.

Los work-items se organizan en work-groups (equivalentes a los bloques de threads de CUDA). A los work-groups se les asigna un único work-group ID con la misma dimensionalidad que el espacio de índice que se utiliza para los work-items. A los work-items se les asigna un ID único local dentro de un work-group para que un único work-item pueda ser identificado de manera unívoca por su global ID o por una combinación de su ID local y work-group ID. Los work-items de un work-group dado se ejecutan de manera concurrente en lo elementos de proceso de una única unidad de computación.

El espacio de índice soportado por OpenCL se denomina NDRange (parecido a grid de CUDA). Un NDRange es un espacio de índice N-dimensional, donde N es uno, dos o tres. Un NDRange se define por una matriz de enteros de longitud N especificando la extensión del espacio de índice en cada dimensión a partir de un índice de desplazamiento F (cero por defecto). Cada work-item tiene un ID global y un ID local que son tuplas N-dimensionales. Los componentes del ID global son valores en el rango de F, a F más el número de elementos en esa dimensión menos uno.

Cada elemento de trabajo se puede identificar de dos maneras: en términos de un índice global, y en términos de un índice del grupo de trabajo además de un índice local dentro de un grupo de trabajo.

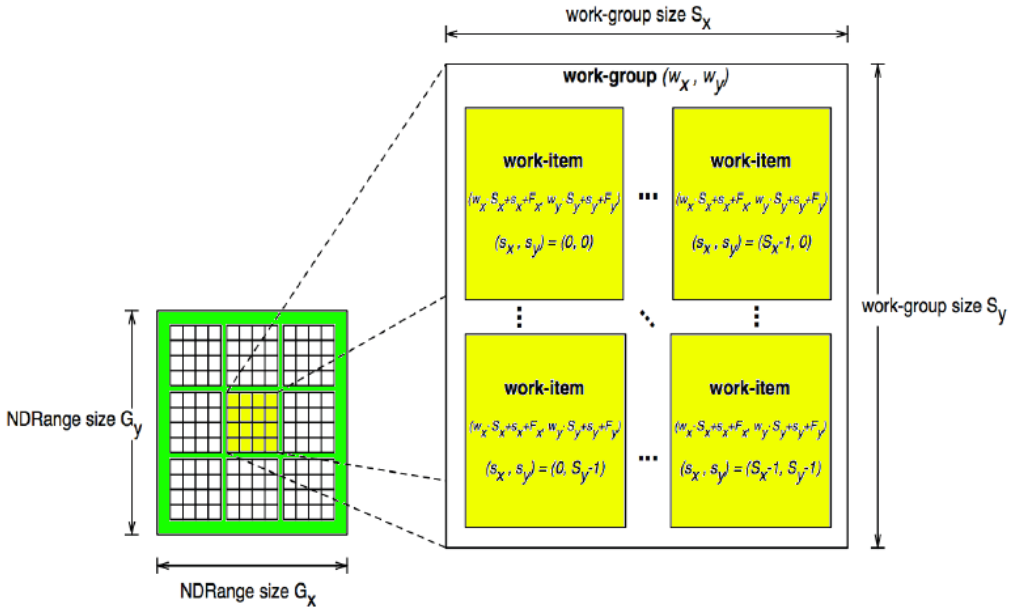

Figura 3.11: Explicación del NDRange

Por ejemplo, consideremos un espacio de índice de 2 dimensiones en la figura 3.11. Definimos el espacio de índice de entrada para los work-items (G_x, G_y), el tamaño de cada work-group (S_x, S_y) y el ID de desplazamiento global (F_x, F_y). Los índices globales se definen en G_x por G_y. Los índices locales se definen en un espacio de índice en S_x por S_y donde el número de work-items de un work-group es $S_x * S_y$. Dado el tamaño de cada grupo de trabajo y el número total de elementos de trabajo se puede calcular el número de grupos de trabajo. En un espacio de dos dimensiones el espacio de índice se utiliza para identificar un work-group. Cada work-item se identifica por su identificador global (g_x, g_y) o por la combinación de la identificación de work-group (w_x, w_y), el tamaño de cada work-group (S_x, S_y) y el ID local (s_x, s_y) dentro del grupo de trabajo de tal manera que [15]:

$$(g_x, g_y) = (w_x * S_x + s_x + F_x, w_y * S_y + s_y + F_y)$$

3.5.3. Modelo de memoria

Los work-items cuando están ejecutando un kernel tienen acceso a cuatro regiones de memoria distintas como se ilustra en la figura 3.12 [15]:

- Memoria Global: Esta región de memoria permite la lectura/escritura a todos los work-items en todos en los work-groups. Los work-items pueden leer o escribir en cualquier elemento de un objeto de memoria. Las lecturas y escrituras en la memoria global puede ser almacenadas en caché en función de las capacidades de que el dispositivo.
- Memoria constante: Una región de memoria global que se mantiene constante durante la ejecución de un kernel. El host asigna e inicializa los objetos de memoria colocada en la memoria constante.
- Memoria local: Es una región de memoria local a un work-group. Esta región de memoria se puede utilizar para asignar variables que son compartidas por todos los work-item en ese work-group. Puede ser implementado como regiones de memoria dedicada en el dispositivo de OpenCL. Alternativamente, la región de memoria local puede ser asignada a los sectores de la memoria global.
- Memoria Privada: Es privada a un work-item. Las variables definidas en memoria privada de un work-item no son visibles a otros work-items.

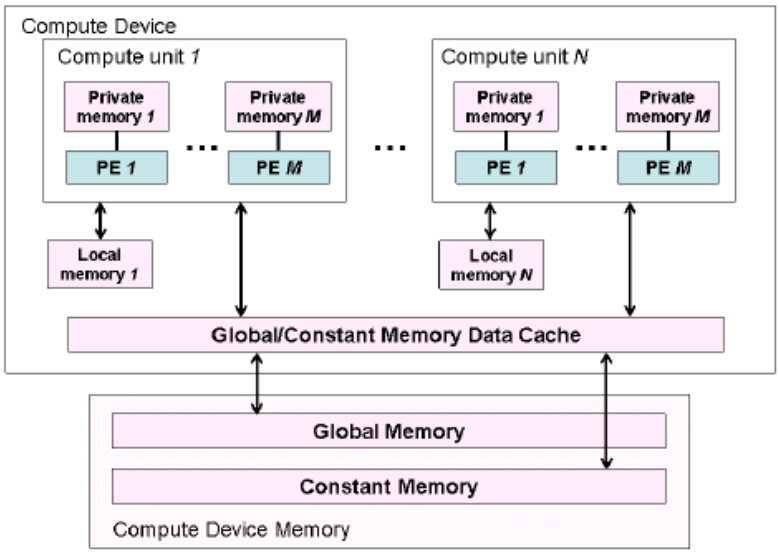

Figura 3.12: Modelo de memoria en OpenCL

A continuación se detalla según las regiones de memoria, las capacidades de reserva y acceso a la memoria tanto del host como del kernel:

	Privada	Local	Constante	Global
Host	No reservable Inaccesible	Reserva dinámica Inaccesible	Reserva dinámica Lectura/Escritura	Reserva dinámica Lectura/Escritura
Kernel	Reserva estática Lectura/Escritura	Reserva estática Lectura/Escritura	Reserva estática Solo lectura	No reservable Lectura/Escritura

Tabla 3.3: Tipo de acceso en el modelo de memoria de openCL

Con esto el host crea objetos en la memoria del dispositivo mediante comandos openCL de manera que se produce una interacción host/memoria del dispositivo. Dicha interacción puede ser las operaciones de copia explícita y/o mapeado de regiones de objetos de memoria en el espacio de direcciones del host.

OpenCL usa una consistencia relajada entre los work-items. De tal forma que la memoria local es consistente para los work-items de un work-group tras una

barrera de grupo, lo mismo sucede para la memoria global pero no se garantiza consistencia entre work-groups diferentes [15].

3.5.4. Modelo de programación

La ejecución OpenCL soporta los modelos de programación de paralelismo de datos y de tareas, así como híbridos de estos dos modelos [15]:

- El modelo de programación paralela en datos define que, un cálculo en términos de una secuencia de instrucciones se aplica a varios elementos de un objeto de memoria. Además el espacio de índice asociado al modelo de ejecución de OpenCL define los work-items y cómo se mapean los datos en los work-items. En un modelo estrictamente paralelo de datos, existe una asignación uno a uno entre el work-item y el elemento en un objeto de memoria sobre el cual un kernel puede ser ejecutado en paralelo. Sin embargo, openCL implementa una versión relajada del modelo de programación paralela en datos donde no se requiere un mapeo estricto uno a uno. Así, el paralelismo de datos es un modelo muy natural dada la ejecución replicada de núcleos sobre un espacio n-dimesional.
- El modelo de programación paralelo en tareas define un modelo en el que se ejecuta una sola instancia de un kernel independiente de cualquier espacio de índice. Es equivalente a la ejecución de un núcleo en una unidad de computación con un work-group que contenga un solo work-item.

4. Algoritmo PPI

A continuación se hará una breve explicación de las técnicas básicas para el procesamiento de imágenes hiperespectrales para pasar posteriormente a la explicación del algoritmo PPI.

4.1.Técnicas para el procesamiento de imágenes. Algoritmo PPI

Debido a la inmensa cantidad de información espectral que tienen las imágenes hiperespectrales es necesario aplicar distintas técnicas de transformación, justamente para disminuir el contenido de datos redundantes. Una de estas técnicas es la de compresión espacialmente aplicando un índice de pureza de píxel (Pixel Purity Index o PPI), que permite representar la distribución en el espacio de los píxeles espectralmente más puros.

La mayor parte de las técnicas de procesamiento de imágenes hiperespectrales presuponen que las mediciones que realizan los sensores, en cada punto determinado de la imagen, se constituyen por diferentes materiales a un nivel de subpíxel, es decir, que cada punto está definido por una combinación lineal de firmas asociadas a componentes espectralmente puros denominados endmembers [16] (esto es lo denominado modelo lineal de mezcla). Esta mezcla a nivel subpíxel puede producirse por la insuficiencia de resolución en el sensor hiperespectral. Este modelo responde bien cuando los componentes espectralmente puros está están separados en el espacio. A continuación detallaremos un poco más en qué consiste el modelo lineal de mezcla.

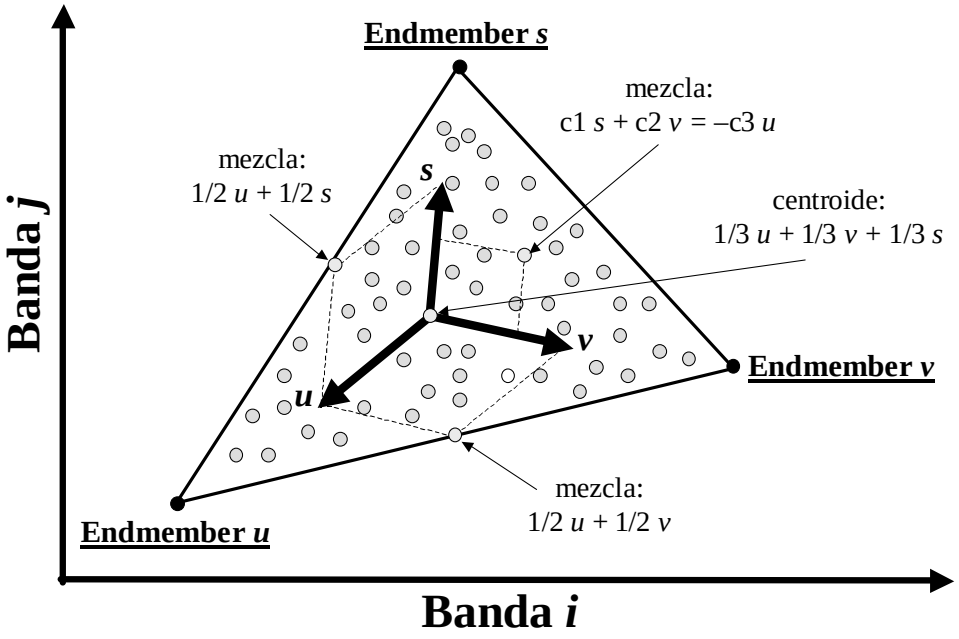

Figura 4.1: La ilustración nos muestra como se visualiza el modelo lineal de mezcla

Como podemos observar en la figura 4.1, podemos hacer una simplificación del modelo lineal de mezcla para dos bandas, cada una de ellas representada en un eje. También podemos ver como los tres puntos más extremos que delimitan a todos los demás son los denominados endmembers, es decir, los puntos espectralmente más puros, esto quiere decir que los vectores asociados a esos puntos forman un nuevo sistema de referencia con origen el centroide del triángulo, a partir de los cuales, todos los demás se pueden poner como combinación lineal de los endmembers.

La elección de los endmembers se debe realizar de manera correcta para obtener los puntos espectralmente más puros. Una de las maneras de obtenerlos es la utilizada por el algoritmo PPI (Píxel Purity Index). El algoritmo PPI fue desarrollado por Boardman en [6], es un algoritmo de procesamiento de imágenes hiperespectrales que se basa en la generación aleatoria en la nube de puntos de vectores unitarios de manera que cada punto se proyecta sobre cada vector unitario generado. Hecho esto, se identifica los puntos más extremos en cada vector generado, a continuación se incrementa un contador en los puntos extremos (índice de pureza), que indica el número de veces que ha sido seleccionado como

punto extremo. Después de haberse un número de iteraciones razonable, se seleccionan como endmembers aquellos puntos que más veces han sido seleccionados. A continuación se detalla de manera más profunda el algoritmo PPI:

1. En primer lugar el algoritmo inicializa a 0 el índice de pureza de cada punto.
2. A continuación se genera un vector unitario aleatorio llamado skewer cuyo objetivo es dividir la nube de puntos en dos.
3. Posteriormente, como ya hemos mencionado, se proyectan cada punto de la imagen hiperespectral en el skewer para identificar los puntos extremos, los cuales se les incrementa el índice de pureza en una unidad.
4. Los pasos 2 y 3 se realizan tantas veces como número de iteraciones sean especificadas.
5. Realizado lo anterior estamos en disposición de utilizar un valor umbral dado por parámetro para seleccionar aquellos puntos que lo superen. Estos puntos se seleccionarán como puntos espectralmente más puros.
6. Los píxeles seleccionados podrá visualizarse con una herramienta interactiva que permita visualizar diagramas de dispersión denominada *N-Dimensional Visualizer* que se trata de una herramienta de ENVI que permite realizar diagramas de dispersión de los primeros autovectores obtenidos.

Este algoritmo lo describiremos a continuación de manera más formal. Inicialmente necesitaremos como entrada el cubo hiperespectral de datos F de N dimensiones, el nº de endmembers que se obtendrán (E), el número de vectores aleatorios (K), y por ultimo un valor umbral para seleccionar los endmembers. Las salidas serán el conjunto de E endmembers $\{e_i\}_{i=1}^{E}$. Con esto, el algoritmo se describe de la siguiente manera:

1. Generación de los K skewers aleatorios: $\{skewer_j\}_{j=1}^{K}$

2. Proyecciones de los puntos en el skewers: para cada uno, toda la muestra de vectores de píxeles f_i del conjunto de datos original, se proyecta sobre el skewer j a través del producto escalar para encontrar una muestra de vectores en sus extremos (máximo y mínimo), así, construimos el conjunto de extremos para el skewer j $S_{extrema}(skewer_j)$. Con esto la operación realizada será:

$$|f_i \cdot skewer_i|$$

Con la obtención de skewers extremos anterior, podría pasar que la misma muestra de vectores, fuesen seleccionados por más de un conjunto de extremos a la vez ($S_{extrema}(skewer_j)$). Para solucionarlo definiremos:

$$I_s(f_i) = \begin{cases} 1 \; si \; f_i \in S \\ 0 \; si \; f_i \notin S \end{cases}$$

Donde $I_s(f_i)$ indica si el elemento f_i pertenece o no al conjunto S de puntos extremos.

3. Una vez sabemos el si se repiten la muestra de vectores, ponemos a cada punto el número de veces que ha sido seleccionado como punto extremo, para ello usamos la función definida anteriormente de la siguiente manera:

$$N_{PPI}(f_i) = \sum_{j=1}^{K} I_{S_{extrema}(skewer_j)} \cdot (f_i)$$

4. A continuación se obtienen los endmembers, que son aquellos píxeles cuyo número de veces que hayan sido seleccionados ($N_{PPI}(f_i)$) supere el umbral dado por parámetro.

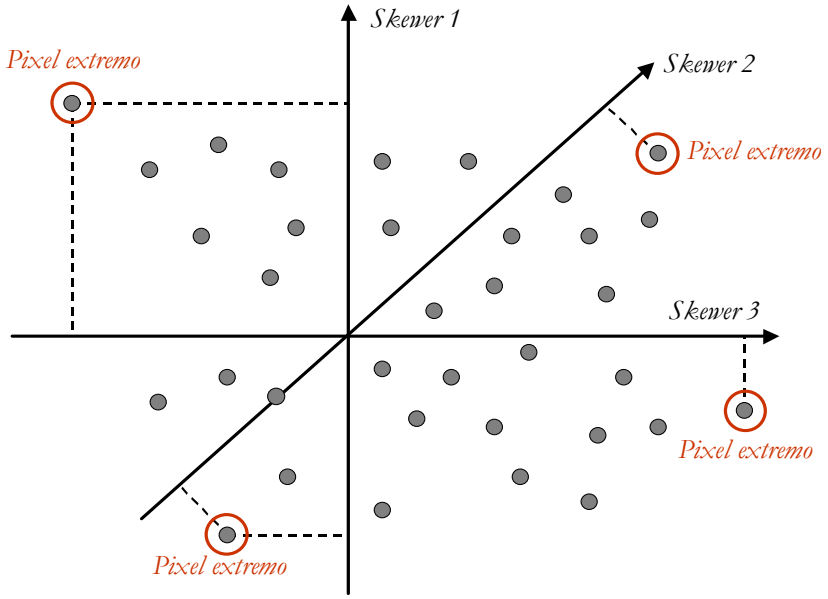

Figura 4.2: Representación gráfica del algoritmo PPI.

4.2.Paralelización de PPI en GPU

En este apartado detallaremos como ha sido paralelizado el algoritmo PPI en los kernels implementados para una mejor compresión del PFC.

Para la ejecución de este algoritmo existen tres fases claramente diferenciadas, la generación de skewers aleatorios (vectores aleatorios), el cálculo de la pureza de cada píxel y la selección de los endmembers. La primera fase ha sido implementada en un único kernel, y las otras dos fases del algoritmo han sido implementadas en otro kernel, ambas han sido implementadas tanto en CUDA como en openCL. A continuación mostraremos cada una de las fases de una manera más detallada:

1. Generación de los skewers aleatorios: para la generación de los skewers se ha implementado de manera paralela un algoritmo denominado Mersenne Twister [17-18] que se trata de un generador de números pseudo-aleatorios que proporciona la generación rápida de números de muy alta calidad, habiendo sido diseñado específicamente para solventar muchas de los fallos encontrados en algoritmos más antiguos. Por tanto este algoritmo lo usaremos para la generación de aleatorios ya que no todas las versiones de CUDA y

openCL implementan en la API generadores de números aleatorios que nos serán muy útiles en la obtención de los skewers aleatorios. Se generarán tantos números aleatorios (skewers aleatorios) como número de iteraciones necesitemos en el algoritmo PPI.

2. La segunda fase del algoritmo PPI consiste en el cálculo de la pureza de cada píxel de manera paralelizada. Dada una imagen de entrada (d_imagen), los skewers de entrada generados en el punto anterior (d_random), y las dimensiones de dicha imagen de entrada (num_lines, num_samples y num_bands), en esta fase obtendremos la pureza de manera que, sobre cada skewer serán proyectados todos los píxeles de la imagen, teniendo en cuenta que la proyecciones sobre un skewer se procesará en un único hilo/work-item (según estemos en CUDA/openCL). Por tanto, si necesitamos n iteraciones de entrada **(número de bloques * número de hilos)** tenemos que haber generado n skewers (en d_random) y en cada uno de ellos serán proyectados todos los puntos de d_imagen.
3. El último paso es la selección de los endmembers. En este paso tenemos que obtener los índices de los valores mayor y menor que hemos obtenido en el paso anterior y lo almacenamos en d_res_parcial, para posteriormente fuera del kernel incrementar el valor en una unidad en el elemento que ocupa el lugar dado por este índice extraído. Por ejemplo, si en la estructura d_res_parcial el valor más alto se encuentra en la posición 30 y el más bajo en la 289 (figura 4.3), se incrementarán en uno las posiciones 30 y 289 la imagen resultante que comienza valiendo 0 en todos sus elementos y será la que almacene los resultados finales del algoritmo. La tarea de extraer el máximo y el mínimo se realiza de forma lineal. Finalmente tras todas las iteraciones obtenemos la matriz de resultados de la imagen usada para mostrar la imagen procesada.

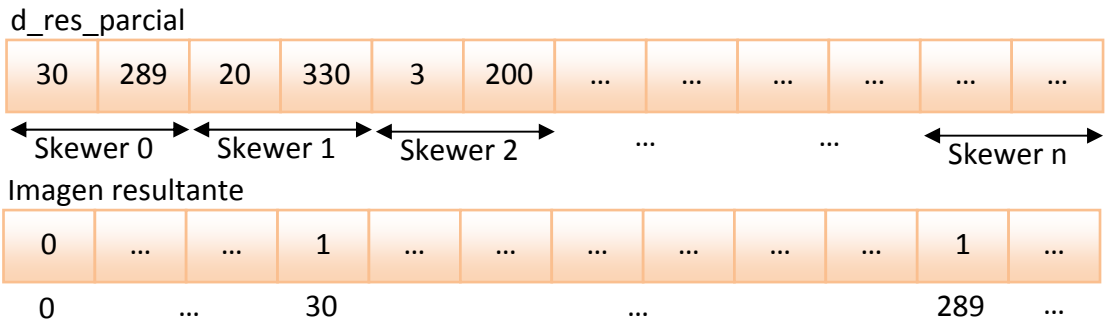

Figura 4.3: Ilustración que muestra como son las estructuras de datos usadas.

4.3.PPI en CUDA vs PPI en openCL

En este apartado haremos una exposición comparativa de cómo se ha implementado el algoritmo PPI para el lenguaje de programación CUDA para a continuación hacer lo propio para el lenguaje de programación openCL para cada uno de los ficheros que forman PPI en este PFC (figura 4.4). El objetivo de esto es el de hacer ver al lector las similitudes y diferencias que se encuentran en las implementaciones desarrolladas para el presente PFC. También es una buena comparativa en el caso que el lector quiera ver qué hacer para desarrollar un algoritmo en ambos lenguajes. Cabe destacar que para que las comparativas sean lo más realistas posibles, el código PPI en ambos lenguajes ha sido desarrollado de la manera más parecida que se ha podido conseguir.

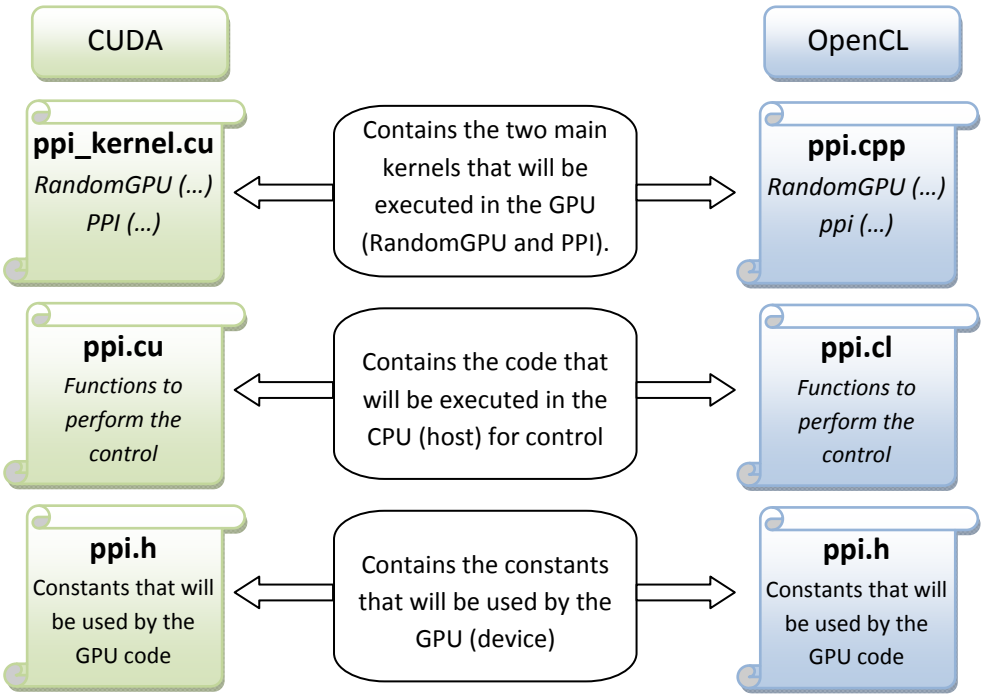

Figura 4.4: Ilustración que muestra los ficheros que forman PPI en CUDA y en openCL

Como podemos observar en la figura 4.4, para la implementación del algoritmo ppi tanto en CUDA como en openCL se han usado principalmente tres archivos que contienen las distintas partes que se indican en dicha figura:

- Comencemos con la exposición comparativa de los ficheros que contienen el kernel (parte que se ejecuta en GPU) en ambos lenguajes. Como se puede observar en la figura 4.5 en estos ficheros no se encuentran demasiadas diferencias entre ambos lenguajes, debido en parte a que con openCL se ha mantenido la sintaxis de CUDA, también se debe a que en openCL se equipara la noción de work-item con la de hilo en CUDA, y la noción de work-group (openCL) con la de bloque (CUDA), y por consiguiente, se ejecuta en openCL una instancia del kernel por cada work-item, igual que en CUDA. Veamos entonces las diferencias para el kernel que ejecuta ppi. Podemos observar en la figura 4.5 una comparación entre parte

del kernel ppi en openCL con la misma parte de CUDA. En la etiqueta número 1 de cada código, podemos ver la diferencia de cómo se define la imagen de entrada a procesar. En la etiqueta número 2 podemos ver cómo obtener el id global de un work-item/hilo entre todos los demás (vemos como en openCL en este caso es mucho más sencillo de conseguir utilizando una función específica para ello). En la etiqueta número 3 podemos observar la definición de variables privadas en openCL (locales en CUDA), y las locales a un work-group en openCL son compartidas en CUDA con los demás hilos. En la etiqueta número 4, en la parte de openCL tenemos que la id local de un work-item dentro de work-group, mientras que en CUDA tenemos la obtención de la id local de un hilo dentro de un bloque de hilos. En la etiqueta número 5 podemos observar la diferencia entre como se sincronizan las barreras en openCL y en CUDA, en openCL es necesario indicar si la barrera es local (CLK_LOCAL_MEM_FENCE) o global (CLK_GLOBAL_MEM_FENCE).

OpenCL

```
__kernel void ppi(__global const float * d_im
    __private uint idx =  get_global_id(0)
    __private float pemax; // producto escalar maximo
    __private float pemin; // producto escalar minimo
    __private float pe; //producto escalar
    __private int v,d;
    __private int imax=0;
    __private int imin=0;
    __local float s_pixels[Tamanio_Vector];
    __private float l_rand[224];

    pemax=MIN_INT;
    pemin=MAX_INT;
    //Copiamos un pixel de la memoria global a los registros
    for (uint k=0; k<num_bands; k++){
        l_rand[k]=d_random[idx*num_bands+k];
    }

    for(uint it=0; it<num_lines*num_samples/N
        barrier(CLK_LOCAL_MEM_FENCE);
        //Copiamos N_Pixels pixels a la memoria compartida
        if(get_local_id(0)<N_Pixels){
            for(uint j=0; j<num_bands; j+
                s_pixels[get_local_id(0)+
            }
        }
        barrier(CLK_LOCAL_MEM_FENCE);//o CLK_LO
```

CUDA

```
__global__ void PPI(float * d_imagen, float *d_rand
{
    int idx =  blockDim.x * blockIdx.x+threadIdx.x;

    float pemax; // producto escalar maximo
    float pemin; // producto escalar minimo
    float pe; //producto escalar
    int v,d;
    int imax=0;
    int imin=0;
    pemax=MIN_INT;
    pemin=MAX_INT;

    __shared__ float s_pixels[Tamanio_Vector];
    float l_rand[224];
    //Copiamos un pixel de la memoria global a los registros
    for (int k=0; k<num_bands; k++){
        l_rand[k]=d_random[idx*num_bands+k];
    }
    for(int it=0; it<num_lines*num_samples/N_Pixels
        //Copiamos N_Pixels pixels a la memoria compartida
        if(threadIdx.x<N_Pixels){
            for(int j=0; j<num_bands; j++){
                s_pixels[threadIdx.x+N_Pixels*j
            }
        //}
    }
    syncthreads();
```

Figura 4.5: Comparación visual de los kernels implementados en CUDA y openCL

Como hemos podido ver en la figura anterior, ambos lenguajes son muy similares en cuanto al kernel, con las pequeñas variaciones que hemos hecho mención en el párrafo anterior. Pero principalmente, la diferencia la podemos encontrar en la parte de host (recordemos que se trata de la parte que se ejecuta en CPU) la cual mostraremos a continuación.

- Seguidamente exponemos la comparativa de los ficheros que contienen la parte de host en ambos lenguajes, para ello en primer lugar mostraremos a nivel general y de forma simplificada como se lanza la ejecución de un kernel cualquiera en openCL, y a continuación, haremos lo mismo con CUDA:

En openCL para ejecutar un kernel cualquiera hay que seguir los siguientes pasos:

1. Primero obtenemos de las plataformas de openCL a usar que se detectan en el sistema, y seleccionamos la deseada.
2. Seguidamente creamos de un contexto para todos los dispositivos. Un contexto es el entorno en el que los núcleos se ejecutan y el dominio en el que se define la sincronización y gestión de memoria.
3. A continuación, obtenemos los dispositivos de openCL a usar, por ejemplo se pueden detectar varias GPU's que pueden ejecutar los kernels en un mismo sistema. Seleccionaremos aquel dispositivo que más nos interese.
4. Posteriormente, creamos las colas de comandos. Las operaciones de OpenCL que se someten a una cola de comandos para su ejecución. Por ejemplo, los comandos de OpenCL emiten los kernels para su ejecución en un dispositivo de cálculo (GPU's, CPU's,…), manipular objetos de memoria, etc.

5. Después de lo anterior crearemos los buffers de datos. Los buffers se pueden definir como una zona de memoria accesible desde el kernel a través de un puntero, que se ejecuta en un dispositivo, y el host los usa para pasar parámetros al kernel a través de la API de openCL.
6. Inmediatamente después de crear los buffers, cargamos el kernel, pasamos los parámetros de entrada y de salida, y compilamos los núcleos a ejecutar mediante la API de openCL.
7. Finalmente la ejecución de los mismos con la API de openCL.
8. Una vez terminada la ejecución del kernel, para obtener los resultados, en el host será necesario leer los resultados de las operaciones realizadas.

A continuación, para hacernos una idea mostraremos un código genérico (en parte pseudocódigo) del host para ejecutar un algoritmo en openCL, pero a pesar de ser genérico, ha sido el código base para nuestro PPI en openCL. Cabe destacar que el objetivo de mostrar el código genérico es el de compararlo con el de CUDA, y por lo tanto, este código es un código simplificado sin definiciones de variables y sin control de errores para una comparación más sencilla:

```
//Obenemos las plataformas disponibles y seleccionamos una
clGetPlatformIDs(0, NULL, &num_platforms);
platforms=(cl_platform_id *) malloc(sizeof(cl_platform_id)*num_platforms);
clGetPlatformIDs(num_platforms, platforms, NULL);
selected_platform=0;
//Creamos un contexto para dicha plataforma
context=clCreateContextFromType(cps, CL_DEVICE_TYPE_ALL,NULL,NULL,&err);
//seleccion del dispositivo
clGetContextInfo(context,CL_CONTEXT_DEVICES,0,NULL,&size_devices);
num_devices=size_devices/sizeof(cl_device_id);
devices= (cl_device_id*) malloc(size_devices);
selected_device=0;
//creamos la cola de comandos
clCreateCommandQueue(context, devices[selected_device],
                     CL_QUEUE_PROFILING_ENABLE,NULL);
//Creamos los buffers de datos ( una cola para cada variable)
A=clCreateBuffer(context, CL_MEM_WRITE_ONLY, TAMAÑO-VARIABLE-A,NULL,NULL);
B=...
...
Z=...
//Cargamos y compilamos el kernel deseado y le pasamos los parámetros
clCreateProgramWithSource(context,1,&source,sourceSize,&err);
clBuildProgram(program, num_devices, devices, NULL, NULL, NULL);
cl_kernel kernel=clCreateKernel(program,"NombreKernel",&err);
    //Al kernel le paso todos los n argumentos de entrada y salida
clSetKernelArg(kernel,0,TAMAÑO-A,&A);
clSetKernelArg(kernel,1,TAMAÑO-B,&B);
...
clSetKernelArg(kernel,n,TAMAÑO-Z,&Z);
//Pongo la ejecución del kernel en la cola especificando el tamaño del grupo de trabajo y el tamaño de cada grupo
clEnqueueNDRangeKernel(command_queue,kernel, 1,NULL,
                       &global_work_size, &local_work_size,
                       0,NULL,&event);
cl_ulong start,end;
clWaitForEvents(1, &event);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
                        sizeof(cl_ulong), &start, NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
                        sizeof(cl_ulong), &end, NULL);
clReleaseEvent(event);
//Leo el resultado
clEnqueueReadBuffer(command_queue,d_random,CL_TRUE, 0,
                    RAND_N * sizeof(cl_float),h_random,0,NULL,NULL);
```

Figura 4.6: Ejemplo de host simplificado para ejecutar un programa en openCL

Al igual que hemos hecho en openCL, en CUDA veremos a continuación como ejecutar un kernel cualquiera siguiendo los siguientes pasos:

1. En primer lugar hacemos la reserva de memoria necesaria que se usarán como parámetros del kernel.

2. A continuación copiamos los datos de entrada a dicha reserva de memoria, es decir, copiamos los datos de entrada del host hacia el device.
3. Posteriormente podemos poner ya en ejecución el kernel indicando el número de bloque e hilos y pasando por parámetro las reservas de memoria reservada.
4. Para finalizar será necesario obtener los resultados que han sido generados en el kernel.

```
//Variables que se usarán como parámetros de entrada o salida.
h_A =  malloc(TAMAÑO-A);
h_B =  malloc(TAMAÑO-B);
...
h_Z =  malloc(TAMAÑO-Z);
//Reservamos la memoria necesaria para los parámetros del kernel.
cudaMalloc((void**)&d_A, TAMAÑO-A);
cudaMalloc((void**)&d_B, TAMAÑO-B);
...
cudaMalloc((void**)&d_B, TAMAÑO-B);
//Copiamos los parámetros de entrada de host a device
cudaMemcpy(d_A, h_A, TAMAÑO-A, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, TAMAÑO-B, cudaMemcpyHostToDevice);
...
cudaMemcpy(d_Z, h_Z, TAMAÑO-Z, cudaMemcpyHostToDevice);
//Ejecución kernel PPI
PPI <<<N_BLOQUES,N_HILOS>>>(d_A, d_B, ..., d_Z);
cudaThreadSynchronize();
//Copiamos los resultados de la GPU a la CPU, suponiendo que d_A es parámetro de salida
cudaMemcpy(h_A, d_A, TAMAÑO-A, cudaMemcpyDeviceToHost);
```

Figura 4.7: Ejemplo de host simplificado para ejecutar un programa en CUDA

- En el archivo ppi.h no ha habido apenas variación porque contiene las definiciones de constantes que se usan en los demás ficheros y por tanto no es necesario hacer comparativa alguna.

Como hemos podido apreciar en estas comparativas, en los kernels no hemos encontrado demasiadas diferencias, pero para poner en ejecución desde el host un kernel, es mucho más engorroso hacerlo en openCL que en CUDA. Ello se debe a que openCL es un lenguaje multiplataforma, y por tanto, es necesario obtener las

plataformas que hay disponibles y seleccionar las deseadas. En openCL también es necesario construir el kernel en tiempo de ejecución, como consecuencia de ello hay que hacer uso de más API's dando lugar a mayor cantidad de código fuente en el host. En CUDA no es necesario todas estas cosas ganando en facilidad de programación, pero con el inconveniente de ser un lenguaje con menor portabilidad (sólo es portable entre algunas GPU's nVidia).

En resumen, en lo referente a la programación, con openCL ganamos en portabilidad, pero con CUDA ganamos en facilidad de programación.

5. Imágenes de entrada y resultados Obtenidos

A continuación se mostrarán las imágenes utilizadas para el proyecto haciendo una breve descripción de las mismas. Seguidamente a las imágenes de entrada, se hará una presentación de las imágenes obtenidas como resultados de las distintas ejecuciones, presentando además unos resultados comparativos de los tiempos de ejecución de los algoritmos implementados tanto en CUDA como en openCL.

5.1.Imágenes de entrada utilizadas

La primera imagen hiperespectral a presentar es una imagen que ha sido generada sintéticamente por ordenador, para realizar las pruebas. Las características más importantes expresadas en la cabecera de la imagen son:

Característica de la cabecera	Valor
Nombre de la imagen	sintetica.bsq
Número de samples	100
Número de líneas	100
Número de bandas	224
Desplazamiento de la cabecera	0
Tipo de archivo	Envi estándar
Tipo de datos	2 (Entero)

Tabla 5.1: Características de la imagen hiperespectral sintética utilizada

La razón por la cual se ha usado en primer lugar una sintética en lugar de una real, es que con las sintéticas tenemos la posibilidad de evaluar los resultados con mayor objetividad al generar esta imagen de una manera más controlada debido a que el grado de pureza que tiene cada píxel puede controlarse a priori. Seguidamente expondremos como se ha generado la imagen sintética, indicando como se han obtenido las firmas espectrales.

Para la generación de la imagen hiperespectral sintética, se ha partido de una imagen tomada por el sensor AVIRIS en el año 1997 de la zona de Jasper Rige en el

condado de California. A partir de ella podemos usar las firmas empleadas para crear simulaciones de imágenes hiperespectrales. A continuación mostraremos cual ha sido el proceso para obtener dicha imagen. Primero se indica la abundancia de cada determinado material para cada píxel de la imagen teniendo en cuenta que cada firma espectral se define con un conjunto de abundancias que cumplen restricciones de no-negatividad y de suma unitaria. Y a continuación es necesario añadir ruido a la imagen hiperespectral, para ello se ha generado números aleatorios entre -1 y 1 siguiendo una distribución normal y por tanto su desviación estándar es 1 y la media es 0, y seguidamente se añade este ruido generado, a cada píxel de la imagen. A continuación se ilustra como ha sido el proceso de generación de la imagen:

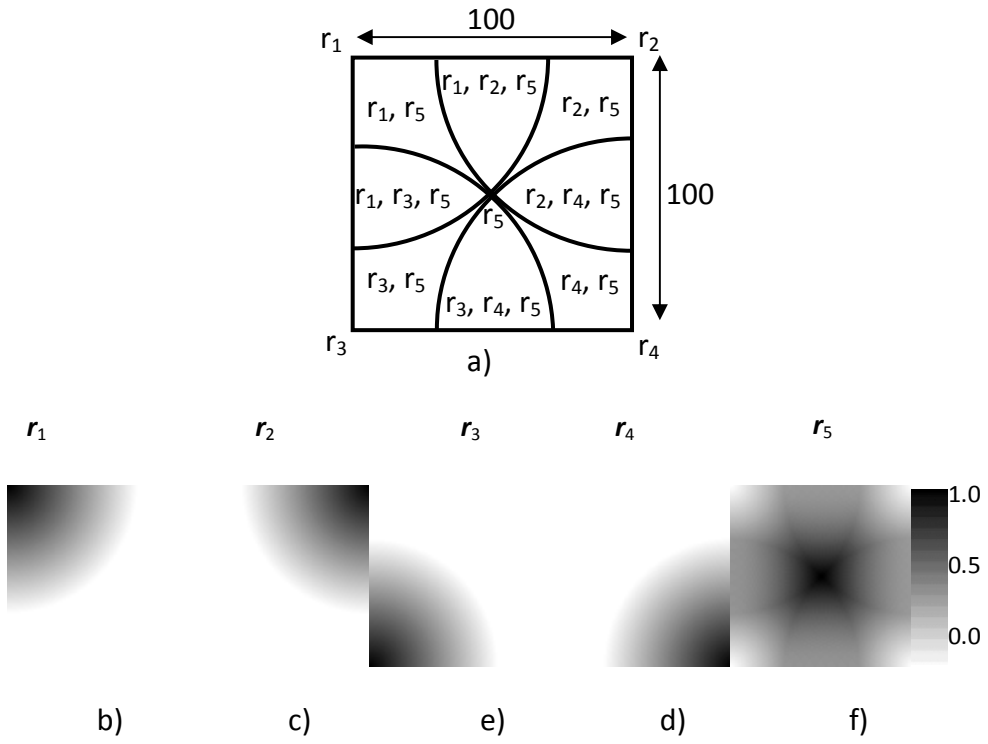

Figura 5.1: Ilustración de cómo se ha formado la imagen sintética.

En la figura 5.1 podemos observar en a) como se distribuye la imagen en los cinco píxeles puros (distintos endmembers que tratará de encontrar el algoritmo ppi), expresados como r_1, r_2, r_3, r_4 y r_5. De la imagen b hasta la f, se muestran los

mapas de abundancia utilizados para la generación de la imagen sintética ("sintética.bsq").

Al simular una imagen obtenida con el sensor AVIRIS, podemos ver en la tabla 5.1 que el número de bandas es de 224. Como hemos mencionado, el sensor AVIRIS obtiene imágenes en la banda de frecuencias de 400 nanómetros 2500 nanómetros, es decir, que obtiene imágenes en el espectro visible e infrarrojo, y por tanto, la imagen *sintetica.bsq* se puede ver a la vista del ojo humano. La imagen hiperespectral utilizada es la siguiente:

Figura 5.2: Representación de la imagen hiperespectral sintética con distintas bandas de color

En esta imagen podemos observar diferentes colores y ha sido generada con el programa ENVI coloreando distintas bandas de manera que una de las bandas perteneciente al color rojo se ha representado en la capa roja de la imagen, una de las bandas perteneciente al color verde se ha representado en la capa verde de la imagen y una de las bandas perteneciente al color azul se ha representado en la capa azul de la imagen. De aquí deducimos que la imagen será parecida a como la

veríamos a simple vista, pero no lo es porque estamos perdiendo la mayor parte del espectro visible, además también concluimos estamos perdiendo la representación del espectro de la imagen.

Otra imagen utilizada para la obtención de los resultados ha sido la imagen cuprite que se trata de una imagen de una zona minera de cobre, las características de la imagen son las siguientes:

Tipo de campo	Valor
Nombre de la imagen	Cuprite.bsq
Número de samples	350
Número de líneas	350
Número de bandas	188
Desplazamiento de la cabecera	0
Tipo de archivo	Envi estándar
Tipo de datos	2 (Entero)

Tabla 5.2: Características de la imagen hiperespectral cuprite utilizada

Al igual que antes presentamos la imagen coloreada en este caso para la imagen cuprite:

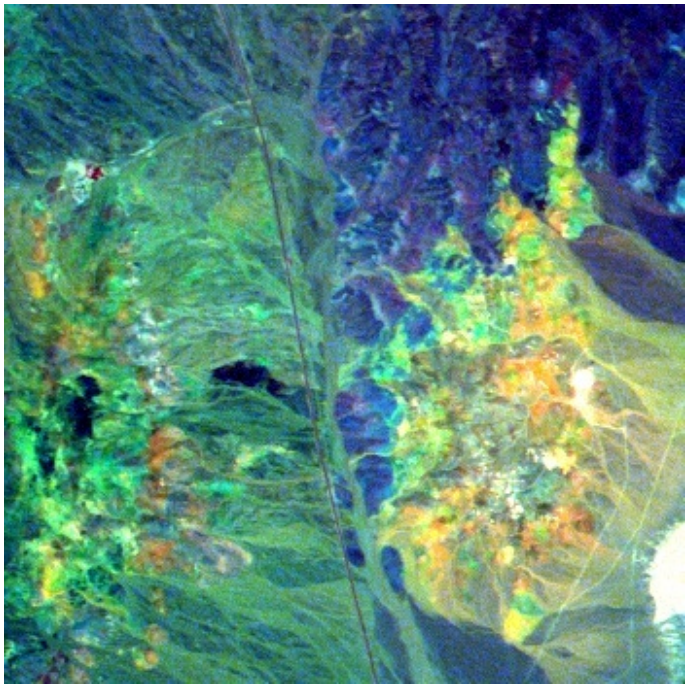

Figura 5.3: Representación de la imagen hiperespectral cuprite con

Como sabemos, las imágenes que hemos presentado anteriormente nos permiten evaluar los tiempos de ejecución de los algoritmos implementados. Pero para evaluar la calidad de las imágenes sintéticas generadas es necesario hacer un análisis de la *precisión estricta* de la imagen sintética.

Para el análisis de precisión estricta utilizaremos la imagen de salida generada en el algoritmo PPI, es decir, la imagen de dimensiones $TILLNE_{samples} * TILLTE_{lines}$ que contiene el número de veces que un píxel (x, y) ha sido seleccionado como endmember. Para el análisis de precisión necesitamos saber cómo es la imagen de entrada que procesará el algoritmo PPI implementado. Así, como hemos mencionado, el análisis de precisión se hará mediante la imagen "sintetica.bsq", debido a que es una imagen que ha sido generada de manera controlada, y por tanto, sabemos a priori cuál es la imagen resultado generada por PPI.

Con esto, la precisión estricta considera un acierto si el píxel extremo que hemos seleccionado en PPI es realmente un endmember, por tanto en el caso de la imagen sintetica.bsq, un acierto es un endmember seleccionado en cualquiera de las cuatro esquinas o en el centro de la imagen. Para saber si un píxel de la matriz está en la misma posición que un endmember podemos usar el siguiente método: leemos por separado cada uno de los mapas y los almacenamos en matrices

diferentes. A continuación recorremos celda a celda nuestra matriz de resultado y si en alguno de los mapas el valor de la celda cuya posición coincide con la que se está recorriendo es de 255 entonces nos encontramos en una posición donde hay un endmember y como consecuencia sumamos el valor de nuestra celda a una variable que llamaremos aciertos.

Para calcular la precisión estricta, en primer lugar debemos calcular el número total de aciertos, es decir, un acierto es la suma de los valores de las cuatros esquinas y del centro de la imagen resultado de la ejecución de PPI. Una vez hemos calculado los aciertos, calculamos el número de incrementos, es decir, el número de veces que ha sido seleccionado como endmember un píxel cualquiera de la imagen. Para ello sumamos todos los valores de la imagen resultado generada por el algoritmo PPI. Finalmente, para calcular la precisión estricta podemos aplicar la siguiente ecuación:

$$Precisión\ estricta = \frac{nº\ aciertos}{nº\ incrementos} * 100$$

Figura 5.4:Explicación de cómo se realiza el análisis de precisión

5.2.Resultados en la tesla del CETA-CIEMAT

En este apartado en primer lugar mostraremos las imágenes resultados que se han obtenido para unos parámetros dados, y a continuación, mostraremos una serie de gráficas de tiempos de ejecución para las distintas pruebas que han sido realizadas en el clúster de GPU's del CETA-ciemat de Trujillo, posteriormente se hará un análisis comparativo de las mismas y observaremos el rendimiento de ambos lenguajes.

5.2.1. Imágenes obtenidas como resultado de las ejecuciones

Algunas de las imágenes resultado obtenidas han sido las siguientes. En las siguientes imágenes resultados, los puntos de la imagen han sido coloreados con diferentes colores en función del número de veces que han sido seleccionados como endmember en una determinada iteración.

Figura 5.5: Escala de colores de los píxeles de la imagen.

De izquierda a derecha, de menos veces seleccionado a más veces.

En primer lugar ilustramos el resultado de la ejecución de la imagen "sintetica.bsq" para la ejecución en CUDA:

Nombre	Precisión estricta	Iteraciones	Número bloques	Número hilos	Tiempo de ejecución Kernel ppi
sintetica.bsq	**20.46%**	**15360**	**120**	**128**	**41346.56ms**

Tabla 5.3: Datos sobre la ejecución de la imagen sintetica.bsq en CUDA

La imagen resultado de la ejecución del algoritmo en CUDA se ilustra a continuación:

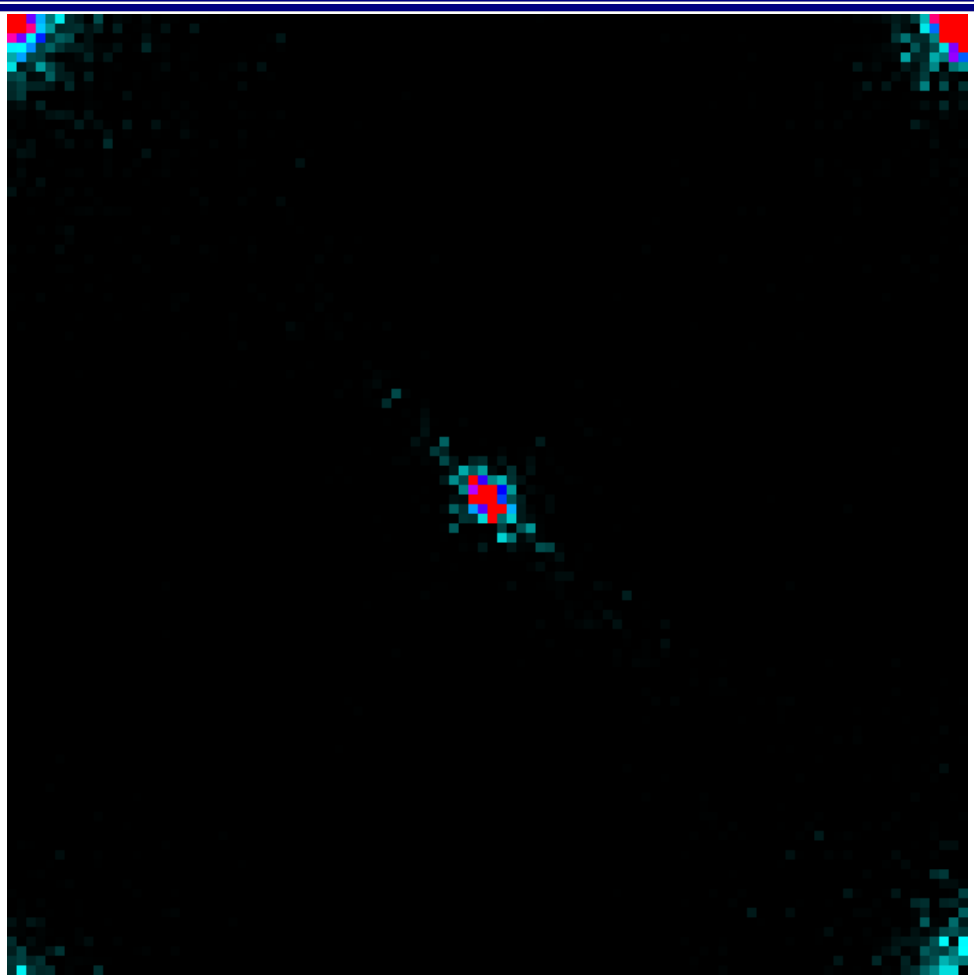

Figura 5.6: Resultado de la ejecución en CUDA para 15360 iteraciones de sintetica.bsq

Para comprender mejor la imagen podemos mencionar que como el algoritmo PPI ha encontrado más o menos los endmembers que planteábamos cuando generamos la imagen sintética (sintetica.bsq), como vemos se encuentran cada uno en una esquina y otro en el centro de la imagen, justo como señalamos cuando generamos la imagen.

Otra imagen resultado generada ha sido la obtenida de la ejecución de "Cuprite.bsq" en CUDA para los siguientes datos. Cabe mencionar que para la imagen cuprite, como no es una imagen real, no se le puede calcular la precisión estricta debido a que no se puede conocer cuáles son a priori los endmembers de la imagen.

Nombre	Iteraciones	Número bloques	Número hilos	Tiempo de ejecución Kernel ppi
Cuprite.bsq	15360	120	128	42462.95ms

Tabla 5.4: Datos sobre la ejecución de la imagen Cuprite.bsq en CUDA

La imagen resultado de la ejecución del algoritmo en CUDA se ilustra a continuación:

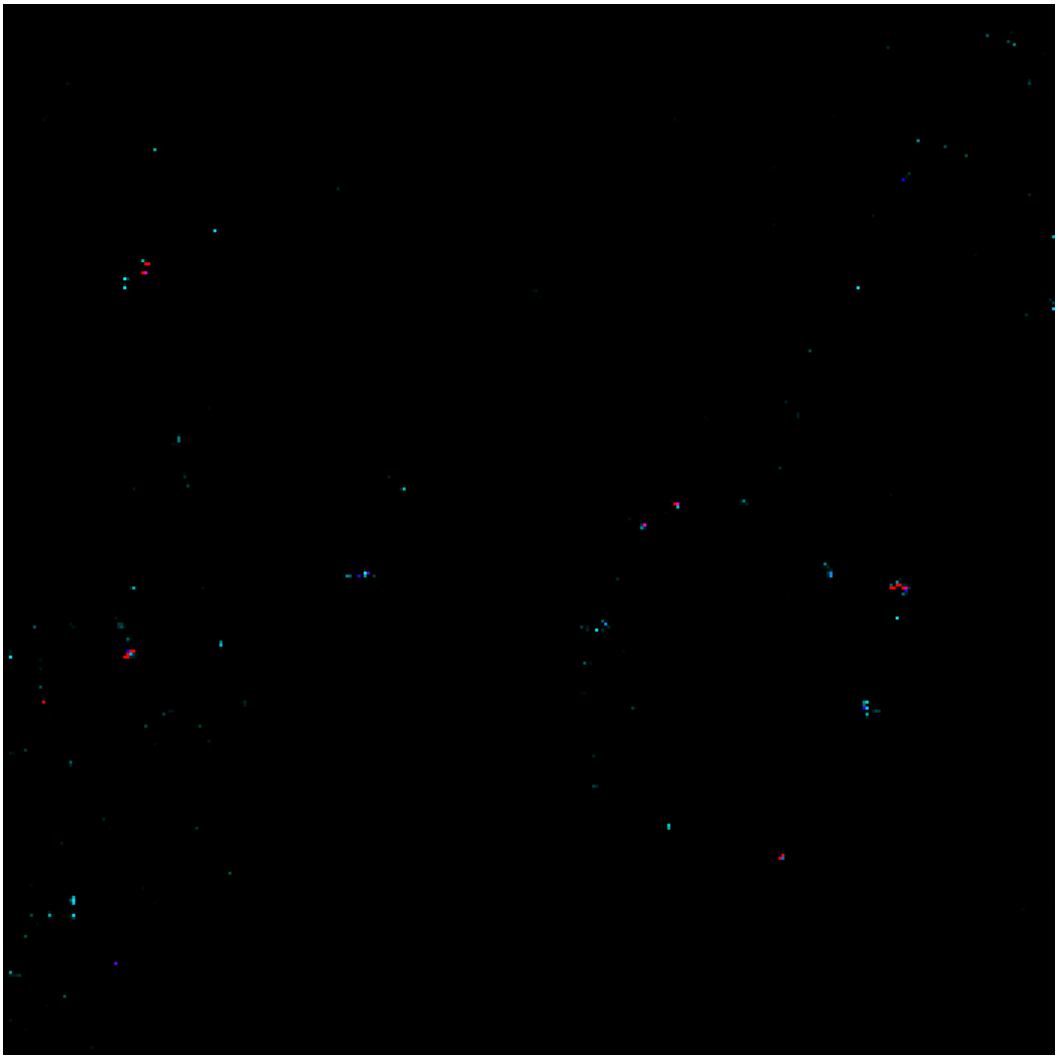

Figura 5.7: Resultado de la ejecución en CUDA para 15360 iteraciones de Cuprite.bsq

Al igual que antes, podemos ver como los píxeles seleccionados como endmembers han sido coloreados con diferentes colores en función del número de veces que hayan sido seleccionados. También de la imagen podemos ver, a

diferencia de la imagen anterior, como no hay píxeles endmembers que sigan un determinado patrón como en la sintética, es decir, que los endmember se distribuyen de manera regular por la imagen. Esto se debe a que es una imagen real y por tanto los distintos materiales puros que podemos encontrar en la imagen no se encuentran en una zona concreta sino como podemos encontrarlos a lo largo de toda la imagen, bien es cierto que podríamos tener una imagen real de satélite en los que los píxeles de la imagen si siguiesen un patrón determinado pero no es lo habitual.

Anteriormente hemos visto las imágenes resultado para las ejecuciones en CUDA, seguidamente haremos lo propio para el leguaje openCL. Para openCL primero ilustramos el resultado de la ejecución de la imagen “sintetica.bsq” con los siguientes datos:

Nombre	Precisión estricta	Iteraciones	Número bloques	Número hilos	Tiempo de ejecución Kernel ppi
sintetica.bsq	**20.92%**	**15360**	**120**	**128**	**3251.82ms**

Tabla 5.5: Datos sobre la ejecución de la imagen sintetica.bsq en openCL

La imagen resultado de la ejecución del algoritmo en CUDA se ilustra a continuación:

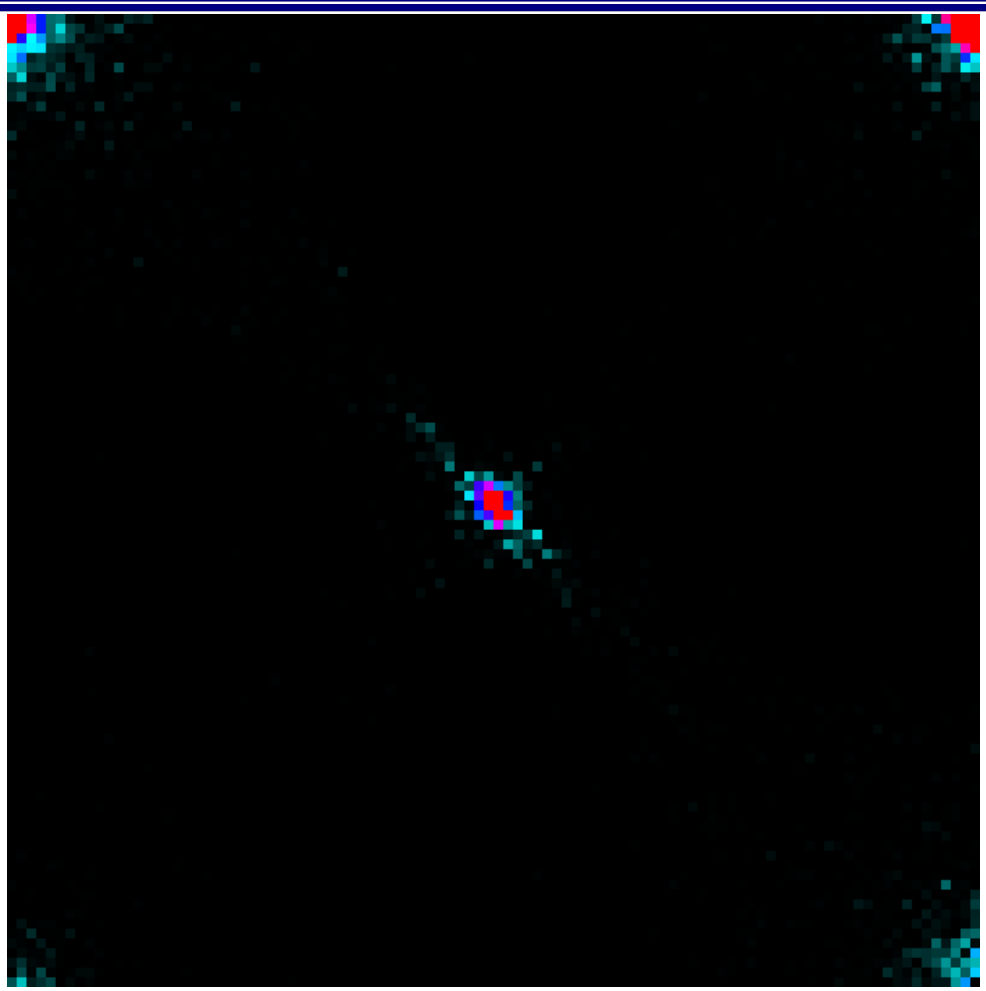

Figura 5.8: Resultado de la ejecución en openCL con 15360 iteraciones de sintetica.bsq

Como podemos observar, el resultado es muy similar al obtenido para la imagen sintética en CUDA y por tanto, podemos mencionar que como el algoritmo PPI ha encontrado más o menos los endmembers que planteábamos cuando generamos la imagen sintética (sintetica.bsq), como vemos se encuentran cada uno en una esquina y otro en el centro de la imagen, justo como señalamos cuando generamos la imagen.

Otra imagen resultado generada ha sido la obtenida de la ejecución de "Cuprite.bsq" en openCL con los siguientes datos. Cabe mencionar que para la imagen cuprite, como no es una imagen real, no se le puede calcular la precisión estricta tampoco en openCL debido a que no se puede conocer cuáles son a priori los endmembers de la imagen.

Nombre	Iteraciones	Número bloques	Número hilos	Tiempo de ejecución Kernel ppi
Cuprite.bsq	15360	120	128	34126.32ms

Tabla 5.6: Datos sobre la ejecución de la imagen Cuprite.bsq en openCL

La imagen resultado de la ejecución del algoritmo en CUDA se ilustra a continuación:

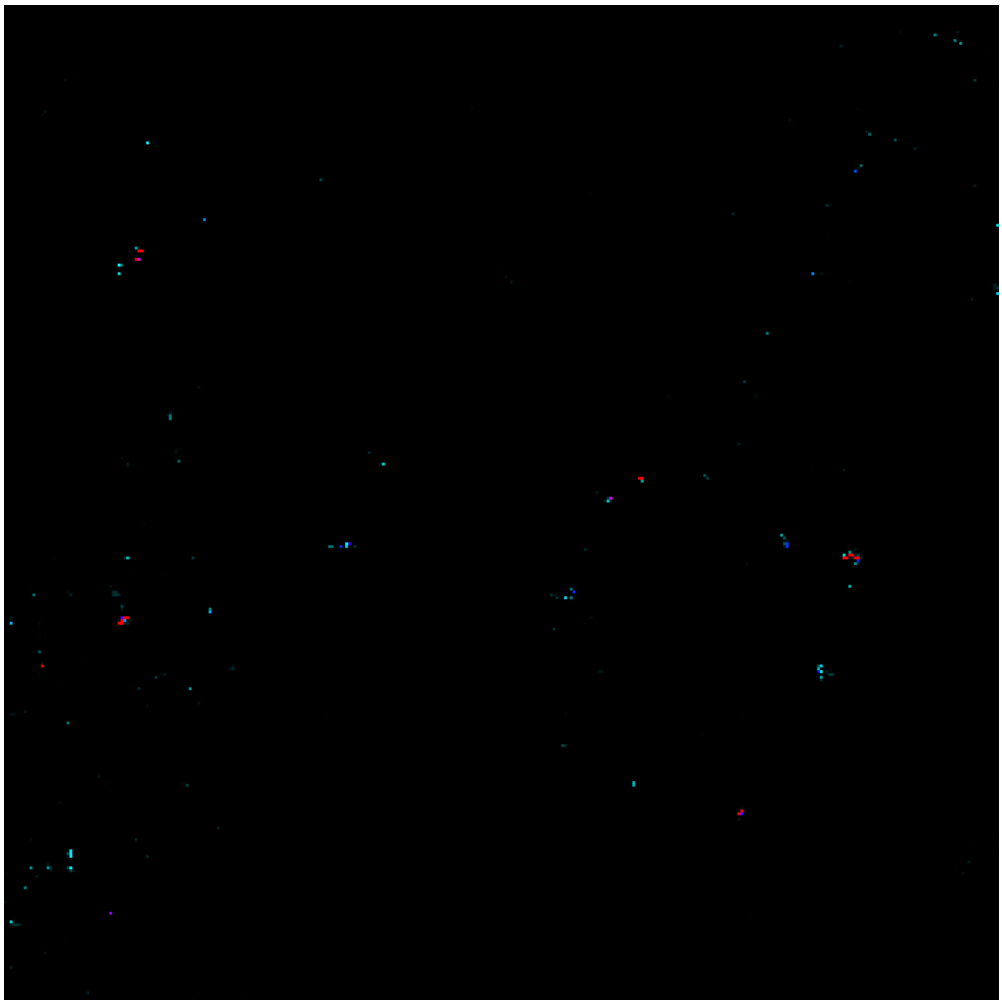

Figura 5.9: Resultado de la ejecución en openCL para 15360 iteraciones de Cuprite.bsq

Al igual que antes con la ejecución de CUDA, podemos ver como no hay píxeles endmembers que sigan un determinado patrón como en la sintética, es decir, que los endmember se distribuyen de manera regular por la imagen. También vemos

como la imagen no es demasiado diferente a la generada por el algoritmo de CUDA (como es normal).

5.2.2. Comparativa de los tiempos de ejecución obtenidos

Para explicar mejor los tiempos de ejecución de PPI, tanto en el algoritmo implementado en CUDA como en el implementado en openCL, se ha medido el rendimiento de las dos partes esenciales en el algoritmo que se ejecutan en la GPU. Esas dos partes son la generación de números aleatorios para los skewers (la denominaremos lanzar random) y la ejecución del algoritmo PPI en si (la denominaremos lanzar PPI). Para esto en primer lugar vamos a mostrar los tiempos de ejecución para 1000 iteraciones tanto para CUDA como para openCL, para las imágenes “sintetica.bsq” y “Cuprite.bsq”. Posteriormente haremos lo propio con 15360 iteraciones.

- Para 1000 iteraciones: en las gráficas podremos observar como variarán los tiempos de ejecución para diferentes valores tanto el número de work-groups como el de work-items en openCL o el número de bloques como el de hilos en CUDA. Seguidamente mostramos los resultados de la ejecución para la imagen “sintetica.bsq”:

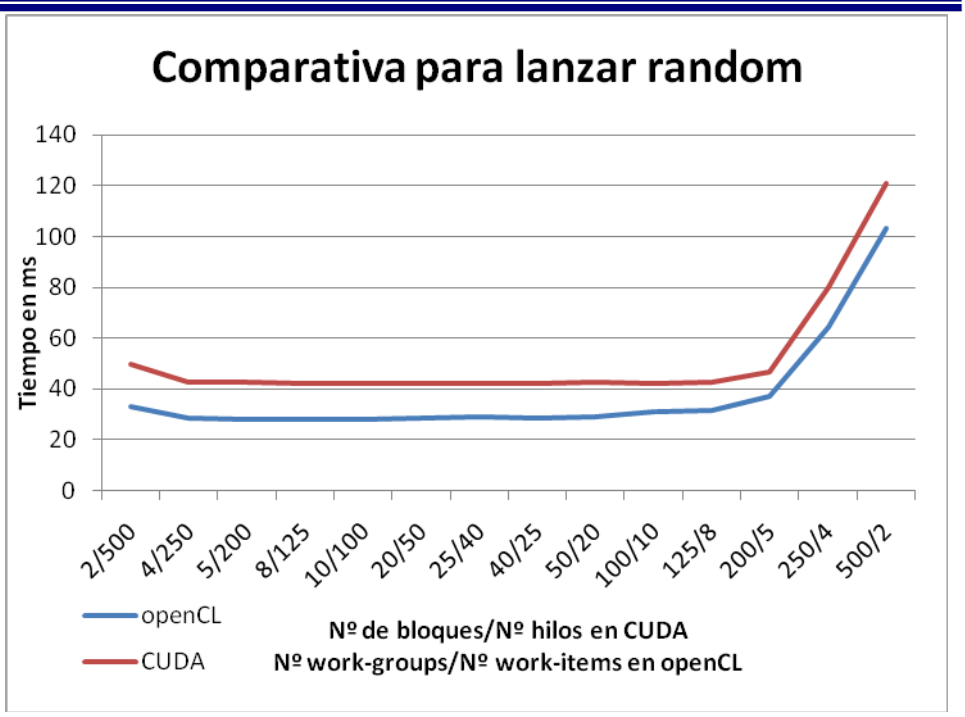

Figura 5.10:Comparativa gráfica CUDA vs openCL, 1000 iteraciones, sintética (I)

Podemos observar en la figura anterior, cómo varía el tiempo de ejecución cuando generamos lo skewers aleatorios, en función de la configuración de nº de bloques y número de hilos. Como vemos, openCL, para las pruebas que hemos realizado con la configuración hardware señalada en puntos anteriores, es más rápido en la generación de skewers.

Otra observación que podemos destacar es que, por un lado es que tanto en CUDA como openCL con pocos bloques/work-groups y muchos hilos/work-items (ejemplo: 2 bloques 500 hilos), los tiempos de ejecución no son óptimos debido a que al haber pocos bloques, habrá muchos hilos colaborando y en las barreras habrá mayor probabilidad de espera entre hilos. Por otro lado, vemos en la figura que con muchos bloques y pocos hilos (ejemplo: 250 bloques y 4 hilos) los tiempos de ejecución se disparan en ambos lenguajes. Esto se debe a que con esta configuración de bloques/work-groups e hilos/work-items el número de bloques que se pueden ejecutar en de manera concurrente es para la GPU Tesla C1060 de 240 (uno por cada multiprocesador), y por tanto, si tenemos que ejecutar más bloques que esa cifra, lo hará uno detrás de otro, es decir, en serie. Aparte de lo

anterior, también influye el hecho que a menos hilos colaborando y más bloques, mayor tiempo de ejecución.

A continuación mostramos la gráfica comparativa para las 1000 iteraciones y la imagen "sintetica.bsq" (igual que en la figura 5.10) para el kernel que se encarga del procesamiento del algoritmo PPI:

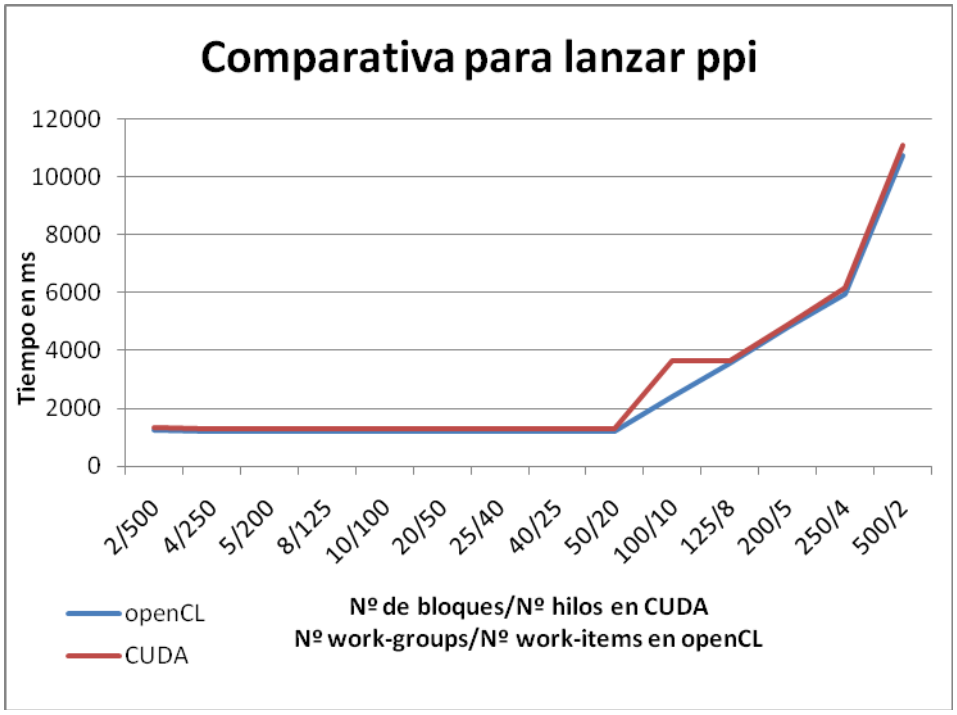

Figura 5.11: Comparativa gráfica CUDA vs openCL, 1000 iteraciones, sintética (II)

En la figura 5.11 podemos destacar como la diferencia del tiempo de ejecución en ambos lenguajes es ínfima, y por tanto, podríamos afirmar que para 1000 iteraciones en la imagen "sintetica.bsq" no hay prácticamente diferencia en los tiempos de ejecución en ambos lenguajes. A pesar de esta afirmación, observamos que ligeramente la ejecución en el lenguaje openCL es más rápida que la ejecución en el lenguaje CUDA. Igual que en la imagen anterior podemos ver en la imagen que cuando el número de bloques se empieza a hacer grande, y por tanto el número de hilos es bajo, los tiempos de ejecución empiezan a hacerse cada vez mayores. Esto puede suceder por lo mismo que hemos mencionado anteriormente.

Anteriormente hemos hecho la comparativa para "sintetica.bsq", ahora haremos lo mismo para la imagen "Cuprite.bsq". Igual que en las figuras 5.10 y 5.11,

el número de iteraciones procesadas será 1000. A continuación mostramos la gráfica comparativa de los tiempos de ejecución para la generación de números aleatorios:

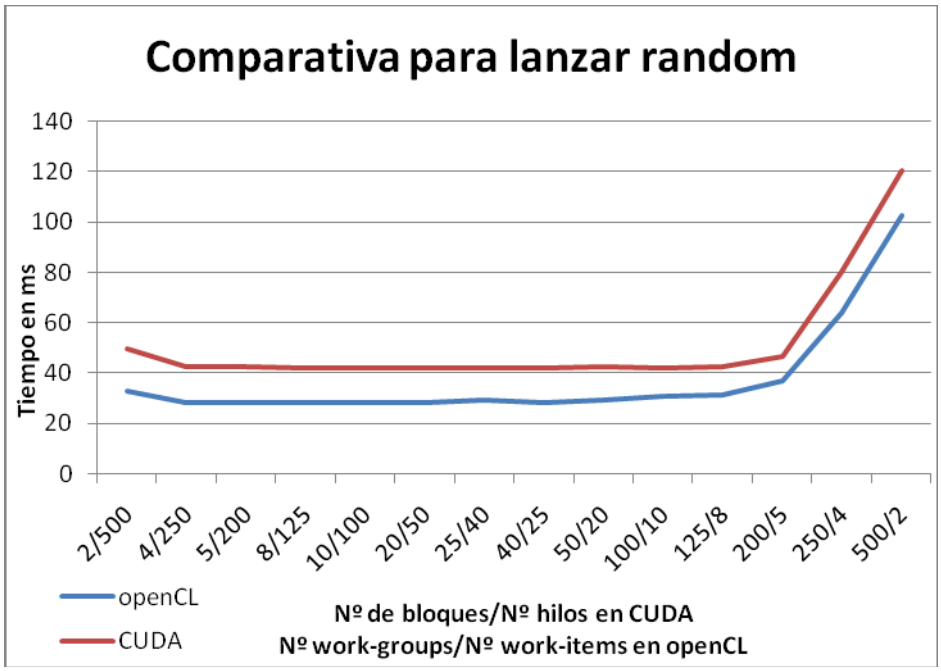

Figura 5.12: Comparativa gráfica CUDA vs openCL, 1000 iteraciones, cuprite (I)

Como vemos, no hay demasiada diferencia en la forma de la gráficas en ambos lenguajes, ni si quiera en los tiempos de ejecución de las mismas. Esto se debe a que se generan el mismo número de aleatorios, y por tanto, la ejecución de la generación de aleatorios no es diferente. Con esto, podremos llegar a las mismas conclusiones que obtuvimos para la imagen sintética para la generación de los skewers aleatorios.

Ahora mostraremos la gráfica comparativa de los tiempos de ejecución para las distintas configuraciones de las ejecuciones del kernel PPI. Al igual que antes el número de iteraciones será 1000 para la imagen Cuprite.bsq:

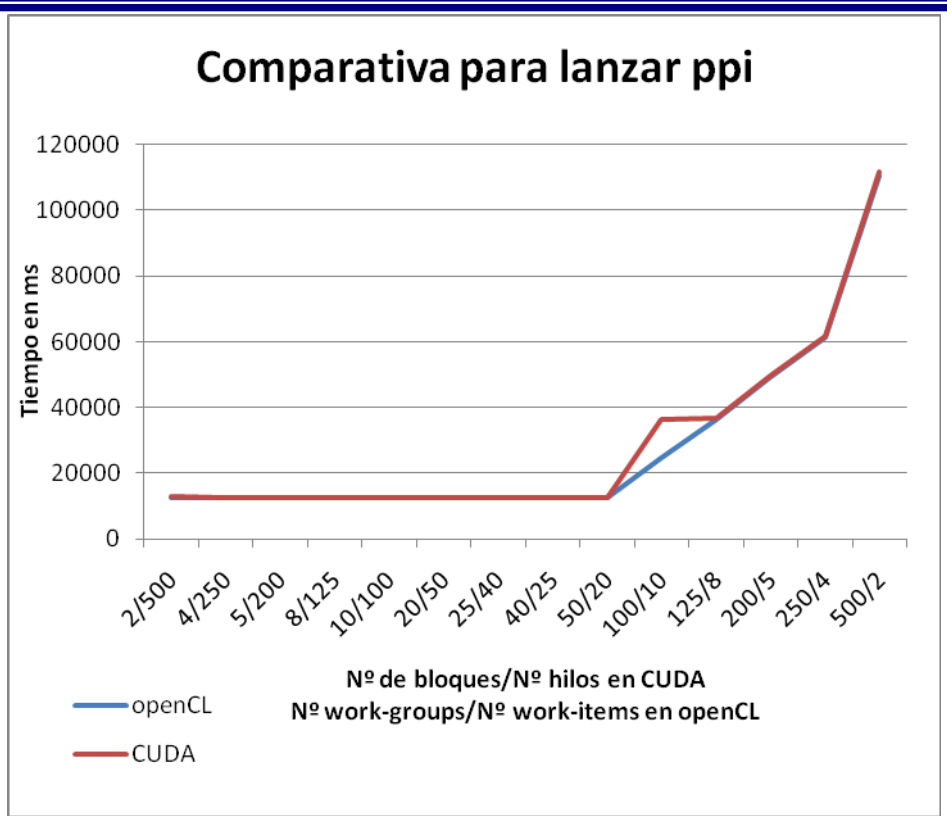

Figura 5.13: Comparativa gráfica CUDA vs openCL, 1000 iteraciones, cuprite (II)

Podemos ver en la figura anterior como la gráfica es muy similar a la obtenida para la imagen "sintetica.bsq", y por tanto, las conclusiones obtenidas son similares. La conclusión nueva que podemos obtener de la ejecución puede ser que con el procesamiento de una imagen real de mayor resolución, recordemos que la de Cuprite es de 350 por 350 píxeles, serán necesario mayor un tiempo de ejecución. Por ejemplo en la figura 5.13 para 25 bloques y 40 hilos el algoritmo PPI se ejecuta en unos 12000 milisegundos, mientras que en esta misma ejecución para la imagen sintética (figura 5.11) se ejecuta en unos 1200 milisegundos (un orden de unas 10 veces menor para éstas imágenes concretamente).

- Para 15360 iteraciones: a continuación haremos la comparativa para las imágenes de entrada que serán "sintetica.bsq" y "Cuprite.bsq", y el número de iteraciones indicada. En la gráfica podemos observar como varían los tiempos de ejecución para diferentes valores tanto el número de work-

groups como el de work-items en openCL o el número de bloques como el de hilos en CUDA.

Con los datos de entradas mencionados a continuación se muestra las siguientes gráficas comparativas:

1. Lanzar random (kernel para generación de skewers aleatorios) para el algoritmo PPI implementado en openCL vs el implementado en CUDA para la imagen “sintetica.bsq”.
2. Lanzar ppi (kernel para la ejecución de PPI propiamente dicho) para el algoritmo implementado en openCL vs el implementado en CUDA para la imagen “sintetica.bsq”.
3. Lanzar random (kernel para generación de skewers aleatorios) para el algoritmo PPI implementado en openCL vs el implementado en CUDA para la imagen “cuprite.bsq”.
4. Lanzar ppi (kernel para la ejecución de PPI propiamente dicho) para el algoritmo implementado en openCL vs el implementado en CUDA para la imagen “cuprite.bsq”.

1. Lanzar random para el algoritmo PPI implementado en openCL vs el implementado en CUDA para la imagen “sintetica.bsq”.

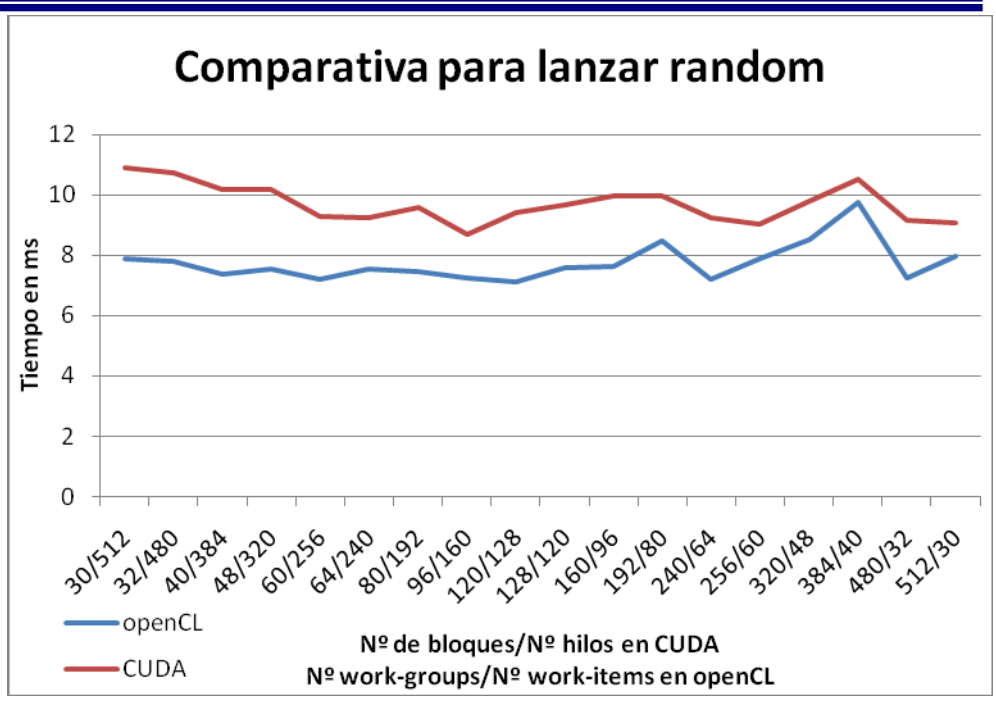

Figura 5.14: Comparativa gráfica CUDA vs openCL, 15360 iteraciones, sintética (I)

Podemos observar en la gráfica que el algoritmo de generación de números aleatorios, que genera 24002560 skewers, se ejecuta en muy pocos milisegundos debido en gran parte al aprovechamiento del paralelismo ofrecido por la GPU. También podemos ver en la gráfica, que como hemos observado en puntos anteriores, no hay demasiada diferencia en los tiempos de ejecución para ejecuciones en los distintos algoritmos de generación de números aleatorios (solo hay una diferencia de 1 ms aproximadamente en este caso). Asimismo, podemos advertir en la gráfica que el algoritmo de generación de skewers implementado en openCL, en general, consigue mejores tiempos de ejecución que el implementado en CUDA como ya hemos advertido anteriormente.

2. Lanzar ppi para el algoritmo implementado en openCL vs el implementado en CUDA para la imagen “sintetica.bsq”.

   Como vemos a continuación, el tiempo de ejecución para ambos algoritmos es similar al igual que ocurría con el algoritmo anterior. La diferencia básica está en el tiempo de ejecución que es mucho mayor (entre dos y doce segundos según el caso). Asimismo, apreciamos un

tiempo de ejecución ligeramente superior en algoritmo de CUDA al igual que en la gráfica anterior.

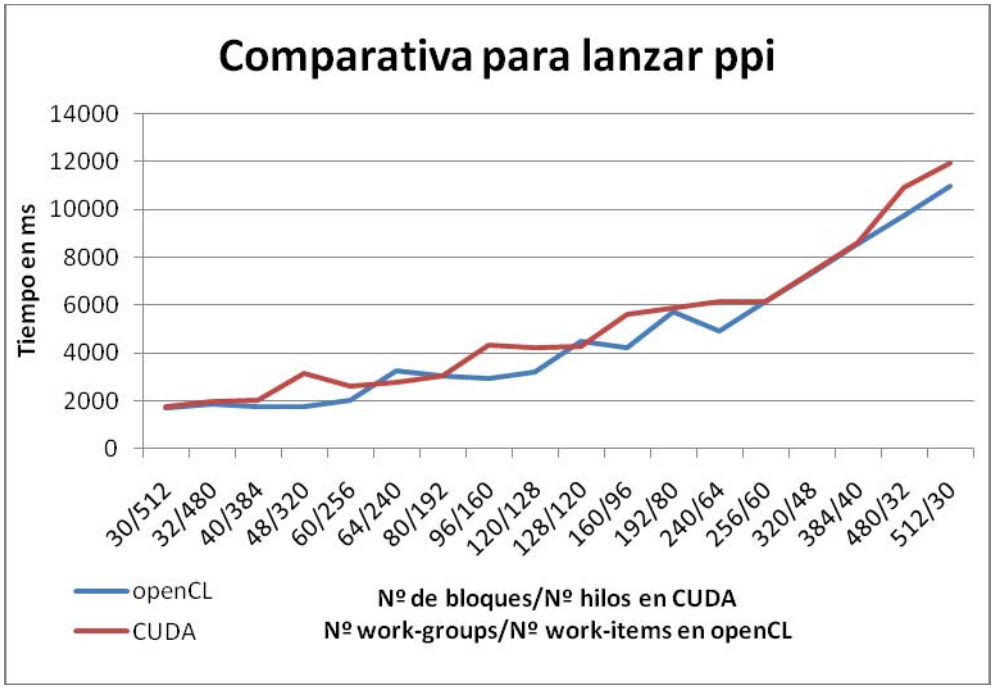

Figura 5.15: Comparativa gráfica CUDA vs openCL, 15360 iteraciones, sintética (II)

En la gráfica también podemos apreciar como al aumentar el número de bloques (número de work-groups en openCL) y disminuyendo con ello el número de hilos (número de work-items en openCL), el tiempo de ejecución aumenta, posiblemente debido a que con el aumento de bloques y la disminución del número de hilos por bloque, hay menos comunicación entre los distintos hilos por estar más divididos y por tanto aumenta el tiempo de ejecución, también como afirmamos anteriormente puede deberse a que a partir de 240 bloques si seguimos aumentando, habrá bloques que se ejecutarán de manera secuencial aumentando con ello el tiempo de ejecución.

3. Lanzar random para el algoritmo PPI implementado en openCL vs el implementado en CUDA para la imagen “cuprite.bsq”.

Igual que en el lanzar random de la imagen sintética, podemos ver en la gráfica, que no hay demasiada diferencia en los tiempos de ejecución para ejecuciones en los distintos algoritmos de generación de números aleatorios (solo hay una diferencia de 1 ms aproximadamente).

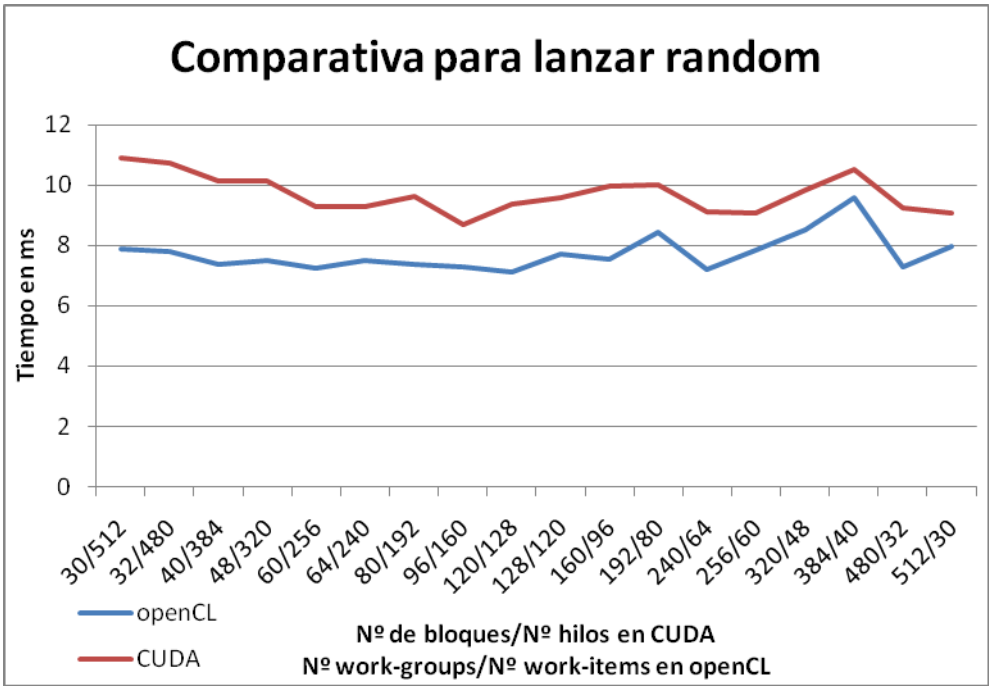

Figura 5.16: Comparativa gráfica CUDA vs openCL, 15360 iteraciones, cuprite (I)

4. Lanzar ppi para el algoritmo implementado en openCL vs el implementado en CUDA para la imagen "cuprite.bsq".

   Igual que en las gráficas de las imágenes anteriores, el tiempo de ejecución para ambos algoritmos es similar. La diferencia básica está en el tiempo de ejecución que es mucho mayor (la ejecución que menos tarda en ejecutarse, tarda unos 20 segundos).

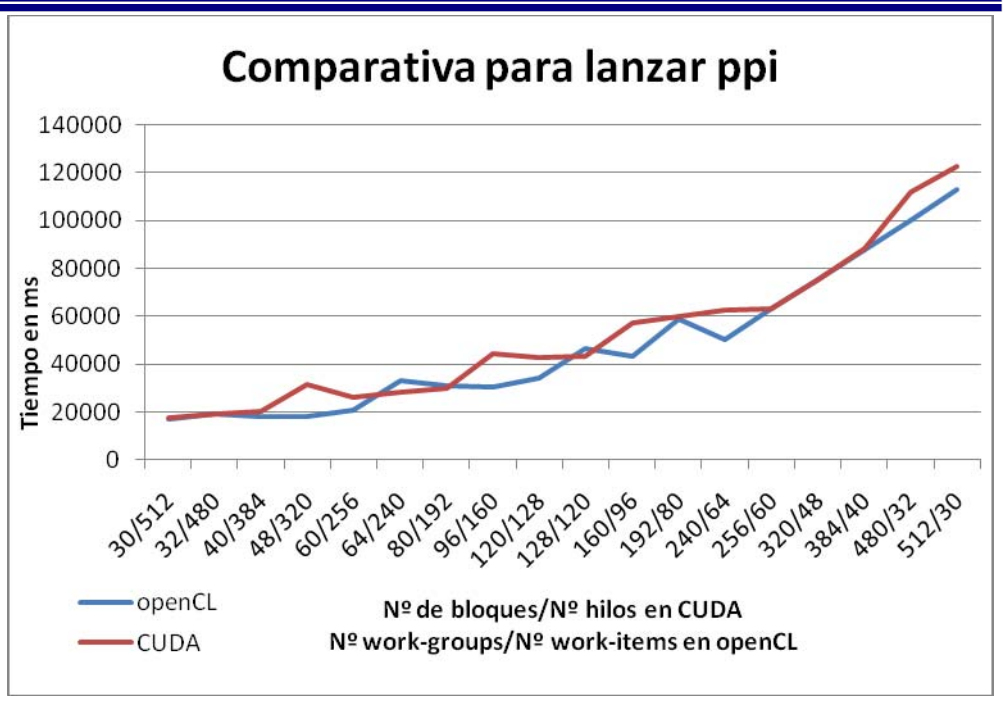

Figura 5.17: Comparativa gráfica CUDA vs openCL, 15360 iteraciones, cuprite (II)

Como venimos viendo en las gráficas anteriores, a partir de los 240 bloques los tiempos de ejecución en ambos lenguajes aumentan en mayor medida por las razones ya mencionadas, para esta imagen el aumento de tiempo es muy notable pues el tiempo de ejecución llega a tomar valores entre 110 y 120 segundos. A pesar de esto, los tiempos de ejecución no aumentan solo a partir de los 240 bloques sino que hasta esa cifra de bloques, también podemos observar que los tiempos van aumentando, aunque en menor medida, debido a que con el aumento del número de bloques se disminuye la colaboración de hilos como ya hemos mencionado con anterioridad.

6. Conclusiones y trabajos futuros

En este apartado haremos la exposición de las conclusiones obtenidas en este PFC, también haremos una tabla en la cual se expondrán una comparativa a modo de resumen de CUDA y openCL. Y para finalizar expondremos una series de trabajos futuros o posibles ampliaciones al presente PFC.

A primera vista de ambos lenguajes, podemos comenzar exponiendo la conclusión sobre la comparativa referente al modelo de programación. Respecto al modelo de programación hemos observado a lo largo de todo el proyecto fin de carrera, como ambos lenguajes son muy similares, hemos visto como los hilos de CUDA se corresponden con los work-items de openCL, y los bloques de hilos de CUDA se corresponden con los work-groups de openCL. A pesar de la gran similitud en el modo de entender ambos lenguajes, se observan pequeñas diferencias, por ejemplo, en el modelo de memoria que pueden dar lugar a error cuando se programa, como bien se ha mencionado en el apartado de los errores encontrados, debido a posibles confusiones con los nombres que se les da a cada tipo de memoria, especialmente cuando de hablamos de memoria local de CUDA y memoria local de openCL, que como ya hemos dicho anteriormente no es lo mismo. La principal ventaja de estas similitudes mencionadas es que la portabilidad de los kernels de CUDA a openCL o viceversa en general es muy fácil, el engorro principal se da en la parte de host debido a que no hay apenas correspondencia en ambos lenguajes. Por tanto, la parte de kernel es muy similar debido a la gran similitud en la arquitectura que hay entre ambos lenguajes pero la parte de host es muy diferente.

Precisamente en la parte de host se encuentran una de las principales diferencias que podemos hallar en ambos lenguajes. Mientras que en CUDA la programación del host para ejecutar un determinado kernel es muy sencilla, la programación en openCL para poner en ejecución ese mismo kernel (habiendo sido portado convenientemente a openCL, por supuesto) es muy diferente y mucho más complicado que en CUDA. De todas formas hay dos API's openCL que pueden ser

utilizadas, la API de C que es compleja (usada en este PFC) y la API de C++ un poco menos compleja que la API de C. A pesar de esto, con ambas API's la programación openCL del host es mucho más compleja que la de CUDA. Por tanto podemos concluir que portar la parte de host de un lenguaje a otro es bastante complejo por sus diferencias.

Centrándonos ahora en la comparativa de los tiempos de ejecución, hemos podido apreciar cuáles han sido los tiempos de ejecución en ambos lenguajes de programación, hemos visto como en nuestras ejecuciones con el algoritmo PPI implementado el algoritmo openCL es ligeramente más rápido en general (aunque no demasiado) que el implementado en CUDA, no obstante para ciertos casos hemos visto como es al contrario (ver tabla 6.1, se trata de uno de los datos de una gráfica anterior).

Nº Bloques/Nº hilos en CUDA Nº work-group/Nº work-items en openCL	Tiempo de ejecución de CUDA en ms.	Tiempo de ejecución de openCL en ms.
30/512	17413,69	17326,45
32/480	19371,14	19004,5
40/384	20276,3	18172,57
48/320	31463,18	18065,13
60/256	26400	20890,62
64/240	28115,65	33422,63
80/192	30226,28	31129,17
96/160	44606,04	30656,91
120/128	42667,75	34316,55
128/120	43160,3	46478,63
160/96	57128,81	43319,97

Tabla 6.1: Comparación de tiempos de ejecución en ambos lenguajes

Así, con los tiempos de ejecución obtenidos podemos afirmar que openCL para ciertas situaciones puede llegar a ser más rápido que en CUDA y en otras situaciones puede CUDA llegar a ser más rápido que openCL.

Independientemente de que openCL sea más o menos rápido, este lenguaje de programación ofrece una mayor portabilidad de código de unos sistemas a otros que con el lenguaje de programación CUDA no podemos obtener. Esto quiere decir

que el lenguaje openCL se puede ejecutar en GPU's que tienen la arquitectura de nVidia igual que CUDA, pero con la diferencia que openCL se puede ejecutar también en GPU's que tienen la arquitectura de ATI. Aunque no se haya hablado mucho en este PFC, los kernels openCL también pueden ser ejecutados en CPU con varios núcleos (siempre que la CPU lo soporte), si bien el paralelismo será mucho menor que si se ejecuta en GPU.

Otra de las conclusiones que se pueden obtener es en relación con el algoritmo PPI en sí, este algoritmo obtiene los endmembers de manera correcta, pero se hace de una manera poco directa, es decir, se hace una proyección de *todos* los píxeles de la imagen sobre cada skewer para encontrar los endmembers para ver los píxeles extremos, en lugar de ir directamente a una zona de la imagen para encontrarlos, por esta razón se han desarrollados otros algoritmos que se encargan de solucionar este inconveniente valorado [22].

A continuación mostramos una tabla comparativa de ambos lenguajes a modo de resumen:

CUDA	OpenCL
Se ejecuta en GPU's para nVidia únicamente. (sólo en modelos que soporten GPGPU)	Multiplataforma, tanto en nVidia como ATI (sólo en modelos que soporten GPGPU) como en procesadores con varios núcleos puede ser ejecutado
Compila los núcleos en un proceso que se ejecuta en tiempo de compilación	Compila los núcleos en un proceso que se ejecuta en tiempo de ejecución
Es menos detallado que OpenCL, esto es obvio para el algoritmo PPI desarrollado	Requiere mayor nivel de detalle en el host lo que complica su programación
Tiempos de ejecución muy parejos en las pruebas realizadas, aunque ligeramente por delante openCL	
Según diferentes estudios suele rendir mejor CUDA que openCL aunque en determinados casos puede ser al contrario como en el nuestro. [23-24]	

Tabla 6.2: Comparación CUDA vs OpenCL

También podemos valorar cómo la computación GPGPU consigue aprovechar el rendimiento de las GPU's que podemos tener en casa en aplicaciones que necesitan gran capacidad de programación paralela como pueden ser las operaciones matemática con matrices que pueden tener gran cantidad de datos. A esto hay que añadir que la computación GPU es una alternativa muy económica a otras alternativas de computación paralela como la computación a través de los clústeres. Mientras que las GPU's actuales de última generación tienen un precio comprendido entre 300 y 600 €, los precios de los clústeres son bastante superiores. Además de todo esto, si el procesamiento GPGPU lo queremos hacer a bordo del avión o el medio de transporte donde se instale el sensor hiperespectral, es mucho más sencillo hacer el procesamiento en una GPU por cuestiones de peso y dimensiones.

Para finalizar podemos exponer una serie de posibles trabajos futuros a este PFC:

- Uno de los posibles trabajos futuros puede ser la implementación del algoritmo PPI en un clúster de GPU permitiendo procesar imágenes de mayor tamaño.
- Puede ser interesante también la comparación de la implementación de un algoritmo paralelo en GPU con ese mismo algoritmo paralelo en un clúster o incluso en una FPGA implementada en VHDL.
- Hemos mencionado también que el algoritmo openCL puede ejecutarse en una CPU multiprocesador, podría ser interesante realizar una aplicación u algoritmo en C haciendo uso de los hilos y esa misma aplicación u algoritmo en openCL y comparar el rendimiento cuando se ejecutan en CPU, también la realizada en openCL se podría hacer unas ejecuciones en GPU para ver las diferencias con la CPU.
- Por otra lado, y con vistas a contrastar los resultados obtenidos en el estudio a otras aplicaciones, otra línea futura de trabajo podría ser probar la implementación GPU con distintos tipos de imágenes, uno de los problemas que tiene CUDA y openCL, es actualmente no dispone de funciones para realizar la carga de ficheros de imágenes directamente en espacios de

memoria de los kernels en ejecución, excepto en CUDA para las imágenes de tipo PPM y PGM, con lo cual se produce una carga de imagen quizá poco optimizada.

- Para finalizar, aunque no se ha hablado de la nueva API de Microsoft de DirectX 11, podemos comentar que es una nueva versión de Direct3D de la que podemos destacar la tecnología de computación de shaders que permite que la GPU no sea solamente usada para gráficos 3D, sino como medio para la computación GPGPU, por tanto los desarrolladores puedan beneficiarse de las ventajas de las tarjetas gráficas para procesamiento paralelo. Por tanto sería interesante hacer la comparativa de tiempos con la API de DirectX como extensión en este PFC.

7. Bibliografía

[1] Dimitris Manolakis, David Marden, and Gary A. Shaw, “Hyperspectral Image Processing for Automatic Target Detection Applications”. Lincoln laboratory journal volume 14, number 1, 2003.

[2] Green, R.O. et al., “Imaging spectroscopy and the airborne visible/infrared imaging spectrometer (AVIRIS),” Remote Sens. Environ., vol. 65, pp. 227-248, 1998.

[3] Landgrebe, D., “Hyperspectral Image Data Analysis”, IEEE Signal Processing Magazine, vol. 19, no. 1, pp. 17-28, 2002.

[4] Landgrebe, D., “Multispectral Data Analysis, A Signal Theory Perspective”, 1998.

[5] Robert Green y Betina PavriGreen. “AVIRIS In-Flight Calibration Experiment, Sensitivity Analysis, and Intraflight Stability”, en Proc. IX NASA/JPL Airborne Earth Science Workshop, Pasadena, CA, 2000.

[6] Boardman, J., Kruse, F., & Green, R. (1995). “Mapping target signatures via partial unmixing of AVIRIS data,” in Summaries of JPL Airborne Earth Science Workshop, asadena, CA.

[7] Goetz, A. F., & Kindel, B. (1999). "Comparison of Unmixing Result Derived from AVIRIS, High and Low Resolution, and HYDICE images at Cuprite, NV". Proc. IX NASA/JPL Airborne Earth Science Workshop.

[8] Zhe Fan, Feng Qiu, Arie Kaufman, Suzanne Yoakum-Stover. “GPU Cluster for High Performance Computing”. Center For Visual Computing and Department of Computer Science (Stony Brook University).

[9] Board Specification. Tesla C1060. Computing Processor Board

[10] Marc Ebner, Markus Reinhardt, and Jürgen Albert. “Evolution of Vertex and Pixel Shaders”, Universität Würzburg, Lehrstuhl für Informatik II, Germany.

[11] NVIDIA GeForce 8800 GPU Architecture Overview (November 2006 v0.9).

[12] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide, Versión 1.1 (29/11/07)

[13] NVIDIA's Next Generation. CUDA Compute Architecture: Fermi.

[14] Specification of Tesla C1060 Computing Processor Board (September 2008).

[15] The OpenCL Specification Khronos OpenCL. Working Group Version: 1.1 Editor: Aaftab Munshi.

[16] Plaza, A., Martínez, P., Pérez, R.M., Plaza, J., "A quantitative and comparative analysis of endmember extraction algorithms from hyperspectral data," IEEE Trans. On Geoscience and Remote Sensing, vol. 42, no. 3, pp. 650-663, March 2004.

[17] Makoto Matsumoto, Keio University/Max-Planck_Institut fur Mathematik, Takuji Nishimura, "Mersenne Twister. A 623-dimensionally equidistributed uniform pseudorandom number generator", Keio University.

[18] Victor Podlozhnyuk, "Parallel mersenne twister" NVIDIA, Junio 2007.

[19] Erik Lindholm, John Nickolls, Stuart Oberman, John Montrym, "nVidia Tesla: A Unified Graphics and Computing Architecture". Marzo–abril 2008

[20] nVidia, "NVIDIA CUDA Compute Unified Device Architecture. Programming Guide", versión 1.1, November 2007

[21] Pablo Lamilla Álvarez, Shinichi Yamagiwa, Francisco José Abad Cerdá, "Design and implementation of a highperformance stream-based computing platform on multigenerational GPUs.", Universidad Politécnica de Valencia, September 2010.

[22] José M. P. Nascimento, José M. B. Dias, "Vertex Component Analysis: A Fast Algorithm to Unmix Hyperspectral Data".

[23] Kamran Karimi, Neil G. Dickson, Firas Hamze, "A Performance Comparison of CUDA and OpenCL", D-Wave Systems Inc., Burnaby, British Columbia, Canada.

[24] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, Jack Dongarra, "From CUDA to OpenCL: Towards a Performance-portable Solution for

Multi-platform GPU Programming", University of Tennessee Knoxville and University of Manchester, April 2011.

[25] Jason sanders, Edward Kandrot ,“CUDA by example”, July 2010.

8. Apéndice

8.1.Lenguaje CUDA

En este apartado se hará un pequeño tutorial indicando como es el lenguaje CUDA y qué debe hacerse para programar en este lenguaje. Debemos que señalar que se trata de un tutorial básico para programar en CUDA, con lo cual no se comenta toda la API de CUDA.

Para hacer un programa en CUDA será necesario, como ya hemos comentado en alguna ocasión, tener dos archivos (host y kernel). A continuación haremos una explicación de cómo hay que completar cada uno de ellos:

- En el **kernel** pondremos las instrucciones que ejecutaremos de manera paralela en la GPU. Un kernel se programa de manera muy similar a una función de lenguaje C, pero teniendo en cuenta que lo programemos debemos hacerlo pensando que el kernel se ejecutará en todos los hilos a la vez (paralelos), y por tanto, los hilos deben programarse de manera que colaboren entre sí resolviendo el problema de manera paralela. Un ejemplo de kernel puede ser el siguiente [19]:

```
void addMatrix
    (float *a, float *b, float *c, int N)
{
    int i, j, idx;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            idx = i + j*N;
            c[idx] = a[idx] + b[idx];
        }
    }
}
void  main()
{
    . . .
    addMatrix(a, b, c, N);
}
```

(a)

```
__global__ void addMatrixG
    (float *a, float *b, float *c, int N)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = i + j*N;
    if (i < N && j < N)
1       c[idx] = a[idx] + b[idx];
}
```

```
void  main()
{
    dim3 dimBlock (blocksize, blocksize);
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
2   addMatrixG<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```

(b)

Figura 8.1: a) Código serie en C b) Código CUDA en paralelo. Ejemplo de programas que suman arrays

En la figura anterior, el rectángulo *número 1 es el kernel* paralelizado del algoritmo que suma dos arrays, y resuelve lo mismo que vemos en el código serie en C (a). Podemos ver en el rectángulo número 1 un kernel que debe situarse en el fichero de kernels. Como vemos, un kernel no es más que una función como otra cualquiera en la cual delante de la etiqueta *void* le indicamos el modificador *__global__* que nos permite indicar que dicha función es un kernel que solo puede ser puesto en ejecución por la CPU en la GPU. Los modificadores posibles en una función son:

__global__: son invocados por la CPU.

__device__: son invocados por la GPU.

Además de de esto debemos saber que los parámetros de la función en tiempo de ejecución se encuentran en la memoria global de la GPU (accesibles por todos los hilos). Si en lugar de usar la memoria global a todos los hilos, queremos usar otros tipos de memoria, dentro del kernel podemos declarar variables igual que en C poniendo delante alguno de estos modificadores:

__shared__: memoria compartida a todos los hilos de un bloque.

__constant__: para declarar constantes en GPU. Las constantes se inicializan en GPU a través de la API.

Las variables que no tienen definido ninguno de los anteriores modificadores son variables locales a cada hilo. Como vemos en la figura anterior todas las variables están declaradas como locales a cada hilo.

En el kernel de la figura podemos ver tres tipos de variables que usa CUDA, dichas variables son intrínsecas en CUDA y son las siguientes: la variable *blockIdx.x* nos devuelve el índice de bloque en el cual se está ejecutando, mientras que *blockDim.x* nos devuelve la dimensión definida de los bloques en hilos del kernel, por otro lado la variable *threadIdx.x* nos devuelve el id de la hebra actual dentro de un bloque

donde estamos ejecutando el código. Otro identificador posible es: gridDim.x (dimensiones de grid en bloques).

Todas estas variables utilizadas son útiles en cuando paralelizamos un código para hacer que unos hilos ejecuten una parte de código y otros hilos ejecuten otras partes de código, o para usar índices para el acceso a vectores y/o matrices. Siguiendo con el ejemplo anterior, vemos que estas variables han sido usadas para calcular un índice, de manera que cada hilo se encarga de una suma de un elemento de la matriz.

- En el **host** pondremos las instrucciones que ejecutaremos en serie en la CPU. Por tanto, en el host tendremos una función implementada en C que nos permitirá pasar por parámetro los datos al kernel y definir el tamaño de grid, bloque e hilo además de poner en ejecución el kernel correspondiente. Con esto, podemos ver en la figura anterior en el recuadro número 2, un ejemplo simple de cómo se puede hacer un host de manera sencilla. En éste vemos la definición del tamaño de bloque y del tamaño de grid de tres dimensiones, pero también se podría definir de dos dimensiones (teniendo únicamente bloques e hilos) de la siguiente manera:

  NombreKernel <<<n_bloques,n_hilos>>>(…parámetros del kernel…);

Ahora haremos una breve exposición de funciones básicas de la API de CUDA para usarla en el host. No existe una función de inicialización explícita para la API en tiempo de ejecución, se inicializa la primera vez que se llama una función de tiempo de ejecución [20].

Las funciones siguientes se utilizan para gestionar los dispositivos presentes en el sistema:

cudaGetDeviceCount() y cudaGetDeviceProperties()

Proporcionan una manera de enumerar estos dispositivos y recuperar sus propiedades:

```
int deviceCount;
```

```
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, device);
}
```

cudaSetDevice ()

Se utiliza para seleccionar el dispositivo asociado al host:

```
cudaSetDevice (device);
```

Un dispositivo debe ser seleccionado antes de cualquier función __global__ o se llame a cualquier otra función de la API de gestión de memoria. Si esto no se hace con una llamada explícita a cudaSetDevice (), se selecciona automáticamente el dispositivo 0 y cualquier llamada posterior explícita a cudaSetDevice () no tendrá ningún efecto.

Las funciones de gestión de la memoria se utilizan para asignar y liberar memoria del dispositivo, el acceso a la memoria asignada para cualquier variable declarada en el espacio de memoria global, y la transferencia de datos entre el host y la memoria del dispositivo.

Memoria lineal se asigna usando cudaMalloc() o cudaMallocPitch() y liberado con cudaFree().

El siguiente ejemplo asigna una matriz de punto flotante de 256 elementos en la memoria lineal:

```
float* devPtr;
cudaMalloc((void**)&devPtr, 256 * sizeof(float));
```

cudaMallocPitch()

Se recomienda para la asignación de matrices 2D, ya que se asegura de que la asignación está debidamente rellenada para cumplir con los requisitos de alineación, asegurando el mejor rendimiento cuando se accede a las direcciones de fila o la realización de copias entre las matrices 2D y otras regiones de la memoria del dispositivo (con las funciones de cudaMemcpy2D()). El campo devuelto debe ser utilizado para acceder a elementos de la matriz. El siguiente ejemplo asigna una matriz 2D de anchura x altura (width x height) amplia de

valores de punto flotante y muestra cómo recorrer los elementos de la matriz en el código de dispositivo:

```
// host code
float* devPtr;
int pitch;
cudaMallocPitch((void**)&devPtr, &pitch,
width * sizeof(float), height);
myKernel<<<100, 512>>>(devPtr, pitch);
```

```
// device code
__global__ void myKernel(float* devPtr,
                         int pitch)
{
 for (int r = 0; r < height; ++r) {
 float* row = (float*)((char*)devPtr + r * pitch);
   for (int c = 0; c < width; ++c) {
         float element = row[c];
   }
 }
}
```

Las matrices de CUDA se distribuyen utilizando cudaMallocArray () y se liberan usando cudaFreeArray (). cudaMallocArray () requiere una descripción del formato creado con cudaCreateChannelDesc ().

cudaMalloc()

cudaError_t cudaMalloc(void** devPtr, size_t count);

Asigna número de bytes de memoria lineal en el dispositivo y vuelve en devPtr * un puntero a la memoria asignada. La memoria asignada está convenientemente preparada para cualquier tipo de variable. La memoria no se borra.

cudaFree()

cudaError_t cudaFree(void* devPtr);

Libera el espacio de memoria apuntado por devPtr, que debe haber sido devuelto por una llamada previa a cudaMalloc(). De lo contrario, o si

cudaFree(devPtr) ya ha sido llamado antes, se devuelve un error. Si devPtr es 0, no se realiza.

cudaMemcpy()

```
cudaError_t cudaMemcpy(void* dst, const void* src, size_t count,
                       enum cudaMemcpyKind kind);
cudaError_t cudaMemcpyAsync(void* dst, const void* src,
                       size_t count,
                       enum cudaMemcpyKind kind,
                       cudaStream_t stream);
```

Copia count bytes del área de memoria apuntada por src a la zona de memoria apuntada dst, donde kind es uno de cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost o cudaMemcpyDeviceToDevice, y especifica la dirección de la copia. Las áreas de memoria no se pueden superponer.

cudaMemcpyAsync () es asíncrona y, opcionalmente, puede estar asociado a un stream pasando un argumento stream distinto de cero.

cudaMemset()

```
cudaError_t cudaMemset(void* devPtr, int value, size_t count);
```

Rellena los primeros count bytes del área de memoria apuntada por devPtr con valor constante value.

8.2.Lenguaje openCL

En este apartado se hará un pequeño tutorial indicando como es el lenguaje openCL y qué debe hacerse para programar en este lenguaje. Debemos que señalar que se trata de un tutorial básico para programar en openCL, con lo cual no se comenta toda la API de openCL.

Para hacer un programa en openCL será necesario, como ya hemos comentado en alguna ocasión, tener dos archivos (host y kernel). A continuación haremos una explicación de cómo hay que completar cada uno de ellos:

- En el **kernel** pondremos las instrucciones que ejecutaremos de manera paralela en la GPU. Un kernel se programa de manera muy similar a una función de lenguaje C, pero teniendo en cuenta que lo programemos debemos hacerlo pensando que el kernel se ejecutará en todos los hilos a la vez (paralelos), y por tanto, los hilos deben programarse de manera que colaboren entre sí resolviendo el problema de manera paralela. Un ejemplo de kernel puede ser el siguiente [21]:

```
__kernel void VectorAdd(__global const float* a,
                        __global const float* b,
                        __global float* c,
                        int iNumElements)
{
    //get index into global data array
    int iGID = get_global_id(0);

    //bound check (equivalent to the limit on a 'for' loop for standard/serial C code)
    if (GID >= iNumElements)
    {
        return;
    }

    //add the vector elements
    c[iGID] = a[iGID] + b[iGID];
}
```

Figura 8.2: Ejemplo de código fuente de un kernel que suma dos vectores

En la figura anterior, podemos ver *el kernel* paralelizado del algoritmo que suma dos vectores. Dicho kernel debe situarse en el fichero de kernels. Como vemos, un kernel no es más que una función como otra cualquiera en la cual delante de la etiqueta *void* le indicamos el modificador __*kenel* que nos permite indicar que dicha función es un kernel que solo puede ser puesto en ejecución por la CPU en la GPU. Con el calificador __*kernel* las siguientes reglas se aplican a las funciones que se declaran: puede ser ejecutado en el dispositivo sólo, puede ser llamado por el anfitrión, es sólo una

llamada a la función normal si una función __kernel es llamado por otro núcleo.

Además de de esto debemos saber que los parámetros de la función en tiempo de ejecución se encuentran en la memoria global de la GPU (accesibles por todos los hilos) como podemos ver en la figura 7.2 con el calificador __global. Si en lugar de usar la memoria global a todos los hilos, queremos usar otros tipos de memoria, dentro del kernel podemos declarar variables igual que en C poniendo delante alguno de estos modificadores:

__*global*: es el nombre del espacio de direcciones que se utiliza para referirse a los objetos de la memoria (buffer o de objetos de la imagen) asignados de la reserva de memoria global. Memoria compartida a todos los hilos de un bloque.

__*local*: es el nombre del espacio de direcciones se utiliza para describir las variables que deben ser asignados en la memoria local y compartida por todos los elementos de trabajo de un grupo de trabajo. Los punteros al espacio de direcciones __local se permiten como argumentos a funciones (incluidas las funciones __kernel). Las variables declaradas en el espacio de direcciones __local dentro de una función __kernel debe ocurrir en una función __kernel.

__*constant*: para declarar constantes en GPU. Estas variables de sólo lectura se pueden acceder por todos (global) los work-items del kernel durante su ejecución. Los punteros al espacio de direcciones __constant se permiten como argumentos a funciones (incluidas las funciones __kernel) y para las variables declaradas dentro de funciones.

__*private*: son variables privadas a un hilo, por tanto no se comparten ni con los hilos del mismo work-group ni con los demás hilos, es decir, sencillamente no se comparten con nadie.

Las variables que no tienen definido ninguno de los anteriores modificadores son variables privadas a cada hilo. Como vemos en la

figura anterior la variable "iGID" de la figura anterior están declaradas como privadas a cada hilo.

- En el **host** pondremos las instrucciones que ejecutaremos en serie en la CPU. Por tanto, en el host tendremos una función implementada en C que nos permitirá pasar por parámetro los datos al kernel y definir las dimensiones del NDRange, bloque e hilo además de poner en ejecución el kernel correspondiente. Para poner un kernel en ejecución, es necesario hacer los pasos que comentamos en el apartado "PPI en CUDA vs PPI en openCL", y siendo ese patrón, podemos definir el host para ejecutar el kernel de la figura 7.2. En la figura 7.3, para no complicar más la definición del host, se ha simplificado la definición de las variables omitiendo alguna de las menos importantes, también se han omitido por la misma razón la comprobación de errores en el host. En dicha figura vemos como para ejecutar el kernel openCL son necesarias muchas más funciones de la API que en CUDA con lo cual se hace un poco más complicado la implementación de esta. A continuación mostramos el código:

```
void main(int argc, char** arv){
    cl_mem a, b, c;
    int N=30;
    int selected_plataform;
    size_t global_work_size=64;
    size_t local_work_size=128;
    float h_a=(float*) malloc(N* sizeof(float));
    float h_b=(float*) malloc(N* sizeof(float));
    float h_c=(float*) malloc(N* sizeof(float));
    ...
    //inicializamos los parámetros de entrada h_a y h_c
    //Obenemos las plataformas disponibles y seleccionamos una
    clGetPlatformIDs(0, NULL, &num_platforms);
    platforms=(cl_platform_id *) malloc(sizeof(cl_platform_id)*num_platforms);
    clGetPlatformIDs(num_platforms, platforms, NULL);
    selected_platform=0;

    //Creamos un contexto para dicha plataforma
    context=clCreateContextFromType(cps, CL_DEVICE_TYPE_ALL,NULL,NULL,&err);
    //seleccion del dispositivo
    clGetContextInfo(context,CL_CONTEXT_DEVICES,0,NULL,&size_devices);
    num_devices=size_devices/sizeof(cl_device_id);
    devices= (cl_device_id*) malloc(size_devices);
    selected_device=0;
    //creamos la cola de comandos
    clCreateCommandQueue(context, devices[selected_device],
                         CL_QUEUE_PROFILING_ENABLE,NULL);
    //Creamos los buffers de datos ( una cola para cada variable)
    a=clCreateBuffer(context, CL_MEM_READ_ONLY, N *sizeof(cl_float),NULL,NULL);
    b=clCreateBuffer(context, CL_MEM_READ_ONLY, N *sizeof(cl_float),NULL,NULL);
    c=clCreateBuffer(context, CL_MEM_WRITE_ONLY, N *sizeof(cl_float),NULL,NULL);
    //copiamos el contenido de a y b (parámetros de entrada) al buffer.
    err=clEnqueueWriteBuffer(command_queue, a,CL_TRUE, 0,
                             N *sizeof(cl_float),h_a,0,NULL,NULL);
    err=clEnqueueWriteBuffer(command_queue, b,CL_TRUE, 0,
                             N *sizeof(cl_float),h_b,0,NULL,NULL);
    //Cargamos y compilamos el kernel desado y le pasamos los parámetros
    clCreateProgramWithSource(context,1,&source,sourceSize,&err);
    clBuildProgram(program, num_devices, devices, NULL, NULL, NULL);
    cl_kernel kernel=clCreateKernel(program,"NombreKernel",&err);
        //Al kernel le paso todos los n argumentos de entrada y salida
    clSetKernelArg(kernel,0,N *sizeof(cl_float),&a);
    clSetKernelArg(kernel,1,N *sizeof(cl_float),&b);
    clSetKernelArg(kernel,2,N *sizeof(cl_float),&c);
    clSetKernelArg(kernel,3, sizeof(cl_int), N);
    //Pongo la ejecución del kernel en la cola especificando el tamaño del grupo de trabajo y el tamaño de cada grupo
    clEnqueueNDRangeKernel(command_queue,kernel, 1,NULL,
                           &global_work_size, &local_work_size,
                           0,NULL,&event);
    cl_ulong start,end;
    clWaitForEvents(1, &event);
    clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
                            sizeof(cl_ulong), &start, NULL);
    clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
                            sizeof(cl_ulong), &end, NULL);
    clReleaseEvent(event);
    //Leo el resultado
    clEnqueueReadBuffer(command_queue, c,CL_TRUE, 0,
                        N * sizeof(cl_float),h_c,0,NULL,NULL);
    //El resultamos lo tendremos en la variable h_c
}
```

Figura 8.3: Host para el ejemplo de código fuente de un kernel que suma dos vectores

Ahora haremos una breve exposición de funciones básicas de la API de openCL para usarla en el host [15].

Las funciones siguientes se utilizan para gestionar los dispositivos presentes en el sistema y los objetos openCL:

clGetPlatformIDs()

```
clGetPlatformIDs (cl_uint num_entries,
                  cl_platform_id *platforms,
                  cl_uint *num_platforms)
```

La lista de las plataformas disponibles se puede obtener mediante esta función.

clCreateContextFromType()

```
clCreateContextFromType(const cl_context_properties *properties,
                        cl_device_type device_type,
                        void (CL_CALLBACK *pfn_notify)(const char *errinfo,
                        const void *private_info, size_t cb,
                        void *user_data),
                        void *user_data,
                        cl_int *errcode_ret)
```

Crea un contexto OpenCL de un tipo de dispositivo que identifica el dispositivo específico para su uso.

clGetContextInfo()

```
cl_int clGetContextInfo (cl_context contexto,
    cl_context_info param_name,
    param_value_size size_t,
    void * param_value,
    param_value_size_ret * size_t)
```

Se puede utilizar para consultar información acerca de un contexto.

clCreateCommandQueue()

```
cl_command_queue clCreateCommandQueue (cl_context context,
    cl_device_id device,
    cl_command_queue_properties properties,
    cl_int *errcode_ret)
```

Crea una cola de comandos en un dispositivo específico. La cola de comandos se puede utilizar para la cola de un conjunto de operaciones (conocidas como comandos) en orden. Tener múltiples colas de comandos permite a las aplicaciones poner a la cola de varios comandos independientes sin necesidad de sincronización. Hay que tener en cuenta que esto debería funcionar, siempre y cuando estos objetos no se comparten. El intercambio de objetos a través de múltiples colas de comandos requerirá de la aplicación para realizar la sincronización adecuada.

clCreateBuffer()

```
cl_mem clCreateBuffer (cl_context context,
                       cl_mem_flags flags,
                       size_t size,
                       void *host_ptr,
                       cl_int *errcode_ret)
```

Función que se usa para crear objetos buffer, los cuales podremos usar por ejemplo para pasar por parámetros los datos de entrada del kernel a ejecutar.

clEnqueueReadBuffer() y clEnqueueWriteBuffer()

```
cl_int clEnqueueReadBuffer (cl_command_queue command_queue,
    cl_mem buffer,
    cl_bool blocking_read,
    size_t offset,
    size_t cb,
    void *ptr,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list,
    cl_event *event)
```

```
cl_int clEnqueueWriteBuffer (cl_command_queue command_queue,
        cl_mem buffer,
        cl_bool blocking_write,
        size_t offset,
        size_t cb,
        const void *ptr,
        cl_uint num_events_in_wait_list,
        const cl_event *event_wait_list,
        cl_event *event)
```

Estas dos funciones se usa para leer y escribir en los objetos de memoria que hemos creado anteriormente con la llamada a la API de openCL *clCreateBuffer()*. Estas dos funciones son por tanto útiles para escribir el dato de entrada en el objeto que se le pasa al kernel y/o leer los datos obtenidos del kernel.

clCreateProgramWithSource()

```
cl_program clCreateProgramWithSource (cl_context context,
        cl_uint count,
        const char **strings,
        const size_t *lengths,
        cl_int *errcode_ret)
```

Crea un objeto de programa para un contexto, y carga el código fuente especificado por las cadenas de texto en la matriz de cadenas en el objeto del programa. Los dispositivos asociados con el objeto del programa son los dispositivos asociados con el contexto.

clBuildProgram()

```
cl_int clBuildProgram (cl_program programa,
        cl_uint num_devices,
        const cl_device_id * device_list,
        const char * las opciones,
        void (CL_CALLBACK pfn_notify *) (cl_program programa,
        void * user_data),
```

void * user_data)

Construye (compila y enlaza) un programa ejecutable a partir de la fuente del programa o binario para todos los dispositivos o un dispositivo específico en el contexto de OpenCL relacionados con el programa. OpenCL permite a los ejecutables del programa que se construirá usando un archivo fuente o uno binario. *clBuildProgram* debe ser llamado para el programa creado utilizando *clCreateProgramWithSource* o *clCreateProgramWithBinary* para construir el ejecutable del programa de uno o más dispositivos relacionados con el programa

cl_kernel()

clCreateKernel cl_kernel (cl_program programa,

const char * kernel_name,

cl_int errcode_ret *)

Para crear un objeto kernel de programa.

clSetKernelArg()

cl_int clSetKernelArg (kernel cl_kernel,

cl_uint arg_index,

arg_size size_t,

const char * arg_value)

Se utiliza para establecer el valor del argumento de un argumento específico de un kernel.

clEnqueueNDRangeKernel ()

cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue,

cl_kernel kernel,

cl_uint work_dim,

global_work_offset const size_t *,

const size_t * global_work_size,

const size_t * local_work_size,

cl_uint num_events_in_wait_list,

const cl_event * event_wait_list,

cl_event evento *)

Encola un comando para ejecutar el kernel en un dispositivo.

clWaitForEvents ()

cl_int clWaitForEvents (cl_uint num_events, const cl_event event_list *)

Espera en el host hasta que los comandos identificados se completan. Un comando es considerado completo si su estado de ejecución es CL_COMPLETE o un valor negativo. Los eventos especificados en la event_list como puntos de sincronización.

clGetEventProfilingInfo()

cl_int clGetEventProfilingInfo (caso cl_event,

cl_profiling_info param_name,

param_value_size size_t,

void * param_value,

param_value_size_ret * size_t)

Devuelve la información de perfiles para el comando asociado con event.

clReleaseEvent()

cl_int clReleaseEvent (cl_event evento)

Liberamos el evento creado anteriormente.

8.3.Herramientas utilizadas

En el desarrollo del presente PFC han sido necesarias diferentes herramientas que han sido útiles para todo el tipo tareas de desarrollo necesarias para el PFC:

- Notepad++: es un editor de texto WYSIWIG que ha sido usado para la codificación de gran parte del código fuente.
- SSHSecureShellClient: es un programa que sirve para conectarse de manera remota usando el protocolo ssh, y ha sido usado tanto para conectarse al clúster de GPU's del CETA-ciemat y como a la GPU Tesla que se encuentra en el laboratorio de Hypercomp. Este programa ha sido útil para la compilación y ejecución del código fuente.

- Xming: Es un programa que implementa el sistema de ventanas X de Linux y que ha sido usado conjuntamente con SSHSecureShellClient para ejecutar aplicaciones de manera gráfica de manera remota tanto en la GPU's del CETA-ciemat como a la GPU Tesla que se encuentra en el laboratorio de Hypercomp.
- Microsoft Word: Es un editor de texto con el cual ha sido realizada la presente memoria.
- Paint: Editor gráfico que ha sido usado para editar varias de las imágenes usadas en esta documentación del PFC.
- Adobe Photoshop: Igual que el programa anterior, editor gráfico que ha sido usado para editar varias de las imágenes usadas en esta documentación del PFC.
- Dev-C++: Programa usado para parte de la codificación y para copiar con formato el texto para poner el código fuente en la presente documentación del PFC.
- Google Chrome: Navegador web usado para la búsqueda de información sobre los lenguajes CUDA y openCL, tutoriales, y manuales de usuarios de dicho lenguajes…
- Microsoft PowerPoint: Es un editor de presentaciones que ha sido necesario para la creación de la presentación.
- Envi en su versión 4.5: es un programa que sirve para ver de manera visual y procesar imágenes hiperespectrales.

Además de estas herramientas software que se han ejecutado en local, han sido necesarias otras que han sido necesarias que han sido usadas de manera remota:

- Linux versión Red Hat 4.1.2-44: Es el sistema operativo que está instalado en la máquina remota en el CETA-ciemat usado de manera remota para las compilaciones y ejecuciones. Dicho sistema operativo tiene activo un servidor ssh para recibir conexiones.
- Otra herramienta ha sido el compilador de CUDA, la versión de CUDA que está instalada en dicha máquina es la 4.0.1 que soporta la compilación y ejecución del lenguaje openCL en su versión 1.0.

8.4.Problemas encontrados

En el desarrollo del PFC ha sido necesario superar los diferentes problemas que se han ido encontrando a lo largo de toda la fase de desarrollo del PFC. Algunos de los problemas se muestran a continuación acompañados de la solución que se les ha dado para solventarlos:

- Una vez comenzamos con el desarrollo del PFC, los primeros problemas con los cuales nos encontramos son: el completo desconocimiento de cómo funciona la arquitectura GPGPU, cómo es el lenguaje CUDA, cómo es el lenguaje openCL; qué es, qué hace el algoritmo PPI y para qué sirve; qué son y cómo se estructura una imagen hiperespectral,… Para solucionar todos estos problemas se ha acudido a los manuales de referencia que aparecen en la bibliografía [10-15, 20]. También se ha leído documentación sobre imágenes hiperespectrales, el algoritmo PPI, papers sobre el algoritmo PPI,… La mayoría de los documentos están en la bibliografía.
- Después de resolver el problema de desconocimiento inicial que hubo en el desarrollo de este proyecto fin de carrera, surgieron problemas con en lo relacionado con el diseño del algoritmo openCL y la codificación: uno de los primeros problemas fue que la versión de CUDA instalada en el laboratorio de hypercomp no soportaba la ejecución de openCL (las versiones actuales de CUDA permiten compilar openCL para ejecutarlo sobre las GPU's nVidia). Debido a problemas ajenos a este PFC no se pudo actualizar a una versión más actual, y por tanto, para solucionar este inconveniente se procedió a la creación de una cuenta en el CETA-ciemat ubicado en Trujillo.
- Siguiendo con el tipo de problemas anterior, el siguiente ha sido la adaptación del código fuente CUDA a código openCL, el código fuente del algoritmo PPI inicialmente se programó para CUDA pero para hacer las comparaciones es necesario que se ejecuten en ambos lenguajes, por tanto, se empezó pasando el código fuente del algoritmo de generación de skewers (generación de números aleatorios), es en este punto donde

se encontró uno de los primeros problemas importantes. Para la generación de skewers aleatorios es necesario tener un generador de números aleatorios que se ejecute en GPU, en las versiones actuales de del lenguaje CUDA no hay problema pues se implementa un generador de números aleatorios en GPU, pero en la versión 1.0 del lenguaje openCL, que ha sido con la que se ha trabajado durante el desarrollo de este PFC, no sucede lo mismo. Para solucionar este problema se ha optó por la generación de números aleatorios a través de un algoritmo de generación de números aleatorios denominado Mersenne Twister, que permite la generación de gran cantidad de números de manera rápida.

- Una vez resuelto el problema cómo generar skewers, surgió un problema con la implementación del algoritmo. El problema se debía a que openCL soporta el paso por parámetros y ciertas asignaciones de estructuras de datos con ciertas limitaciones que afectaban directamente al traducir la implementación del algoritmo Mersenne Twister a openCL, como consecuencia de esto, debe cambiarse la manera de cómo asignan estructuras de datos para que ambos sean lo más parecido posible para realizar las comparaciones con la mayor realidad posible.
- También otro de los problemas con la generación de números aleatorios ha sido que solo generaba una matriz con un número aleatorio y el resto estaba a 0's. El problema estaba al obtener el número truncando la precisión a entero pues generan números float.
- En la parte de diseño y compilación de openCL, debido a la complejidad del host (por falta de experiencia con openCL) también han surgido numerosos problemas con el ajuste de dimensiones del NDRange, por error fue inicializado a 2, y como consecuencia no se conseguía la ejecución de ningún programa porque se quedaba bloqueado como si estuviera en un bucle infinito, pero no había ninguno, por tanto fue uno de los problemas más difíciles de detectar. Al ejecutar aplicaciones, por ejemplo de 120 work-groups por 128 work-items como se había definido un espacio de 2 dimensiones, probablemente la GPU se desbordase con

la gran cantidad de hilos $((120 * 128)^2 = 235.929.600\ \text{hilos})$ y de work-items, por tanto, la solución fue simplemente cambiar la dimensión a 1.

- Otro de los problemas importantes relacionados con la codificación, fue en la codificación del código fuente del kernel al confundir las variables locales de CUDA con las variables locales de openCL, que no es lo mismo como se deduce de las explicaciones en los apartados de la arquitectura de CUDA y openCL. Por tanto, como sabemos, la memoria local en CUDA se corresponde con la privada de openCL, y la memoria local de openCL se corresponde con la compartida de CUDA.
- En codificación también hubo algún problema al obtener los índices locales y globales de cada work-item al obtenerlos los de otra dimensión (get_local_id(1) y get_global_id(1) en lugar de get_local_id(0) y get_global_id(0)).
- Otro de los problemas más importantes fue que ante los mismos skewers de entrada (sin hacerlo de manera aleatoria), en cada ejecución del algoritmo no se obtenían imágenes resultado iguales, y esto es un error. Después de mucho investigar el asunto se llegó a la conclusión que el problema se debía al uso incorrecto de la función __syncthreads() en CUDA y barrier(CLK_LOCAL_MEM_FENCE) en openCL por lo que la sincronización sincronizando mal los hilos.

Todos estos problemas han sido los de mayor importancia, pero ha habido muchos más aunque menos importantes.

8.5.Código fuente

8.5.1. Archivo ppi.h común a ambos lenguajes

```
#ifndef PPI_H
#define PPI_H
#ifndef ppi_h
#define ppi_h



#define      DCMT_SEED 4172
#define  MT_RNG_PERIOD 607


typedef struct{
    unsigned int matrix_a;
    unsigned int mask_b;
    unsigned int mask_c;
    unsigned int seed;
} mt_struct_stripped;


#define   MT_RNG_COUNT 4096
#define          MT_MM 9
#define          MT_NN 19
#define       MT_WMASK 0xFFFFFFFFU
#define       MT_UMASK 0xFFFFFFFEU
#define       MT_LMASK 0x1U
#define      MT_SHIFT0 12
#define      MT_SHIFTB 7
#define      MT_SHIFTC 15
#define      MT_SHIFT1 18



#endif
#endif
```

8.5.2. Código en CUDA

Archivo de host

```
//PPI: este fichero contiene el código del algoritmo PPI en su versión paralela GPU

//Parámetros de la ejecución;
//        1.-       Ruta de la cabecera de la imagen .hdr
//        2.-       Ruta de la imagen .bsq
//        3.-       Parámetro para pruebas, no se usa
//        4.-       Número de bloques
//        5.-       Número de hilos
// Número de iteraciones = Número de bloques * Número de hilos

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <cutil_inline.h>
#include "ppi.h"
#include <errno.h>
#include <iostream>
#define MENSAJES_ACTIVOS 0



////////////////////////////////////////////////////////////////////////////////
// Funciones comunes del host y del device
////////////////////////////////////////////////////////////////////////////////
//ceil(a / b)
```

```
extern "C" int iDivUp(int a, int b){
    return ((a % b) != 0) ? (a / b + 1) : (a / b);
}

//floor(a / b)
extern "C" int iDivDown(int a, int b){
    return a / b;
}

//Alineamiento al siguiente multiplo más cercano de b
extern "C" int iAlignUp(int a, int b){
    return ((a % b) != 0) ?  (a - a % b + b) : a;
}

////Alineamiento al anterior multiplo más cercano de b
extern "C" int iAlignDown(int a, int b){
    return a - a % b;
}


#include "ppi_kernel.cu"



////////////////////////////////////////////////////////////////////////////////
// Data configuration
////////////////////////////////////////////////////////////////////////////////
const int    PATH_N = 24000000;
const int N_PER_RNG = iAlignUp(iDivUp(PATH_N, MT_RNG_COUNT), 2);
const int    RAND_N = MT_RNG_COUNT * N_PER_RNG;




// Variables globales
int num_samples;//Numero de samples
int num_lines;//Numero de líneas
int num_bands;//Numero de bandas
int data_type;//Tipo de datos
long int VD;
long int lines_samples;// num_lines*num_samples

//Función: Read_header: Lee la cabecera del fichero .hdr y obtiene las bandas, líneas
y samples que tiene
// la imagen.
//Entrada: char filename_header[200]: Nombre del fichero de cabecera
//Salida: variables globales con las líneas, bandas y samples, num_lines, num_bands y
num_samples
void Read_header(char filename_header[200]){
          FILE *fp;
          char line[20];

          if(strstr(filename_header, ".hdr")==NULL){
                    printf("ERROR: El fichero %s no contiene el formato adecuado.Debe
tener extension hdr\n", filename_header);
                    system("PAUSE");
                    exit(1);
          }

          if ((fp=fopen(filename_header,"r"))==NULL){
                    printf("ERROR %d. No se ha podido abrir el fichero .hdr de la
imagen: %s \n",errno, filename_header);
                    system("PAUSE");
                    exit(1);
          }
          else{
                    fseek(fp,0L,SEEK_SET);
                    while(fgets(line, 20, fp)!='\0'){
                              if(strstr(line, "samples")!=NULL)
                                        num_samples = atoi(strtok(strstr(line, " = "),
" = "));

                              if(strstr(line, "lines")!=NULL)
                                        num_lines = atoi(strtok(strstr(line, " = "), "
= "));

                              if(strstr(line, "bands")!=NULL)
```

```
                                        num_bands = atoi(strtok(strstr(line, " = "), "
= "));

                                if(strstr(line, "data type")!=NULL)
                                        data_type = atoi(strtok(strstr(line, " = "), "
= "));

                        }//while
                        lines_samples=num_lines*num_samples;
                        fclose(fp);
                }//else
}


//Función: Load_image: Carga la imagen contenida en el fichero .bsq y la almacena en
image_vector.
//Entrada:          char image_filename[200]: nombre del fichero que contiene las
imagen. Formato bsq
//                          float *image_vector: Vector que contendrá la imagen.
//Salida: Devuelve un vector float con los datos de la imagen.
float *Load_Image(char image_filename[200], float *h_imagen){
        FILE *fp;
        //FILE *fo;
        short int *tipo_short_int;
        double *tipo_double;
        float *tipo_float;


        if(strstr(image_filename, ".bsq")==NULL){
                printf("ERROR: El fichero %s no contiene el formato adecuado. Debe
tener extension bsq\n", image_filename);
                exit(1);
        }
        if ((fp=fopen(image_filename,"rb"))==NULL){
                printf("ERROR %d. No se ha podido abrir el fichero .bsq que
contiene la imagen: %s \n", errno, image_filename);
                exit(1);
        }
        else{
                fseek(fp,0L,SEEK_SET);
                switch(data_type){
                        case 5:{
                                        tipo_double = (double *) malloc (num_lines *
num_samples * num_bands * sizeof(double));

        fread(tipo_double,1,(sizeof(double)*lines_samples*num_bands),fp);
              //Pasamos los datos de la imagen a float
                                        for(int i=0; i<num_lines * num_samples *
num_bands; i++){
                                                h_imagen[i]=(float)tipo_double[i];
                                        }
                                        free(tipo_double);
                                        break;
                        }

                        case 2:{
                                        tipo_short_int = (short int *) malloc
(num_lines * num_samples * num_bands * sizeof(short int));
                                        fread(tipo_short_int,1,(sizeof(short
int)*lines_samples*num_bands),fp);
                //Pasamos los datos de la imagen a float
                                        for(int i=0; i<num_lines * num_samples *
num_bands; i++){
                                                h_imagen[i]=(float)tipo_short_int[i];
                                        }
                                        free(tipo_short_int);
                                        break;
                        }

                        case 4:{
                                        tipo_float = (float *) malloc (num_lines *
num_samples * num_bands * sizeof(float));

        fread(tipo_float,1,(sizeof(float)*lines_samples*num_bands),fp);
              //Pasamos los datos de la imagen a float
                                        for(int i=0; i<num_lines * num_samples *
num_bands; i++){
```

```
                                        h_imagen[i]=tipo_float[i];
                                }
                                free(tipo_float);
                                break;
                        }
                }
                fclose(fp);
        }
        return h_imagen;
}

//Función: EscribirResultado: Guarda la imagen de pureza contenida en imagen en el
fichero especificado por resultado_filename.
//Entrada:          char resultado_filename[200]: nombre del fichero donde se
almacenará la imagen. Formato bsq
//                  int *imagen: Vector que contiene la imagen de pureza.
//Salida: No devuelve nada.
void EscribirResultado( int *imagen, char resultado_filename[200]){
        FILE *fp;
        if ((fp=fopen(resultado_filename,"wb"))==NULL){
                printf("ERROR %d. No se ha podido abrir el fichero resultados: %s
\n",errno, resultado_filename);
                system("PAUSE");
                exit(1);
        }
        else{

                fseek(fp,0L,SEEK_SET);

                fwrite(imagen,1,(num_lines * num_samples * sizeof(int)),fp);

        }
        fclose(fp);
}


////////////////////////////////////////////////////////////////////////////////
// Main program
////////////////////////////////////////////////////////////////////////////////
int main(int argc, char **argv){
        float *d_Rand;
    float*h_RandGPU;
    double gpuTime;
    unsigned int hTimer;

        const int N_BLOQUES = atoi(argv[3]);
        const int N_HILOS = atoi(argv[4]);

    // Lectura de la cabecera
    Read_header(argv[1]);
    if(MENSAJES_ACTIVOS==1){
                printf(" num_lines = %d\n", num_lines);
                printf(" num_samples = %d\n", num_samples);
                printf(" num_bands = %d\n", num_bands);
        }

    // Punteros para la memoria del host
    float *h_imagen;
    int *h_res_total;
    int *h_res_parcial;

    // Punteros para la memoria del device
    float *d_imagen;
    int *d_res_parcial;

    // Asignación de memoria para host y device
    h_imagen = (float*) malloc(num_lines * num_samples * num_bands * sizeof(float));
    h_res_total = (int*) malloc(num_lines * num_samples * sizeof(int));
    h_res_parcial = (int*) malloc(N_HILOS*N_BLOQUES*2*sizeof(int));

    cudaMalloc((void**)&d_imagen, (num_lines * num_samples * num_bands *
sizeof(float)));
    cudaMalloc((void**)&d_res_parcial, (N_HILOS*N_BLOQUES*2*sizeof(int)));

    //Leemos la imagen
    Load_Image(argv[2], h_imagen);
```

```
    for(int i=0; i<num_lines*num_samples; i++){
            h_res_total[i]=0;
    }
    for (int j=0; j<N_HILOS*N_BLOQUES*2; j++){
            h_res_parcial[j]=-1;
    }

    //Copia host -> device
    cudaMemcpy(d_imagen, h_imagen, (num_lines * num_samples * num_bands *
sizeof(float)), cudaMemcpyHostToDevice  );
    cudaMemcpy(d_res_parcial, h_res_parcial, (N_HILOS*N_BLOQUES*2* sizeof(int)),
cudaMemcpyHostToDevice  );

    cutilCheckError( cutCreateTimer(&hTimer) );

        if(MENSAJES_ACTIVOS==1)
                printf("Initializing data for %i samples...\n", PATH_N);
        h_RandGPU  = (float *)malloc(RAND_N * sizeof(float));
        cutilSafeCall( cudaMalloc((void **)&d_Rand, RAND_N * sizeof(float)) );


        srand(time(NULL));
        int r=rand();

        //Generación de los números aleatorios
        if(MENSAJES_ACTIVOS==1)
                printf("Generating random numbers on GPU...\n");
        cutilSafeCall( cudaThreadSynchronize() );
        cutilCheckError( cutResetTimer(hTimer) );
        cutilCheckError( cutStartTimer(hTimer) );
        RandomGPU<<<N_BLOQUES,N_HILOS>>>(d_Rand, N_PER_RNG, r);
        cutilCheckMsg("RandomGPU() execution failed\n");
        cudaThreadSynchronize();
        cutilCheckError( cutStopTimer(hTimer) );
        gpuTime = cutGetTimerValue(hTimer);
        if(MENSAJES_ACTIVOS==1)
                printf("Generated samples : %i \n", RAND_N);



        //Ejecución kernel PPI
        PPI <<<N_BLOQUES,N_HILOS>>>(d_imagen, d_Rand, d_res_parcial, num_lines,
num_samples, num_bands);
        cudaThreadSynchronize();

    //Paramos el cronometro
        cutStopTimer(hTimer);

        float elapsedTimeRnd=cutGetTimerValue(hTimer);
        printf("Procesos:; Hilos:; algoritmo; tiempo(ms.)\n");
        printf("%d; %d; lanzarRandom; %.2f\n", N_BLOQUES, N_HILOS, gpuTime);
        printf(";; lanzarPPI; %.2f\n", elapsedTimeRnd);

        ////////////////////////////////////////////////////////////////////////
////////////////////////
        //Copiamos los resultados de la GPU a la CPU
        cudaMemcpy(h_res_parcial, d_res_parcial, (N_HILOS*N_BLOQUES*2*sizeof(int)),
cudaMemcpyDeviceToHost);
        //Agrupamos los resultados
        for(int i=0; i<N_HILOS*N_BLOQUES*2; i++){
                h_res_total[h_res_parcial[i]]=h_res_total[h_res_parcial[i]]+1;
        }

        //Escribimos la imagen de pureza
        EscribirResultado(h_res_total, argv[5]);

        ////////////////////////////////////////////////////////////////////////
////////////////////////

        //Liberamos la memoria
        if(MENSAJES_ACTIVOS==1)
        printf("Shutting down...\n");
        cudaFree(d_Rand);
        cudaFree(d_imagen);
        cudaFree(d_res_parcial);
```

```
        free(h_RandGPU);
        free(h_imagen);
        free(h_res_total);
        free(h_res_parcial);

    cutilCheckError( cutDeleteTimer( hTimer) );
    cudaThreadExit();
}
```

Archivo de kernel

```
/*
 * Copyright 1993-2009 NVIDIA Corporation.  All rights reserved.
 *
 * NVIDIA Corporation and its licensors retain all intellectual property and
 * proprietary rights in and to this software and related documentation and
 * any modifications thereto.  Any use, reproduction, disclosure, or distribution
 * of this software and related documentation without an express license
 * agreement from NVIDIA Corporation is strictly prohibited.
 *
 */

#include "ppi.h"
#define MIN_INT ((float) (1 << (sizeof(float) * 8 - 1)))//Entero mínimo
#define MAX_INT (-(MIN_INT + 1))//Entero máximo



__device__ static mt_struct_stripped ds_MT[MT_RNG_COUNT];
static mt_struct_stripped h_MT[MT_RNG_COUNT];



////////////////////////////////////////////////////////////////////////////////
// Write MT_RNG_COUNT vertical lanes of NPerRng random numbers to *d_Random.
// For coalesced global writes MT_RNG_COUNT should be a multiple of warp size.
// Initial states for each generator are the same, since the states are
// initialized from the global seed. In order to improve distribution properties
// on small NPerRng supply dedicated (local) seed to each twister.
// The local seeds, in their turn, can be extracted from global seed
// by means of any simple random number generator, like LCG.
////////////////////////////////////////////////////////////////////////////////
__global__ void RandomGPU(
    float *d_Random,
    int NPerRng, int seed
){
    const int      tid = blockDim.x * blockIdx.x + threadIdx.x;
    const int THREAD_N = blockDim.x * gridDim.x;

    int iState, iState1, iStateM, iOut;
    unsigned int mti, mti1, mtiM, x;
    unsigned int mt[MT_NN];

    for(int iRng = tid; iRng < MT_RNG_COUNT; iRng += THREAD_N){
        //Load bit-vector Mersenne Twister parameters
        mt_struct_stripped config = ds_MT[iRng];

        //Initialize current state
        //mt[0] = config.seed;
          mt[0] =seed*tid;
        for(iState = 1; iState < MT_NN; iState++)
            mt[iState] = (1812433253U * (mt[iState - 1] ^ (mt[iState - 1] >> 30)) +
iState) & MT_WMASK;

        iState = 0;
        mti1 = mt[0];
        for(iOut = 0; iOut < NPerRng; iOut++){
            //iState1 = (iState +     1) % MT_NN
            //iStateM = (iState + MT_MM) % MT_NN
            iState1 = iState + 1;
            iStateM = iState + MT_MM;
```

```
            if(iState1 >= MT_NN) iState1 -= MT_NN;
            if(iStateM >= MT_NN) iStateM -= MT_NN;
            mti  = mti1;
            mti1 = mt[iState1];
            mtiM = mt[iStateM];

            x    = (mti & MT_UMASK) | (mti1 & MT_LMASK);
            x    =  mtiM ^ (x >> 1) ^ ((x & 1) ? config.matrix_a : 0);
            mt[iState] = x;
            iState = iState1;

            //Tempering transformation
            x ^= (x >> MT_SHIFT0);
            x ^= (x << MT_SHIFTB) & config.mask_b;
            x ^= (x << MT_SHIFTC) & config.mask_c;
            x ^= (x >> MT_SHIFT1);

            //Convert to (0, 1] float and write to global memory
            d_Random[iRng + iOut * MT_RNG_COUNT] = -0.5+(((float)x + 1.0f) /
4294967296.0f);
        }
    }
}


//Numero de pixeles que se almacenarán en memoria compartida.
# define N_Pixels 9
//TamaÃ±o del vector que se almacenará en memoria compartida.
# define Tamanio_Vector (224*N_Pixels)
//Funcion: kernel PPI. Es el algoritmo PPI propiamente dicho.
//Entradas: float *d_imagen: estructura que almacena los datos de la imagen.
//                  float *d_random: estructura que almacena los vectores aleatorios.
//                  float *d_res_parcial:estructura que almacenara los resultados tras
la ejecucion del algoritmo.
//                  int num_lines: número de líneas de la imagen.
//                  int num_samples: numero de columnas de la imagen.
//                  int num_bands: numero de bandas de la imagen.
//Salidas: la funcion no devuelve nada.
__global__ void PPI(float * d_imagen, float *d_random, int *d_res_parcial, int
num_lines, int num_samples, int num_bands)
{
          int idx =  blockDim.x * blockIdx.x+threadIdx.x;

          float pemax; // producto escalar maximo
          float pemin; // producto escalar minimo
          float pe; //producto escalar
          int v,d;
          int imax=0;
          int imin=0;
          pemax=MIN_INT;
          pemin=MAX_INT;

          __shared__ float s_pixels[Tamanio_Vector];
          float l_rand[224];
          //Copiamos un pixel de la memoria global a los registros
          for (int k=0; k<num_bands; k++){
                    l_rand[k]=d_random[idx*num_bands+k];
          }
                    for(int it=0; it<num_lines*num_samples/N_Pixels; it++){
                              //Copiamos N_Pixels pixeles a la memoria compartida
                              if(threadIdx.x<N_Pixels){
                                        for(int j=0; j<num_bands; j++){

          s_pixels[threadIdx.x+N_Pixels*j]=d_imagen[(it*N_Pixels+threadIdx.x)+(num_lin
es*num_samples*j)];
                                        }
                                   //}
                              }
                              __syncthreads();

          ////////////////////////////////////////////////////////////////////////
                              //Recorremos el vector de pixeles

                              for (v=0; v <N_Pixels; v++) // El vector tiene
Tamanio_Vector elementos
                              {
                                        // calculamos producto escalar
```

```
                                        pe = 0;
                                        for (d=0; d < num_bands; d++){
                                                pe = pe +
l_rand[d]*s_pixels[v+N_Pixels*d];
                                        }
                                        // detectamos los extremos
                                        if (pe > pemax) {
                                                imax=it*N_Pixels+v;
                                                pemax=pe;
                                        }
                                        if (pe < pemin) {
                                                imin=it*N_Pixels+v;
                                                pemin=pe;
                                        }
                                }


                        }
//         Actualizamos d_res_parcial
          d_res_parcial[idx*2]=imax;
          d_res_parcial[idx*2+1]=imin;


}
```

8.5.3. Código en openCL

Archivo de host

```
//PPI: este fichero contiene el código del algoritmo PPI en su versión paralela GPU
//Parámetros de la ejecuciÃ³n;
//        1.-       Ruta de la cabecera de la imagen .hdr
//        2.-       Ruta de la imagen .bsq
//        3.-       Parámetro para pruebas, no se usa
//        4.-       Número de bloques
//        5.-       Número de hilos
// Número de iteraciones = Número de bloques * Número de hilos

#ifdef __APPLE__
          #include <OpenCL/opencl.h>
#else
          #include <CL/cl.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#define NN 1024
#include <string.h>
#include <fstream>
#include <iostream>
#include <time.h>
#define MENSAJES_ACTIVOS 0

#include <time.h>
#include "ppi.h"
#include <errno.h>
#include <string.h>

////////////////////////////////////////////////////////////////////////////////
// Funciones comunes del host y del device
////////////////////////////////////////////////////////////////////////////////
//ceil(a / b)
extern "C" int iDivUp(int a, int b){
    return ((a % b) != 0) ? (a / b + 1) : (a / b);
}

//floor(a / b)
extern "C" int iDivDown(int a, int b){
    return a / b;
}
```

```
//Alineamiento al siguiente múltiplo más cercano de b
extern "C" int iAlignUp(int a, int b){
    return ((a % b) != 0) ?  (a - a % b + b) : a;
}

////Alineamiento al anterior múltiplo más cercano de b
extern "C" int iAlignDown(int a, int b){
    return a - a % b;
}


////////////////////////////////////////////////////////////////////////////////
// Data configuration
////////////////////////////////////////////////////////////////////////////////
const int    PATH_N = 24000000;
const int N_PER_RNG = iAlignUp(iDivUp(PATH_N, MT_RNG_COUNT), 2);
const int    RAND_N = MT_RNG_COUNT * N_PER_RNG;


// Variables globales
int num_samples;//Numero de samples
int num_lines;//Numero de líneas
int num_bands;//Numero de bandas
int data_type;//Tipo de datos
long int VD;
long int lines_samples;// num_lines*num_samples

//Funcion: Read_header: Lee la cabecera del fichero .hdr y obtiene las bandas,
líneas y samples que tiene
// la imagen.
//Entrada: char filename_header[200]: Nombre del fichero de cabecera
//Salida: variables globales con las líneas, bandas y samples, num_lines, num_bands
y num_samples
void Read_header(char filename_header[200]){
          FILE *fp;
          char line[20];
          char nameWithExtension[204];

          strcpy(nameWithExtension, filename_header);
          strcat(nameWithExtension, ".hdr");
          if ((fp=fopen(nameWithExtension,"r"))==NULL){
                    if (MENSAJES_ACTIVOS==1)
                              printf("ERROR %d. No se ha podido abrir el fichero .hdr
de la imagen: %s \n",errno, nameWithExtension);
                    //system("PAUSE");
                    exit(1);
          }
          else{
                    fseek(fp,0L,SEEK_SET);
                    while(fgets(line, 20, fp)!='\0'){
                              if(strstr(line, "samples")!=NULL)
                                        num_samples = atoi(strtok(strstr(line, " = "),
" = "));

                              if(strstr(line, "lines")!=NULL)
                                        num_lines = atoi(strtok(strstr(line, " = "), "
= "));

                              if(strstr(line, "bands")!=NULL)
                                        num_bands = atoi(strtok(strstr(line, " = "), "
= "));

                              if(strstr(line, "data type")!=NULL)
                                        data_type = atoi(strtok(strstr(line, " = "), "
= "));

                    }//while
                    lines_samples=num_lines*num_samples;
                    fclose(fp);
          }//else
}


//Funcion: Load_image: Carga la imagen contenida en el fichero .bsq y la almacena en
image_vector.
//Entrada:         char image_filename[200]: nombre del fichero que contiene las
imagen. Formato bsq
```

```
//                          float *image_vector: Vector que contendra la imagen.
//Salida: Devuelve un vector float con los datos de la imagen.
float *Load_Image(char image_filename[200], float *h_imagen){
        FILE *fp;
        //FILE *fo;
        short int *tipo_short_int;
        double *tipo_double;
        float *tipo_float;
        char nameWithExtension[204];

        strcpy(nameWithExtension, image_filename);
        strcat(nameWithExtension, ".bsq");
        /*if(strstr(image_filename, ".bsq")==NULL){
                printf("ERROR: El fichero %s no contiene el formato adecuado.
Debe tener extension bsq\n", image_filename);
                exit(1);
        }*/
        if ((fp=fopen(nameWithExtension,"rb"))==NULL){
                if (MENSAJES_ACTIVOS==1)
                        printf("ERROR %d. No se ha podido abrir el fichero .bsq
que contiene la imagen: %s \n", errno, nameWithExtension);
                exit(1);
        }
        else{
                fseek(fp,0L,SEEK_SET);
                switch(data_type){
                        case 5:{
                                tipo_double = (double *) malloc (num_lines *
num_samples * num_bands * sizeof(double));

        fread(tipo_double,1,(sizeof(double)*lines_samples*num_bands),fp);
              //Pasamos los datos de la imagen a float
                                for(int i=0; i<num_lines * num_samples *
num_bands; i++){
                                        h_imagen[i]=(float)tipo_double[i];
                                }
                                free(tipo_double);
                                break;
                        }

                        case 2:{
                                tipo_short_int = (short int *) malloc
(num_lines * num_samples * num_bands * sizeof(short int));
                                fread(tipo_short_int,1,(sizeof(short
int)*lines_samples*num_bands),fp);
                //Pasamos los datos de la imagen a float
                                for(int i=0; i<num_lines * num_samples *
num_bands; i++){
                                        h_imagen[i]=(float)tipo_short_int[i];
                                }
                                free(tipo_short_int);
                                break;
                        }

                        case 4:{
                                tipo_float = (float *) malloc (num_lines *
num_samples * num_bands * sizeof(float));

        fread(tipo_float,1,(sizeof(float)*lines_samples*num_bands),fp);
              //Pasamos los datos de la imagen a float
                                for(int i=0; i<num_lines * num_samples *
num_bands; i++){
                                        h_imagen[i]=tipo_float[i];
                                }
                                free(tipo_float);
                                break;
                        }
                }
                fclose(fp);
        }
        return h_imagen;
}

//Funcion: EscribirResultado: Guarda la imagen de pureza contenida en imagen en el
fichero especificado por resultado_filename.
//Entrada:          char resultado_filename[200]: nombre del fichero donde se
almacenará la imagen. Formato bsq
```

```
//                int *imagen: Vector que contiene la imagen de pureza.
//Salida: No devuelve nada.
void EscribirResultado( int *imagen, char resultado_filename[200]){
          FILE *fp;
          char nameWithExtension[204];

          strcpy(nameWithExtension, resultado_filename);
          strcat(nameWithExtension, ".res");
          if ((fp=fopen(nameWithExtension,"wb"))==NULL){
                    if(MENSAJES_ACTIVOS==1)
                              printf("ERROR %d. No se ha podido abrir el fichero
resultados: %s \n",errno, nameWithExtension);
                    system("PAUSE");
                    exit(1);
          }
          else{

                    fseek(fp,0L,SEEK_SET);

                    fwrite(imagen,1,(num_lines * num_samples * sizeof(int)),fp);

          }
          fclose(fp);
}

inline void checkErr(cl_int err, const char * name)
{
          if (err != CL_SUCCESS) {
                    std::cerr << "ERROR: " << name
                    << " (" << err << ")" << std::endl;
                    exit(EXIT_FAILURE);
          }
}


void showAvailablePlatforms(cl_uint num_platforms, cl_platform_id *platforms) {
          int i;
          size_t param_value_size;
          char *param_value;
          cl_int err;

          for(i=0;i<num_platforms;i++) {

                    std::cout << "================" << std::endl;
                    std::cout << "Plataforma " << i << std::endl;

                    err=clGetPlatformInfo(platforms[i],CL_PLATFORM_NAME,0,
NULL,&param_value_size);
                    checkErr(err,"clGetPlatformInfo");
                    param_value=(char *) malloc(param_value_size);

          err=clGetPlatformInfo(platforms[i],CL_PLATFORM_NAME,param_value_size,
param_value,NULL);
                    checkErr(err,"clGetPlatformInfo");
                    std::cout << "Nombre : " << param_value << std::endl;

                    err=clGetPlatformInfo(platforms[i],CL_PLATFORM_VENDOR,
0,NULL,&param_value_size);
                    checkErr(err,"clGetPlatformInfo");
                    param_value=(char *) malloc(param_value_size);

          err=clGetPlatformInfo(platforms[i],CL_PLATFORM_VENDOR,param_value_size,
param_value,NULL);
                    checkErr(err,"clGetPlatformInfo");
                    std::cout << "Fabricante : " << param_value << std::endl;

                    err=clGetPlatformInfo(platforms[i],CL_PLATFORM_VERSION,0,
NULL,&param_value_size);
                    checkErr(err,"clGetPlatformInfo");
                    param_value=(char *) malloc(param_value_size);

          err=clGetPlatformInfo(platforms[i],CL_PLATFORM_VERSION,param_value_size,
param_value,NULL);
                    checkErr(err,"clGetPlatformInfo");
                    std::cout << "VersiÃ³n : " << param_value << std::endl;

                    std::cout << "================" << std::endl;
```

```
		}
}

void showAvailableDevices(cl_uint num_devices, cl_device_id *devices) {
		int i;
		size_t param_value_size;
		char *param_value;
		cl_int err;

		for(i=0;i<num_devices;i++) {

			std::cout << "================" << std::endl;
			std::cout << "Dispositivo " << i << std::endl;

			err=clGetDeviceInfo(devices[i],CL_DEVICE_NAME,
0,NULL,&param_value_size);
			checkErr(err,"clGetDeviceInfo");
			param_value=(char *) malloc(param_value_size);
			err=clGetDeviceInfo(devices[i],CL_DEVICE_NAME,param_value_size,
param_value,NULL);
			checkErr(err,"clGetDeviceInfo");
			std::cout << "Nombre : " << param_value << std::endl;

			std::cout << "================" << std::endl;
		}
}


double lanzarRandom(cl_float *h_random,char *argv[]){
		const int N_WORK_GROUP = atoi(argv[2]);
		const int N_WORK_ITEM_WORK_GROUP = atoi(argv[3]);

		// Lectura de la cabecera
	Read_header(argv[1]);
		if(MENSAJES_ACTIVOS==1){
			printf(" num_lines = %d\n", num_lines);
			printf(" num_samples = %d\n", num_samples);
			printf(" num_bands = %d\n", num_bands);
		}

		// Punteros para la memoria del device
		cl_mem d_random;

		cl_int err;
		cl_context context;
		cl_device_id *devices;
		size_t size_devices;
		cl_int num_devices;
		cl_platform_id *platforms;
		cl_uint num_platforms;
		cl_context_properties properties;
		size_t N=NN;
		size_t global_work_size;
		size_t local_work_size;
		unsigned int i,j,k,selected_platform,selected_device;
		cl_float x;

		global_work_size=N_WORK_GROUP*N_WORK_ITEM_WORK_GROUP;
		local_work_size=N_WORK_ITEM_WORK_GROUP;

		//Obtenemos las plataformas disponibles y seleccionamos una
		err= clGetPlatformIDs(0, NULL, &num_platforms);
		checkErr(err,"clGetPlatformIDs");
		if(MENSAJES_ACTIVOS==1)
			std::cout << "Detectada(s) " << num_platforms << " plataforma(s)"
<< std::endl;
		platforms=(cl_platform_id *) malloc(sizeof(cl_platform_id)*num_platforms);
		err= clGetPlatformIDs(num_platforms, platforms, NULL);
		checkErr(err,"clGetPlatformIDs");
		selected_platform=0;
		if(MENSAJES_ACTIVOS==1){
			showAvailablePlatforms(num_platforms,platforms);
			std::cout << "Seleccionamos la plataforma " << selected_platform
<< std::endl;
			std::cout << "Creamos un contexto para los dispotivos presentes
en la plataforma " << std::endl;
```

```
		}

		//Creamos un contexto para dicha plataforma ****************
		cl_context_properties cps[3] = {CL_CONTEXT_PLATFORM,
(cl_context_properties)platforms[selected_platform], 0};
		context=clCreateContextFromType(cps, CL_DEVICE_TYPE_ALL,NULL,NULL,&err);
		checkErr(err,"clCreateContextFromType");

		err = clGetContextInfo(context,CL_CONTEXT_DEVICES,0,NULL,&size_devices);
		checkErr(err,"clGetContextInfo");

		num_devices=size_devices/sizeof(cl_device_id);
		if(MENSAJES_ACTIVOS==1)
			std::cout << "Detectado(s) " << num_devices << " dispositivo(s) "
<< " en la plataforma " << selected_platform << std::endl;
		devices= (cl_device_id*) malloc(size_devices);
		err =
clGetContextInfo(context,CL_CONTEXT_DEVICES,size_devices,devices,NULL);
		checkErr(err,"clGetContextInfo");

		if(MENSAJES_ACTIVOS==1)
			showAvailableDevices(num_devices,devices);
		selected_device=0;
		cl_command_queue command_queue=clCreateCommandQueue(context,
devices[selected_device], CL_QUEUE_PROFILING_ENABLE,NULL);
		checkErr(err,"clCreateCommandQueue");
		if(MENSAJES_ACTIVOS==1)
			std::cout << "Creamos una cola de comandos para el dispositivo "
<< selected_device << std::endl;
		//*************************************************************

		//Creamos una cola para cada variable.***********
		// Asignación de memoria para device
		d_random=clCreateBuffer(context, CL_MEM_WRITE_ONLY, RAND_N
*sizeof(cl_float),NULL,NULL);
		checkErr(err,"clCreateBuffer");
		//****************************************

		//Leemos el código del kernel a ejecutar de un fichero
		std::ifstream file("./ppi.cl");
		checkErr(file.is_open() ? CL_SUCCESS : -1, "ifstream() no puede acceder al
fichero");
		std::string sourceString( std::istreambuf_iterator<char>(file),
(std::istreambuf_iterator<char>()));

		//Convertimos el string leído a char *
		const char *source = sourceString.c_str();
		size_t sourceSize[]={ strlen (source) };
		cl_program program =
clCreateProgramWithSource(context,1,&source,sourceSize,&err);
		checkErr(err,"clCreateProgramWithSource");


		// Creo un ejecutable para todos los dispositivos disponibles
		err = clBuildProgram(program, num_devices, devices, NULL, NULL, NULL);
		if (err != CL_SUCCESS) {
			std::cerr << "ERROR: Compilacion del Kernel" << std::endl;
			char* build_log;
			size_t log_size;
			clGetProgramBuildInfo(program, devices[0], CL_PROGRAM_BUILD_LOG,
0, NULL, &log_size);
			build_log = new char[log_size+1];
			clGetProgramBuildInfo(program, devices[0], CL_PROGRAM_BUILD_LOG,
log_size, build_log, NULL);
			build_log[log_size] = '\0';
			if(MENSAJES_ACTIVOS==1)
				std::cout << build_log << std::endl;
			delete[] build_log;
			exit(EXIT_FAILURE);
		}

		// Creo un kernel usando el programa que acabo de construir
		cl_kernel kernel=clCreateKernel(program,"RandomGPU",&err);
		checkErr(err,"clCreateKernel");

		//********************** PARA EL PASO DE ARGUMENTOS
**********************************
```

```
		//Al kernel le paso como argumento los 2 vectores de entrada, el vector
resultado y el tamaño del vector
		if(MENSAJES_ACTIVOS==1)
			printf("Initializing data for %i samples...\n", PATH_N);
		srand(time(NULL));
		uint r=rand();
		err=clSetKernelArg(kernel,0,sizeof(cl_mem),&d_random);
		checkErr(err,"clSetKernelArg");

		err=clSetKernelArg(kernel,1,sizeof(cl_int),&N_PER_RNG);
		checkErr(err,"clSetKernelArg");

		err=clSetKernelArg(kernel,2,sizeof(cl_uint),&r);
		checkErr(err,"clSetKernelArg");

		cl_event event;
		if(MENSAJES_ACTIVOS==1)
			printf("Poniendo en cola la ejecucion del kernel RandomGPU\n");
		//Pongo la ejecuciÃ³n del kernel en la cola especificando el tamaño del
grupo de trabajo y el tamaño de cada grupo
		err=clEnqueueNDRangeKernel(command_queue,kernel, 1,NULL, &global_work_size,
&local_work_size,0,NULL,&event);
		checkErr(err,"clEnqueueNDRangeKernel");

		cl_ulong start,end;
		clWaitForEvents(1, &event);
		clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
sizeof(cl_ulong), &start, NULL);
		clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(cl_ulong),
&end, NULL);
		clReleaseEvent(event);

		double elapsedTime = (double)1.0e-6 * (end - start);

		//Leo el resultado
		err=clEnqueueReadBuffer(command_queue,d_random,CL_TRUE, 0, RAND_N *
sizeof(cl_float),h_random,0,NULL,NULL);
		checkErr(err,"clEnqueueReadBuffer");

		if(MENSAJES_ACTIVOS==1)
			printf("Shutting down lanzarRandom...\n");
		//Cerramos la cola de comandos y liberamos la cola de comandos, el kernel,
el programa y el contexto
		clFinish(command_queue);
		clReleaseKernel(kernel);
		clReleaseProgram(program);
		clReleaseCommandQueue(command_queue);
		clReleaseContext(context);
		return elapsedTime;
}

double lanzarPPI(float *h_random,char *argv[]){
		const int N_WORK_GROUP = atoi(argv[2]);
		const int N_WORK_ITEM_WORK_GROUP = atoi(argv[3]);

		// Lectura de la cabecera
    Read_header(argv[1]);
		if(MENSAJES_ACTIVOS==1){
			printf(" num_lines = %d\n", num_lines);
			printf(" num_samples = %d\n", num_samples);
			printf(" num_bands = %d\n", num_bands);
		}

		// Punteros para la memoria del host
		float *h_imagen;
		int *h_res_total;
		cl_int *h_res_parcial;
		// Punteros para la memoria del device
		cl_mem d_imagen, d_res_parcial, d_random;

		cl_int err;
		cl_context context;
		cl_device_id *devices;
		size_t size_devices;
		cl_int num_devices;
		cl_platform_id *platforms;
		cl_uint num_platforms;
```

```
          cl_context_properties properties;
          size_t N=NN;
          size_t global_work_size;
          size_t local_work_size;
          unsigned int i,j,k,selected_platform,selected_device;
          cl_float x;

          global_work_size=N_WORK_GROUP*N_WORK_ITEM_WORK_GROUP;
          local_work_size=N_WORK_ITEM_WORK_GROUP;

          //Obenemos las plataformas disponibles y seleccionamos una
          err= clGetPlatformIDs(0, NULL, &num_platforms);
          checkErr(err,"clGetPlatformIDs");
          if(MENSAJES_ACTIVOS==1)
                    std::cout << "Detectada(s) " << num_platforms << " plataforma(s)"
<< std::endl;
          platforms=(cl_platform_id *) malloc(sizeof(cl_platform_id)*num_platforms);
          err= clGetPlatformIDs(num_platforms, platforms, NULL);
          checkErr(err,"clGetPlatformIDs");
          selected_platform=0;
          if(MENSAJES_ACTIVOS==1){
                    showAvailablePlatforms(num_platforms,platforms);
                    std::cout << "Seleccionamos la plataforma " << selected_platform
<< std::endl;
                    std::cout << "Creamos un contexto para los dispotivos presentes
en la plataforma " << std::endl;
          }

          //Creamos un contexto para dicha plataforma ****************
          cl_context_properties cps[3] = {CL_CONTEXT_PLATFORM,
(cl_context_properties)platforms[selected_platform], 0};
          context=clCreateContextFromType(cps, CL_DEVICE_TYPE_ALL,NULL,NULL,&err);
          checkErr(err,"clCreateContextFromType");

          err = clGetContextInfo(context,CL_CONTEXT_DEVICES,0,NULL,&size_devices);
          checkErr(err,"clGetContextInfo");

          num_devices=size_devices/sizeof(cl_device_id);
          if(MENSAJES_ACTIVOS==1)
                    std::cout << "Detectado(s) " << num_devices << " dispositivo(s) "
<< " en la plataforma " << selected_platform << std::endl;
          devices= (cl_device_id*) malloc(size_devices);
          err =
clGetContextInfo(context,CL_CONTEXT_DEVICES,size_devices,devices,NULL);
          checkErr(err,"clGetContextInfo");

          if(MENSAJES_ACTIVOS==1)
                    showAvailableDevices(num_devices,devices);
          selected_device=0;
          cl_command_queue command_queue=clCreateCommandQueue(context,
devices[selected_device], CL_QUEUE_PROFILING_ENABLE,NULL);
          checkErr(err,"clCreateCommandQueue");
          if(MENSAJES_ACTIVOS==1)
                    std::cout << "Creamos una cola de comandos para el dispositivo "
<< selected_device << std::endl;
          //*********************************************************

          //Creamos una cola para cada variable.***********
          // Asignacion de memoria para host y device
          //para host
          h_imagen = (float*) malloc(num_lines * num_samples * num_bands *
sizeof(float));
          h_res_total = (int*) malloc(num_lines * num_samples * sizeof(int));
          h_res_parcial = (cl_int*)
malloc(N_WORK_ITEM_WORK_GROUP*N_WORK_GROUP*2*sizeof(int));

          //para device
          d_imagen=clCreateBuffer(context, CL_MEM_READ_ONLY,
num_lines*num_samples*num_bands*sizeof(cl_float),NULL,NULL);
          checkErr(err,"clCreateBuffer");
          d_random=clCreateBuffer(context, CL_MEM_READ_ONLY, RAND_N
*sizeof(cl_float),NULL,NULL);
          checkErr(err,"clCreateBuffer");
          d_res_parcial=clCreateBuffer(context, CL_MEM_WRITE_ONLY,
N_WORK_ITEM_WORK_GROUP*N_WORK_GROUP*2*sizeof(cl_int),NULL,NULL);
          checkErr(err,"clCreateBuffer");
          //Leemos la imagen
```

```
		Load_Image(argv[1], h_imagen);
		for(int i=0; i<num_lines*num_samples; i++){
			h_res_total[i]=0;
		}
		for (int j=0; j<N_WORK_ITEM_WORK_GROUP*N_WORK_GROUP*2; j++){
			h_res_parcial[j]=-1;
		}

	//Copiamos del host al dispositivo como cudaMemcpy
		err=clEnqueueWriteBuffer(command_queue,d_imagen,CL_TRUE, 0,
num_lines*num_samples*num_bands*sizeof(cl_float),h_imagen,0,NULL,NULL);
		checkErr(err,"clEnqueueWriteBuffer");

		err=clEnqueueWriteBuffer(command_queue,d_random,CL_TRUE, 0, RAND_N
*sizeof(cl_float),h_random,0,NULL,NULL);
		checkErr(err,"clEnqueueWriteBuffer");
		//*****************************************

		//Leemos el código del kernel a ejecutar de un fichero
		std::ifstream file("./ppi.cl");
		checkErr(file.is_open() ? CL_SUCCESS : -1, "ifstream() no puede acceder al
fichero");
		std::string sourceString( std::istreambuf_iterator<char>(file),
(std::istreambuf_iterator<char>()));

		//Convertimos el string leído a char *
		const char *source = souríString.c_str();
		size_t sourceSize[]={ strlen (source) };
		cl_program program =
clCreateProgramWithSource(context,1,&source,sourceSize,&err);
		checkErr(err,"clCreateProgramWithSource");

		// Creo un ejecutable para todos los dispositivos disponibles
		err = clBuildProgram(program, num_devices, devices, NULL, NULL, NULL);
		if (err != CL_SUCCESS) {
			std::cerr << "ERROR: CompilaciÃ³n del Kernel" << std::endl;
			char* build_log;
			size_t log_size;
			clGetProgramBuildInfo(program, devices[0], CL_PROGRAM_BUILD_LOG,
0, NULL, &log_size);
			build_log = new char[log_size+1];
			clGetProgramBuildInfo(program, devices[0], CL_PROGRAM_BUILD_LOG,
log_size, build_log, NULL);
			build_log[log_size] = '\0';
			if(MENSAJES_ACTIVOS==1)
				std::cout << build_log << std::endl;
			delete[] build_log;
			exit(EXIT_FAILURE);
		}
/*****************************************************
// CL_DEVICE_MAX_WORK_GROUP_SIZE
size_t workgroup_size;
clGetDeviceInfo(devices[0], CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(workgroup_size),
&workgroup_size, NULL);
printf("  CL_DEVICE_MAX_WORK_GROUP_SIZE:\t%u\n", workgroup_size);
// CL_DEVICE_MAX_WORK_ITEM_SIZES
size_t workitem_size[3];
clGetDeviceInfo(devices[0], CL_DEVICE_MAX_WORK_ITEM_SIZES, sizeof(workitem_size),
&workitem_size, NULL);
printf("  CL_DEVICE_MAX_WORK_ITEM_SIZES en 1Âª dimension:\t%u \n",
workitem_size[0]);
*****************************************************/
		// Creo un kernel usando el programa que acabo de construir
		cl_kernel kernel=clCreateKernel(program,"ppi",&err);
		checkErr(err,"clCreateKernel");

		//************************ PARA EL PASO DE ARGUMENTOS
**********************************
		//Al kernel le paso como argumento los 2 vectores de entrada, el vector
resultado y el tamaño del vector
		err=clSetKernelArg(kernel,0,sizeof(cl_mem),&d_imagen);
		checkErr(err,"clSetKernelArg");

		err=clSetKernelArg(kernel,1,sizeof(cl_mem),&d_random);
		checkErr(err,"clSetKernelArg");

		err=clSetKernelArg(kernel,2,sizeof(cl_mem),&d_res_parcial);
```

```
		checkErr(err,"clSetKernelArg");

		err=clSetKernelArg(kernel,3,sizeof(cl_uint),&num_lines);
		checkErr(err,"clSetKernelArg");

		err=clSetKernelArg(kernel,4,sizeof(cl_uint),&num_samples);
		checkErr(err,"clSetKernelArg");

		err=clSetKernelArg(kernel,5,sizeof(cl_uint),&num_bands);
		checkErr(err,"clSetKernelArg");

		cl_event event;
		//Pongo la ejecución del kernel en la cola especificando el tamaño del
grupo de trabajo y el tamaño de cada grupo
		if(MENSAJES_ACTIVOS==1)
			std::cout << "Poniendo en ejecucion el kernel.\n";
		err=clEnqueueNDRangeKernel(command_queue,kernel,1,NULL,&global_work_size,
&local_work_size,0,NULL,&event);
		checkErr(err,"clEnqueueNDRangeKernel");


		cl_ulong start,end;
		clWaitForEvents(1, &event);
		clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
sizeof(cl_ulong), &start, NULL);
		clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(cl_ulong),
&end, NULL);
		clReleaseEvent(event);

		double elapsedTime = (double)1.0e-6 * (end - start);

		//Leo el resultado
		err=clEnqueueReadBuffer(command_queue,d_res_parcial,CL_TRUE, 0,
N_WORK_ITEM_WORK_GROUP*N_WORK_GROUP*2*sizeof(cl_uint),h_res_parcial,0,NULL,NULL);
		checkErr(err,"clEnqueueReadBuffer");
		//Agrupamos los resultados
		for(int i=0; i<N_WORK_ITEM_WORK_GROUP*N_WORK_GROUP*2; i++)
			h_res_total[h_res_parcial[i]]=h_res_total[h_res_parcial[i]]+1;

		//Escribimos la imagen de pureza
		EscribirResultado(h_res_total, argv[1]);

		if(MENSAJES_ACTIVOS==1)
			printf("Shutting down...\n");
		//Cerramos la cola de comandos y liberamos la cola de comandos, el kernel,
el programa y el contexto
		clFinish(command_queue);
		clReleaseKernel(kernel);
		clReleaseProgram(program);
		clReleaseCommandQueue(command_queue);
		clReleaseContext(context);
		return elapsedTime;
}

int main(int argc, char *argv[]) {
		double elapsedTimeRnd=0;
		double elapsedTimePPI=0;

		if (argc != 4){
			if(MENSAJES_ACTIVOS==1){
				printf("ERROR: nº de parametros incorrecto\n");
				printf("Uso: ./ppi <NombreDeLaImagen> <N. work-groups>
<N. de work-items/work-group>\n");
			}
			return 0;
		}

		if(MENSAJES_ACTIVOS==1)
			printf("\n\n------------ Iniciando Algoritmo PPI para OpenCL -----
----\n");

		cl_float *h_random =  (float *)malloc(RAND_N * sizeof(float));

		if(MENSAJES_ACTIVOS==1)
			printf("\n************ Generando vectores aleatorios
**************\n");
		elapsedTimeRnd=lanzarRandom(h_random, argv);
```

```
		/*printf("\n************ Obteniendo de archivo los aleatorios
**************\n");
		char aux[200];
		for(int i=0; i< RAND_N; i++){
			std::cin>>aux;
			h_random[i]=atof(aux);
		}*/

		if(MENSAJES_ACTIVOS==1)
			printf("\n******************* Lanzando PPI
************************\n");
		elapsedTimePPI = lanzarPPI(h_random, argv);

		printf("N_WORK_GROUP:; N_WORK_ITEMS/WORK_GROUP:; algoritmo;
tiempo(ms.)\n");
		printf("%i; %i; lanzarRandom; %.2f\n", atoi(argv[2]), atoi(argv[3]),
elapsedTimeRnd);
		printf(";;lanzarPPI; %.2f\n", elapsedTimePPI);

		delete h_random;
		return 1;
}
```

Archivo de kernel

```
#include "ppi.h"
#define MIN_INT ((float) (1 << (sizeof(float) * 8 - 1)))//Entero minima
#define MAX_INT (-(MIN_INT + 1))//Entero máximo

__constant static mt_struct_stripped d_MT[MT_RNG_COUNT];

//
__kernel void RandomGPU(__global float *d_Random, uint NPerRng, uint seed){
		__private uint	tid = get_local_size(0) * get_group_id(0) +
get_local_id(0);
		__private uint THREAD_N = get_local_size(0) * get_num_groups(0);

		__private uint iState, iState1, iStateM, iOut;
		__private uint mti, mti1, mtiM, x;
		__private uint mt[MT_NN];

		for(uint iRng = tid; iRng < MT_RNG_COUNT; iRng += THREAD_N){
			//Load bit-vector Mersenne Twister parameters
			mt_struct_stripped config;
			//config = ds_MT[iRng];

			//Initialize current state
			//mt[0] = config.seed;
			mt[0] =seed*tid;
			for(iState = 1; iState < MT_NN; iState++)
				mt[iState] = (1812433253U * (mt[iState - 1] ^ (mt[iState
- 1] >> 30)) + iState) & MT_WMASK;

			iState = 0;
			mti1 = mt[0];
			for(iOut = 0; iOut < NPerRng; iOut++)
			{
				iState1 = iState + 1;
				iStateM = iState + MT_MM;
				if(iState1 >= MT_NN) iState1 -= MT_NN;
				if(iStateM >= MT_NN) iStateM -= MT_NN;
				mti  = mti1;
				mti1 = mt[iState1];
				mtiM = mt[iStateM];

				x  = (mti & MT_UMASK) | (mti1 & MT_LMASK);
				x  =  mtiM ^ (x >> 1) ^ ((x & 1) ? config.matrix_a : 0);

				mt[iState] = x;
				iState = iState1;

				//Tempering transformation
				x ^= (x >> MT_SHIFT0);
```

```
                                x ^= (x << MT_SHIFTB) & config.mask_b;
                                x ^= (x << MT_SHIFTC) & config.mask_c;
                                x ^= (x >> MT_SHIFT1);

                                //Convert to (0, 1] float and write to global memory
                                d_Random[iRng + iOut  * MT_RNG_COUNT] = -0.5+(((float)x
+ 1.0f) / 4294967296.0f);
                        }
                }
}


//Numero de pixeles que se almacenarán en memoria compartida.
# define N_Pixels 9
//Tamaño del vector que se almacenará en memoria compartida.
# define Tamanio_Vector (224*N_Pixels)
//Funcion: kernel PPI. Es el algoritmo PPI propiamente dicho.
//Entradas: float *d_imagen: estructura que almacena los datos de la imagen.
//                  float *d_random: estructura que almacena los vectores aleatorios.
//                  float *d_res_parcial:estructura que almacenara los resultados tras
la ejecucion del algoritmo.
//                  int num_lines: numero de lineas de la imagen.
//                  int num_samples: numero de columnas de la imagen.
//                  int num_bands: numero de bandas de la imagen.
//Salidas: la funcion no devuelve nada.
__kernel void ppi(__global const float * d_imagen, __global const float
*d_random,__global int *d_res_parcial, const uint num_lines, const uint num_samples,
const uint num_bands) {
          __private uint idx =  get_global_id(0);//get_local_size(0) *
get_group_id(0)+get_local_id(0);

          __private float pemax; // producto escalar maximo
          __private float pemin; // producto escalar minimo
          __private float pe; //producto escalar
          __private int v,d;
          __private int imax=0;
          __private int imin=0;
          __local float s_pixels[Tamanio_Vector];
          __private float l_rand[224];

          pemax=MIN_INT;
          pemin=MAX_INT;
          //Copiamos un pixel de la memoria global a los registros
          for (uint k=0; k<num_bands; k++){
                    l_rand[k]=d_random[idx*num_bands+k];
          }

          for(uint it=0; it<num_lines*num_samples/N_Pixels; it++){
                    barrier(CLK_LOCAL_MEM_FENCE);
                    //Copiamos N_Pixels pixels a la memoria compartida
                    if(get_local_id(0)<N_Pixels){
                                        for(uint j=0; j<num_bands; j++){

          s_pixels[get_local_id(0)+N_Pixels*j]=d_imagen[(it*N_Pixels+get_local_id(0))+
(num_lines*num_samples*j)];
                                        }
                    }
                    barrier(CLK_LOCAL_MEM_FENCE);
          ////////////////////////////////////////////////////////////////////////
                    //Recorremos el vector de pixels

                    for (v=0; v <N_Pixels; v++) // El vector tiene Tamanio_Vector
elementos
                    {
                              // calculamos producto escalar
                              pe = 0;
                              for (d=0; d < num_bands; d++){
                                        pe = pe + l_rand[d]*s_pixels[v+N_Pixels*d];
                              }
                              // detectamos los extremos
                              if (pe > pemax) {
                                        imax=it*N_Pixels+v;
                                        pemax=pe;
                              }
                              if (pe < pemin) {
                                        imin=it*N_Pixels+v;
                                        pemin=pe;
```

```
                        }
                    }
                }
                // Actualizamos d_res_parcial
                d_res_parcial[idx*2]=imax;
                d_res_parcial[idx*2+1]=imin;
}
```