Universidad de Valladolid

E.T.S.I. Telecomunicación

TRABAJO FIN DE MÁSTER

MÁSTER EN INGENIERÍA DE TELECOMUNICACIÓN

Desarrollo y optimización de algoritmos paralelos de cálculo general sobre el framework OpenCLIPER

Autor:
Julián González Díaz

Tutor:
Federico Simmross Wattenberg

Valladolid, septiembre de 2022

Título:	**Desarrollo y optimización de algoritmos paralelos de cálculo general sobre el framework OpenCLIPER**
Autor:	**Julián González Díaz**
Tutor:	**Federico Simmross Wattenberg**
Departamento:	**TSCIT**

TRIBUNAL

Presidente:	**Santiago Aja Fernández**
Secretario:	**Pablo Casaseca de la Higuera**
Vocal:	**Manuel Rodríguez Cayetano**
Presidente Suplente:	**Juan Ignacio Asensio Pérez**
Secretario Suplente:	**Luis Miguel San José Revuelta**
Vocal Suplente:	**Miguel Bote Lorenzo**

Fecha:

Calificación:

Resumen

En el ámbito médico, las pruebas de diagnóstico mediante imagen son una de las técnicas más empleadas hoy en día. Con el objetivo de agilizar todo el procesamiento de imágenes, nace la necesidad de emplear dispositivos eficientes como las GPU para esta tarea. La implementación de algoritmos eficientes sobre estos dispositivos permite agilizar los tiempos de procesamiento, así como mejorar la calidad de la imagen médica, ayudando a que se pueda llevar a cabo un mejor diagnóstico. Con este objetivo nació OpenCLIPER, como un framework diseñado para permitir a los investigadores centrarse en el problema de diseño de esto algoritmos, simplificando gran parte de la complejidad que supone realizar desarrollos para estos dispositivos GPU.

Sin embargo, estudiando los algoritmos existentes dentro del framework OpenCLIPER en la actualidad, se detecta una carencia de algoritmos básicos, que realicen operaciones fundamentales sobre las que se puedan construir nuevos algoritmos más complejos. Debido a este problema, surge este proyecto, que tiene como objetivo aportar nuevos algoritmos de procesamiento de imagen para resolver esta carencia el framework y mejorar sus capacidades. Para ello se desarrollarán e integrarán en la plataforma algoritmos que nos permitan realizar el producto de matrices, elevar una matriz a una potencia, calcular la norma-N de un vector, realizar el producto escalar y vectorial de dos vectores, y calcula la convolución de una matriz frente a un *kernel* de convolución. El objetivo principal es la optimización de dichos algoritmos de forma que la integración sea lo más eficiente posible. Finalmente, se llevará a cabo un análisis de rendimiento de los algoritmos implementados, que nos permitirá comparar su eficiencia con plataformas de procesamiento de referencia, evaluar la eficiencia de los mismos y detectar diferente mejoras que nos permitan seguir desarrollándolos y mejorándolos en el futuro.

Palabras Clave

GPU, OpenCL, OpenCLIPER, Convolución Matrices, Multiplicación Matrices, Procesamiento Imagen, Norma-N, Producto Escalar, Producto Vectorial

Abstract

In the medical field, diagnostic imaging tests are one of the most widely used techniques today. In order to streamline all image processing, the need arises to use efficient devices such as GPUs for this tasks. The implementation of efficient algorithms on these devices speeds up processing times, as well as improves the quality of the medical image, helping to carry out a better diagnosis. With this objective, OpenCLIPER was born as a framework designed to allow researchers to focus on the problem of designing these algorithms, simplifying much of the complexity involved in developing on these GPU devices.

However, studying the existing algorithms within the OpenCLIPER framework at present, a lack of basic algorithms is detected, which perform fundamental operations on which new more complex algorithms can be built. Due to this problem, this project arises, which aims to provide new image processing algorithms to solve this lack of the framework and improve its capabilities. For this, algorithms will be developed and integrated into the platform that allow us to perform the product of matrices, raise a matrix to a power, calculate the Euclidean norm of a vector, perform the dot and vector product of two vectors, and calculate the convolution of a matrix versus a convolution *kernel*. The main objective is the optimization of these algorithms so that the integration will be as efficient as possible. Finally, a performance analysis of the implemented algorithms will be carried out, which will allow us to compare their efficiency with reference processing platforms, evaluate their efficiency and detect different improvements that will allow us to continue developing and improving them in the future.

Keywords

GPU, OpenCL, OpenCLIPER, Matrix Convolution, Matrix Multiplication, Image Processing, Norm-N, Scalar Product, Vectorial Product

Agradecimientos

En primer lugar, me gustaría agradecer todo el esfuerzo y paciencia a mi tutor, Federico Simmross Wattenberg, por darme la posibilidad de hacer este Trabajo de Fin de Máster y por su tiempo dedicado pese a todas las vueltas que ha dado el mismo.

Por otro lado, a mi pareja, Azahara, por aguantarme y apoyarme, ayudándome a sacar fuerzas día a día. Sin tu apoyo habría sido mucho más difícil sacar este proyecto adelante.

También a mi familia, que gracias a su insistencia han conseguido que nunca abandonara la esperanza de poder acabarlo.

Por último, a todos mis amigos y compañeros, cuyo consejo y ánimos en los momentos malos me han ayudado a no tirar la toalla.

Índice General

Capítulo 1

Introducción

Los sistemas de computación actuales están integrados por cada vez más dispositivos dedicados. Por ejemplo, en cualquier ordenador personal como los que usamos diariamente tenemos la CPU, encargada de las operaciones principales de gestión de recursos y cálculo; una GPU, encargada del procesamiento gráfico; una tarjeta de red, cuya tarea es el envío y recepción de datos a través de diferentes redes de comunicaciones; una tarjeta de sonido, encargada de procesar la entrada y salida de audio; y existen muchos otros. Estos dispositivos nos permiten mejorar el rendimiento de los sistemas sin aumentar en gran medida el coste gracias a su diseño especializado en la tarea para la que están dedicados. La coordinación entre todos los elementos de un sistema, encargando cada tarea al dispositivo específicamente diseñado para ella, nos permite mejorar el rendimiento gracias a un mejor uso de los recursos [1].

Además, cada vez es más importante el procesamiento de imágenes en nuestra vida cotidiana. Desde las redes sociales que usamos diariamente hasta las técnicas de diagnóstico médico, las imágenes son entidades que nuestros dispositivos tienen que almacenar y procesar continuamente. Esto unido a que los dispositivos son cada vez más potentes y a la necesidad de que sean cada vez más eficientes, por ejemplo en cuanto a consumo de energía, provoca que sea muy importante buscar la eficiencia en el tratamiento de imagen.

Para esta tarea, dispositivos dedicados como las GPU, presentes actualmente en todo tipo de sistemas informáticos cobran un papel fundamental, pasando de ser elementos dedicados a la generación de gráficos a ser herramientas de cálculo muy avanzadas gracias a su diseño. Frente a la limitación que supone el uso de hardware de propósito general como las CPU, las GPU nos permiten ahorrar tiempo y recursos, liberando además a la CPU para otras tareas. Por ello es fundamental el avance en algoritmos que aprovechen este entorno de computación heterogéneo al máximo, permitiéndonos ahorrar tiempo y

mejorar los resultados en estas tareas.

Dentro del desarrollo de algoritmos para GPU existen principalmente dos plataformas. Por un lado, CUDA es una plataforma propietaria de Nvidia desarrolla exclusivamente para su uso en las tarjetas gráficas de la misma marca. Es una solución que tiene gran soporte y muchas herramientas ya integradas, pero que está limitada a las GPU de esta marca exclusivamente [10]. Por otro lado, OpenCL es la plataforma de código libre más extendida. Mantenida por el grupo Khronos, tiene como principal objetivo poder emplearse en cualquier dispositivo, y admite ejecución de algoritmos sobre todo tipo de GPU y CPU diferentes. Sin embargo, esto provoca que la gestión de la plataforma no sea sencilla, y requiera un conocimiento profundo de la misma. [6]

En este contexto en el que podemos emplear la potencia de cálculo que nos ofrecen las GPU para procesamiento de imagen nace OpenCLIPER. OpenCLIPER es un framework para procesamiento y reconstrucción de imagen médica desarrollado sobre OpenCL. Este framework tiene el objetivo de simplificar a los desarrolladores las complejidad de OpenCL permitiéndoles centrarse en el desarrollo de los algoritmos. A pesar de que al arquitectura de esta herramienta ofrece muchas posibilidades a los desarrolladores, se observan una serie de carencias debido a que el número de algoritmos que tiene implementado en la actualidad es bastante limitado. [7]

Este trabajo tiene el objetivo de mejorar OpenCLIPER aumentando la variedad de algoritmos de los que dispone. Para ello, se van a desarrollar diferentes algoritmos de procesamiento de imagen empleando las facilidades que nos ofrece el framework de forma que puedan integrarse en el mismo. El enfoque principal de la implementación será la búsqueda de la máxima eficiencia, centrando el proceso de diseño de la optimización de los algoritmos para GPU sobre este framework. Además, se llevará a cabo un análisis de rendimiento, comparando los tiempos de procesado con una herramienta de referencia y evaluando la mejora alcanzada con el objetivo de estudiar sus fortalezas y debilidades, planteando posibles mejoras para la evolución de dichos algoritmos.

1.1 Motivaciones

El reto del proyecto será conseguir algoritmos funcionales buscando la máxima eficiencia para ciertas operaciones de procesamiento de imagen, aprovechado la capacidad de

la GPU para el procesamiento de datos vectoriales. Pese a que el hardware evoluciona cada día más y más, esta mejora no es suficiente si no se dispone de algoritmos diseñados específicamente para aprovechar todas las ventajas del mismo. Además, estos dispositivos tienen una estructura y fundamentos completamente diferentes a los dispositivos CPU que han sido usados habitualmente para esta tarea, por lo que es necesario un enfoque completamente diferente para el desarrollo de estos algoritmos. [5].

De forma general, la cantidad de información que contiene una imagen depende de su tamaño. Cuando más grande sea la misma, mayor capacidad de cómputo se requiere para procesar todos los detalles que contiene [12]. Esto también genera la necesidad de mejores algoritmos, dado que en muchos entornos cada vez es más importante la resolución y calidad de la imagen, elevando la carga computacional de los algoritmos que procesan las mismas.

Centrándonos específicamente en la imagen médica, es muy importante poder aprovechar la mejora en la tecnología para dar un mejor y más rápido diagnóstico a los pacientes. Hace años, cuando se hacía una prueba que implicaba imagen médica, había que esperar tiempo desde la adquisición de datos hasta que la prueba estaba lista para ser analizada por un profesional. Además, la calidad de estas imágenes era relativamente baja. En la actualidad, se dispone de la tecnología para que los resultados de estas pruebas estén disponibles casi al instante y con una calidad mucho más elevada. Es fundamental desarrollar algoritmos que nos permitan aprovechar los recursos de la mejor manera posible, facilitando y mejorando la precisión del diagnóstico.

OpenCLIPER es una herramienta diseñada para facilitar el desarrollo de algoritmos eficientes para mejorar este problema, pero como se indicaba anteriormente tiene ciertas carencias. Dado que su arquitectura permite facilitar enormemente la tarea del desarrollo de algoritmos y ejecutar procesamientos complejos, mejorar el número de algoritmos de los que dispone resulta de gran interés. Estos nuevos algoritmos permitirán no solo emplear el framework para más tareas y procesos, sino también servir como referencia para que en el futuro se puedan construir sobre los mismos algoritmos más complejos.

1.1.1 Objetivos

Vistas las motivaciones y campos de mejora descritos, el objetivo principal del proyecto es el desarrollo de algoritmos de procesado de imagen en GPU, aprovechando la versatilidad de la plataforma OpenCL para este propósito y mejorando su facilidad de uso con el framework OpenCLIPER. Por un lado, el uso de OpenCL viene motivado principalmente por su versatilidad [6]. Uno de los principales objetivos del proyecto es que pueda emplearse en numerosos dispositivos, y que no sea necesario el uso de una licencia de pago para su uso en entorno empresarial. El uso del framework OpenCLIPER se debe a las facilidades que nos brinda para el desarrollo sobre OpenCL, reduciendo en gran medida su complejidad [15]. Además, este framework hace más sencillo la mejora de los algoritmos, ya que realiza de forma transparente toda la parte de gestión de la plataforma OpenCL, por lo que mejoras y cambios en la base del algoritmo se propagan de forma transparente sobre la plataforma de desarrollo. Esto permite enriquecer el propio framework, ya que la incorporación a su estructura de estos algoritmos, le permitirá soportar más operaciones y dotarlo de más funcionalidades.

De forma concreta, el objetivo del proyecto es la implementación de los siguientes algoritmos de procesamiento de imagen:

- **Algoritmo de cálculo de la norma-N de un vector** [2], que permitirá calcular la norma-N de cualquier orden de un vector de cualquier tamaño y cuyos elementos pueden ser números reales y números complejos.
- **Algoritmo de producto escalar de vectores** [9][16], que proporciona el resultado de calcular el producto escalar de dos vectores de cualquier tamaño.
- **Algoritmo de producto algebraico de matrices** [13], que permitirá realizar el producto de dos matrices de cualquier tamaño, y cuyos elementos pueden ser números reales y números complejos.
- **Algoritmo de potencia de una matriz** [13], que se basará en el algoritmo anterior para elevar una matriz cuadrada a un exponente que sea un número natural.
- **Algoritmo de convolución de matrices** [12], el cual es un algoritmo básico para el procesamiento de imágenes mediante filtros, y proporciona la matriz resultado de realizar la convolución entre dos matrices.

1.1.2 FASES Y MÉTODOS

En el desarrollo de este trabajo se han seguido las siguientes etapas para la consecución de los objetivos:

1. **Etapa de formación**. En esta parte inicial, se ha realizado un estudio documental de los algoritmos, las plataformas y el paradigma de programación. El objetivo ha sido obtener los conocimientos previos necesarios para la ejecución del proyecto, dividido en los siguientes pasos:
    1. Repasar los fundamentos teóricos, técnicos y matemáticos de la información contenida dentro de las imágenes, así como de su representación lógica a nivel computacional.
    2. Revisar la teoría matemática subyacente a los algoritmos a desarrollar, así como su implementación computacional.
    3. Estudiar el paradigma de programación empleado por los dispositivos de tipo GPU, basado en el procesamiento vectorial de datos. Además de estudiar las características de estos dispositivos y sus fundamentos.
    4. Comprensión de la plataforma de programación OpenCL, tanto a nivel lógico como estructural. Como añadido, ver como explotar de mejor manera sus virtudes y evitar su debilidades.
    5. Aprender el uso del framework OpenCLIPER, tanto sus fundamentos como su estructura, y las facilidades que nos da a la hora de integrar los algoritmos con OpenCL.
2. **Etapa de diseño**. Con todos los conocimientos teóricos, se han planteado diferentes soluciones y algoritmos para resolver el problema:
    1. Diseño de diferentes enfoques al problema, de forma teórica.
    2. Análisis de las soluciones. Ventajas e inconvenientes de cada una de forma teórica.
    3. Decisión final sobre la versión a implementar.
3. **Implementación y optimización**. Una vez definidos detalladamente los algoritmos a implementar, se ha desarrollado el código de los mismos progresivamente, haciendo pruebas y buscando mejorar la eficiencia:

1. Implementación del algoritmo de multiplicación de números reales, desarrollo del proceso de multiplicación en OpenCLIPER y creación de un programa de prueba, que permita depurar errores y validar los resultados con datos controlados.
2. Creación del algoritmo de multiplicación para números complejos.
3. Adaptación del proceso de multiplicación de OpenCLIPER para que elija dinámicamente el algoritmo a utilizar en función de los datos, y validación del algoritmo para números complejos y el flujo de selección de algoritmo.
4. Implementación del algoritmo de calculo de norma-N en GPU, creación del proceso de OpenCLIPER para soportar el algoritmo y validación del flujo con datos de prueba.
5. Implementación del algoritmo de calculo de norma-N en GPU para números complejos, modificación del proceso de OpenCLIPER para que elija dinámicamente el algoritmo adecuado y validación del flujo completo con datos de prueba.
6. Creación del proceso de OpenCLIPER para calcular la potencia de una matriz en base al algoritmo de multiplicación y validación del flujo contra datos de prueba.
7. Implementación del algoritmo de calculo de producto escalar de vectores, creación del proceso de OpenCLIPER para soportar el algoritmo y validación del flujo contra datos de prueba.
8. Implementación del algoritmo de calculo de producto escalar de vectores para números complejos, modificación del proceso de OpenCLIPER para que elija dinámicamente el algoritmo adecuado y validación del flujo completo con datos de prueba.
9. Implementación del algoritmo de convolución y el *kernel* de *padding* para el procesamiento de los datos de entrada.
10. Creación del proceso de convolución en OpenCLIPER así como todo el flujo de preprocesado y ejecución con un programa de prueba.
11. Validación de la implementación, con datos de prueba y depurado de errores.

4. **Estudio del rendimiento**. Tras implementar y validar los algoritmos, se ha llevado

acabo un análisis de rendimiento, con el objetivo de evaluar la integración y poder plantear mejoras:

1. Recogida de datos y comparación entre el rendimiento de CPU y GPU ejecutando el algoritmo de multiplicación, para matrices de números reales y números complejos de tamaño variable.
2. Empleo del programa Matlab para ejecutar las operaciones equivalentes en multiplicación de matrices, recogida de datos de rendimiento y realización de comparativa.
3. Recogida de datos y comparación entre el rendimiento de CPU y GPU ejecutando el algoritmo de potencia de matriz, para matrices de números reales y números complejos, variando el tamaño de la matriz y la potencia.
4. Empleo del programa Matlab para ejecutar las operaciones equivalentes a la potencia de matriz, recogida de datos de rendimiento y realización de comparativa.
5. Recogida de datos y comparación entre el rendimiento de CPU y GPU ejecutando el algoritmo de norma-N de un vector, para vectores de números reales y números complejos, variando el tamaño del vector y la potencia.
6. Empleo del programa Matlab para ejecutar las operaciones equivalentes a la norma-N de un vector, recogida de datos de rendimiento y realización de comparativa.
7. Recogida de datos y comparación entre el rendimiento de CPU y GPU ejecutando el algoritmo de producto escalar de vectores, para vectores de números reales y números complejos, variando el tamaño de los vectores.
8. Empleo del programa Matlab para ejecutar las operaciones equivalentes al producto escalar de vectores, recogida de datos de rendimiento y realización de comparativa.
9. Recogida de datos y comparación entre el rendimiento de CPU y GPU ejecutando el algoritmo de convolución, variando el tamaño de las matrices.
10. Empleo del programa Matlab para ejecutar las operaciones equivalentes a la convolución de matrices, recogida de datos de rendimiento y realización de comparativa.

1.1.3 Medios materiales

Para la realización de este proyecto han sido necesarias las siguientes herramientas y medios materiales, que se exponen a continuación:

Hardware

- Servidor de cálculo, con las siguientes características:
  - **Procesador:** Intel(R) i7-4790 CPU 3.60GHz.
  - **Memoria RAM:** 8 GB.
  - **GPU:** NVIDIA GeForce RTX 2080 Ti.
  - **Sistema Operativo:** Ubuntu SMP x64 20.04.
  - **Versión de OpenCL:** OpenCL C 1.2.

Software

- **KDevelop [4]:** editor de código que soporta múltiples lenguajes de programación, en este caso los requeridos C y C++. De código libre.
- **Latex [22]:** procesador de textos de código libre, empleado para generar la memoria de este proyecto.
- **Matlab R2020b [19]:** lenguaje de programación matemático, que nos ha permitido procesar los resultados de rendimiento de forma estadística. También empleado para realizar simulaciones y evaluar el rendimiento de los algoritmos desarrollados en comparativa a los tiempos de procesamiento de referencia de esta herramienta.
- **TigerVNC [20]:** aplicación de escritorio remoto que funciona sobre SSH. Empleada para conectarse al servidor de cálculo, tanto para el desarrollo del código como para la realización de pruebas de rendimiento.

1.2 Estructura del documento

Para finalizar esta introducción al documento, se van a exponer los diferentes capítulos de los que trata el mismo, de forma resumida.

Para comenzar, en el capítulo 2 se plantean los fundamentos de la computación heterogénea y cómo este paradigma tiene múltiples ventajas frente a la computación homogénea clásica. Además, se revisaran los diferentes lenguajes más extendidos para desarrollo de aplicaciones para computación paralela, justificando por qué se escoge OpenCL para este proyecto. Tras la revisión, se expondrá al detalle tanto la arquitectura de la plataforma OpenCL. Esto permitirá entender sus características y todo el desarrollo posterior.

En el siguiente punto, el capítulo 3, se revisará a fondo el framework OpenCLIPER, que servirá de apoyo para el desarrollo y la implementación de los algoritmos sobre OpenCL. Inicialmente, se revisarán las características principales de esta herramienta. Tras esto, se verá al detalle su arquitectura y fundamentos. Por último, se expondrá un pequeño ejemplo de un programa desarrollado, para poder mostrar tanto las características expuestas como poder ayudar a la compresión de los algoritmos complejos posteriores.

A continuación, en el capítulo 4, se hará un salto para exponer los fundamentos matemáticos de los algoritmos que se van a implementar, así como mostrar las fórmulas matemáticas y los datos sobre los que se aplican los diferentes algoritmos. Se llevará una revisión al detalle, muy centrado en avanzar el flujo que se implementará en la práctica y comentando algunas propiedades necesarias a tener en cuenta para la correcta ejecución de las operaciones.

Volviendo al entorno de programación, se llevará a cabo el desarrollo completo de los algoritmos en el capítulo 5. Para todos los algoritmos, se comenzará exponiendo el código que ejecutará la GPU y sus diferentes variantes en función del tipo de dato o del uso de memoria. Tras esto, se detallarán paso a paso el proceso de OpenCLIPER que se encargará de la gestión del código que ejecuta la GPU, desde recibir los datos del usuario hasta devolver la salida. Finalmente, se expondrá un ejemplo de programa de prueba, donde se verá el código que debe desarrollar cualquier usuario para poder ejecutar el algoritmo sobre las datos que desee. Es importante destacar que debido a la arquitectura de OpenCLIPER, muchos de estos procesos que gestionan los algoritmos y los programas de ejemplo son prácticamente iguales, dado que es el objetivo principal de framework es precisamente simplificar este código necesario para la ejecución de algoritmos. Por tanto, se detallará únicamente la parte diferencial necesaria para cada algoritmo, y se detallará de forma completa solo en el primer apartado para el algoritmo y proceso de cálculo de la norma-N de un vector.

Tras el capítulo de implementación se desarrolla el capítulo 6. En este detallaremos cómo se han llevado a cabo las pruebas de rendimiento y el procesamiento de los datos obtenidos de las mismas. Además, mostraremos estos datos en gráficos comparativos, que nos permitirán evaluar la eficiencia de los algoritmos, así como detectar puntos de mejora en los mismos.

Finalmente, en el capítulo 7, se expondrá el aprendizaje obtenido durante el desarrollo de este proyecto, así como las conclusiones del mismo. Por último, se detallarán diversas líneas futuras que puede seguir el proyecto para continuar su desarrollo.

Capítulo 2

Computación Heterogénea

Los sistemas de computación actuales están compuestos por una gran variedad de diferentes tipos de procesadores como las CPU, las GPU, las tarjetas de red, tarjetas de sonido y otros. Estos dispositivos se comunican entre sí para coordinarse en las diferentes tareas para las cuales cada uno de estos procesadores han sido diseñados específicamente. Estos entornos heterogéneos tienen múltiples ventajas, ya que permiten organizar las tareas de forma que cada una se lleve a cabo por el procesador más adecuado, lo cual aumenta el rendimiento del sistema en su conjunto.

Los sistemas homogéneos tienen como limitación fundamental su baja versatilidad. Este tipo de sistemas son muy buenos en las tareas para las cuales han sido diseñados, pero su eficiencia baja enormemente para el resto. Sin embargo, los sistemas heterogéneos se comportan de forma más flexible en entornos donde la variedad de tipos de tareas y carga computacional es alta y se adaptan mucho mejor a diferentes requisitos. Además, la coordinación entre los diversos componentes permite elaborar algoritmos más complejos, que resuelvan de forma más rápida tareas complejas [5].

En esta sección vamos a hablar de las diferencias entre la computación secuencial y la computación paralela y cómo se coordinan ambas para la ejecución de programas. Tras esto, hablaremos de las diferentes plataformas de programación que más se emplean actualmente para el desarrollo de aplicaciones de computación paralela y finalizaremos este apartado exponiendo la empleada en este trabajo: OpenCL. Sobre ésta desarrollaremos en detalle desde sus fundamentos hasta tu arquitectura, ya que será fundamental entenderla completamente para poder adaptar a la misma los algoritmos de cálculo propuestos en este trabajo mejorando su eficiencia.

2.1 Computación secuencial frente a paralela

La computación secuencial es aquella en la cual las diferentes instrucciones que componen el programa se ejecutan una detrás de otra. El flujo del programa resultante coincide con el orden en el que se sitúan las diferentes instrucciones, y el estado del programa en cada momento depende solo de las instrucciones ejecutadas hasta dicho punto [3].

El componente especializado para llevar a cabo tareas de computación secuencial es la CPU o *Central Processing Unit*. Este dispositivo se compone de unos pocos núcleos de procesamiento complejos, que son capaces de manejar numerosos flujos de control y coordinar tareas complejas. Sin embargo, tiene ciertas limitaciones para la realización de cálculos matemáticos, y no es especialmente eficiente ejecutando algoritmos que requieran muchos cálculos simultáneos debido a su flujo de instrucciones secuenciales.

Frente a esta, la computación paralela consiste en la realización de un gran número de tareas que se ejecutan simultáneamente. De esta forma, resolver un problema consiste en dividirlo en pequeñas tareas independientes para que puedan ejecutarse todas al mismo tiempo. Para ello, un gran número de núcleos de cálculos, generalmente sencillos, llevan a cabo cada una de estas tareas sobre los datos, o sobre un subconjunto de los mismos, de forma que la solución final sea una composición de los datos de salida de las diferentes tareas [1].

El dispositivo especialista en esta tarea es la GPU o *Graphics Processing Unit*. Este componente está formado por numerosos núcleos sencillos llamados unidades de computo o *Compute Units*, que son capaces de realizar operaciones matemáticas. La gran especialidad de la GPU es lanzar el mismo algoritmo sencillo para que sea ejecutado al mismo tiempo en su gran cantidad de núcleos sobre distintos datos, de forma que la cantidad de operaciones en el mismo espacio de tiempo es mucho mayor que la que puede llevar a cabo la CPU. A este tipo de procesado, que se estudiará con más detalle en el apartado 2.2.1, en el que se lanzan gran numero de ejecuciones del mismo algoritmo sobre conjuntos de datos sucesivos, se denomina procesamiento vectorial. Esto se debe a que los datos están almacenados en grupos continuos en la memoria, donde para cada ejecución, las diferentes unidades de cómputo procesan de forma simultánea cada grupo de datos continuos almacenados en memoria. El resultado será el conjunto de soluciones obtenidas en la ejecución de cada unidad de cómputo, ordenadas adecuadamente en memoria.

Un ejemplo de esto es el procesamiento de imágenes. Para aplicar transformaciones sobre una imagen completa, muchas veces se puede dividir la tarea en aplicar una transformación sencilla a cada uno de los píxeles que forman la imagen, y luego recomponer la imagen de salida. La computación paralela nos permite que todas las operaciones sobre cada pixel se ejecuten a la vez, lo que supone una gran ventaja frente al escenario en computación secuencial en el que habría que procesar cada pixel a continuación del anterior.

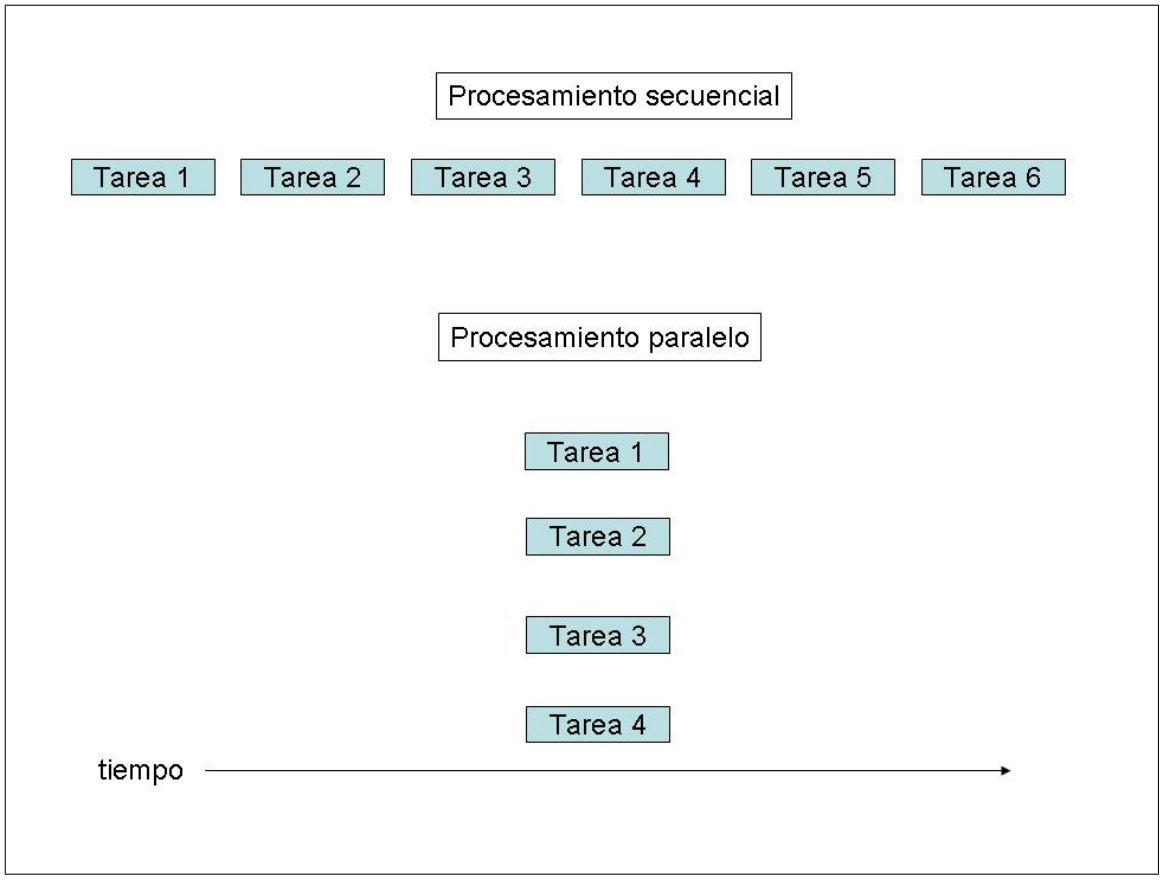

Figura 2.1: Ejemplo de procesamiento secuencial y paralelo.

En un escenario real, la forma de trabajar es una coordinación entre la CPU y la GPU, de forma que estas se coordinan para llevar a cabo la tarea compleja. La CPU se encarga de recibir los datos y distribuirlos adecuadamente en la memoria para que sean accesibles por la GPU, y coordina la ejecución indicando a la GPU qué tareas tiene que realizar. La GPU realiza todos los cálculos aprovechando el paralelismo, y devuelve a la memoria el resultado de las operaciones. Por último, será la CPU la encargada de recoger la información de salida, componerla y almacenarla en función de las necesidades del usuario.

2.2 Plataformas para computación paralela

En el desarrollo de aplicaciones y algoritmos que puedan aprovechar al máximo las capacidades es las GPU y la ejecución de procesamiento en paralelo, existen varias plataformas dedicadas específicamente a esta tarea. Dentro de las principales vamos a destacar las dos más extendidas. Por un lado, como plataforma propietaria desarrollada por la empresa Nvidia [11] para la programación de sus tarjetas gráficas, tenemos CUDA [10]. Por otro lado, y con otro enfoque distinto, tenemos el lenguaje OpenCL [6]. OpenCL es un lenguaje de código abierto desarrollado por el grupo Khronos [17], que tiene el objetivo de poder utilizarse en cualquier tipo de dispositivo de cálculo, incluso lanzar los algoritmos de GPU sobre CPU.

CUDA es una plataforma y modelo de programación de computación paralela desarrollado por Nvidia con propósito general sobre las GPU de su propia marca. Mediante CUDA, los desarrolladores pueden aprovechar toda la potencia de las GPU desarrolladas por Nvidia para acelerar programas aprovechando la computación paralela. Las aplicaciones CUDA se puede desarrollar empleando lenguajes de programación como C, C++ o Fortran. Además, numerosas aplicaciones ya emplean CUDA para mejorar el rendimiento en programas que requieren gran capacidad de cálculo como Matlab o aplicaciones desarrolladas por Adobe o Autodesk. Si bien dispone de un potente *toolkit*, así como amplia documentación y acceso a cursos desarrollados por la propia Nvidia, es propietario, y solo está disponible para tarjetas gráficas de la misma marca. Además, no permite su uso comercial de forma gratuita [10].

En contraposición, OpenCL es la plataforma más extendida de código libre. Está diseñada para poder ejecutarse sobre diferentes tipos de dispositivos, como CPU y GPU y actualmente casi todos los dispositivos GPU de todas las marcas lo soportan (incluidas las propias tarjetas gráficas de Nvidia y la mayoría de CPU del mercado). Soporta un amplio abanico de aplicaciones, y su amplia disponibilidad en la mayoría de los dispositivos le permite ser la herramienta por excelencia para explotar al máximo los entornos heterogéneos. Además, el hecho de que sea de código libre facilita que se disponga de una amplia gama de tutoriales, documentación, información pública, especificaciones abiertas, etc. Para desarrollo, soporta los lenguajes C y C++. La principal desventaja frente a CUDA es la falta de algoritmos altamente optimizados ya disponibles. Además, dado que

busca ser multiplataforma y poder ejecutarse prácticamente en cualquier dispositivo, está peor optimizada que lenguajes dedicados a unas plataformas concretas, como puede ser CUDA sobre las GPU de Nvidia. Sin embargo, dada su versatilidad, es la plataforma elegida para el desarrollo de este trabajo, dado que se adapta mejor al enfoque del mismo y las ventajas superan en gran medida a las ofrecidas por CUDA en el entorno en el que se enmarca este trabajo [6].

2.2.1 Arquitectura de OpenCL

La arquitectura de OpenCL se puede definir de acuerdo a la siguiente jerarquía de modelos, en función del enfoque estructural que observemos de la plataforma:

- **Modelo de plataforma:** como podemos observar, este modelo se ilustra en la Figura 2.2. Este modelo consiste en un *host*, que estaría conectado con 1 o más dispositivos OpenCL. Estos dispositivos OpenCL podríamos identificarlos como los dispositivos físicos de los que disponemos en nuestra máquina, como serían una CPU o un GPU. Dentro de cada dispositivo OpenCL encontramos las unidades de computación, que estarían formadas por los diferentes elementos de procesamiento. Es en estos elementos de procesamiento son donde se realizan las ejecuciones de los algoritmos de computación definidos. Estos algoritmos definidos para la ejecución en los diferentes elementos de procesamiento se denominan *kernels*. Es importante remarcar que este modelo muestra la visión de OpenCL del hardware que hay por debajo. La relación entre los diferentes elementos del modelo y el hardware real que soporta OpenCL dependerá de cada dispositivo, y será OpenCL quién lo defina buscando la máxima eficiencia y permitiendo una gran flexibilidad. Este nivel de abstracción gestionado por OpenCL es lo que permite que el mismo código pueda ser ejecutado por dispositivos muy diferentes, ya que será OpenCL el que en función del hardware físico defina los niveles lógicos que ejecutarán la aplicación.

  Uno de los retos que supone este modelo de plataforma de OpenCL es conseguir que los diferentes algoritmos complejos que se van a implementar en este trabajo puedan dividirse en pequeñas tareas más sencillas que puedan ejecutarse de forma simultánea. Aunque se detallará para cada algoritmo en el apartado de implementación en el capítulo 5, de forma general el objetivo será que cada *kernel* calcule un

pixel de la imagen de salida en función de la información de entrada, aunque esto dependerá del algoritmo en cuestión.

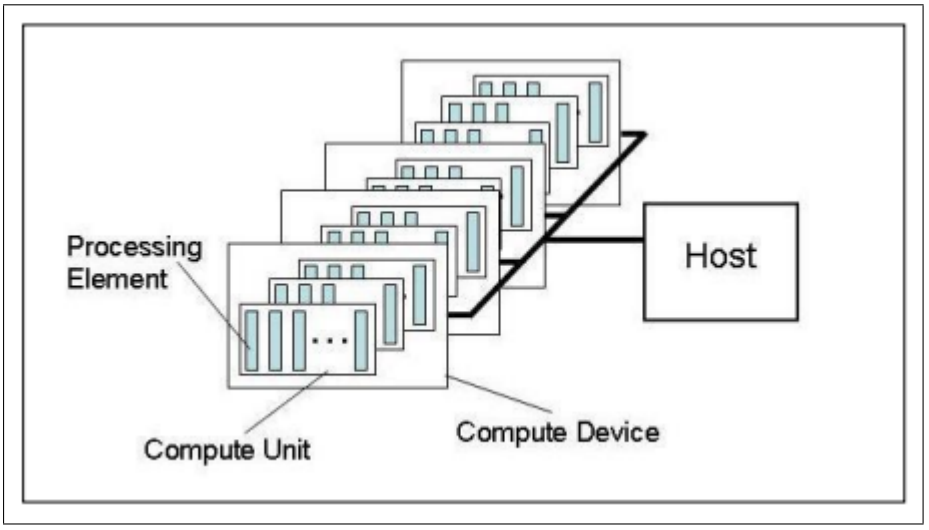

Figura 2.2: Modelo de plataforma en OpenCL [6].

- **Modelo de ejecución:** en OpenCL, este modelo se define en dos niveles diferentes de ejecución. El primer nivel de ejecución viene dado por el programa principal que será ejecutado por el *host* y que se encargará de definir el entorno, dispositivos y recursos a emplear, entrada/salida de datos, etc. El programa principal ejecutado por el *host* inicialmente creará y gestionará el contexto y definirá los dispositivos de trabajo empleando funciones de la API de OpenCL. Tras esto, se encargará de inicializar los datos y crear los buffers de memoria de entrada y salida para que el dispositivo pueda obtener los datos que debe procesar. Finalmente, obtendrá el código del *kernel*, y se encargará de encolar las ejecuciones de este algoritmo, esperando a que la salida de todo el procesado esté disponible en la memoria para su gestión (almacenamiento en ficheros, salida por pantalla, etc).

  El segundo nivel de ejecución está formado por los *kernels*, que serán ejecutados por 1 o más dispositivos OpenCL y contienen los algoritmos de cálculo propiamente dichos. Cada unidad que realiza una ejecución de un *kernel* sobre los datos indicados se denomina *work-item*. Estos se agrupan en grupos de ejecución denominados *work-groups*. Todos los *work-items* ejecutarán el mismo algoritmo, pero cada uno sobre un conjunto de datos diferentes dispuestos por el *host*. De la misma forma, las diferentes salidas serán almacenadas en la memoria accesible por el *host*, según el programa principal haya dispuesto. Esta forma de ejecución de múltiples ejecuciones

paralelas se denomina computación vectorial, y la forma habitual de disponer los datos de entrada y salida es que cada *work-item* acceda a posiciones consecutivas en la memoria, almacenando la salida de igual manera ordenada en memoria.

Como se detallará en el capítulo 3, la principal virtud de OpenCLIPER es añadir una capa de abstracción simplificando todas las tareas de este primer nivel del modelo de ejecución, que está relacionado más con la configuración de la plataforma y las tareas del *host* que con los algoritmos de cálculo en sí mismos.

- **Modelo de memoria:** este describe la estructura y comportamiento de la memoria tal y como la plataforma OpenCL opera sobre ella. Este modelo permite conocer con cómo la memoria empleada es visible por la unidades de ejecución y sus relaciones, de forma que puedan disponerse los datos de la forma más adecuada y sean accesibles en todos momento por las unidades de ejecución. Podemos dividir esta memoria en 4 regiones principales, relacionadas como se muestra en la figura 2.3 [6]:

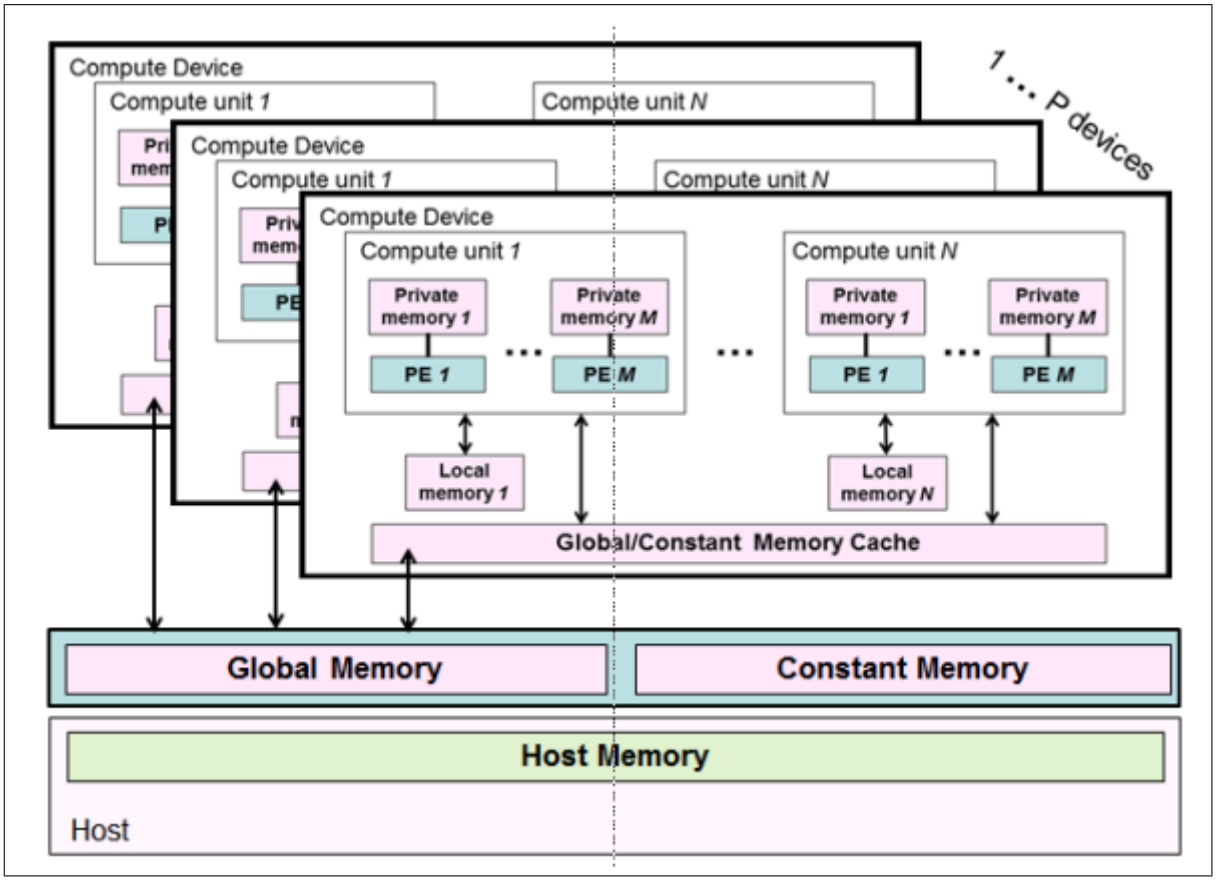

Figura 2.3: Modelo de memoria en OpenCL [6].

- **Memoria global:** esta región de memoria permite lectura/escritura por parte

de cualquier *work-item* definido dentro de cualquier *work-group* para cualquier dispositivo de dentro de cualquier contexto. Esta memoria puede ser cacheada dependiendo de las capacidades del dispositivo. La principal desventaja de la misma es que es la de acceso más lento.

- **Memoria constante:** es una región de la memoria global que permanece inalterada durante la ejecución de todas las instancias de los *kernels*. Resulta muy útil para almacenar información ya inicializada que no va a ser modificada (solo lectura). Tiene la ventaja de que es más rápida que la memoria global, y es muy útil para guardar los datos de partida.
- **Memoria local:** región de memoria exclusiva para cada *work-group*. Tiene la ventaja de ser notablemente más rápida que las dos anteriores (memoria global y memoria constante), y sirve para compartir la información dentro de un único *work-group*. El problema principal de la misma es que es muy limitada, y no es útil para guardar grandes cantidades de información. Suele emplearse para variables comunes a todos los *work-item*s de ejecución dentro de cada *work-group*.
- **Memoria privada:** región de memoria exclusiva para cada *work-item*. Muy pequeña pero en cambio es la más rápida de todas. Suele emplearse para guardar variables temporales de cada item de trabajo como acumuladores para sumatorios, o variables para control de bucles.

Los diferentes niveles de memoria en los que se estructura OpenCL son una de las claves para poder optimizar los algoritmos en esta plataforma. Si bien los niveles más altos, como la memoria global o constante son muy grandes, el acceso a los mismos por parte de los *work-item* es más lento, por lo que en función del algoritmo y los datos puede ser más interesante emplear los niveles más bajos. Sin embargo, esto también tiene un coste, ya que tanto la memoria local y privada no solo son más pequeñas, sino que además la visibilidad de la información que contienen es limitada a un *work-group* para memoria local o *work-item* en el caso de memoria privada. Además, el coste computacional de copiar la información de los niveles altos a los niveles bajos puede suponer mayor carga que el hecho de trabajar sobre memoria de nivel alto directamente, en función del algoritmo y los accesos a memoria necesarios.

¿Cómo debe cada algoritmo gestionar estos niveles de memoria? ¿En qué casos es

rentable trabajar sobre niveles de memoria más bajos a pesar del coste de copiar la información desde las memorias de nivel alto? Cuándo la información en las memorias de nivel bajo no quepa completamente, ¿es más rentable estructurar los algoritmos de forma que operen sobre subconjuntos de la información completa de forma iterativa en memorias de bajo nivel, o qué trabajen con toda la información en paralelo pero en memorias más lentas? Todas estas preguntas son fundamentales con el objetivo de optimizar los algoritmos, y tienen diferentes respuestas según el algoritmo que se quiera implementar y optimizar. Para cada algoritmo, detallaremos todas estas cuestiones y las decisiones de diseño en el capitulo 5.

Capítulo 3

El framework OpenCLIPER

OpenCLIPER es un framework basado en OpenCL pensado para procesado y reconstrucción de imagen médica. Su objetivo es simplificar todas las complejidades que implica OpenCL en cuanto a gestión de plataformas, memoria, etc., para permitir que los desarrolladores se centren en los *kernels* y sus algoritmos. Esta herramienta facilita y automatiza tareas sobre OpenCL tales como descubrimiento e inicialización de dispositivos, transferencia de datos y gestión de memoria, carga de *kernels* o compilado y gestión de errores [7]. Desarrollado en C++, OpenCLIPER está pensado para ofrecer una serie de servicios principales para los desarrolladores. Estos servicios son los siguientes:

- **Gestión de dispositivos de computación:** en OpenCL, toda la inicialización de dispositivos y su descubrimiento debe realizarse manualmente. Este es el precio que pagar de soportar tantos dispositivos diferentes, y de ofrecer esa amplia versatilidad. En este apartado, OpenCLIPER proporciona la facilidad de poder seleccionar e inicializar el dispositivo más adecuado en función de una serie de criterios (tipo de dispositivo, marca, versión de OpenCL y más) con una sola llamada. Además, también de acuerdo a estos criterios, el framework selecciona el de mayor rendimiento a priori en función de sus características. OpenCLIPER está pensado para trabajar con la versión 1.2 de OpenCL, ya que es la más soportada por la amplia mayoría de dispositivos actualmente.

- **Almacenamiento y manipulación de datos:** otro problema general cuando trabajamos con dispositivos dedicados es el sincronismo entre todos los diferentes mapas de memoria que encontramos entre el host y los dispositivos dedicados. Trabajar con conceptos como el contexto y la cola de comandos en OpenCL puede resultar complejo, sobre todo cuando se busca optimizar las transferencia de información. OpenCLIPER simplifica todo este trabajo en una sola llamada. Con esta se configura y

transfiere los datos desde el host al dispositivo de procesamiento, con la información preparada para su acceso y manipulación.

Para que el usuario pueda mantener el control de esta información en todo momento, cada conjunto de datos se almacena siempre de forma lineal y continua dentro de la memoria del dispositivo, incluso cuando se trata de un conjunto de datos muy heterogéneo. El enfoque es el procesamiento de la información por bloques, de forma que cada *kernel* conoce la posición inicial de cada uno de los bloques de información, pudiendo acceder a esta correctamente. El desarrollador no necesita conocer esta distribución de memoria, ya que OpenCLIPER se encargará de transferirla y ordenarla para que sea accesible por los *kernels*. Además, puede cargar automáticamente numerosos formatos de ficheros de imagen y datos como PNG, TIFF, MAT, CFL o ficheros en RAW.

- **Mantenimiento de *kernels*:** Por último, OpenCLIPER facilita en gran medida la carga de *kernels*, compilado y gestión de errores de forma sencilla. Este framework permite toda la gestión de los *kernels* con una sola llamada, incluso si hay diferentes *kernels* en un fichero, estos están disponibles con su nombre. Para esta tarea, y para simplificar el lanzamiento de los mismos, OpenCLIPER introduce el concepto de *Process*. Estos son clases personalizables que heredan de una interfaz estándar definida, y todos funcionan de la misma forma: inicializar los valores de entrada y salida, configurar los parámetros necesarios y ejecución. En caso de que se requiera cierto preprocesado de los datos antes de lanzar un *kernel*, los procesos disponen de una fase de inicialización, que permite separar este preprocesado y llevarlo a cabo de forma controlada.

3.1 Arquitectura de OpenCLIPER

La estructura del framework está formada por un conjunto de clases que se relacionan entre sí. Cada una cumple una funcionalidad, desde la inicialización del dispositivo a la ejecución del *kernel* o el mantenimiento de la información. El diagrama de clases lo podemos ver en la figura 3.1. A continuación, vamos a desarrollar una breve descripción de las tareas que realiza cada una:

- ***CLapp:*** es la clase principal de OpenCLIPER. Actúa como una interfaz con OpenCL

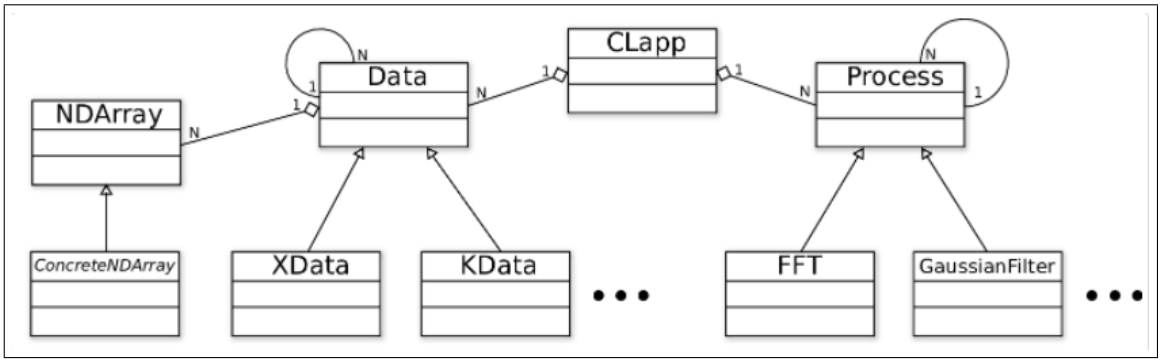

Figura 3.1: Clases del framework OpenCLIPER [8].

y se encarga de almacenar y gestionar toda la información relativa a las plataformas y dispositivos. Además, contiene la lista de objetos para ser procesados, y gestiona toda la memoria, tratando de minimizar las transferencias y realizando los mapeados memoria-buffer buscando el máximo rendimiento.

- ***Data:*** clase abstracta de la cual derivan los diferentes objetos que contienen imágenes o volúmenes de datos n-dimensionales. Cada objeto Data consiste en un grupo de uno o más objetos *NDArray* que pueden ser de diferentes dimensiones. En general, cada adquisición contiene datos heterogéneos que son almacenados en un solo objeto de este tipo. Mantiene el alineamiento de memoria, lo que permite a los *kernels* asumir constantes y predecibles cadenas en cada dispositivo. De forma general, OpenCLIPER provee dos especializaciones de esta clase: *XData*, pensada para datos con interpretación física directa en memoria; y *KData*, pensada para datos estructurados en el llamado espacio-k, empleado en determinadas modalidades de imagen médica.

- ***NDArray:*** representa una señal, imagen o volumen de datos estructurados en N dimensiones. Se define como clase abstracta para que pueda ser empleado por cualquier tipo de dato específico en función de la máquina. La ventaja de esto es que nuevas especializaciones de datos no tienen que lidiar con un profundo análisis de cómo se almacenan los mismos, de forma que el acoplamiento entre clases se mantiene al mínimo.

- ***Process:*** es la interfaz para los algoritmos encargados del procesamiento de datos. Es una clase abstracta que los desarrolladores deben heredar para implementar sus propios procesos. Su propósito es estandarizar toda la fase de preparación necesaria

para ejecutar los algoritmos, de forma que no se necesite un conocimiento profundo de OpenCL para poder empezar a trabajar con ello. Los procesos además se pueden llamar en cascada sin que esto implique una penalización en rendimiento gracias a su diseño.

3.2 Ejemplo de integración sobre OpenCLIPER

Para poder visualizar realmente las ventajas que nos ofrece el framework, vamos a ver a continuación un ejemplo completo empleando el mismo. Este va a consistir en un algoritmo que calcula la matriz transpuesta sobre una matriz de entrada. El objetivo es entender como se construyen la aplicaciones empleando este framework, de forma que facilite la comprensión de los algoritmos desarrollados [15].

```
#include<OpenCLIPER/kernels/hostKernelFunctions.h>

kernel void transpose(global complexType* input, global complexType* output) {

    uint x = get_global_id(0);
    uint y = get_global_id(1);


    uint rowStrideIn = getSpatialDimStride(input, 1, 0);
    uint rowStrideOut = getSpatialDimStride(output, 1, 0);


    output[rowStrideOut * y + x] = input[rowStrideIn * x + y];
}
```

Figura 3.2: Kernel OpenCL que niega una matriz [15].

Como podemos ver en la figura 3.2, este es un *kernel* sencillo que transpone una matriz. Para ello, intercambia las filas y columnas. Comienza obteniendo el identificador único de cada *work-item* con la función *get_global_id* proporcionada por OpenCL. Las dos llamadas con el argumento 0 y 1 se refieren a las dos dimensiones que definen el conjunto de *work-items*, cada uno operando sobre el elemento de la matriz de datos de entrada con los mismos índices. Como detalle, podemos ver que solo necesitamos como argumentos de entrada las matrices de datos y de salida, ya que las dimensiones de las mismas para poder recorrerlas nos las proporciona una llamada a la función *getSpatialDimStride* definida por OpenCLIPER, que lee el tamaño de los datos sobre los objetos directamente. Esto nos simplifica los argumentos de entrada del *kernel*, lo que implica también simplificar su

llamada. Esta dimensión se utiliza para poder acceder en memoria al elemento correcto de la matriz de entrada y de salida, ya que debido a que la memoria es unidimensional, la forma de recorrer la matriz es conociendo el tamaño de las filas.

En la última línea se ve el detalle de como calcular la posición del elemento, conociendo el tamaño de la fila, y el indice bidimensional del mismo (que coincide con el del *work-item* encargado de procesarlo). Este algoritmo consiste en intercambiar los identificadores, de forma que el que representa la fila en la entrada represente la columna en la salida y viceversa. Esto es así porque la forma de disponer las matrices bidimensionales en la memoria unidimensional es colocar las filas una a continuación de la anterior. Por tanto acceder a un elemento de forma general consiste en la operación $id_{fila} \cdot tamaño_{fila} + id_{columna}$. De igual forma, la estructura de la memoria y como paralelizar siempre que consideremos cada *work-item* como representación de un elemento de una matriz se explicará con mucho más detalle en el capítulo 5, donde se detallan los algoritmos implementados en este trabajo.

```
#include <OpenCLIPER/hostKernelFunctions.hpp>
#include <OpenCLIPER/CLapp.hpp>
#include <Transpose.hpp>

namespace OpenCLIPER {

void Transpose::init() {
    kernel=getApp()->getKernel("transpose");
}

void Transpose::launch() {
    kernel.setArg(0,*(getInput()->getDeviceBuffer()));
    kernel.setArg(1,*(getOutput()->getDeviceBuffer()));

    cl::NDRange globalSize = cl::NDRange(getOutput()->getSpatialDimSize(0,0),
                                         getOutput()->getSpatialDimSize(1,0));

    queue.enqueueNDRangeKernel(kernel, cl::NullRange, globalSize, cl::NullRange, NULL, NULL);
}

}
```

Figura 3.3: Proceso que gestiona la ejecución del *kernel* [15].

Revisando la figura 3.3, podemos ver que tanto la inicialización como la ejecución del *kernel* es muy sencilla. Por un lado, en la parte de inicialización del proceso, se inicializa la variable de clase *kernel*, donde con una llamada a la *app* (que nos devuelve el objeto *CLapp*) se le solicita que nos devuelva el *kernel* llamado *traspose*. Simplemente con este paso, ya tenemos cargado en *kernel* en la variable.

En la segunda parte, en la función de lanzamiento, en primera instancia se configuran

los argumentos que va a emplearse de entrada y salida. Para ello, se llama a las funciones de *CLapp* que nos traen los buffers de memoria donde se almacena la información de los respectivos argumentos. En el siguiente paso, se define el grupo de items de trabajo necesario para procesar la petición, que en este caso coincide con el rango de nuestra matriz de entrada, de forma que cada item procese un elemento de dicha matriz. Finalmente, se encola el *kernel* para su ejecución directamente sobre la cola ya creada y gestionada por *CLapp*.

```
#ifndef TRANSPOSE_HPP
#define TRANSPOSE_HPP

#include <OpenCLIPER/Process.hpp>

namespace OpenCLIPER {

class Transpose: public Process {
    public:
        using Process::Process;

        void init();
        void launch();
};

}
#endif
```

Figura 3.4: Cabeceras del proceso que gestionan la ejecución del *kernel* [15].

Los ficheros de cabecera son sencillos, como se puede ver en la figura 3.4. Ya que todos los métodos se heredan de la clase padre, simplemente se definen los dos métodos empleados, que en este caso no necesitan argumentos ni de entrada ni devuelven nada.

```
#include <OpenCLIPER/CLapp.hpp>
#include <OpenCLIPER/XData.hpp>
#include <LPISupport/ProgramConfig.hpp>
#include "Transpose.hpp"

using namespace OpenCLIPER;
int main() {
    std::shared_ptr<CLapp> pCLapp;

    try {
        pCLapp = CLapp::create();
        pCLapp->loadKernels(LPISupport::ProgramConfig::exeDir() + "/transpose.cl");


        auto pData = new std::vector<complexType>({0, 1, 2, 3, 4, 5, 6, 7, 8, 9});
        auto pIn = std::make_shared<XData>(pCLapp, 5, 2, pData);
        pIn->show();

          auto pOut = std::make_shared<XData>(pCLapp, 2, 5, TYPEID_COMPLEX);

          auto transpose = std::make_shared<Transpose>(pCLapp, pIn, pOut);
          transpose->init();
          transpose->launch();

          pOut->show();
      }
      catch(cl::BuildError& e) {
          CLapp::dumpBuildError(e);
      }
      catch(CLError& e) {
          std::cerr << CLapp::getOpenCLErrorInfoStr(e, "main");
      }
      catch(std::exception& e) {
          LPISupport::Utils::showExceptionInfo(e, "main");
      }
  }
```

Figura 3.5: Main del programa que transpone una matriz [15].

Finalmente, como podemos ver en la figura 3.5, el programa que ejecuta el *kernel* para obtener el resultado de la operación de transponer una matriz es sencillo. En la primera parte, creamos nuestra *CLapp*, y el fichero que contiene los *kernels* que vamos a emplear, en este caso *transpose.cl*. Tras esto, creamos nuestra matriz de datos, y lo almacenamos en una variable de tipo *XData*. Para ello, le damos nuestra variable que contiene nuestra *app*, las dimensiones y los valores de entrada definidos en forma de vector. En el siguiente paso, creamos otra instancia de la clase *XData*, que mantendrá nuestra matriz de salida, con las dimensiones adecuadas (en este caso intercambiadas ya que estamos calculando la matriz transpuesta).

En el siguiente paso, creamos una instancia de nuestra clase *Transpose*, que es el proceso

encargado de la inicialización y ejecución del *kernel*, así como de obtener los resultados en nuestra variable de salida. En este caso, le pasamos como variables de entrada nuestra *CLapp*, nuestro objeto *pIn* con los datos de entrada y nuestro objeto *pOut* donde deberá almacenar la salida. A continuación, llamamos a las funciones de inicialización y tras esto a la de lanzamiento, ya que hemos programado para que realicen todas las operaciones necesarias. Por último, mostramos la salida por pantalla con la función *show*, aunque podríamos tambien emplear la función *save* para guardarla en un fichero con diferentes formatos, por ejemplo MAT (Matlab file).

Capítulo 4

Algoritmos de computación

En esta sección vamos a detallar la base matemática sobre la que se sustenta la construcción de los algoritmos diseñados en este proyecto. Para ello, se partirá de unas explicaciones sencillas sobre qué es una matriz y como se realizan las operaciones que vamos a implementar, así como las condiciones necesarias que deben cumplir las matrices de datos para que estas operaciones se puedan llevar a cabo.

Partiremos de la definición de matriz. Entendemos una matriz como un arreglo rectangular de números, denominados elementos de la matriz, dispuestos en filas y columnas. El tamaño de una matriz se describe a partir de los números de filas y columnas en las que están dispuestos sus elementos, definido como $(M \cdot N)$, donde M es el número de filas y N es el número de columnas. Un matriz tiene la forma que podemos ver en la figura 4.1.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Figura 4.1: Forma básica de una matriz definida [13].

Vista su forma, una matriz puede entenderse como un vector de dos dimensiones, o de forma extendida como un vector de vectores. Comprender este punto es fundamental, ya que para aplicar los algoritmos de cálculo sobre GPU, es fundamental entender que la principal forma de procesamiento de estos dispositivos consiste en la ejecución de un algoritmo sobre datos vectoriales, aplicando los cálculos al mismo tiempo cada una de las componentes de un vector de entrada. De esta forma, se procesan al mismo tiempo todos

los elementos del vector, lo que por extensión puede ser aplicado también a una matriz [13].

4.1 Norma-N de un vector

Observando los algoritmos implementados en OpenCLIPER para operaciones vectoriales, uno de los primeros algoritmos de interés para los que no existe implementación es la norma-N. Esta nos permite definir la distancia en un espacio N-dimensional. De forma general, se puede definir la norma-p en un espacio N-dimensional según la fórmula 4.1 [2].

$$||\vec{x}||_p = \sqrt[p]{\sum_{k=1}^{N} |x_i|^p} \tag{4.1}$$

Vista la fórmula de la norma-N, se puede definir un algoritmo computacional que siga los siguientes pasos para el cálculo de la norma-p de una vector N-dimensional:

1. Calcular la potencia para cada elemento del vector. Se toma el valor absoluto del mismo y se eleva a la potencia p.
2. Se van almacenando todos los valores en un acumulador común, donde se va calculando el sumatorio de todos los elementos elevados a p.
3. Una vez todos los elementos se han computado y el sumatorio está calculado, calculamos la raíz, y guardar el resultado en la salida.

Para extender el cálculo de la norma-N a un vector de números complejos, simplemente se considera la operación sobre el módulo del número complejo, de forma que los elementos dentro del sumatorio que son elevados a p serían los módulos de los números complejos que conforman el vector. El módulo de un número complejo se calcula siguiendo la siguiente fórmula 4.2 [14], donde x es la parte real e y es la parte imaginaria de z:

$$|z| = \sqrt{x^2 + y^2} \tag{4.2}$$

Una vez vistas las fórmulas definidas para el cálculo de la norma-N tanto para un vector

de elementos reales como para un vector de elementos complejos, resulta fundamental reflexionar sobre los siguientes puntos de cara a la implementación del algoritmo:

- **Cómo paralelizar las operaciones:** parece interesante en este caso que la paralelización se base en que cada elemento del vector sea procesado por un *work-item* diferente, que se encargue del cálculo de la potencia y el módulo (en caso de número complejo) y lo agregue al acumulador.
- **Memoria para almacenar el acumulador:** en este punto, el acumulador es un espacio de memoria común que requiere ser actualizado por todos los *work-item*, por lo que sería interesante en términos de optimización usar la memoria de más bajo nivel visible para todos ellos debido al alto número de accesos. Además, también habrá que realizar las actualizaciones de forma atómica, ya que es fundamental evitar que los *work-items* se sobrescriban entre ellos cuando actualicen este espacio de memoria común.
- **Sincronización previa al cálculo de la raíz :** otro aspecto relevante es poder identificar cuando todos los *work-item* han finalizado sus cálculos y el acumulador contiene el valor final para poder realizar la raíz correctamente.
- **Cómo realizamos la raíz final :** si bien todos los *work-items* operan sobre cada elemento del vector, la raíz y el almacenamiento del resultado final en la variable de salida solo debería llevarse a cabo una vez. Se requerirá una revisión de forma que aseguremos que este cálculo solo se hace una vez y en el momento en el que el acumulador contenga el valor final del sumatorio

Todos estos puntos necesarios serán revisados durante el diseño del algoritmo en el capítulo 5. En este apartado, se detallará la toma de decisiones para solventar estos retos de implementación, buscando optimizar el algoritmo con el objetivo de reducir lo máximo posible el tiempo de ejecución del mismo.

4.2 Producto de matrices

Dentro de las operaciones con matrices, una de las más usuales es el producto de dos matrices. Este producto de matrices puede definirse como se muestra en la figura 4.2 [13].

Definición Si A es una matriz de $m \times n$ y B es una matriz de $n \times r$, entonces el ***producto*** $C = AB$ es una matriz de $m \times r$. La entrada (i, j) del producto se calcula del modo siguiente:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$$

Figura 4.2: Definición de producto de dos matrices [13].

De esta forma, podemos extender que el producto de dos matrices consiste en el sumatorio del producto de los elementos de cada vector fila del multiplicando, por cada vector columna del multiplicador. En la matriz de salida, la posición de cada elemento vendría entonces determinada por la fila del multiplicando y la columna del multiplicador sobre las cuales se ha operado. Así, si quisiéramos calcular el elemento $c_{2,1}$ del producto de dos matrices, consistiría en multiplicar la fila 2 del multiplicando por la columna 1 del multiplicador elemento a elemento, y realizar el sumatorio. Otra forma de escribirlo es la que podemos ver en la ecuación 4.3.

$$c_{i,j} = \sum_{1}^{n} a_{i,n}b_{n,j} \tag{4.3}$$

Una vez comprendido cómo se lleva a cabo la multiplicación de matrices, es muy importante exponer cuándo esta operación puede llevarse a cabo, partiendo de la propia definición de la multiplicación. Parece claro en este caso que las matrices no tienen por qué ser iguales para poder realizar el producto entre ellas, pero debido a que estamos multiplicando los elementos del vector fila de A por los elementos del vector columna de B, estos vectores deben tener las mismas dimensiones. Por tanto el número de columnas de A (número de elementos del vector fila) y el número de filas de B (elementos del vector columna) deben de ser iguales. Por extensión, las dimensiones de nuestra matriz resultado serán el mismo número de filas de A y el mismo número de columnas de B.

Definido el algoritmo y teniendo en cuenta el detalle y arquitectura de OpenCL y OpenCLIPER, parece interesante la paralelización en función de la matriz de salida buscando optimizar el tiempo de procesamiento. Es decir, que cada *work-item* calcule un elemento de la matriz de salida, recorriendo la fila y columna correspondiente y realizando el sumatorio. Este planteamiento provoca que haya elementos del multiplicando y multiplicador

que tengan que ser accedidos por diferentes *work-items* al mismo tiempo, lo cual puede aumentar la latencia por accesos a zonas comunes de memoria, lo que puede reducir la eficiencia. Dado este problema, resultará interesante evaluar si puede ser rentable copiar en memoria local cierta información o no. También estudiar la mejor forma de disponer la información en memoria buscando la mayor eficiencia.

4.2.1 Aplicación del producto sobre matrices de números complejos

Para poder aplicar el algoritmo de multiplicación de matrices a números complejos, primero debemos comenzar comprendiendo detalladamente como se realiza el producto de dos números complejos cualesquiera. De forma general, podemos definir un número complejo como un numero con la forma $z = x + yi$ donde x e y son números reales e i es la unidad imaginaria, igual a $\sqrt{-1}$. Se denomina entonces a x como la parte real del número complejo z ($Re\{z\}$) y a y como la parte imaginaria de z ($Im\{z\}$)[21]. Hay otras formas de representar un número complejo, como pueden ser en forma de fasor o coordenadas polares, pero no se entrará en más detalle de estas representaciones debido a que no tienen interés en la aplicación desarrollada.

De esta forma definida, se puede entender el producto de dos números complejos como el producto de un binomio. Este producto puede definirse de la siguiente forma. Sean $z = a + bi$ y $w = c + di$, se define su producto mediante la ecuación 4.4.

$$z \cdot w = (a + bi)(c + di) = ac + adi + bci + bdi^2 = (ac - bd) + (ad + bc)i \tag{4.4}$$

Una vez definido el producto entre números complejos, la multiplicación de matrices de números complejos se reduce a la aplicación del algoritmo expuesto en el apartado 4.2, pero cuando multipliquemos dos elementos de la matriz se llevará a cabo este producto binomial, teniendo en cuenta que dentro de las matrices el número complejo estará representado siempre de forma binomial [14].

Buscando optimizar el algoritmo entran en juego los mismos factores que para el caso de números reales. Sin embargo, es importante destacar que realizar el producto de números complejos implica más operaciones de cálculo que en el caso de números reales, por lo que

el coste computacional de multiplicar dos números complejos es más alto. Esto a nivel de optimización puede implicar que sea menos necesario copiar los datos a memorias de bajo nivel, ya que aunque exista solapamiento de *work-items* en las lecturas de memoria, el hecho de que el coste computacional de las operaciones sea mayor puede reducir el impacto do los accesos a memoria, y por tanto la necesidad de copiar los datos a memorias más rápidas.

4.2.2 Potencia de una matriz

A partir de las características de la multiplicación de matrices, desarrollar un algoritmo que eleve una matriz a una potencia natural consiste en multiplicaciones sucesivas de dicha matriz. Sin embargo, se deben tener en cuenta ciertas consideraciones en cuanto al orden de las operaciones y a las dimensiones de las matrices ya que para obtener el resultado correcto de la operación esta debe realizarse de forma ordenada.

- **Dimensiones de la matriz:** dado que elevar una matriz a una potencia consiste en multiplicaciones sucesivas de una matriz consigo misma, y según las propiedades vistas en el apartado anterior, el número de filas y de columnas de la matriz deben ser iguales. Esto implica que, por tanto, para elevar una matriz a una potencia esta debe ser una matriz cuadrada. Esto deberá estar controlado para que solo se pueda ejecutar el algoritmo sobre matrices cuadradas.
- **Orden de las operaciones:** en este caso, vamos a multiplicar N veces una matriz, por lo que el producto en cada paso debe ser el resultado de la operación anterior por la matriz inicial. Esto se puede ver en la ecuación 4.5:

$$A^n = A \cdot A \cdot A... \ N \ veces = A^2 \cdot A \cdot A... \ (N-1) \ veces \tag{4.5}$$

  Con este ejemplo se puede observar que la matriz donde se van acumulando las sucesivas multiplicaciones siempre debe cumplir el rol de multiplicando. Si bien esto puede parecer obvio, es muy importante tenerlo en cuenta para que el algoritmo lea la memoria correctamente y se hagan las operaciones en un orden correcto para no cometer errores.

Con estas consideraciones, y teniendo en cuenta que ya existirá implementado un algo-

ritmo que realice el producto de matrices previamente, parece interesante que sea OpenCLIPER el encargado de gestionar las llamadas al mismo para hacer estas multiplicaciones sucesivas. Gracias a los procesos que nos ofrece este framework, se podrá implementar un flujo transparente al usuario que se encargue de esta gestión ordenada de la memoria en la que se hacen las operaciones en orden. Con este planteamiento, surgen otra serie de problemas a tener en cuenta como es la necesidad de una matriz auxiliar para ir almacenando el resultado parcial de los productos, así como minimizar el número de operaciones en memoria. El objetivo del flujo en la implementación de este algoritmo será el evitar tener que realizar copias entre regiones de memoria, de forma que minimicemos las operaciones a realizar los productos, en base a ir modificando que matrices cumplen los roles de multiplicando, multiplicador y resultado en cada interacción. El minimizar las copias y los accesos a memoria nos permitirá optimizar el algoritmo, ya que se podrá reducir al máximo la carga computacional, siendo esta igual a la carga de realizar N multiplicaciones por separado sin ninguna otra operación extra.

4.3 Producto escalar de vectores

El producto escalar de vectores es una operación entre dos vectores que se define como el producto de las magnitudes de ambos vectores y el coseno del ángulo entre ellos, como se ve en la fórmula 4.6 [16]:

$$A \cdot B = |A||B| \cos\theta,\ 0 < \theta < \pi \tag{4.6}$$

Pero para nuestro algoritmo, dado un vector en función de los componentes, calcular sus módulos y el ángulo que forman para calcular su producto escalar resulta muy lento y es una forma poco eficiente para realizar esta operación. Por tanto, se utilizará la forma de cálculo del producto escalar a partir de sus componentes, que se detalla en la fórmula 4.7 [16]:

$$A \cdot B = \sum_{k=1}^{n} a_k b_k \tag{4.7}$$

Como podemos ver, el algoritmo consistirá en sumar los productos de las componentes

de los vectores. Desde este enfoque, se puede entender el producto escalar de vectores como un caso particular del producto de matrices, donde el primer vector cumpla el rol de vector columna y el segundo vector sea el vector fila. Sería el equivalente a realizar el producto algebraico del primer vector transpuesto con el segundo vector. Esto cumpliría la regla de las dimensiones del producto algebraico de matrices ya que $A(1 \cdot N) \cdot B(N \cdot 1) = C(1 \cdot 1)$, y sería la operación equivalente. Es importante destacar, como condición obligatoria que deben cumplir los vectores de entrada, que para poder realizar su producto escalar deben estar referenciados en la misma dimensión (mismo número de elementos).

Una vez estudiada la operación y dado que es un caso particular del producto matricial, es interesante evaluar si merece la pena diseñar un algoritmo específico para esta operación, o si se puede tratar al igual la potencia de una matriz, como un caso que emplee el algoritmo de producto algebraico de matrices gestionando los datos de entrada adecuadamente. También estudiar que estrategia de paralelización será la más interesante, así como que nivel de memoria puede resultar más eficiente en este caso. Esto nos permitirá optimizar este algoritmo para que sea lo más eficiente posible.

4.3.1 Producto de vectores de números complejos

Para el caso en el que el producto escalar sea calculado entre dos vectores cuyos elementos son números complejos, se aplica la misma fórmula que define el producto escalar en 4.6. Pero dado que nuestros vectores vienen dados en componentes, la forma más sencilla es llevar a cabo la operación de la misma forma que en el caso real, haciendo el cálculo en base a las componentes. Por ello, se empleará la siguiente fórmula 4.8 [9].

$$Z \cdot W = \sum_{k=1}^{n} z_k \overline{w_k} \tag{4.8}$$

Como se puede observar, la diferencia entre el cálculo para vectores con componentes reales y vectores con componentes complejas es que se debe tomar el complejo conjugado del segundo vector sobre el que se está calculando el producto escalar.

A nivel operacional, de cara al algoritmo que se va a implementar, esto no supone ningún problema, ya que la operación de cada producto de pares en cada *work-item* es idéntica a la empleada en la multiplicación de matrices de números complejos, pero tenien-

do en cuenta el cambio de signo necesario. El resto de consideraciones para llevar a cabo la implementación son las mismas a las detalladas para vectores de componentes reales.

4.4 ALGORITMO DE CONVOLUCIÓN DE MATRICES

La convolución es una operación muy empleada en procesado de imágenes para aplicación de filtros y consiste en un procesado entre nuestra imagen, como matriz de datos de entrada, y una matriz de referencia. A la matriz de referencia sobre la que se convoluciona la imagen se denomina *kernel* de convolución[1]. En este punto, se debe entender una imagen como una matriz de datos bidimensional, donde cada elemento de la matriz contiene la información de intensidad de un pixel. Típicamente, esta intensidad se define con un byte, por lo que puede tomar valores entre 0 y 255. En caso de imágenes a color, cada imagen viene definida por tres matrices, cada una para los colores básicos rojo, verde y azul.

Una vez comprendida la matriz de entrada de referencia en este proceso y asumiendo que es una imagen en blanco y negro (lo que implica que se define con una sola matriz), podemos exponer la definición de la convolución entre esta matriz de datos, D, y un *kernel* de convolución, K.

Dada una matriz $D_{m \cdot n}$ y una matriz $K_{k \cdot k}$ denominado *kernel* de convolución, $K_{k \cdot k}$ con $k = 2N + 1 <= m, n$; se define la convolución de matrices D y K como una nueva matriz $R = D * K$ cuyos elementos se definen a partir de la expresión 4.9[12].

$$r_{i,j} = \frac{1}{c} \sum_{r=1}^{2N+1} \sum_{s=1}^{2N+1} d_{i-N+r-1, j-N+s-1} c_{r,s} \; donde \; c = \sum_{i,j=1}^{2N+1} k_{i,j} \tag{4.9}$$

De forma visual, la convolución consiste en, sobre la imagen de datos, superponer en cada pixel el *kernel* de convolución, y hacer el sumatorio de todas las parejas de valores que quedan superpuestas. De esta forma, lo que implica a nivel teórico es que cada pixel mezcla su información con los pixeles colindantes en ciertas proporciones definidas en el *kernel* de convolución, de forma que se pueden conseguir efectos como desplazamientos, enfoques, desenfoques, *blur* y muchos otros. Se muestra de forma gráfica en la figura 4.3.

[1] No confundir este *kernel* de convolución, que consiste en una matriz de datos con la que se convoluciona una matriz imagen, con el *kernel* de código ejecutado por el dispositivo OpenCL mencionado hasta ahora.

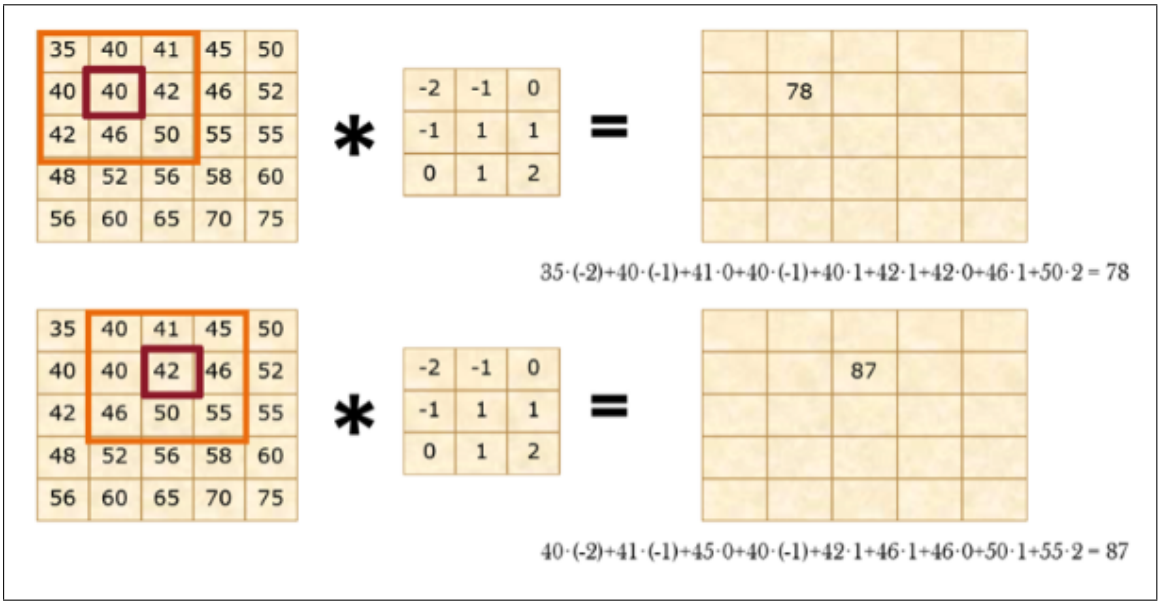

Figura 4.3: Ejemplo de operación de convolución [12].

Como podemos observar en la definición de la convolución, tenemos ciertas limitaciones en el *kernel* de convolución para que la operación sea posible, que describimos a continuación.

- ***Kernel cuadrado:*** para poder definir un centro del *kernel*, que debe superponerse al pixel que se está procesando, el *kernel* debe ser una matriz cuadrada. Esto implica que su número de filas y de columnas debe ser igual, en este caso k.
- ***Kernel de dimensiones impares:*** muy relacionado con el anterior punto, este *kernel* debe tener dimensiones impares. Es por esto por lo que en la definición se definen las dimensiones del *kernel* como k = 2N+1, de forma que siempre sea cuadrado e impar. Además, este valor natural N será muy importante a la hora de implementar el algoritmo, como veremos más adelante en esta sección.

Una vez vistos estos puntos, vamos a estudiar cómo debe ser la imagen o matriz de datos para poder procesarse completamente. En principio, vemos que no tiene limitaciones más allá de las ya mencionadas, pero surge un problema cuando queremos evaluar la operación en las regiones límite de la imagen. Como podemos observar en la figura 4.3, cuando coloquemos el *kernel* sobre los pixeles del borde, parte del *kernel* quedará fuera de la imagen. De forma general, se pueden seguir varias estrategias para solucionar este problema:

- Completar con ceros los valores de alrededor necesarios.
- Repetir los valores del borde.
- Completar con los valores de la parte simétrica opuesta.

Para evitar distorsionar la información, la opción más utilizada y que se llevará a la práctica en este proyecto es completar con ceros los valores de alrededor, creando un "marco" en la imagen. Este proceso es denominado *padding* y el tamaño del marco viene indicado por las dimensiones del *kernel*. En este punto, es donde entra en juego el valor anteriormente mencionado N sobre el que calculábamos la dimensión k de la matriz *kernel*. Calcular la matriz de *padding* sobre la que ya podemos aplicar el *kernel* sobre todos los puntos de la matriz de datos consiste en agregar un "marco" de ceros de tamaño N alrededor. También podemos ver esta operación como agregar 2N filas y 2N columnas de ceros, una inicial y otra final respectivamente. Nuestra matriz de *padding* tendrá entonces de dimensiones $(m+2N, n+2N)$, siendo nuestra matriz de datos (m, n). Podemos ver un ejemplo en la figura 4.4.

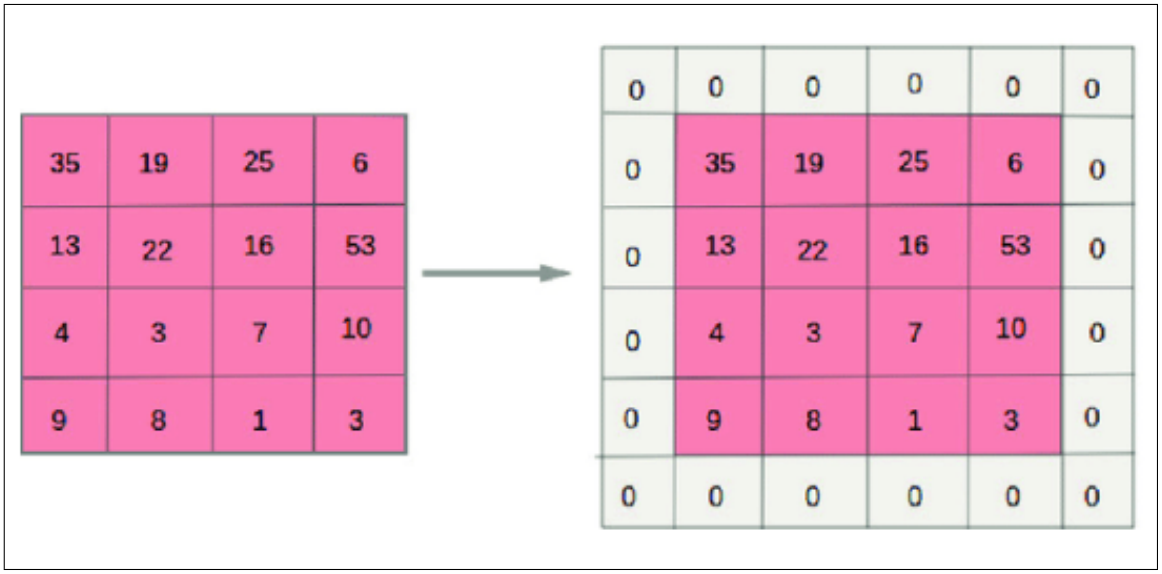

Figura 4.4: Operación de *padding* sobre una matriz con $N = 1$.

En la figura 4.4 podemos ver como llevaríamos a cabo el *padding* en una matriz de datos previamente a la convolución. Consistiría en definir una matriz de "ceros" del tamaño de la matriz de datos original más el *padding* necesario, y la rellenaríamos colocando la matriz en el centro. Esto es lo que le da esta forma de "marco" característica (además teniendo en

cuenta que el 0 representa el color negro o la ausencia de color). Una vez con esta matriz con *padding* ya realizaríamos la convolución.

Visto el algoritmo de convolución a implementar, vemos claro que tenemos dos fases. Primero una fase donde se calcula la matriz de *padding* y una segunda donde ya realizamos la convolución.

Para la primera fase, se puede ver que lo más interesante es que se defina una matriz de "ceros" con el tamaño adecuado, y después cada *work-item* del kernel de *padding* se encargue de mover cada elemento de la matriz de datos a la matriz de *padding* calculando la nueva posición que debe ocupar.

Para la segunda fase, resulta más complejo decidir cómo estructurar la información y cómo definir los *work-items*. Buscando la optimización del algoritmo, parece interesante plantear la misma estrategia que para la multiplicación y que cada elemento de la matriz de salida sea calculado por cada *work-item* definido. Sin embargo, en este caso podemos ver que entonces la matriz que contiene el kernel de convolución debería se recorrida completamente por todos los *work-items*, lo que puede suponer muchísimas lecturas de una zona común de memoria. Para optimizar el algoritmo será necesario plantear si la memoria local podría almacenar este kernel para que esos accesos comunes fueran mucho más rápidos. Al igual que en el resto de casos, todos los detalles de la implementación se estudiarán a continuación en el capitulo 5.

Capítulo 5

Diseño e Implementación

En esta capítulo se va a detallar como se ha llevado a cabo la implementación de los diferentes algoritmos sobre OpenCL a través del framework OpenCLIPER. Primero, se describirán los *kernels* en sí mismos, lo cual es la parte fundamental del algoritmo y donde se realizan los cálculos. Tras esto, se expondrá cómo se integra este *kernel* con OpenCLIPER, para que sea fácil de utilizar y sencillo de implementar para procesado de datos reales. Para ello, se definirá un nuevo proceso de OpenCLIPER, a partir las necesidades del *kernel* y del algoritmo, y que se encargará de procesar y validar los datos de entrada, crear las variables y buffers intermedios cuando sean necesarios, definir y reservar los workgroups y *work-items* más eficientes en función de los datos, lanzar la ejecución y devolver la salida a la memoria accesible por el procesador.

Finalmente, se mostrará un ejemplo de programa que, sobre la implementación realizada sobre OpenCLIPER, se encargue de aplicar el algoritmo a unos datos de entrada, almacenando los datos de salida. Con esto se pretende mostrar cómo se debe usar el proceso de OpenCLIPER definido y así la gran ventaja que supone el uso de este framework: que una vez integrado, el usuario pueda usar este *kernel* para realizar los cálculos que necesite se va a reducirá a unos pocos pasos.

5.1 Norma-N de un vector

Se va a comenzar el diseño y desarrollo de los algoritmos de computación por el cálculo de la norma-N de un vector. Para la norma-N, tenemos como entradas un vector de datos, que pueden ser reales o complejos, y un numero natural N, que será la norma de cálculo. Este algoritmo nos permitirá calcula la norma de orden N de un vector de cualquier tamaño.

Para comenzar, se va a plantear como se va a llevar a cabo la paralelización de los cálculos. Vista la teoría, la implementación más directa se llevará a cabo de forma que cada hilo de ejecución, computado por cada *work-items*, se encargue de calcular la potencia de orden N de cada elemento del vector y almacenarlo en una variable común que realizará las funciones de sumatorio. De esta forma, se calcularán en paralelo todas las potencias, porque es la operación de mayor coste computacional dentro del algoritmo. Una vez calculadas todas las potencias y agregadas a la variable común que gestiona el sumatorio, se realizará la raíz final, almacenando el resultado en la variable de salida.

Teniendo en cuenta la gestión de memoria, el vector de datos se mantendrá en memoria global. Dado que en este caso cada *work-item* solo necesitará leer la memoria global una vez, no tenemos zonas de memoria donde diferentes *work-items* tengan que hacer lecturas a zonas comunes de memoria. Debido a esto, no parece necesario copiar el vector de datos a memoria local. En cuanto a la potencia N, aunque tenga que ser leída por todo los hilos solo va a ser leída una vez por cada uno, por lo que se mantendrá también en memoria global. Sin embargo, las variable que va a mantener el acumulador y que va a tener que ser actualizada por todos los *work-items* sí parece interesante almacenarla en memoria local. Dado que solo se define un único *work-group* en este ejecución, será accesible para todos los *work-items* en memoria local, y su acceso de lectura y escritura será mucho más rápido que en memoria global. No se puede utilizar la memoria privada en este caso, ya que esta memoria es única para cada *work-item*, y en el acumulador tendrán que acceder todos ellos.

Vistos los detalles de implementación en este algoritmo, vamos a ver el kernel desarrollado, con todos los detalles y soluciones para los diferentes problemas encontrados.

5.1.1 KERNEL PARA NORMA-N DE UN VECTOR

```
1  kernel void euclideanNormReal(global realType* A, int N, global realType* C, local float* acc) {
2
3      //Get the thread IDs
4      const int globalCol = get_global_id(0); // Col ID of C (0..N)
5
6      //Inicialize the accumulator
7      if(globalCol == 0){*acc=0;}
8
9      //Stop to synchronice all mamory inicialization
10     barrier(CLK_LOCAL_MEM_FENCE);
11
12     //Atomic read-modify-load on the acumulator the power of the element
13     atomic_add_local(acc,pown(fabs(A[globalCol]),N));
14
15     //Stop to synchronice all workitems
16     barrier(CLK_LOCAL_MEM_FENCE);
17
18     //Save on the output the root
19     if(globalCol == 0){
20         C[0] = pow(*acc,1/(float)N);
21     }
22 }
```

Figura 5.1: *Kernel* para el algoritmo de norma-N para componentes reales.

En la figura 5.1 tenemos el *kernel* desarrollado para números reales. Este será ejecutado por cada uno de los *work-items*, realizando las siguientes operaciones:

1. ***Línea 4: obtener el ID:*** como veremos también en todo el resto de *kernels*, el primer paso es obtener los identificadores de cada *work-item*. Esto se utiliza para identificar el hilo de ejecución, ya que en este caso va a coincidir el identificador del *work-item* con el elemento del vector que va a procesar.

2. ***Línea 7: poner a cero el acumulador:*** dado que el acumulador está almacenado en memoria local, este debe ponerse a 0 para asegurarnos que no contiene basura previamente al cálculo del sumatorio. Para ello, y dado que esta acción solo tiene que realizarse una vez, indicamos que solo el *work-item* con el id igual a cero sea el encargado de realizar esta operación.

3. ***Línea 10: punto de sincronización:*** en este punto, se establece una barrera. esta función definida por OpenCL sirve para forzar la espera de unos determinados hilos en un momento concreto, para forzar un punto de sincronismo. En este caso, queremos que el sincronismo se lleve a cabo para que todos los hilos se paren hasta que el acumulador esté puesto a 0, y nadie escriba antes de esto. Por ello, se pone una barrera que afecta a todos los *work-items* dentro del *work-group* con el uso del parámetro *"CLK_LOCAL_MEM_FENCE"*.

4. ***Línea 13: cálculo de la potencia y suma en el acumulador:*** esta instrucción tiene varios pasos importantes en una sola línea, por lo que se desglosará en los siguientes puntos:

   - ***pown(fabs(A[globalCol]),N):*** en este punto, se llama a la función de bajo nivel de OpenCL que nos permite calcular una potencia de un número real sobre un número natural. Esta función es la que mejor se ajusta a nuestro algoritmo, ya que OpenCL dispone de otras funciones en caso de, por ejemplo, querer elevar un numero real sobre un número real. En este caso, elevamos el elemento del vector cuyo índice coincide con el identificador cada *work-item* a la potencia N.
   - ***atomic_add_local():*** esta función se ha diseñado específicamente para actualizar de forma atómica el acumulador. Esto significa que esta función lleva a cabo una lectura, agregación y escritura del valor de cada hilo al acumulador sin que ningún otro hilo altere el valor del acumulador en el proceso. Esto es necesario para evitar que diferentes hilos de ejecución traten de actualizar el acumulador a la vez, y esto provoque errores porque un sobrescriba el valor actualizado por otro hilo. El diseño de esta función se detalla en la figura 5.2.

```
void atomic_add_local(volatile local float* source, const float operand) {
    float prevVal, newVal;

    do {
        prevVal = *source;
        newVal = prevVal + operand;
    }
    //Evaluate if the update of the value is done where source value is the same as where it was taken for calcs
    while(atomic_cmpxchg((volatile local unsigned int*)source, as_int(prevVal), as_int(newVal)) != as_int(prevVal));
}
```

Figura 5.2: Función que actualiza el acumulador de forma atómica.

Como podemos observar, esta función tiene como valores de entrada el puntero a la variable que se desea actualizar, y el valor que se quiere agregar a la variable. En su funcionamiento, se trata de un bucle que lee el valor existente, almacenándolo en *prevVal*, calcula el nuevo valor en *newVal*, y actualiza el valor llamando a una función de OpenCL llamada *atomic_cmpxchg()*. Esta función que tiene como parámetros el puntero a la variable, el valor original y el valor que se desea actualizar, llevará a cabo la actualización de la variable tras comparar que el valor en el momento de actualizar coincida con el valor que se le ha indicado que tenía recogido en *prevVal*. Solo actualizará el valor si coinciden y

si no, no realizará la actualización del valor. Como salida, devolverá siempre el valor que tenía la variable antes de realizar la comparativa. De esta forma se controla la salida del bucle, ya que el valor habrá sido actualizado exitosamente siempre que la función devuelva el *prevVal*, y no habrá sido actualizada en el resto de casos. En caso negativo, esto volverá a ejecutar el bucle, que leerá el nuevo valor de la variable y volverá a intentar actualizarlo.

Esta función tiene la ventaja de que nos permite asegurarnos que toda la información de todos los hilos en paralelo se almacenará correctamente en la variable, pero tiene la gran desventaja de que, si tratan de actualizar varios hilos al mismo tiempo, el comportamiento de actualización entre ellos será como si fuera un solo hilo haciendo actualizaciones secuenciales. Para mostrarlo con un ejemplo, si 3 hilos tratan de actualizar al mismo tiempo la variable, el primero de los 3 prevalecerá, y los otros 2 tendrán que ejecutar de nuevo el bucle. En el segundo intento, de los 2 que quedan prevalecerá 1 de ellos, y el último tendrá que ejecutar de nuevo el bucle, ahora si ya actualizando la variable ya que es el único. Al final, como 3 hilos trataban de actualizar la variable a la vez, la han actualizado en 3 iteraciones consecutivas, como si fueran ejecuciones secuenciales.

5. ***Línea 16: punto de sincronización :*** después de que todos los *work-items* hayan calculado la potencia de su elemento y se haya actualizado el acumulador, volvemos a poner un punto de sincronización. El objetivo del mismo es asegurar que el acumulador ya ha sido actualizado con todos los cálculos, y contiene el valor final previo a realizar la raíz N.

6. ***Línea 19-20: cálculo de la raíz sobre la variable de salida:*** para finalizar, un elemento tiene que calcular la raíz y almacenar el resultado en la variable de salida. Para ello, dado que esta operación solo se tiene que realizar una vez, volvemos a poner la condición de que solo el *work-item* con ID igual a cero se encargue de ello. Esta operación se realiza llamando a la función *pow* de OpenCL, ya que en este caso la potencia es un número real y no un número entero. El cálculo del resultado se almacena directamente en la salida, donde será devuelto por el *host* hacia el programa del usuario.

Para finalizar, es importante destacar que el flujo para el calculo de la norma-N para un vector de elementos complejos es exactamente igual, pero añadiendo una instrucción para calcular el módulo del número complejo. Para ello utilizamos una función específicamente diseñada para esta tarea que nos proporciona la librería de OpenCL que se llama *hypot*. Esta se llama previamente a calcular la potencia por cada uno de los *work-items*, y con esto se obtiene el módulo que será el valor de entrada del cálculo de la potencia.

5.1.2 Integración de la norma-N con OpenCLIPER

Una vez detallado el *kernel* para el cálculo de la norma-N, vamos a detallar le proceso de OpenCLIPER encargado de gestionar la ejecución del mismo. Como ya hemos visto, este llevará a cabo la creación de buffers, preprocesado, obtención y carga del código del *kernel* y de la cola de ejecución y obtención de la salida. Como se ha expuesto en apartados anteriores, OpenCLIPER define una forma estándar de trabajo, lo que hace que estos procesos sean muy similares entre sí. Por tanto, en este primer algoritmo se expondrá todo el detalle de la implementación del proceso de OpenCLIPER, así como el programa necesario para su ejecución. Sin embargo, para el resto de algoritmos se detallarán las particularidades diferenciales necesarias en cada caso, así como aquellos puntos clave para su ejecución, pero se remitirá a este apartado como flujo genérico para todos los los puntos que no se detallen expresamente.

Como podemos observar en la figura 5.3, todo proceso en OpenCLIPER se compone de dos funciones principales, *init()* y *launch()*. La función *init()* se emplea para hacer preprocesado, y durante este trabajo solo se utilizará en el algoritmo de convolución. En cuanto a la función *launch()*, se encarga desde la recogida de parámetros hasta la ejecución del *kernel*, y se estructura en los siguientes puntos:

1. ***Línea 25-26: obtención de parámetros de lanzamiento:*** por defecto, OpenCLIPER define para cada proceso un argumento de entrada y un argumento de salida. En caso de requerir más, se puede definir una estructura de datos denominada parámetros de lanzamiento que nos permite tener todos los parámetros necesarios. En este caso nuestros parámetros de entrada son el vector de datos y la potencia N. En este caso, como entrada del proceso se define el vector de datos, y como parámetro de lanzamiento la potencia N.

```
17  void EuclideanNorm::init() {
18
19      EUN_CERR("EUN::init()\n");
20  }
21
22  void EuclideanNorm::launch() {
23      try {
24          //Get the launch parameters, the norm
25          auto pLP = std::dynamic_pointer_cast<LaunchParameters>(pLaunchParameters);
26          if (!pLP) pLP = std::unique_ptr<LaunchParameters>(new LaunchParameters());
27
28          EUN_CERR("EUN::launch\n");
29
30          //Validation checks on the inputs and outputs.
31          check_objects(getInput(),pLP->Norm,getOutput());
32
33          //Set input and output OpenCL buffers on device memory
34          cl::Buffer* pInBuf1 = getInput()->getDeviceBuffer();
35          cl::Buffer* pOutBuf = getOutput()->getDeviceBuffer();
36
37          //Choose the kernel depending the vector data type
38          if(getInput()->getElementDataType() == TYPEID_COMPLEX){
39              kernel = getApp()->getKernel("euclideanNormComplex");
40
41          }else{
42              kernel = getApp()->getKernel("euclideanNormReal");
43          }
44
45          //Set kernel parameters
46          kernel.setArg(0, *pInBuf1);
47          kernel.setArg(1, pLP->Norm);
48          kernel.setArg(2, *pOutBuf);
49          kernel.setArg(3, cl::Local(sizeof(float)));
50
51          //Set kernel work items size: number of pixels to process is vector size
52          cl::NDRange globalSize = cl::NDRange(getInput()->getSpatialDimSize(0, 0), getInput()->getSpatialDimSize(1, 0));
53
54          //Execute kernel
55          queue.enqueueNDRangeKernel(kernel, cl::NullRange, globalSize, cl::NDRange(), NULL, NULL);
56      }
57      catch (cl::Error& err) {
58          BTTHROW(CLError(err), "EuclideanNorm::launch");
59      }
60  }
```

Figura 5.3: Proceso de OpenCLIPER que gestiona la ejecución del *kernel* de norma-N.

2. ***Línea 31: función check_objects() :*** esta función auxiliar definida dentro del proceso para validar todos los argumentos de entrada y salida. En ella, se comprueban diferentes condiciones en función del algoritmo, para validar que todas las entradas y salidas sean correctas, estén definidas, tengan los valores, tamaños o tipos de datos esperados, y sea posible con ellas realizar la ejecución correcta del algoritmo.

3. ***Línea 34-35: definición de buffers de entrada-salida :*** en estas líneas, a partir de las variables de entrada, se definen los buffers de OpenCL necesarios para que la información en memoria sea accesible por parte del *kernel* en OpenCL. OpenCLIPER nos facilita esta tarea proporcionándonos una función que nos devuelve este buffer a partir de la variable *XData* definida que almacena los datos.

4. ***Línea 38-43: selección del kernel correcto :*** a continuación, se obtiene el código del *kernel* que se va a ejecutar. En este caso, dado que podemos tener dos *kernels* en función de si los datos del vector de entrada son números reales o números complejos, utilizamos el tipo de dato para seleccionar uno u otro. Este flujo se va a aplicar a los algoritmos definidos en este trabajo, ya que se pueden ejecutar sobre datos reales o complejos.

5. ***Línea 46-49: configuración de parámetros del kernel :*** en este punto, una vez seleccionado el *kernel*, se van a configurar en orden los parámetros que necesita en ejecución. Para ellos, se indican los punteros, las variable y se reserva espacio en memoria para aquellas variables que se emplearán dentro del *kernel* pero que todavía no están definidas. En este caso la última línea reserva espacio en la memoria local para el acumulador.

6. ***Línea 52-53: definición del tamaño de work-group :*** En esta línea definimos el tamaño del conjunto de *work-items* que van a ejecutar el *kernel*. Este punto es fundamental, ya que de él dependen los hilos de procesamiento y los identificadores de los mismos. En este caso, como va a haber tantos *work-items* como elementos del vector de datos de entrada, se indican directamente las dimensiones del mismo para definir el grupo.

7. ***Línea 56-59: encolado del kernel:*** por último, ya con todos los parámetros configurados, se llama a la cola de ejecución, y se añade el *kernel*, indicando el tamaño de *work-group* previamente definido.

5.2 Producto escalar de vectores

Una vez visto el algoritmo que calcula la norma-N, el siguiente algoritmo que se va a implementar es un algoritmo con un planteamiento muy similar. Como se ha detallado en el apartado teórico, el producto escalar de vectores consiste en el sumatorio de los productos de sus componentes. Esta forma de trabajo es muy similar a la norma-N, ya que cada elemento de los vectores de entrada se lee una vez, y todos los resultados de las sumas parciales se agregan en un sumatorio. Por tanto, el planteamiento de paralelización será muy similar al del algoritmo anterior. Cada uno de los *work-items* se encargará de calcular el producto de cada par y agregarlo al sumatorio. Por tanto, se definirá un grupo de *work-items* unidimensional, de longitud igual a la de los dos vectores de datos, de forma que cada *work-item* calcule el producto de la pareja con mismo índice, y la agregue al sumatorio con una operación atómica.

En cuanto a la estructura de los datos en memoria, al igual que en el algoritmo anterior, los vectores de datos se mantendrán en la memoria global. Dado que cada elemento de ambos vectores solo será accedido una vez para lectura por un único *work-item*, no

resulta rentable copiarlos a memoria local. Sin embargo, para el acumulador que calcula el sumatorio, este se almacenará en memoria local dado que tiene que ser actualizado por todos los *work-items*. Esto es posible dado que se va a definir un único *work-group* y por tanto estos acumuladores en memoria local serán accesibles por todos los *work-items*.

En cuanto al código del *kernel* así como el proceso de OpenCLIPER, son prácticamente idénticos con pequeños cambios. Por tanto, respecto al *kernel*, solo se detallarán los puntos diferentes, en este caso sobre el *kernel* que aplica a vectores con elementos complejos. En cuanto al proceso de OpenCLIPER, se referencia al expuesto en la sección 5.3, con el único cambio que el parámetro de lanzamiento en lugar de ser un número entero será un puntero que apuntará al vector B, y se obtendrá el buffer necesario para el acceso al mismo desde el *kernel* de forma homónima a como se obtenía para el vector de entrada en el proceso de la norma-N.

5.2.1 Kernel para el producto escalar

Para analizar el *kernel* diseñado para el producto escalar, vamos a emplear como referencia el que aplica cuando los vectores tienen como elementos números complejos.

Como podemos ver en la figura 5.4, este flujo es muy similar al expuesto en la figura 5.1. Como diferencia podemos ver que los parámetros de entrada son 3 vectores, los dos que contienen los datos de entrada y el vector que almacenará el valor de salida, y los dos acumuladores definidos sobre memoria local, para almacenar la parte real y la parte imaginaria.

```
1  kernel void scalarProductComplex(global complexType* A, global complexType* B, global complexType* C,
2                                   local float* accReal, local float* accImag) {
3
4      //Get the thread IDs
5      const int globalCol = get_global_id(0); // Col ID of C (0..N)
6
7      //Inicialize de accumulator
8      if(globalCol == 0){
9          *accReal=0;
10         *accImag=0;
11     }
12     //Stop to synchronice all mamory inicialization
13     barrier(CLK_LOCAL_MEM_FENCE);
14
15     //Atomic read-modify-load on the acumulator the product of the element
16     atomic_add_local(accReal,((A[globalCol].x*B[globalCol].x)-(A[globalCol].y*(-B[globalCol].y))));
17     atomic_add_local(accImag,((A[globalCol].x*(-B[globalCol].y))+(A[globalCol].y*B[globalCol].x)));
18
19     //Stop to synchronice all workitems
20     barrier(CLK_LOCAL_MEM_FENCE);
21
22     //Save on the output
23     if(globalCol == 0){
24         C[0].x = *accReal;
25         C[0].y = *accImag;
26     }
27 }
```

Figura 5.4: *Kernel* para el cálculo del producto escalar para vectores de componentes complejas.

El resto del flujo es similar: primero obtenemos el índice del *work-item* que ejecuta el hilo, tras esto inicializamos a cero los acumuladores y ponemos una barrera de sincronismo. Tras esto, en las líneas 16 y 17, se realiza el cálculo del producto del complejo contenido en A con el complejo conjugado del número contenido en B. Para calcular el complejo conjugado de los elementos contenidos en B, unicamente cambiamos de signo la parte imaginaria. Este valor calculado se almacena en los acumuladores empleando la función previamente detallada en la figura 5.2 . Una vez todos los hilos han llegado a la barrera de sincronismo y los acumuladores tienen el valor final del sumatorio, simplemente se almacena este valor en la variable de salida, solo una única vez llevada a cabo por el *work-item* con identificado igual a cero.

Es importante destacar que la principal diferencia a nivel de carga computacional entre el *kernel* para el cálculo de la norma-N y este *kernel* para el cálculo del producto vectorial es el coste computacional de calcular el producto frente a calcular la potencia N. Si bien este punto puede no parecer tan importante, será clave para evaluar el rendimiento de estos algoritmos y compararlos entre sí. Haciendo un adelanto de los resultados que podremos estudiar en el capítulo 6, cuanto mayor sea la carga computacional de las operaciones llevadas a cabo en los hilos frente a la carga computacional de las lecturas-escrituras zonas comunes de memoria, mejor será el rendimiento de la GPU frente a la CPU.

Para finalizar, cabe destacar que para el *kernel* empleado cuando las componentes de los vectores son números reales, se reduce a un solo acumulador, y la operación es simplemente el producto de ambos números. También hay que concretar que la única diferencia entre el proceso que gestiona este *kernel* y el que se ha estudiado en la figura 5.3 es que el parámetro de lanzamiento en lugar de ser un número es un puntero de tipo *XData* al igual que el *input*, y que en lugar de pasarlo como argumento del *kernel* por valor, se pasa el buffer OpenCL. Esta es una de las grandes ventajas que podemos ver con el uso de OpenCLIPER en la práctica, no solo el programa del usuario es sencillo y siempre tiene la misma estructura, sino que para los desarrolladores, la estructura del framework permite que para lanzar diferentes *kernels* el proceso encargado de gestionar las ejecuciones sea muy similar en la mayoría de casos, y sea reutilizable facilitando en muchos casos el diseño e integración de nuevos algoritmos de procesamiento.

5.3 PRODUCTO DE MATRICES

Una vez definidos los dos algoritmos que se van a implementar para operaciones con vectores, vamos a ver los algoritmos para operaciones con matrices, comenzando con el algoritmo que calcula el producto de dos matrices. Este algoritmo tomará como valores de entrada dos matrices, A y B y realizará el producto algebraico de las mismas. Inicialmente, vamos a ver el flujo del kernel, una vez implementado sobre OpenCL. Para esta integración, se ha definido que cada *work-item* calcule el producto de un elemento de la matriz de salida. De esta forma, el *work-item* calculará el producto del elemento de la matriz con el mismo indice: $WI_{i,j} \rightarrow C_{i,j}$. Por tanto, se va a definir un conjunto de *work-items* bidimensional cuyo tamaño coincide con la matriz de salida que almacenará el resultado de la operación. Esto implica que el *kernel* realiza el sumatorio de los productos de la fila A_i por la columna B_j, almacenando el resultado de forma ordenada en Ci, j.

Para la disposición en memoria, resulta destacable que aunque en este algoritmo se trabaje con matrices bidimensionales, la memoria solo es unidimensional, por lo que la forma de disponer los datos será un vector donde se colocarán las filas de la matriz una detrás de otra. Esta disposición es la más habitual, la escogida también para este trabajo y es la que va a emplearse para almacenar las matrices en memoria tanto en este algoritmo como en el de convolución. Esta estructura de almacenamiento empleada provoca que, para

poder acceder a un elemento de la matriz a partir de sus índices, sea necesario conocer el ancho de cada fila o el número de columnas de la matriz. Con esta información, acceder a un elemento de la matriz bidimensional sobre la memoria unidimensional consistirá en la siguiente operación definida en la fórmula 5.1:

$$a_{i,j} = A[i \cdot n + j] \tag{5.1}$$

siendo n el número de columnas de A.

En cuanto a nivel de memoria que se va a emplear, cabe preguntarse si con el planteamiento de diseño definido resulta rentable copiar cierta información a memoria local. En este caso, las lecturas a zonas comunes de memoria dependerán del número de filas y de columnas de las matrices de entrada. Dado que cada *work-item* va a realizar el producto de una fila por una columna, el número de lecturas a la misma fila de la matriz A va a ser el número de columnas de la matriz B y viceversa. Esto se debe a que cada fila de A se multiplica por todas las columnas de B. Por tanto, podemos ver que el número de accesos a zonas de memoria común por parte de diferentes *work-items* es relativamente bajo. Por ello se plantea que se trabaje directamente sobre memoria global, excepto para el acumulador de cada *work-item* que tiene sentido que se almacene en la memoria privada. Se estudiará en el capítulo 6 de resultados la calidad de esta aproximación.

Por último, resaltemos que no se considera necesario exponer de forma detallada el proceso de OpenCLIPER que va a gestionar este algoritmo. En este caso, el proceso es idéntico al definido para el producto escalar de vectores, similar a su vez al expuesto en la sección 5.1.2 para la norma-N. Todos los pasos son iguales a los definidos para el producto escalar, a excepción que al definir las dimensiones del *work-group* de trabajo. Para el producto de matrices, las dimensiones del conjunto de *work-items* de trabajo serán las mismas que la matriz de salida C. Esto evidencias las grandes ventajas que supone OpenCLIPER no solo para usuarios sino también para desarrolladores: con construir solo un proceso e irle realizando pequeñas modificaciones se pueden integrar diferentes algoritmos sin tener que preocuparse de toda la gestión de OpenCL.

5.3.1 KERNEL PARA PRODUCTO DE MATRICES

Una vez definida la paralelización que se va a llevar a cabo con cada *work-item* calculando cada elemento de la matriz de salida y definida la estructura de memoria a emplear, se va a revisar el *kernel* implementado para matrices de números reales en la figura 5.5.

```
1  kernel void multiplyReal(global realType* A, global realType* B, global realType* C) {
2
3      //Get the thread IDs
4      const int globalRow = get_global_id(1); // Row ID of C (0..M)
5      const int globalCol = get_global_id(0); // Col ID of C (0..N)
6
7      //Multiply is A(M*K)*B(K*N) = C(M*N)
8      const int n = getSpatialDimSize(B, 0, 0);
9      const int k = getSpatialDimSize(B, 1, 0);
10
11     //Calculate the value, iterating on k
12     private realType acc;
13     acc = 0;
14     for (int i=0; i<k; i++) {
15         acc += A[globalRow*k + i] * B[i*n + globalCol];
16     }
17
18     //Save on the output
19     C[globalRow*n + globalCol] = acc;
20 }
```

Figura 5.5: Kernel de multiplicación de matrices de números reales.

1. ***Línea 4-5: obtención de identificadores de work-item:*** al igual que en el resto de *kernels*, comenzamos obteniendo el los identificadores del *work-item* que ejecuta esta instancia del kernel. Estos nos indican que elemento de la matriz de salida va a ser calculado y a su vez la fila de A y la columna de B que se va a multiplicar a pares.

2. ***Línea 8-9: obtención de las dimensiones de B:*** como se ha indicado en el apartado anterior, para recorrer las matrices bidimensionales sobre la memoria unidimensional es necesario conocer el número de columnas de las mismas, debido a como se han dispuesto en memoria. Para ello en estas dos líneas, empleando una función de OpenCLIPER que nos devuelve las dimensiones de un puntero de una matriz de tipo *XData*, obtenemos el número de filas y de columnas de la matriz B. Recordemos que, debido a las propiedades que deben cumplir las dimensiones de dos matrices para ser multiplicables, el número de filas de B es igual al número de columnas de A, y el número de columnas de B es igual al número de columnas de la matriz resultado C. Por tanto, con estos dos datos, conocemos el número de columnas de todas las matrices implicadas en las operaciones, para poder recorrer

sus valores a partir de los índices.

3. ***Línea 12-13: definir e inicializar el acumulador:*** en las siguientes líneas de define e inicializa el acumulador que va a almacenar el sumatorio de los valores para cada elemento de la matriz de salida. Dado que cada una de estos elementos va a ser calculado exclusivamente por un solo *work-item*, se define esta variable como privada para que su acceso sea lo más rápido posible. Además, no es necesario en este caso tampoco definir la operación con la función de escritura atómica, ya que al ser accedido sobe un único *work-item* no puede haber solapamiento con otros hilos de ejecución.

4. ***Línea 14-16: calcular el valor de cada elemento de la matriz de salida:*** para calcular el valor de cada elemento se utiliza la siguiente fórmula 5.2. Para ello se define un bucle, que va recorriendo estos valores, y calculando la multiplicación, almacenando su valor agregado en el acumulador definido en el paso anterior. Para ello, se recorre el bucle k veces, siendo este valor el número de columnas de A. Vemos que la variable i que controla las ejecuciones del bucle, va recorriendo para la fila de A ($globalRow \cdot k + i$) los valores de la misma y realizando el producto recorriendo la columna B ($i \cdot n + globalCol$).

$$c_{i,j} = \sum_{1}^{k} a_{i,k} \cdot b_{k,j} \tag{5.2}$$

5. ***Línea 19: almacenar el valor de salida de forma ordenada en C:*** por último, y conociendo los índices, almacenamos el resultado final en C, de forma ordenada, como se ha explicado en el apartado de las dimensiones. En este punto, como cada *work-item* tiene unos índices diferentes, cada uno habrá ido realizando la misma operación sobre diferentes datos y guardándola en la posición adecuada.

Una vez estudiado el flujo del *kernel* diseñado para multiplicar matrices de números reales, revisado sobre la figura 5.5, el flujo será exactamente el mismo, pero cambiado la forma de multiplicar cada par. Como se expuso en el apartado 4.2.1, multiplicar dos números complejos representados de forma binomial consiste en el producto de un binomio, donde se separa la parte real y la parte imaginaria. Por tanto, dentro del bucle que recorre la fila y la columna y que debe procesar cada *work-item*, se va a sustituir el producto de dos números reales por el de dos números complejos, implementando la formula 4.4.

Para ello es importante comentar como se definen los datos complejos. A nivel de programación, un número complejo lo definimos como una estructura de datos que contiene dos números reales, uno la parte real y otro la parte imaginaria. Por tanto, nuestros datos consisten en una matriz de estructuras, que ofrecen sus propios métodos para acceder a esos dos valores en coma flotante. De esta forma, extraemos los datos en juego, realizamos la operación separando parte real y parte imaginaria, y guardamos el resultado en dos acumuladores por separado. Una vez finalizada la operación, recorriendo el bucle que realiza el sumatorio hasta k completamente, se guardan ambos valores en la matriz de salida, que también consiste en una matriz de estructuras del mismo tipo.

El resto del algoritmo es idéntico, haciendo todo el proceso de obtener los identificadores de *work-item*, obteniendo las dimensiones indicadas en 5.3.1, iniciando en este caso dos acumuladores separados para parte real e imaginaria, realizando los productos y almacenando el sumatorio final en la matriz C de salida de forma ordenada.

5.4 Potencia de una matriz

Una vez implementado el *kernel* que realiza el producto algebraico de matrices en OpenCLIPER, se va a definir la implementación del algoritmo que calcula la potencia de una matriz. Para ello se va a construir sobre los *kernels* del producto algebraico de matrices ya diseñados, de forma que se van a hacer llamadas sucesivas a los mismos para calcular la potencia. Dado que a nivel de *kernel* se van a reutilizar los empleados para realizar el producto de matrices, se va a definir un flujo dentro del proceso de OpenCLIPER que nos permita gestionar todas estas ejecuciones sucesivas, así como las variables involucradas.

El objetivo de este flujo en OpenCLIPER es reducir al mínimo todas la operaciones extra que no sean las llamas sucesivas al *kernel*. Dado que además de la matriz de datos y la matriz de salida se va a requerir una matriz auxiliar para almacenar los resultados parciales, una de estos objetivos será evitar tener que realizar cualquier copia de datos entre regiones de memoria. Es decir, no tener que copiar información de una matriz a otra. Otro de estos objetivos será que en la última ejecución, la información esté disponible en la matriz de salida directamente para cualquier potencia. Para conseguir todos estos objetivos y conseguir que calcular la potencia solo implique la ejecuciones sucesivas del kernel se ha definido el siguiente proceso de OpenCLIPER.

5.4.1 Integración de la potencia con OpenCLIPER

Dentro del proceso definido en OpenCLIPER, la primera parte es la parte común a todos los procesos tal y como se han construido durante el trabajo. Por ello, no se entrará tanto en destalle de los puntos ya definidos en la sección 5.1.2. El proceso de OpenCLIPER para el cálculo de una matriz elevado a una potencia se muestra en la figura

```
17  void PowerMatrix::init() {
18
19      POM_CERR("POM::init()\n");
20  }
21
22  void PowerMatrix::launch() {
23      try {
24          //Get the launch parameters, the power
25          auto pLP = std::dynamic_pointer_cast<LaunchParameters>(pLaunchParameters);
26          if (!pLP) pLP = std::unique_ptr<LaunchParameters>(new LaunchParameters());
27
28          POM_CERR("POM::launch\n");
29
30          //Validation checks on the inputs and outputs.
31          check_objects(getInput(),getOutput());
32
33          //Create de auxiliar data matrix to hold the operations
34          auto pAux = std::make_shared<XData>(getApp(), getInput()->getSpatialDimSize(0, 0),
35                                              getInput()->getSpatialDimSize(1, 0), getInput()->getElementDataType());
36
37          //Set input and output OpenCL buffers on device memory
38          cl::Buffer* pInBuf1 = getInput()->getDeviceBuffer();
39          cl::Buffer* pAuxBuf = pAux->getDeviceBuffer();
40          cl::Buffer* pOutBuf = getOutput()->getDeviceBuffer();
41
42          //Choose the kernel depending the matrix data type
43          if(getInput()->getElementDataType() == TYPEID_COMPLEX){
44              kernel = getApp()->getKernel("multiplyComplex");
45          }else{
46              kernel = getApp()->getKernel("multiplyReal");
47          }
48
49          // Set kernel work items size: number of pixels to process is image width x height
50          cl::NDRange globalSize = cl::NDRange(getOutput()->getSpatialDimSize(0, 0), getOutput()->getSpatialDimSize(1, 0));
51
52
53          //Number of multipliers to power
54          for(int i=1;i<pLP->Power;i++){
55
56              //If power is pair, the final output is on the odd executions
57              if(pLP->Power%2==0){
58
59                  //First execution is the data over itself
60                  if(i==1){
61
62                      // Set kernel parameters
63                      kernel.setArg(0, *pInBuf1);
64                      kernel.setArg(1, *pInBuf1);
65                      kernel.setArg(2, *pOutBuf);
66
67                      // Execute kernel
68                      queue.enqueueNDRangeKernel(kernel, cl::NullRange, globalSize, cl::NDRange(), NULL, NULL);
69
70                  //On the pair executiones save the operation on the aux matrix
71                  }else if(i%2==0){
```

Figura 5.6: Parte general de proceso de OpenCLIPER para realizar el la potencia de una matriz.

1. ***Línea 25-26: obtención de parámetros de lanzamiento :*** por defecto, OpenCLIPER define para cada proceso un argumento de entrada y un argumento de salida. En caso de requerir más, se puede definir una estructura de datos denominada parámetros de lanzamiento que nos permite tener todos los parámetros necesarios.

En este caso nuestros parámetros de entrada son el vector de datos y la potencia N. En este caso, como entrada del proceso se define el vector de datos, y como parámetro de lanzamiento la potencia N.

2. ***Línea 31: función check_objects() :*** esta función auxiliar definida dentro del proceso para validar todos los argumentos de entrada y salida. En ella, se comprueban diferentes condiciones en función del algoritmo, para validar que todas las entradas y salidas sean correctas, estén definidas, tengan los valores, tamaños o tipos de datos esperados, y sea posible con ellas realizar la ejecución correcta del algoritmo.

3. ***Línea 34-35: definición de buffers de entrada-salida :*** en esta línea se define una nueva matriz vacía, que será la matriz auxiliar necesaria para guardar las operaciones intermedias en las llamadas sucesivas al *kernel*. Esta se define con las mismas dimensiones y el mismo tipo de dato que la matriz de entrada.

4. ***Línea 38-40: definición de buffers de entrada-salida :*** en estas líneas, a partir de las variables de entrada y la matriz auxiliar, se definen los buffer de OpenCL necesarios para que la información en memoria sea accesible por parte del *kernel* en OpenCL. OpenCLIPER nos facilita esta tarea proporcionándonos una función que nos devuelve este buffer a partir de la variable *XData* definida que almacena los datos.

5. ***Línea 43-47: selección del kernel correcto :*** a continuación, se obtiene el código del *kernel* que se va a ejecutar. En este caso, dado que podemos tener dos *kernels* en función de si los datos del vector de entrada son números reales o números complejos, utilizamos el tipo de dato para seleccionar uno u otro. Este flujo se va a aplicar a los algoritmos definidos en este trabajo, ya que se pueden ejecutar sobre datos reales o complejos.

6. ***Línea 50: definición del tamaño de work-group :*** En esta línea definimos el tamaño del conjunto de *work-items* que van a ejecutar el *kernel*. Este punto es fundamental, ya que de él dependen los hilos de procesamiento y los identificadores de los mismos. En este caso, como va a haber tantos *work-items* como elementos de la matriz de datos de entrada y, se indican directamente las dimensiones del mismo para definir el grupo.

7. ***Línea 54-...:*** a partir de este punto, se va a definir el flujo en el que se van realizando la diferentes llamadas al kernel de multiplicación, disponiendo los buffers en cada ejecución de forma que se realice la multiplicación adecuada en cada interacción. Esta disposición variable de los buffers en cada ejecución del kernel es lo que nos va a permitir reducir el cálculo a llamadas sucesivas al kernel si la necesidad de copiar nunca los datos de una zona de memoria a otra. Para facilitar su comprensión se muestra el diagrama de flujo en la figura 5.7.

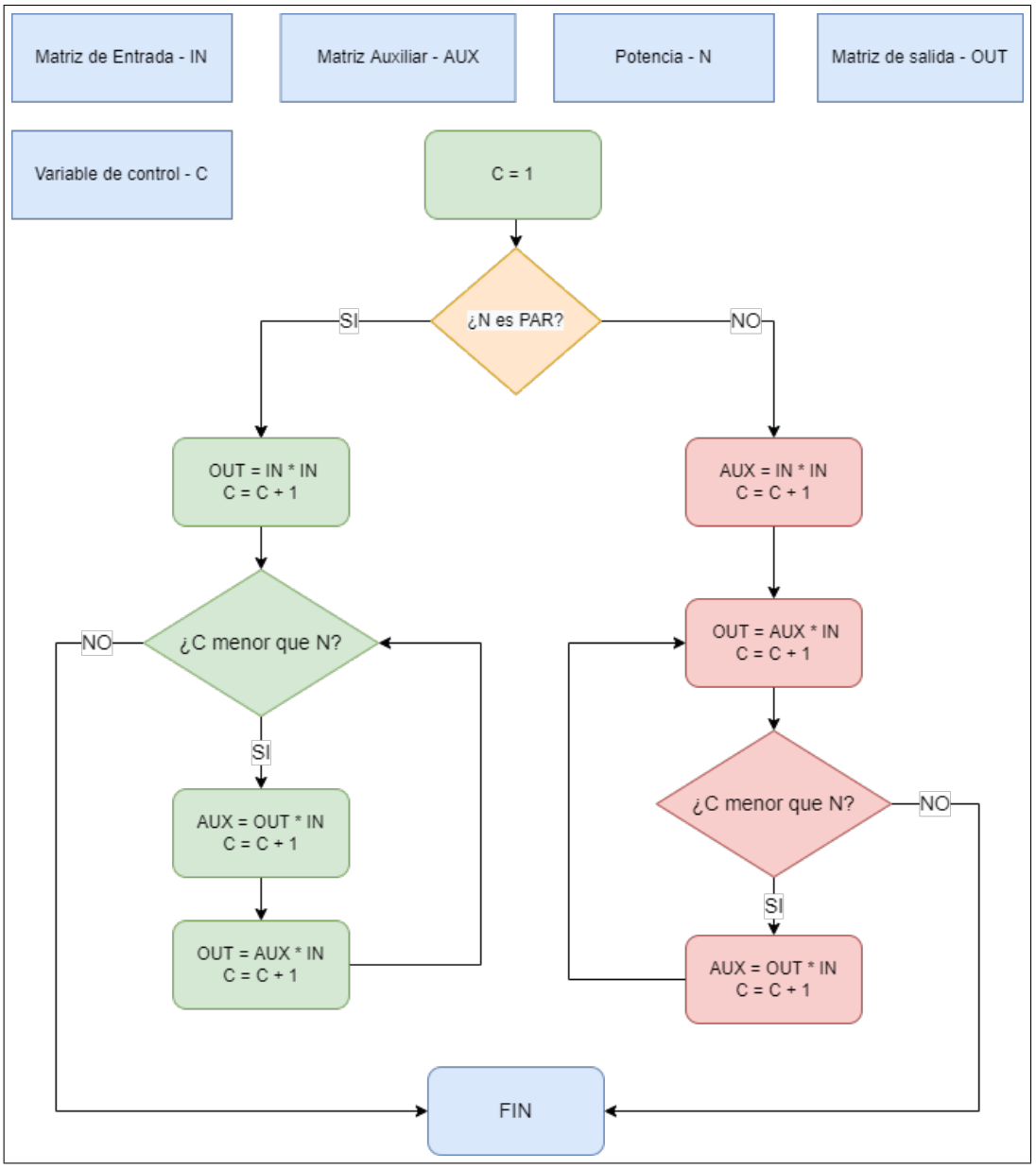

Figura 5.7: Flujo de ejecuciones sucesivas del kernel de multiplicación para el cálculo de una matriz elevado a una potencia.

Como se puede observar en la figura, el flujo va modificando los buffers que se ejecutan en cada instancia en la cola de ejecución de los *kernels*, de forma que en función de la potencia a la que haya que elevar la matriz sigue diferentes flujos. Inicialmente, es importante evaluar si la potencia a calcular es par o impar. Esto se emplea para decidir en primera instancia cuándo se va a usar como matriz resultado la matriz de salida y cuándo la matriz auxiliar. Esto se debe a que si se realiza el cálculo de una potencia par, el numero de ejecuciones del *kernel* va a ser impar y viceversa. Con esto ya diferenciamos dos caminos separados, evitando tener que copiar los datos a la matriz de salida una vez finalizadas todas las ejecuciones.

Una vez definida esta parte, la forma de evitar copiar los datos a la matriz auxiliar, evitar inicializarla con los datos de entrada y en general realizar las mínimas escrituras necesarias sobre esta matriz es ir intercalando las matrices de salida y auxiliar en operaciones sucesivas. De esta forma, cuando la matriz auxiliar cumple el rol de matriz resultado, la matriz de salida cumple el rol de multiplicando y viceversa. De esta manera, se va iterando sucesivamente el bucle, comparando hasta el momento que nuestra variable de control sea igual a la potencia que se desea calcular. Una vez esto ocurre, el bucle y el algoritmo completo finaliza, ya que al definir en función de la paridad de la potencia el orden de alternancia, aseguramos que en la última operación los datos siempre estarán copiados en la matriz de salida.

De esta forma, gracias a aprovechar las ventajas que nos ofrece la estructura de OpenCLIPER, así como el diseño del flujo expuesto, se logran los objetivos de reducir el cálculo de elevar una matriz de datos a una potencia de orden N a una sucesión de multiplicaciones ejecutadas con llamadas al *kernel* diseñado para el producto de dos matrices.

5.5 Convolución de matrices

Por último, y como algoritmo más complejo se detallará la convolución de matrices. Comenzando por la forma de paralelizar el algoritmo, se va a seguir una estrategia muy similar al algoritmo de multiplicación. Para ello se paralelizará de forma que cada *work-item* calculará un elemento de la matriz de salida. De esta forma, se definirá un *work-group* bidimensional en el que cada elemento calculará el sumatorio de los productos de los pares

cuando se superpone la matriz que contiene el *kernel* de convolución[2] centrado en el mismo elemento de la matriz de datos.

En cuanto a la estructura de la memoria, se dispondrán tanto la matriz de datos como la matriz que contiene el *kernel* de convolución de la misma forma que en el caso del producto de matrices. Esto implica que la matriz bidimensional se dispondrá en memoria colocando las filas sucesivamente una detrás de otra. Esto implica que el acceso a cada elemento se llevará a cabo de forma idéntica a la empleada en el algoritmo de multiplicación.

Debido a como se lleva a cabo la operación de convolución, esta estrategia implica una gran cantidad de lectura de zonas de memoria común por un gran número de *work-items*. De hecho, la matriz que contiene el *kernel* de convolución será accedida completamente por todos los *work-items*, lo que puede provocar un cuello de botella si muchos hilos de ejecución tienen que estar esperando a leer los datos necesarios para poder realizar las operaciones. Debido a esto, resulta interesante un diseño en el que se pueda copiar a memoria local la matriz que contiene el *kernel* de convolución. Sin embargo, y como se verá en el capítulo 6, el tamaño de la memoria local es relativamente pequeño, y en general puede haber casos en el que el *kernel* de convolución sea mayor que el espacio de memoria local existente. Por ello se va a realizar un diseño basado en dos *kernels* de procesamiento: uno general aplicable siempre en el que se usará memoria global, y uno que copiará el *kernel* a memoria local y se usará solo cuando se compruebe que la memoria local puede contener el *kernel* de convolución completamente.

Además, es importante destacar que cuanto mayor es el *kernel* de convolución, mayor es el solapamiento de *work-items* que acceden a los mismos elementos de la matriz de datos, pero también mayor número de operaciones debe realizar cada hilo para calcular el sumatorio. Esto se debe a que el kernel, al ocupar más espacio sobre la matriz de datos, más elementos de la matriz de datos van a aplicar en el cálculo de cada elemento de la matriz de salida, ya que el entorno de influencia para la convolución de cada elemento aumenta. Cómo influyen todos estos parámetros de tamaño de la matriz de datos frente a tamaño de kernel, y volumen de operaciones frente a lecturas a zonas comunes de memoria resulta de gran interés para entender la implementación a bajo nivel, y se estudiará a fondo en el siguiente capítulo.

[2]Recordar que cuando se habla de *kernel* de convolución se refiere a la matriz de datos y segundo operando de la operación de convolución, y no al *kernel* de código que contiene las instrucciones de ejecución de cada *work-item*.

Para finalizar, hay que tener en cuenta que en este algoritmo de convolución se requiere agregar el *padding* de ceros a la matriz de datos previamente. Esto implica que antes de ejecutar el *kernel* de cálculo es necesario un procesamiento previo para calcular esta matriz de *padding*. Esto se llevará a cabo con un *kernel* de OpenCL, que se encargue de copiar los valores desde la matriz de datos a la matriz de *padding*, desplazándolos adecuadamente. Como añadido, y como parte del procesamiento previo, el proceso de OpenCLIPER también se encargará de calcular si se puede emplear memoria local en ese caso o no, en función del tamaño de la matriz que contiene el *kernel* de convolución.

5.5.1 KERNEL PARA CONVOLUCIÓN DE MATRICES

Una vez detalladas todas las decisiones de diseño para la implementación de la convolución, se van a ver en detalle los *kernels* de OpenCL implicados en este algoritmo. Inicialmente comenzaremos describiendo el *kernel* que calcula el la matriz de *padding* a partir de la matriz de datos. A continuación, veremos el *kernel* que realiza la convolución empleando la memoria global. Por último, estudiaremos las diferencias y complejidades que tiene el *kernel* que emplea la memoria local para almacenar el *kernel*.

```
87  kernel void paddingKernel(global realType* A, global realType* B, uint padding) {
88
89      //Get the thread IDs
90      const int globalRow = get_global_id(1); // Row ID of C (0..M)
91      const int globalCol = get_global_id(0); // Col ID of C (0..N)
92
93      //Get the column quantity
94      const int n = getSpatialDimSize(A, 0, 0);
95
96      //Save the data on the right place on the padding matrix
97      //(n+2*padding) -> New number of columns on the output.
98      B[(globalRow+padding)*(n+2*padding) + (globalCol+padding)] = A[globalRow*n + globalCol];
99  }
```

Figura 5.8: Kernel para rellenar la matriz de *padding* con los datos.

Podemos ver el código del *kernel* de *padding* en la figura 5.8:

1. ***Línea 87: entrada del kernel:*** en este caso, las entradas del *kernel* van a ser nuestra matriz de datos original, denominada A, nuestra matriz de salida, llamada matriz de padding, B y, por último, el valor de N, siendo este $N = \lfloor \frac{k}{2} \rfloor$ con k la dimensión fila o columna del *kernel* de convolución.

2. ***Líneas 90-91: obtener el work-item ID:*** con una llamada a la API de OpenCL, este nos devuelve los identificadores del *work-item* que está ejecutando esa instancia

del *kernel*. De esta forma podemos paralelizar los cálculos, ya que en función del *work-item* que esté ejecutando el *kernel*, podremos saber a que fila y columna así como en que posición tiene que guardar el valor de salida.

3. ***Línea 94: obtener las dimensiones de A:*** para poder calcular el desplazamiento necesario, necesitamos conocer las dimensiones de A, ya que las de B serán las dimensiones de $A + 2N$, y con estos valores podemos calcular el desplazamiento.

4. ***Línea 98: almacenamiento desplazado en B:*** en este último paso, vamos a almacenar el valor de A pero desplazado adecuadamente en B. Para ello, cada *work-item* guardará el valor del elemento de A que represente según sus índices desplazado en B, según la ecuación 5.3.

$$b_{i,j} = a_{i+N,j+N} \tag{5.3}$$

Una vez visto como se define el *kernel* que copia la matriz de datos dentro de la matriz de *padding*, vamos a explicar paso a paso el diseño del *kernel* que calcula la convolución empleando la memoria global. Podemos ver el código en la figura 5.9.

```
7  kernel void convolutionKernel(global realType* Data, global realType* Kernel, global realType* Output) {
8
9      //Get the thread IDs
10     const int globalRow = get_global_id(1); // Row ID of C (0..M)
11     const int globalCol = get_global_id(0); // Col ID of C (0..N)
12
13     //Get the column quantity and accumulator
14     const int nPadding = getSpatialDimSize(Data, 0, 0);
15     const int nOutput = getSpatialDimSize(Output, 0, 0);
16     const int k = getSpatialDimSize(Kernel, 0, 0);
17
18     //Save the accumulator
19     private float acc_total;
20     private float acc_kernel;
21     acc_total = 0;
22     acc_kernel = 0;
23
24     //Iterating to multiply the kernel over the data
25     for(int i=0;i<k;i++){
26
27         for(int j=0;j<k;j++){
28             acc_total += Data[(globalRow+i)*nPadding + globalCol+j]*Kernel[i*k + j];
29             acc_kernel += Kernel[i*k + j];
30         }
31     }
32
33     //Save the new pixel value on the Output
34     Output[globalRow*nOutput + globalCol] = acc_total/acc_kernel;
35 }
```

Figura 5.9: Kernel de convolución.

1. ***Línea 7: entrada del kernel:*** en este *kernel* también tenemos 3 matrices de entrada, aunque la operación sea diferente. Por un lado, tenemos la matriz de datos, por otro tenemos el *kernel* de convolución y después la matriz de salida, que será donde se almacene el resultado.

2. ***Líneas 10-11: obtener el work-item ID:*** con una llamada a la API de OpenCL, este nos devuelve los identificadores del *work-item* que está ejecutando esa instancia del *kernel*. De esta forma, podemos paralelizar los cálculos, ya que en función del *work-item* que esté ejecutando el *kernel*, podremos saber a qué fila y columna, así como en qué posición tiene que guardar el valor de salida.

3. ***Líneas 14-16: obtener Dimensiones Columna:*** en este caso, necesitamos conocer el número de columnas de las tres matrices implicadas en los cálculos, para poder recorrerlas adecuadamente. Como ya se expuso en el apartado 5.5, la memoria es unidimensional, y las matrices están definidas sucediendo sus filas una detrás de otra. Por tanto, para poder acceder a cada valor a partir de sus índices, necesitamos conocer el número de columnas de cada una de las matrices.

4. ***Lineas 19-22: iniciar el acumulador:*** en este *kernel* vamos a necesitar dos acumuladores diferentes. Uno va a ir calculando el sumatorio de todos los productos, para obtener el dividendo del valor final. El otro, va a ir calculando el sumatorio de los datos únicamente del *kernel*, de forma que sirva como factor de normalización. Esto es necesario como se ve en la fórmula de la ecuación 4.9.

5. ***Líneas 25-31: calcular el sumatorio de la convolución y del normalizador:*** para calcular la convolución, dados los valores conocidos, se va a hacer con bucles en un algoritmo tipo burbuja. Estos van a recorrer los valores en el rango en el que la operación tiene lugar, suponiendo que superponemos el *kernel* sobre el elemento que queremos convolucionar, y aplicamos el sumatorio del producto de los pares de números que se superponen. Este valor se calcula con la siguiente ecuación 5.4. Es importante destacar que puede parecer muy diferente a la ecuación de la convolución en 4.9, pero esto se debe a que esta fórmula se está aplicando a una matriz que ya se le ha agregado el *padding* y por tanto ya tiene intrínseco el desplazamiento que en la fórmula original introducía N.

$$r_{i,j} = \sum_{r=1,s=1}^{k} d_{i+r,j+s} c_{r,s} \tag{5.4}$$

Por último, faltaría indicar como se calcula el factor de normalización, dado este por el sumatorio de todos los valores de la matriz de *kernel* de convolución. Este valor c se calcula en el mismo bucle, como se muestra en la ecuación 5.5.

$$c = \sum_{r=1,s=1}^{k} c_{r,s} \tag{5.5}$$

6. ***Línea 34: almacenamiento del resultado ordenado en la matriz de salida:*** por último, y conociendo los índices, almacenamos el resultado final dividiendo los acumuladores como $\frac{r_{j,i}}{c}$ en *Output* de forma ordenada como se ha explicado en el apartado de las dimensiones. En este punto, como cada *work-item* tiene unos índices diferentes, cada uno habrá ido realizando la misma operación sobre diferentes datos y guardándola en la posición adecuada.

Finalmente, vamos a ver el código del *kernel* que emplea memoria local, copiando el *kernel* de convolución antes de proceder con los cálculos. Todas las líneas sobre las que no se haga referencia serán idénticas a lo que se ha detallado para el *kernel* que emplea memoria global. El flujo es similar, pero con ciertas diferencias que podemos ver en la figura 5.10:

```
41  kernel void convolutionLocalKernel(global realType* Data, global realType* Kernel, global realType* Output,
42                                     local realType* Kernel_local) {
43
44      //Get the thread IDs
45      const int globalRow = get_global_id(1); // Row ID of C (0..M)
46      const int globalCol = get_global_id(0); // Col ID of C (0..N)
47
48      const int localRow = get_local_id(1);
49      const int localCol = get_local_id(0);
50
51      //Get the column quantity
52      const int nPadding = getSpatialDimSize(Data, 0, 0);
53      const int nOutput = getSpatialDimSize(Output, 0, 0);
54      const int k = getSpatialDimSize(Kernel, 0, 0);
55
56      //Copy Kernel to Local Memory
57      if(globalRow<k && globalCol<k){
58          Kernel_local[globalRow*k+globalCol] = Kernel[globalRow*k+globalCol];
59      }
60
61      //Save the accumulator
62      private float acc_total;
63      private float acc_kernel;
64      acc_total = 0;
65      acc_kernel = 0;
66
67      //Stop to syncronice all mamory savings to avoid scratch
68      barrier(CLK_LOCAL_MEM_FENCE);
69
70      //Iterating to multiply the kernel over the data
71      for(int i=0;i<k;i++){
72
73          for(int j=0;j<k;j++){
74              acc_total += Data[(globalRow+i)*nPadding + globalCol+j]*Kernel_local[i*k + j];
75              acc_kernel += Kernel_local[i*k + j];
76
77          }
78      }
79
80      //Save the new pixel value on the Output
81      Output[globalRow*nOutput + globalCol] = acc_total/acc_kernel;
82  }
```

Figura 5.10: Kernel de convolución que emplea memoria local.

1. ***Línea 41-42: entrada del kernel:*** en este *kernel* se agrega una nueva matriz de datos, que es la memoria local reservada para almacenar el *kernel* de convolución. Este espacio de memoria está solo reservado y contiene datos basura, y serán los propios elementos del *kernel* los que copien la información desde la memoria global a la memoria local.

2. ***Línea 48-49: obtención de los ID locales:*** en general siempre se han obtenido los ID globales de los *work-items* con los que se trabajaba. En este caso, dado que se va a emplear memoria local, es importante obtener y operar con los identificadores locales. Estos hacen referencia a todos los *work-items* dentro del mismo *work-group*. De esta forma, nos aseguramos que estamos leyendo y escribiendo de forma que esta sección de memoria sea visible para todos nuestros *work-items*. Es importante destacar que para la ejecución de este *kernel* se define un único *work-group* de las mismas dimensiones que la matriz de salida.

3. ***Línea 57-58: copia los datos del kernel de convolución de memoria global a memoria local:*** en este punto, y de forma que la copia se realice completamente en paralelo, se copia el *kernel* de convolución de memoria global a memoria local, de forma que cada *work-item* copie el elemento que coincide con su ID.

4. ***Línea 68: barrera de sincronismo:*** en este punto, antes de empezar a realizar los cálculos, se emplea una barrera como las ya empleada en el algoritmo de la norma-N, con el objetivo de asegurar que la copia de la memoria se ha realizado completamente, evitando leer datos basura.

5. ***Línea 74-75: operación sobre el los datos en memoria local:*** por último, la única diferencia es que para realizar las operaciones se leerán los datos de la copia realizada en memoria local, lo que acelerará los accesos a memoria y reducirá el cuello de botella provocado por que todos los *work-items* tengan que acceder a las mismas zonas de memoria.

5.5.2 Integración de la convolución con OpenCLIPER

Para poder integrar este *kernel* con el framework OpenCLIPER y que sea fácil de usar por cualquier usuario, se define un nuevo Proceso de OpenCLIPER. Este gestionará los métodos necesarios, de forma que se encargará de recibir los datos de entrada, validarlos,

preparar los pasos necesarios y lanzar las ejecuciones, devolviendo la salida al programa del usuario de forma transparente. Gracias a esta clase, se proporciona una capa de abstracción completa al usuario, de forma que solo tiene que preocuparse de configurar el dispositivo, cargar los datos de entrada y crear el proceso, obteniendo en la salida de la ejecución el resultado. Vamos a ver todo el flujo definido dentro del método *init()* y *launch()*, necesario para obtener los datos, validarlos, y ejecutar el *kernel*.

```
16  void Convolution::init() {
17
18      CONV_CERR("CONV::init()\n");
19
20      //Validation on the input
21      check_init_objects(getInput());
22
23      //Calculate the padding depending of the Kernel size
24      padding = getInput()->getSpatialDimSize(1, 0)/2;
25
26      //Condition to use local memory kernel to faster execution
27      localMemory = false;
28      ulong MemorySize = 0;
29      getApp() -> getDevice().getInfo(CL_DEVICE_LOCAL_MEM_SIZE, &MemorySize);
30      //Compare if kernel matrix could be handled on local memory
31      if((4*getInput()->getSpatialDimSize(1, 0)*getInput()->getSpatialDimSize(0, 0) < MemorySize)){
32
33          localMemory = true;
34
35      }
36  }
```

Figura 5.11: Método *"init"* del proceso convolution de OpenCLIPER.

Vamos a comenzar describiendo el método *"init"* que se muestra en la figura 5.11:

1. ***Entrada de parámetros:*** Este método solo requiere como entrada el *kernel* de convolución definido como *Input* del proceso, ya que dentro de la función de inicialización del proceso se va a hacer la evaluación del mismo. En este punto, el preprocesado consistirá en validar que el *kernel* tiene las dimensiones adecuadas, calcular el *padding* que se debe aplicar a la matriz de datos y evaluar si podemos usar el *kernel* de convolución sobre memoria local, o se debe usar el que emplea memoria global por tema de espacio en memoria.

2. ***Línea 21: validación del kernel de convolución:*** En este punto se comprueba que el *kernel* de convolución es un *kernel* válido. Para ello, se verifican las siguientes condiciones:

   - La matriz ha sido inicializada correctamente
   - Se evalúa que sea una matriz cuadrada.
   - Se comprueba que tanto el número de filas como de columnas sea impar.

3. ***Línea 24: cálculo del padding*** N**:** a continuación, se calcula el *padding* N, siendo este el tamaño de la matriz (fila o columna) entre dos y truncado. $N = \lfloor \frac{k}{2} \rfloor$.

4. ***Líneas 27-35: comprobación del uso del kernel de memoria local:*** por último, se verifica si el tamaño del *kernel* puede ser almacenado en la memoria local, o este es demasiado grande. Si bien la memoria local es rápida, también es pequeña, y dependiendo de cada dispositivo su tamaño cambia. Por ello, a través de una de las funcionalidades que OpenCLIPER extiende de OpenCL, se hace una llamada a la API de OpenCL para obtener el tamaño de la memoria del dispositivo que se ha configurado para realizar la ejecución, evaluando el *kernel* que se debe emplear, y almacenando la condición en una variable privada perteneciente a la propia clase *Convolution*.

Por otro lado, tenemos el método *"launch"*, que ya con toda la información de entrada, será el encargado de realizar todo el proceso de a partir de los datos calcular el *padding* y reutilizar esta matriz de *padding* como entrada al *kernel* de convolución, obteniendo la salida de la operación y devolviéndola al usuario. Vemos el código del proceso para la gestión de la ejecución del *kernel* de *padding* en la figura 5.12.

```
38  void Convolution::launch() {
39      try {
40          CONV_CERR("CONV::launch\n");
41
42          auto pLP = std::dynamic_pointer_cast<LaunchParameters>(pLaunchParameters);
43          if (!pLP) pLP = std::unique_ptr<LaunchParameters>(new LaunchParameters());
44
45          //Security checks on the inputs and outputs.
46          check_launch_objects(getInput(),pLP->Info,getOutput());
47
48          ////////////Calculating Padding Matrix ////////////
49
50          //Step 2: create temp matrix to handle padding data
51          auto pZeroData = new std::vector<realType>((pLP->Info->getSpatialDimSize(0, 0) +
52                                  (2*padding))*(pLP->Info->getSpatialDimSize(1, 0) + (2*padding)),0);
53          auto pZeroDims = new std::vector<dimIndexType>({ (pLP->Info->getSpatialDimSize(0, 0) +
54                                  (2*padding)), (pLP->Info->getSpatialDimSize(1, 0) + (2*padding)) });
55          auto pDataPadding = std::make_shared<XData>(getApp(), pZeroDims, pZeroData);
56
57          kernel = getApp()->getKernel("paddingKernel");
58
59          //Set input and output OpenCL buffers on device memory
60          cl::Buffer* pInBuf1 = pLP->Info->getDeviceBuffer();
61          cl::Buffer* pOutBuf = pDataPadding->getDeviceBuffer();
62
63          //Set kernel parameters
64          kernel.setArg(0, *pInBuf1);
65          kernel.setArg(1, *pOutBuf);
66          kernel.setArg(2, padding);
67
68          //Set kernel work items size: number of pixels to process is data weight x hight
69          cl::NDRange globalSizePadding = cl::NDRange(pLP->Info->getSpatialDimSize(0, 0),
70                                                      pLP->Info->getSpatialDimSize(1, 0));
71          //Execute kernel
72          queue.enqueueNDRangeKernel(kernel, cl::NullRange, globalSizePadding, cl::NDRange(), NULL, NULL);
73
```

Figura 5.12: Método *"launch"*, sección de gestión del kernel de *padding* del proceso convolution de OpenCLIPER.

1. ***Líneas 42-43: lectura de parámetros de lanzamiento:*** por herencia, todo proceso en OpenCLIPER que herede de la clase *Process* tiene espacio para un conjunto de datos de entrada, denominado *input*, y otro para los datos de salida llamado *output*. En este caso no es suficiente, ya que nuestros datos de entrada son dos matrices. La forma más sencilla para pasarle a este método nuestra segunda matriz de datos, es aprovechar los parámetros de lanzamiento que pueden ser maleables para contener todos los datos necesarios. Para ello, se define una estructura de datos adaptada como variable pública de la clase, y se aprovechan los propios métodos ya predefinidos por la clase padre para inicializar los valores. En este caso, el *input* será el *kernel* de convolución, el output será la matriz resultado, y como parámetros de lanzamiento se la pasará la matriz de datos.

2. ***Línea 46: validación de las entradas:*** una vez dentro del método, ya con los parámetros de entrada inicializados y accesibles por el mismo, se validan para comprobar que son correctos para realizar la operación. Las comprobaciones se describen a continuación. Es importante indicar que si alguna de estas condiciones no se cumple, el programa finalizará con un error concreto, que le indicará al usuario el motivo del fallo para que pueda subsanarlo, si es posible:

   - Comprobar que la matriz de datos y la matriz resultado han sido definidas correctamente y no están vacías.
   - Validar que todas las matrices contienen el mismo tipo de dato.
   - Evaluar que la matriz *kernel* tiene dimensiones como máximo iguales a la matriz de datos.
   - Comprobar que la matriz de salida tiene las mismas dimensiones que la matriz de entrada.

   Con este punto cabe destacar, que no es necesario verificar que la matriz de datos y la matriz de resultado sean diferentes. De hecho, aunque está pensado para que sean diferentes, el usuario en su programa puede sobrescribir los datos de entrada con la matriz de salida, y el programa funcionará correctamente. Esto se puede hacer en este caso ya que estamos realizando un paso intermedio donde calculamos la matriz de padding, por lo que el cálculo de la matriz de salida no se hace sobre la matriz original de los datos de entrada, sino sobre la matriz de *padding* intermedia.

3. ***Líneas 51-55: creación de la matriz auxiliar para contener la matriz de padding:*** En este punto, crearemos una matriz vacía de tipo "XData", preparada para contener la matriz de los datos con el padding. Para ello, esta matriz debe tener de dimensiones $(m + N, n + N)$, siendo las dimensiones de $Data = (m, n)$ y N el valor calculado para el *padding* en los parámetros de inicialización. Es importante que esta matriz esté inicializada a 0, ya que la operación consistirá en escribir sobre esta la matriz de datos desplazada adecuadamente, si necesidad de escribir los datos del marco.

4. ***Líneas 57-69: obtención de buffers y kernel, inicialización de argumentos y dimensionamiento de work-items:*** de forma similar a todas las ejecuciones de un *kernel*, antes de su encolado se deben crear los buffers OpenCL que van a contener las variables de entrada, así como definir las dimensiones que tendrá el conjunto de *work-items*. En este caso, estas dimensiones serán las de la matriz de datos, ya que no necesitamos copiar más valores desde una matriz a otra.

5. ***Línea 72: encolado del kernel padding:*** una vez todos los parámetros listos, encolamos el *kernel* de padding, obteniendo como salida la matriz de datos con padding, lista para realizar la convolución. Esta matriz de salida se convertirá en la entrada del siguiente *kernel*, que ya será el que realice la convolución.

En este punto, ya tenemos copiada sobre la matriz auxiliar con *padding* la matriz de datos. Con la información ya preparada en memoria, y en función de el *kernel* que se vaya a emplear para calcular la convolución, se configurarán y realizarán las llamadas adecuadas para la ejecución del *kernel*. Vamos a comenzar por describir el lanzamiento del *kernel* que opera sobre memoria global, y después mostraremos el *kernel* que opera sobre memoria local, detallando las diferencias entre ambos. Podemos observar el código en la figura 5.13.

```
72              if(localMemory){
73                  kernel = getApp()->getKernel("convolutionLocalKernel");
74                  //Set input and output OpenCL buffers on device memory
75                  cl::Buffer* cInBuf1 = pDataPadding->getDeviceBuffer();
76                  cl::Buffer* cInBuf2 = getInput()->getDeviceBuffer();
77                  cl::Buffer* cOutBuf = getOutput()->getDeviceBuffer();
78
79                  //Set kernel parameters
80                  kernel.setArg(0, *cInBuf1);
81                  kernel.setArg(1, *cInBuf2);
82                  kernel.setArg(2, *cOutBuf);
83                  kernel.setArg(3, cl::Local(sizeof(float)*getInput()->getSpatialDimSize(1, 0)*
84                                             getInput()->getSpatialDimSize(0, 0)));
85
86                  //Set kernel work items size: number of pixels to process is data weight x hight
87                  cl::NDRange globalSizeConvolution = cl::NDRange(getOutput()->getSpatialDimSize(0, 0),
88                                                                  getOutput()->getSpatialDimSize(1, 0));
89                  //Execute kernel
90                  queue.enqueueNDRangeKernel(kernel, cl::NullRange, globalSizeConvolution, cl::NDRange() , NULL, NULL);
91                  //Wait all execution to finish on the queue
92                  queue.finish();
93
94              }else{
95                  kernel = getApp()->getKernel("convolutionKernel");
96                  //Set input and output OpenCL buffers on device memory
97                  cl::Buffer* cInBuf1 = pDataPadding->getDeviceBuffer();
98                  cl::Buffer* cInBuf2 = getInput()->getDeviceBuffer();
99                  cl::Buffer* cOutBuf = getOutput()->getDeviceBuffer();
100
101                 //Set kernel parameters
102                 kernel.setArg(0, *cInBuf1);
103                 kernel.setArg(1, *cInBuf2);
104                 kernel.setArg(2, *cOutBuf);
105
106                 //Set kernel work items size: number of pixels to process is data weight x hight
107                 cl::NDRange globalSizeConvolution = cl::NDRange(getOutput()->getSpatialDimSize(0, 0),
108                                                                 getOutput()->getSpatialDimSize(1, 0));
109                 //Execute kernel
110                 queue.enqueueNDRangeKernel(kernel, cl::NullRange, globalSizeConvolution, cl::NDRange() , NULL, NULL);
111             }
112
113         }
```

Figura 5.13: Método *"launch"*, sección de gestión del kernel de convolución del proceso convolution de OpenCLIPER.

1. ***Línea 72: selección del kernel global o local:*** al igual que en el proceso de multiplicación, descargamos al usuario de la carga de decidir el *kernel* más adecuado. De forma análoga, en este punto el proceso convolución elige si usar el *kernel* de memoria global o el de memoria local. Como ya ha calculado en la método de inicialización si el *kernel* de convolución cabe en la memoria local, utilizará el de memoria local siempre que sea posible contener el *kernel*, y no lo usará cuando el *kernel* exceda el tamaño máximo de memoria local disponible.

2. ***Líneas 75-87: obtención de buffers, configuración de argumentos del kernel y tamaño de work-items:*** una vez obtenido el *kernel* que se va a utilizar, se obtienen todos los datos previos para la ejecución del mismo. Como siempre, los buffers de entrada y el tamaño de los *work-items*. Cabe destacar que, en caso de memoria local, tenemos un parámetro más, que es la matriz sobre memoria local vacía. Esta reserva de memoria no se lleva a cabo generando un buffer, sino con una sentencia de la API de OpenCL que nos permite indicar que esa entrada es un espacio reservado de memoria local, de forma que dentro de la función *kernel* exista

el puntero que apunte al espacio de memoria reservado.

3. ***Línea 90: encolado del kernel de convolución:*** Finalmente, con todo listo y en orden, ponemos en cola la ejecución del *kernel* de convolución, obteniendo en la salida nuestra matriz resultado de la convolución de los datos con el *kernel* de convolución.

Por último, detallar que podemos ver que lanzar el *kernel* que emplea memoria global es completamente análogo, pero simplemente hay que configurar un argumento menos porque no hace falta hacer la reserva en memoria local. A continuación, para concluir el apartado de diseño, se va a mostrar un ejemplo de programa que debería diseñar el usuario final para ejecutar una convolución empleando el framework OpenCLIPER.

5.5.3 Ejemplo de programa para la convolución

Una vez expuesto como se define el *kernel* y como es el proceso que gestiona este *kernel*, vamos a ver un ejemplo paso a paso de cómo debería definir un usuario su programa para hacer uso de este proceso. Este programa es similar para todos los procesos implementados en este trabajo, así como para el resto de procesos ya existentes en el framework. Esto va a evidenciar la gran ventaja que supone el uso de OpenCLIPER, y todas las facilidades que da al usuario para que este pueda lanzar el algoritmo sobre los datos que quiera y sin tener que conocer casi nada de OpenCL. Este ejemplo se muestra en la figura 5.14.

1. ***Líneas 23-33: instanciar CLapp y todos los parámetros de OpenCL:*** en OpenCLIPER toda la aplicación, así como la configuración y elección del dispositivo OpenCL y otras muchas funcionalidades son gestionadas a partir de la clase llamada CLapp. El primer paso para cualquier usuario, es crear una instancia de esta clase, indicándole los datos necesarios para que sea capaz de saber que dispositivo debe emplear. Esto se lleva a cabo en una serie de pasos en los que se definen tanto la plataforma como el dispositivo, a partir de una configuración inicial. Esta configuración es un punto en el que OpenCLIPER define varios métodos y parámetros y su gestión es un punto avanzado. Para las necesidades de nuestro programa, simplemente lo definiremos por defecto y dejaremos que sea OpenCLIPER el que elija el mejor dispositivo que detecte en el sistema.

```
22          // Step 0: get a new OpenCLIPER app, initialize computing device and load OpenCL kernel(s)
23          CLapp::PlatformTraits platformTraits;
24          CLapp::DeviceTraits deviceTraits;
25
26          ProgramConfig* pProgramConfig = new ProgramConfig(argc, argv);
27          auto pConfigTraits = std::dynamic_pointer_cast<ProgramConfig::ConfigTraits>(
28          pProgramConfig->getConfigTraits());
29
30
31          deviceTraits = pConfigTraits->deviceTraits;
32          platformTraits = pConfigTraits->platformTraits;
33          pCLapp = CLapp::create(platformTraits, deviceTraits);
34
35          // Step 1: Generate de test data (MxN)*(KxK)
36          const uint m = 513;
37          const uint n = 513;
38          const uint k = 3;
39
40          //A data matrix full for real type
41          auto pData = new std::vector<realType>(m*n);
42          auto pDims = new std::vector<dimIndexType>({ n, m });
43
44          for(int i=0;i<m*n;i++){
45              pData[0][i] = static_cast <float> (rand()) / (static_cast <float> (RAND_MAX/256));
46          }
47
48          auto pIn = std::make_shared<XData>(pCLapp, pDims, pData);
49
50          //B kernel matrix full for real type
51          auto pKernel = new std::vector<realType>(k*k,0);
52          auto pKernelDims = new std::vector<dimIndexType>({ k, k });
53
54          for(int i=0;i<k*k;i++){
55              pKernel[0][i] = static_cast <float> (rand()) / (static_cast <float> (RAND_MAX/256));
56          }
57
58          auto pKernelIn = std::make_shared<XData>(pCLapp, pKernelDims, pKernel);
59
60          //Step 2: create output with correct size
61          auto pOut = std::make_shared<XData>(pCLapp, n, m, TYPEID_REAL);
62
63          // Step 3: create new process, bound to our CL app and set its input/output data sets
64          auto pProcess = Process::create<Convolution>(pCLapp);
65
66          pProcess->setInput(pKernelIn);
67          pProcess->setOutput(pOut);
68
69          pProcess->init();
70
71          auto launchParms = std::make_shared<Convolution::LaunchParameters>(pIn);
72          pProcess->setLaunchParameters(launchParms);
73
74          pProcess->launch();
75
76          pOut->matlabSave("Output.m");
77
78      }
```

Figura 5.14: Programa principal para la ejecución del algoritmo.

2. ***Línea 36-61: creación de datos de entrada XData:*** Para almacenar los datos tanto de entrada como de salida, se deben crear instancia de la clase XData que mantengan esta información. Estas instancias se pueden crear a partir de un vector, dando como parámetros el vector, las dimensiones y el objeto CLapp creado. Pero también tienen constructores para leer una imagen de diversos formatos como TIFF y PNG. Para crear la instancia del objeto XData donde se almacenará la salida, hay que proporcionale la CLapp las dimensiones y el tipo de dato de los elementos de la matriz. En los datos de entrada, el tipo de dato de XData es igual al del vector que

contiene la información. En este ejemplo, se rellenan las entradas con datos aleatorios de 1 Byte de tamaño, como la información de un pixel de una fotografía en blanco y negro.

3. ***Línea 64: instanciar el proceso Convolution:*** esta parte es muy sencilla, ya que solo es necesario llamar al constructor del proceso *Convolution*, sin argumentos.

4. ***Líneas 66-67: set de input y output:*** para configurar el *input* y *output* por defecto se debe sobre el proceso creado llamar a los métodos *"setter"* definidos para este cometido, pasándole como valor la instancia XData ya creada para tal efecto.

5. ***Línea 69: llamar a la función Init():*** en este caso, ya que no necesitamos parámetros de inicialización, solo es necesario llamar a este método de la clase *Convolution*. Con el input ya configurado, este método realizará los cálculos de preprocesado necesarios definidos en el apartado 5.5.2.

6. ***Líneas 71-72: creación de Launch Parameters:*** una vez llamado al método de inicialización, vamos a crear una instancia de los parámetros de lanzamiento, pasando por valor la matriz de datos que nos faltaba, la matriz de datos. Tras esto, sobre la instancia del proceso, se lanza el "setter.ªdecuado ya definido en la clase *Process* para este fin, pasándole la instancia de los parámetros de lanzamiento ya inicializada.

7. ***Línea 74: llamar a la función Launch():*** ya tenemos todos los datos configurados y accesibles por nuestra instancia del proceso *Convolution*. En este punto, solo tenemos que llamar al método *Launch*, sin pasarle ningún parámetro. Cuando finalice la ejecución de este, ya tendremos en la matriz *pOut* el resultado de la ejecución.

8. ***Línea 76: almacenamiento de Output en un fichero:*** Por último, tenemos diferentes métodos que nos proporciona *XData* para almacenar nuestro resultado en la memoria persistente, en forma de fichero, en un fichero MAT, etc..., además de permitirnos mostrar el resultado por pantalla. También existe la posibilidad de que esta matriz de salida sea la futura entrada de otra ejecución de este u otro *kernel*, para lo cual ya estaría completamente disponible y con el formato adecuado.

Como podemos observar, este programa es prácticamente relativamente sencillo y no requiere conocer ningún detalle sobre OpenCL u la gestión de memoria necesaria. Esta

es una de la ventajas más importantes ya mencionadas del uso del framework OpenCLIPER. El usuario no necesita conocer en profundidad los detalles de la implementación del *kernel* o el uso de OpenCL, sino que conociendo la parte básica de OpenCLIPER, y con un programa previo de ejemplo, puede entender a la perfección como construir el programa u obtener sus resultados, con muchas facilidades y sin necesidad de profundizar en conocimientos profundos de programación.

Capítulo 6

Resultados

Una vez revisado de forma detallada como se lleva a cabo la implementación de los algoritmos y su integración con OpenCLIPER, se van a realizar unas pruebas de rendimiento. Con ellas se pretende evaluar los tiempos de ejecución de los algoritmos y su relación con el tamaño de los datos de entrada. Además, se realizará una comparativa entre los algoritmos usando la CPU frente a la GPU para ver cómo se comportan ambos dispositivos. También se evaluarán los algoritmos frente a un programa de referencia, en este caso Matlab, para poder medir la el rendimiento de los diferentes algoritmos y plantear posibles mejoras en función de los resultados.

Para comenzar, se van a detallar las características del dispositivo CPU y GPU empleados en las simulaciones. Estas características, como el número máximo de hilos en paralelo o el tamaño de la memoria local serán fundamentales a la hora de analizar y entender los resultados, pudiendo justificar las curvas y localizar los puntos fuertes de débiles de los algoritmos implementados en función del dispositivo.

Característica	**CPU**	**GPU**
Modelo	Intel(R) i7-4790 CPU 3.60GHz	NVIDIA GeForce RTX 2080 Ti
Versión OpenCL	OpenCL 1.2 CUDA	OpenCL 2.1
Unidades de computación máximas	68	8
Frecuencia de reloj	1545 MHz	3600 MHz
Tamaño de memoria local	48 KB	32 KB
Tamaño de *work-items* definible	1024x1024x64	8192x8192x8192

Tabla 1: Dispositivos de ejecución empleados en las simulaciones de evaluación del rendimiento.

Como se puede ver en la tabla 1, las CPU y GPU son muy diferentes en cuanto a características. Principalmente, la diferencia más importantes a la hora de medir el

rendimiento de ejecución de los *kernels* serán las unidades de computación y la frecuencia de reloj. Por un lado, vemos que la GPU puede ejecutar en paralelo más de ocho veces más hilos de cálculo, los que le permite procesar en paralelo muchos más cálculos en función del *kernel*. Sin embargo, la CPU tiene una frecuencia de reloj más del doble de rápida, lo que implica que en situaciones en las que se produzcan cuellos de botella o puntos en los que os hilos no puedan aprovechar al máximo la paralelización, la CPU será más rápida. En nuestro trabajo, estos puntos críticos serán aquellos en los que diferentes hilos tengan que realizar lecturas o escrituras de las mismas zonas de memoria, ya que estas operaciones no se pueden realizar en paralelo si es está implicada la misma zona de memoria. Como se podrá estudiar en el apartado siguiente, estas particularidades pueden tener un gran impacto en el rendimiento de los *kernels*.

A continuación, antes de entrar en el análisis de estos resultados, es importante reflejar de forma detallada cómo se han hecho las pruebas de rendimiento, qué se mide, cómo se toman los datos y cómo se procesan, de forma que se obtengan resultados estadísticamente fiables. Comenzando por qué tiempos se miden, en todos los casos se miden los tiempos de ejecución del método *init* y *launch* completo de ambos procesos. Se toma este tiempo porque el algoritmo implica más cosas que la simple ejecución del *kernel* y no se puede obviar todas las parte de creación y configuración de variables necesaria. Además, esta parte previa supone un tiempo fijo siempre similar que no influye en la comparativa entre los diferentes dispositivos y tamaños de matrices. Por otro lado, se debe tener en cuenta que el *kernel* de convolución implica la ejecución de dos *kernels*, tanto para el cálculo del *padding* como para la convolución en sí, y la necesidad de este flujo de nuestro algoritmo debe tenerse en cuenta para llevar a cabo una comparación justa. De hecho, como veremos más adelante, combinar los dos *kernels* de convolución en uno de forma que se convolucione con un desplazamiento "al vuelo" que haga el trabajo del *padding* supondría quizás una mejora, pero sería un algoritmo diferente y evaluar solo la ejecución del *kernel* lo pondría en desventaja de forma injusta. Por esto se mide el tiempo del método *init* mas el método *launch* y no solo la ejecución de los *kernels* de forma aislada.

Además, se debe definir el número de simulaciones y el tratamiento estadístico de los datos que se va a llevar a cabo para obtener valores consistentes y que sean válidos a la hora de comparar. Esta parte es fundamental teniendo en cuenta que estamos midiendo rendimientos de un sistema de computación, en el que puede haber gran variabilidad de los

datos debido a múltiples factores, como puede ser la gestión de procesos que lleve a cabo la CPU, que en un momento determinado el sistema se ponga a ejecutar procesos pesados con alto prioridad por algún motivo, que el sistema cambie el modo de rendimiento, que cambie la temperatura y muchos otros. Por ello se lanza una cantidad alta de ejecuciones, y se han realizado todas las pruebas en un periodo corto del tiempo. De esta forma, se busca que los datos sean lo más fiables posibles.

En cuanto a los datos, se ha determinado que se lanzarán 300 ejecuciones para cada prueba. Sobre estos datos, se aplicará un percentil superior e inferior de un 5 %, eliminando el 10 % de los datos más alejados de la mediana. Por un lado, se emplea un número alto de ejecuciones con el objetivo de que la distribución de los datos de rendimiento sea normal. Además, para eliminar posibles outliers debido a estas sobrecargas puntuales en el sistema es para lo que se aplica este percentil del 5 %. Así, una vez con todas las ejecuciones y los datos limpios de outlayers, se calcula la media, que será el valor que se considerará el tiempo promedio de ejecución de ese algoritmo.

A continuación, vamos a estudiar la distribución de varias simulaciones de diferentes algoritmos para ciertos datos de entrada, comprobando que sea una distribución normal, así como analizando sus características. Para realizar la comprobación de que los datos sigan una distribución normal, se ha utilizado el test de Anderson–Darling disponible en Matlab utilizando la función *adtest()* [18]. Las diferentes muestras utilizadas como referencia han pasado el test de forma exitosa, por lo que se considera que con estos parámetros las simulaciones siguen una distribución normal.

Como se puede ver en la figura 6.1, el conjunto de datos de la simulación sigue una distribución normal para este número de ejecuciones. Esto ejemplifica que la medida de la media del tiempo de ejecución es suficiente para evaluar el rendimiento de los algoritmos. Vamos a ver otra muestra de otro algoritmo para comprobar que sea consistente.

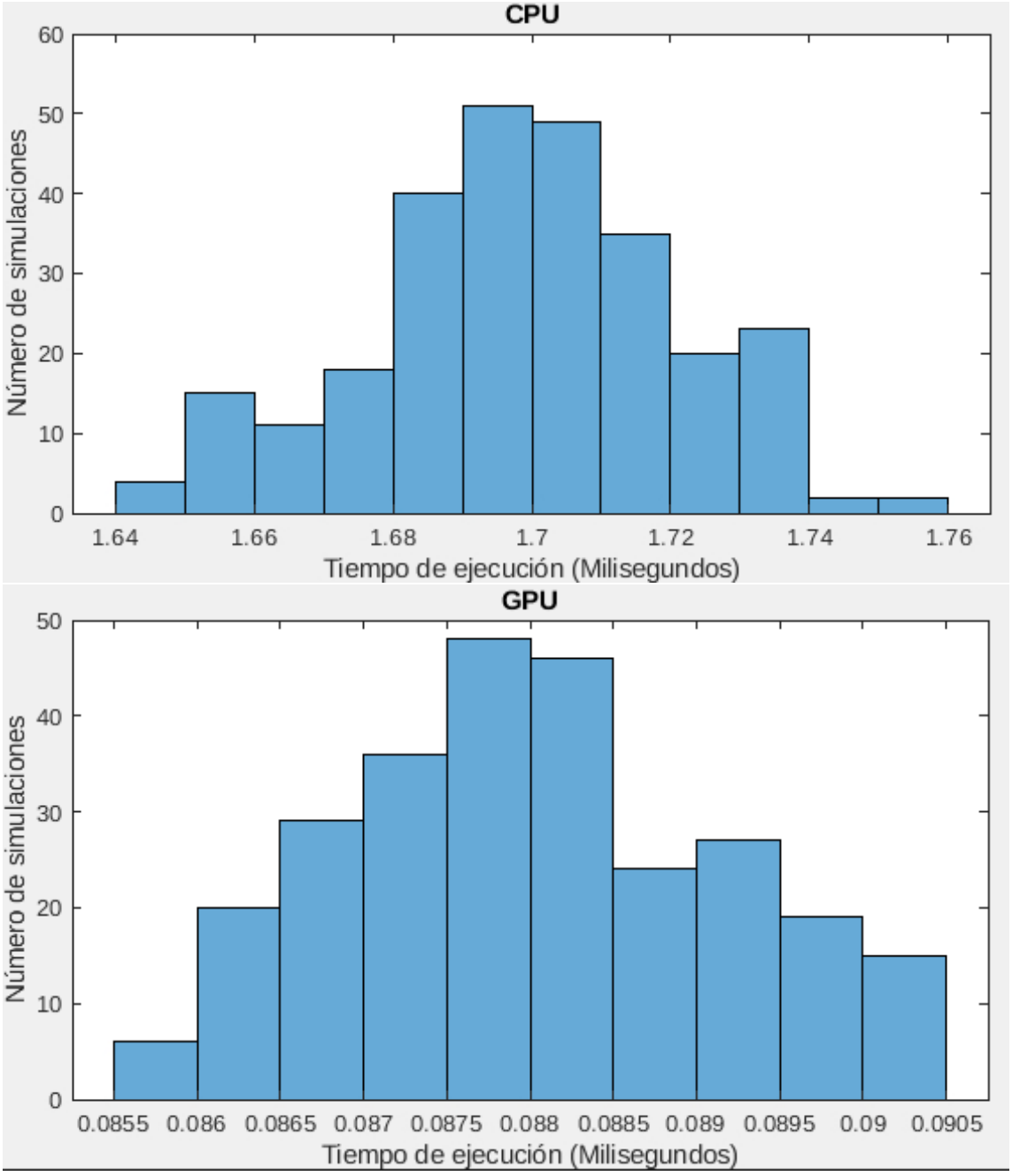

Figura 6.1: Histograma de la simulación de rendimiento para la CPU(arriba) y GPU(abajo) del algoritmos de multiplicación para matrices cuadradas de 256x256.

Un detalle importante que se puede sacar de los datos es la diferencia de varianza entre las GPU y CPU de forma consistente en todas las simulaciones. Para la figura 6.1, La media de la CPU en esa simulación era 1.69 milisegundos por ejecución, con una varianza de $3 \cdot 10^{-3}$. Sin embargo, para la GPU, la media de tiempo de por ejecución era de $8 \cdot 10^{-2}$ milisegundos con una varianza de $1,6 \cdot 10^{-6}$. En términos relativos, esto implica que para

la CPU la varianza era $2,9 \cdot 10^{-4}$ sobre la media, mientras que para la GPU la varianza suponía el $1,5 \cdot 10^{-5}$ sobre la media. Esto nos indica que la CPU tiene una varianza más alta entre diferentes ejecuciones que la GPU. Esto se acentúa más si estudiamos los datos de otros algoritmos, por ejemplo para el algoritmo de la norma-N que podemos ver el la figura 6.2.

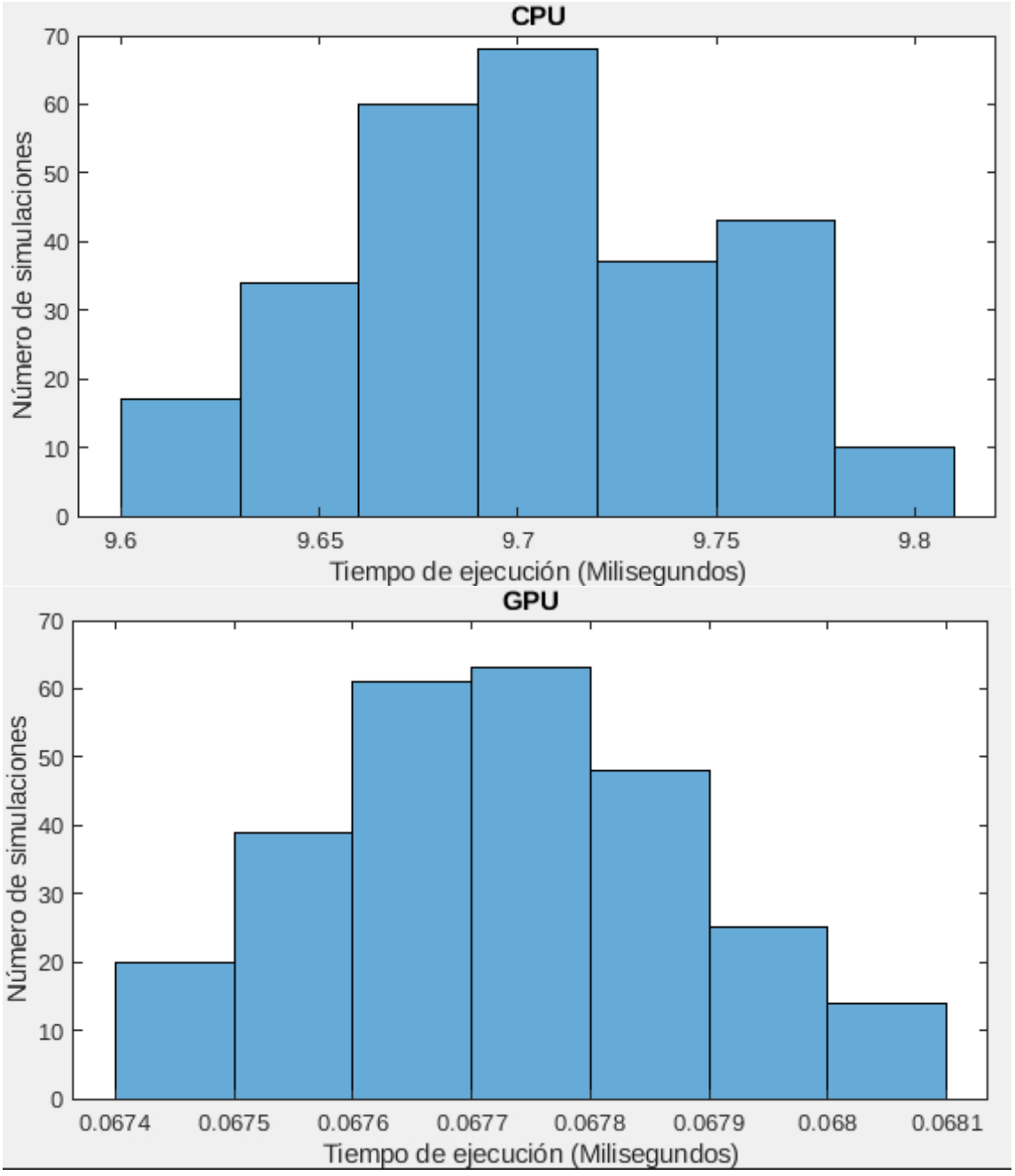

Figura 6.2: Histograma de la simulación de rendimiento para la CPU(arriba) y GPU(abajo) del algoritmo de norma-N para un vector de 10^6 elementos con una norma de 20.

En este caso del cálculo de la norma-N, para la CPU la media del tiempo de ejecución es de $9,7$ milisegundos por ejecución, mientras que la varianza es $3 \cdot 10^{-3}$. Si tomamos los datos de la GPU, la media es de $6,7 \cdot 10^{-2}$ y la varianza es de $2,4 \cdot 10^{-8}$. Esto supone que mientras que la varianza respecto a la media para al CPU es de $2{,}1 \cdot 10^{-3}$, para la GPU este valor es de $3,6 \cdot 10^{-7}$. Esta diferencia observable entre las varianzas de las simulaciones entre la CPU y la GPU se debe a que la GPU es un dispositivo completamente dedicado, mientras que la CPU se encarga de la ejecución de otros procesos, como la gestión del sistema operativo, etc. Esto muestra otra de las grandes ventajas de usar la GPU para los cálculos, ya que al ser un dispositivo dedicado, puede emplearse de forma exclusiva a la ejecución de los algoritmos. En contraposición, aunque la CPU se encuentre con la mínima carga, siempre va a tener una carga debida a la gestión del sistema operativo u otras tareas. De hecho, si durante una ejecución de nuestro algoritmo la CPU diera prioridad a otros procesos como una actualización automática del sistema operativo, el tiempo de ejecución de nuestro proceso podría aumentar enormemente, mientras empleando la GPU nunca existirán estos problemas siempre que no lancemos muchos procesos al mismo tiempo sobre la misma.

Una vez estudiados los datos y configurados los parámetros de simulación, en los siguientes apartados se van a estudiar los resultados de rendimiento obtenidos para los diferentes algoritmos, ejecutados sobre la CPU y GPU, tanto con el código diseñado como en Matlab. Se van a hacer barridos de los diferentes parámetros clave, como tamaños de matrices y vectores o valores de la potencia, de forma que se pueda estudiar la evolución de los tiempos de ejecución según cambian estos parámetros. Esto nos permitirá detectar fortalezas y debilidades del diseño, y proponer mejoras para poder mejorar la eficiencia de los mismos.

6.1 Rendimiento del algoritmo de norma-N

Una vez estudiados los datos, vamos a mostrar los análisis de rendimiento para los diferentes algoritmos, comenzando con el algoritmo que calcula la norma-N de un vector. Para este caso, se van a llevar a cabo dos pruebas diferentes. En la primera, se va a mantener fija la norma, y se va a ir modificando el tamaño del vector de datos, para ver como cambia el tiempo de procesamiento. Tras esta prueba, se fijará un tamaño de vector y

se irá modificando la norma, para ver como influye en el tiempo de procesamiento. Ambas pruebas se van a realizar empleando tanto el *kernel* que opera sobre números reales como el que opera sobre datos complejos.

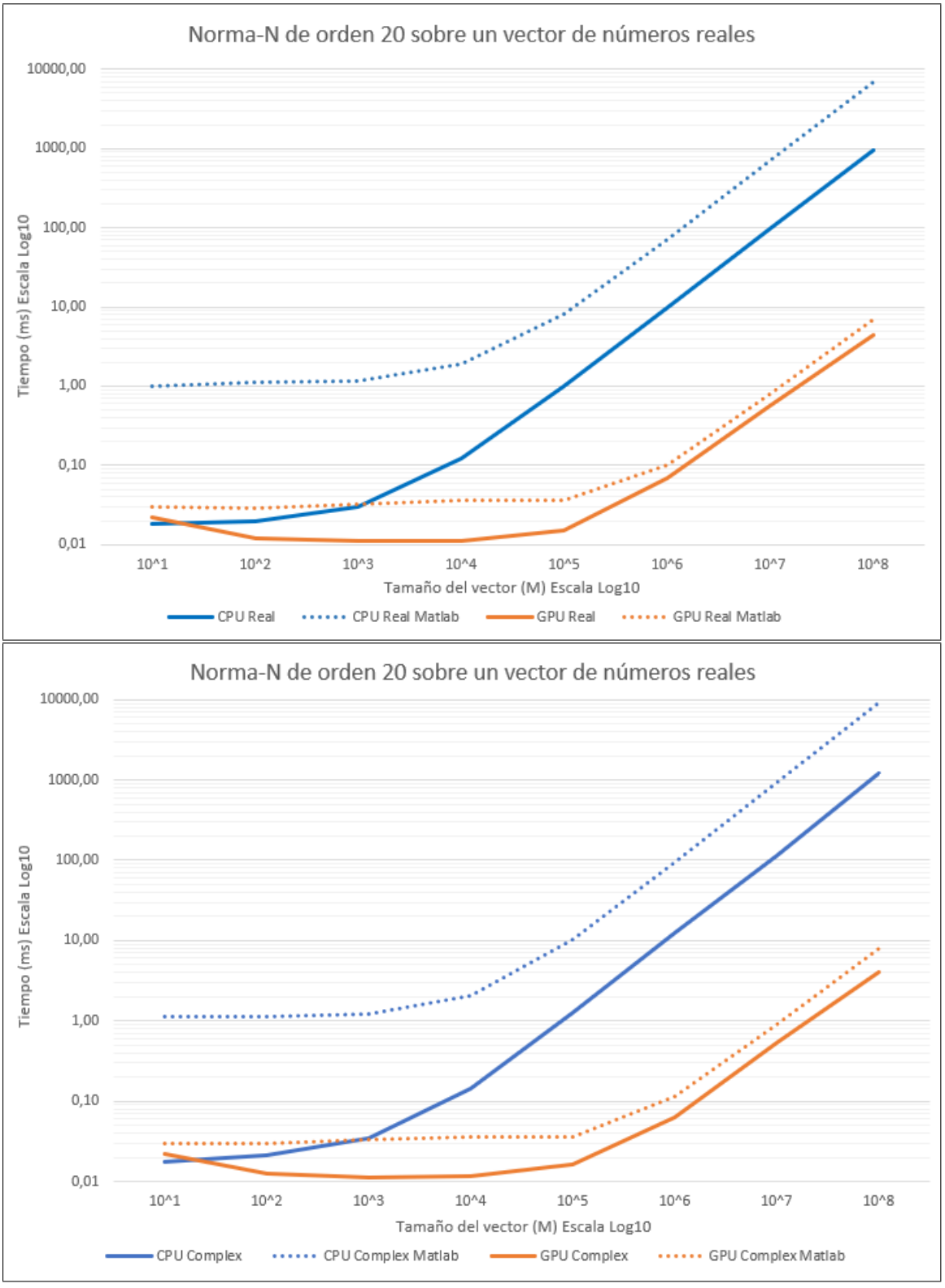

Figura 6.3: Curvas de rendimiento en función del tamaño del vector con norma fija 20 para el algoritmo de cálculo de la norma-N. Arriba para elementos reales y debajo para elementos complejos. Curvas en color naranja liso para la GPU, curva en naranja punteado para la GPU en Matlaba, azul liso para la CPU y azul punteado para la CPU en Matlab.

Como podemos observar en la figura 6.3, la GPU tanto en nuestro algoritmo como para Matlab tiene un mejor rendimiento que la CPU. Si nos fijamos en la comparativa entre la GPU y la CPU podemos ver varios detalles destacables:

- Para tamaños muy pequeños del vector de datos destaca la CPU ya que tiene menos hilos pero son el doble de rápidos. Vemos que a partir de que el vector tiene más de 25 elementos aproximadamente, la GPU comienza a ser más rápida.
- A partir de 10^3 elementos, la curva de la CPU comienza a crecer de forma exponencial, según aumenta el tamaño del vector. En cambio, la curva de la GPU no comienza a tener este crecimiento lineal hasta 10^6.
- La relación entre las curvas es similar independientemente de calcular la norma-N para un vector de números reales o de números complejos. Esto nos indica que la operación de calcular el módulo del complejo, que es la diferencia entre los dos algoritmos, apenas tiene impacto a nivel de carga computacional en comparación al resto de operaciones.

En cuanto a la comparativa de nuestro algoritmo frente a el mismo cálculo realizado con Matlab, podemos ver que nuestro diseño es muy competitivo. Tanto para la operación con CPU y con GPU, nuestro algoritmo siempre ofrece mejores tiempos de procesamiento. Donde notamos una mayor diferencia fijándonos en las curvas de GPU es cuando el tamaño del vector de datos es pequeño, donde Matlab tiene un tiempo mínimo constante hasta que el vector tiene 10^5 elementos. Esto se debe a que Matlab es un lenguaje interpretado, lo que implica un tiempo mínimo mayor que con nuestro algoritmo compilado. Este detalle será consistente, y se podrá apreciar en la comparativa de rendimiento de todos los algoritmos revisados en este capítulo.

A continuación, vamos a estudiar como se comporta el algoritmo cuando definimos un tamaño fijo de vector de 10^5 elementos y vamos variando la norma.

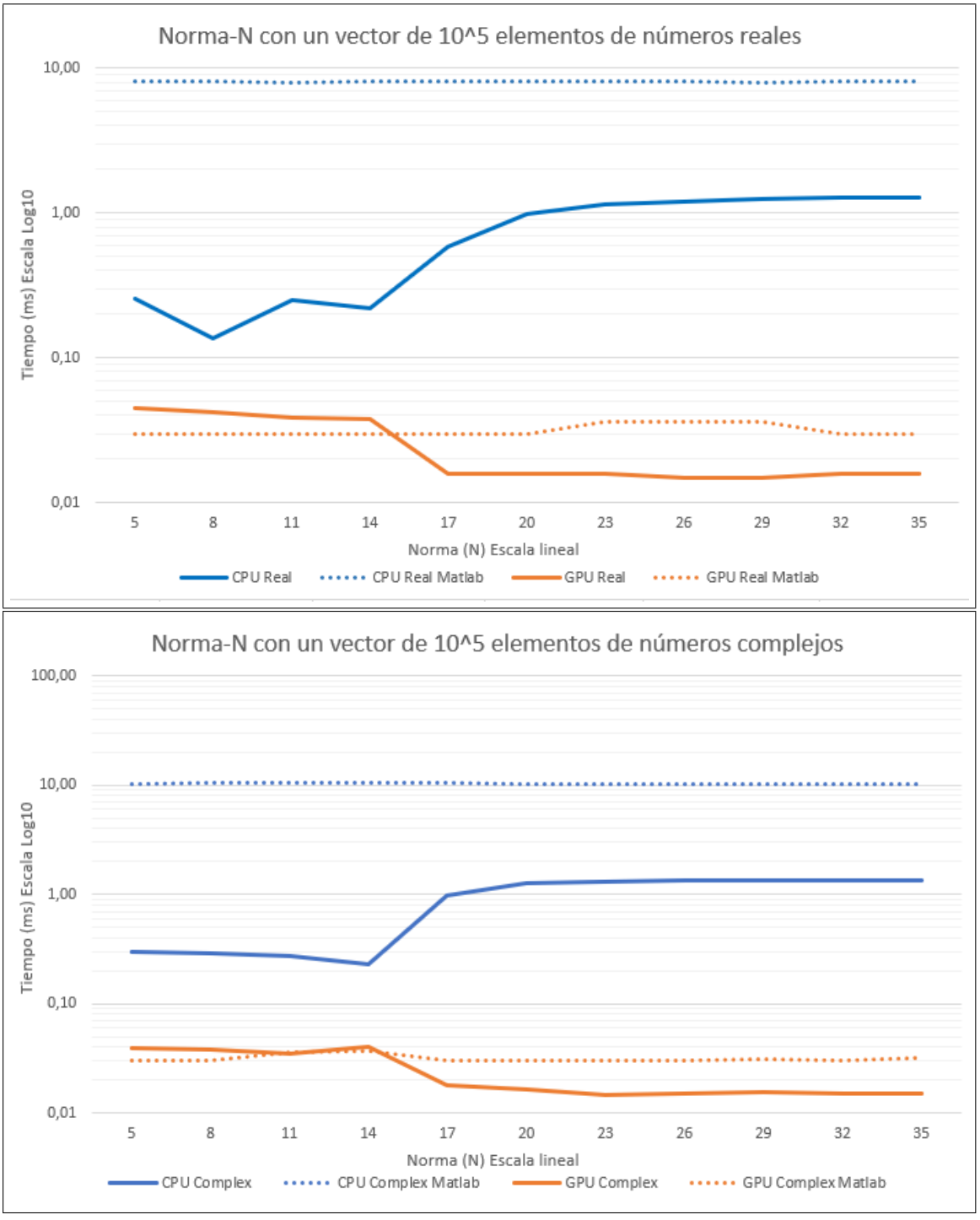

Figura 6.4: Curvas de rendimiento en función del la norma con un vector de tamaño fijo 10^5 para el algoritmo de cálculo de la norma-N. Arriba para elementos reales y debajo para elementos complejos. Curvas en color naranja liso para la GPU, curva en naranja punteado para la GPU en Matlab, azul liso para la CPU y azul punteado para la CPU en Matlab.

Como se puede ver en la figura 6.4, el valor de la norma no tiene un apenas impacto en el rendimiento. Se vemos las curva obtenidas de las simulaciones sobre Matlab, estas son constantes en todo el rango de evaluación, y se corresponden con el punto 10^5 de la figura 6.3. Sin embargo, aunque solo para valores bajos, sí se detecta un cambio de rendimiento para la CPU y la GPU en este caso. Para la CPU, a partir de la norma 14, empieza a empeorar el tiempo de ejecución en un orden de magnitud hasta la norma 23, donde se vuelve estable para valores mayores a este. Si analizamos la GPU, vemos que ocurre justo lo contrario, el tiempo de ejecución mejora a partir de la norma 14 y se vuelve estable. Esto se debe a que la norma se calcula empleando una llamada a una función de OpenCL de bajo nivel que tiene este comportamiento particular. Por último, comparando la ejecución en Matlab frente a la ejecución de nuestro algoritmo, vemos que la CPU de Matlab es un orden de magnitud más lenta siempre, y que la GPU de Matlab es similar, siendo algo peor a partir de la norma 17.

Como conclusión podemos destacar que el algoritmo del cálculo de la norma-N es competitivo. De forma general, y teniendo en cuenta que se ha diseñado para su ejecución específicamente sobre GPU, este algoritmo es más rápido que el mismo cálculo realizado con Matlab para la gran mayoría de los rangos de datos. Además, no se observan limitaciones ni puntos débiles a priori a la vista de los resultados. En este caso, parece que el posible cuello de botella existente en el uso de un acumulador común a todos los hilos para calcular el sumatorio no supone un problema. Será muy interesante la comparativa de este algoritmo y sus resultados con el siguiente algoritmo, el que realiza el producto escalar, ya que ambos tienen una estructura y un planteamiento muy similares, pero se diferencian en ciertos detalles que pueden ser de gran relevancia.

6.2 Rendimiento del algoritmo de producto escalar

En esta sección se va a analizar el rendimiento del algoritmo que realiza el cálculo del producto escalar de dos vectores. Para ello, se va a modificar el tamaño de los mismos, estudiando como afecta este al rendimiento, y realizando la comparativa de forma similar al apartado anterior. Por último, se va a comparar este algoritmo con el diseñado para el cálculo de la norma-N debido a su similitudes.

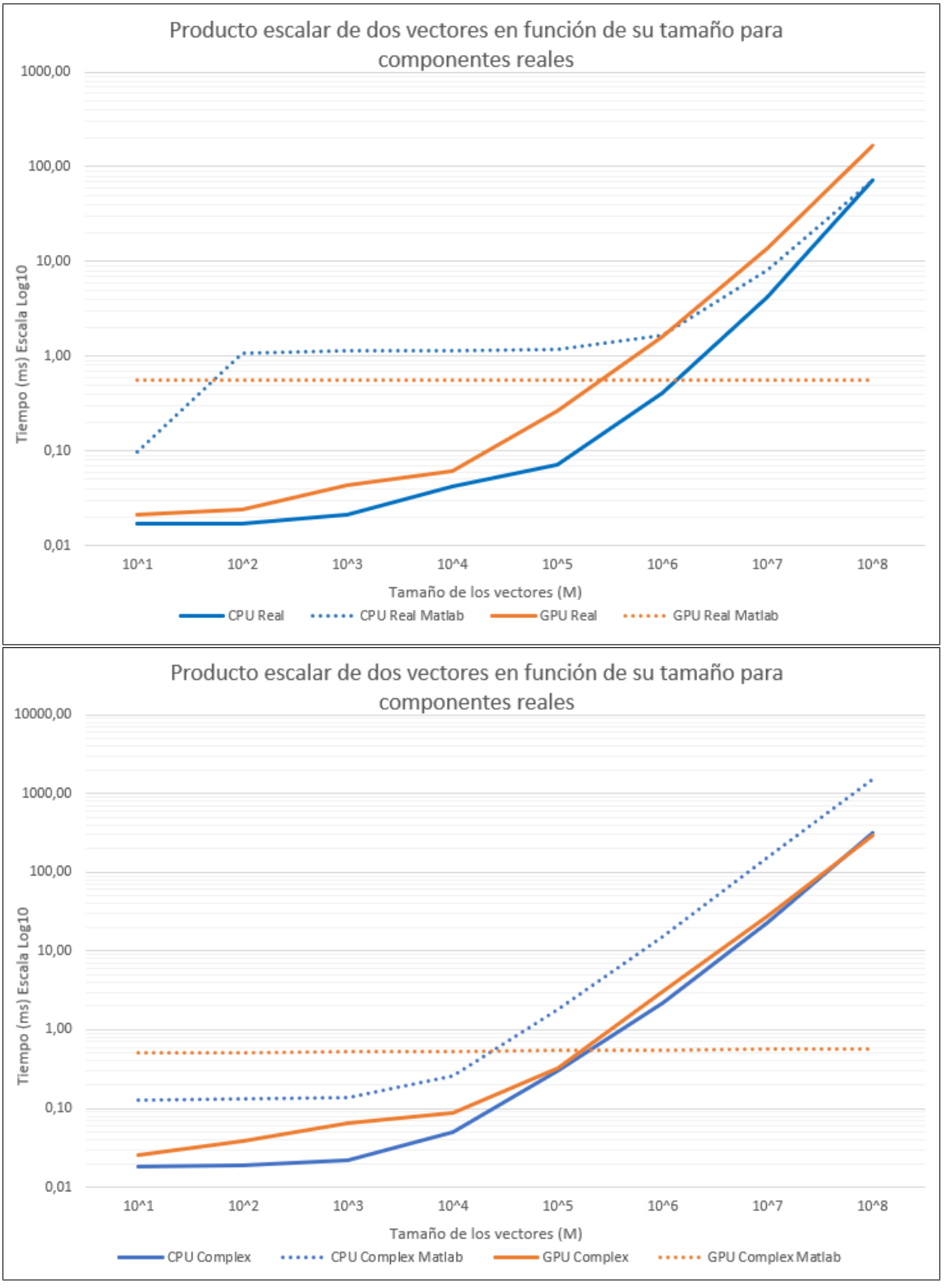

Figura 6.5: Curvas de rendimiento para el producto escalar de vectores en función de su tamaño. Arriba para elementos reales y debajo para elementos complejos. Curvas en color naranja liso para la GPU, curva en naranja punteado para la GPU en Matlab, azul liso para la CPU y azul punteado para la CPU en Matlab.

Como podemos observar en la figura 6.5, lo primero que nos puede sorprender es que en este caso la CPU es más competitiva que la GPU, sobre todo cuando los elementos de los vectores son números reales. Además, esta resulta tan competitiva como la CPU en Matlab, aunque mucho menos que la GPU de Matlab. El gran problema de este algoritmo es que la GPU tiene un pobre desempeño. Viendo la gran diferencia de rendimiento de la GPU en comparación con el algoritmo de norma-N, vamos a estudiar sus diferencias para poder entender este rendimiento.

Si revisamos las secciones 5.1 y 5.2, nos damos cuenta que la principal diferencia entre ambos es que para la norma-N se realiza la operación de elevar un número a una potencia, mientras que en el producto escalar solo se lleva a cabo un producto. Esta diferencia es clave y provoca esta diferencia de rendimiento. La idea consiste en que los diseños de estos algoritmos tenemos dos partes diferenciadas. Por un lado, todas las operaciones de cálculo que se realizan en paralelo por los múltiples hilos, y la actualización atómica del acumulador que se lleva a cabo de forma secuencial (en la práctica, que un solo hilo pueda actualizar el valor del acumulador implica que esta operación es secuencial y cada hilo tiene que "esperar su turno").

La diferencia entre ambos es la parte de la carga computacional implicada en la actualización del acumulador y la parte dedicada al cálculo de las operaciones. Mientras que calcular una potencia supone una carga computacional alta, realizar un producto supone una carga mucho más baja, lo que provoca que de todos los tiempos de operación necesarios para ejecutar cada hilo, en el producto escalar la mayoría del tiempo de ejecución suponga el tiempo de espera para actualizar el acumulador, no siendo así en el algoritmo de la norma-N. Como se ha visto en la introducción, este cuello de botella afecta mucho menos a la CPU que a la GPU, porque si bien tiene menos hilos estos tienen una velocidad de reloj más rápida, por lo que actualizarán el acumulador común más rápido. En cambio si el cálculo supone una gran parte del tiempo de procesamiento, el poder paralelizar un gran número de cálculos sigue suponiendo mayor ventaja que el tiempo de espera para actualizar el acumulador.

Si observamos las dos gráficas de la figura 6.5, también se puede apreciar este comportamiento en la distancia entre las curvas de CPU y GPU si los datos son números reales o números complejos. Como calcular el producto de dos números complejos (4 productos y dos sumas) supone más carga que calcular el producto de dos números reales (1 producto),

esto provoca que para el *kernel* que opera con números complejos el rendimiento entre la CPU y la GPU sea prácticamente igual, mientras que en el caso de números reales la CPU es más rápida en toda la curva que la GPU. Esto se debe al mismo motivo, cuanto más aumente la carga computacional de las operaciones de cálculo que se realizan en paralelo frente a las escrituras a una zona de memoria común que tienen que realizarse en serie, mejor será el desempeño de la GPU frente a la CPU debido a que puede ejecutar muchos más hilos en paralelo, aunque estos sean más lentos debido a la frecuencia de reloj del dispositivo.

Como conclusión se puede evidenciar que la estrategia de un acumulador único, aunque se almacene en memoria local, supone una debilidad para la ejecución de estos algoritmos en GPU, dado que supone un importante cuello de botella. Como propuesta de mejora para evitar esta forma de calcular los sumatorios y poder paralelizar también el cálculo del mismo en cierta medida, parece interesante calcular en una primera fase todos los productos o potencias y almacenarlos en un vector auxiliar. A partir de este vector, se podría calcular el sumatorio sumando los elementos de este vector por parejas, reduciendo el tamaño del vector a la mitad en cada iteración hasta llegar a tener un único elemento, que sería el resultado. Esta estrategia resolvería el problema del acumulador, ya que nos permitiría paralelizar en cierta medida gran cantidad de las sumas necesarias para el cálculo del sumatorio.

6.3 Rendimiento del algoritmo de producto de matrices

Se va a comenzar revisando el algoritmo de multiplicación. Primero, se va a realizar una comparativa de este para matrices reales, cuadradas, y que van aumentando de valor en potencias de 2. De esta forma, se muestra en escala logarítmica en base 2 cómo aumenta el tiempo de ejecución en función del tamaño de las matrices sobre las que se calcula el producto.

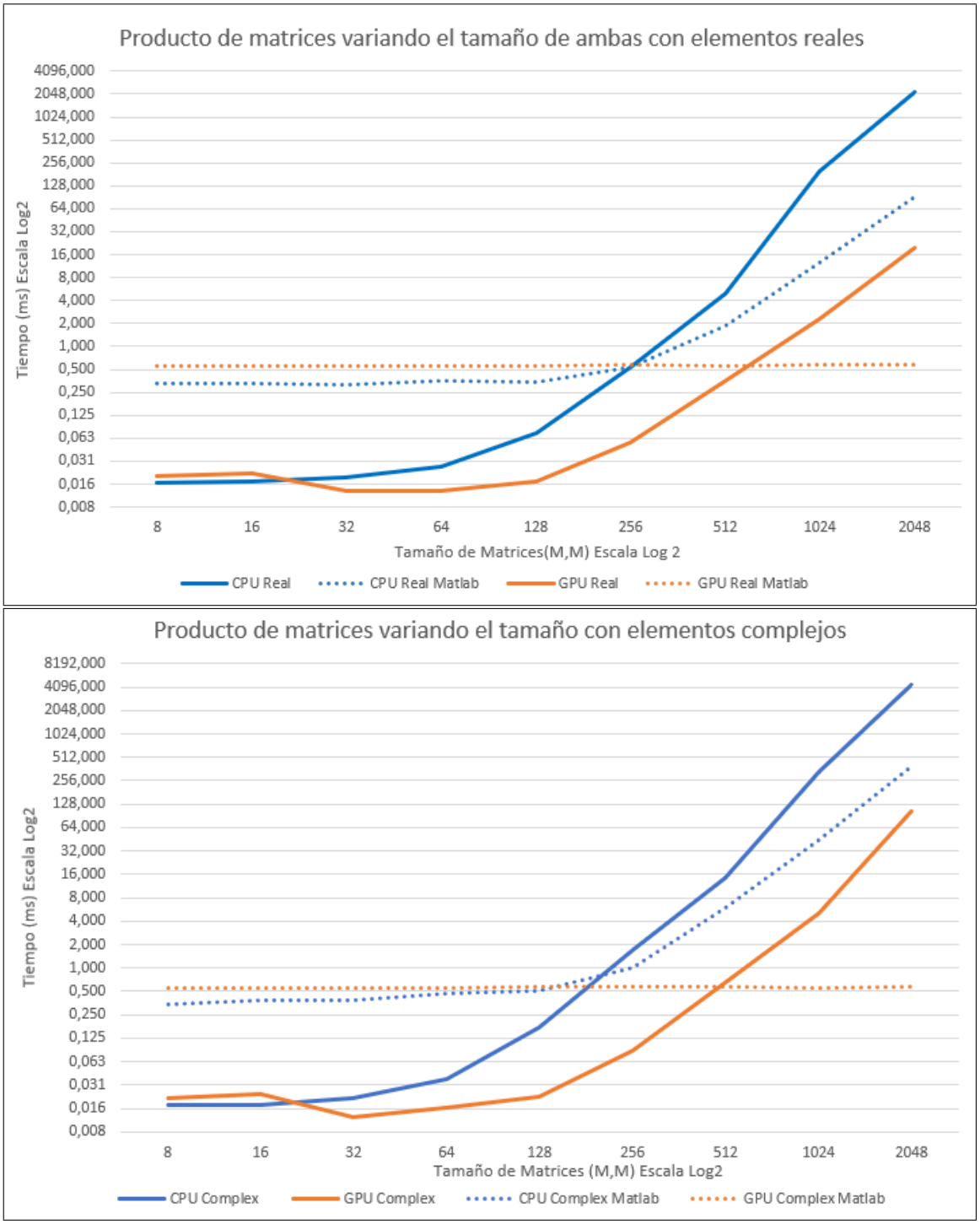

Figura 6.6: Curvas de rendimiento para el producto escalar de vectores en función de su tamaño. Arriba para elementos reales y debajo para elementos complejos. Curvas en color naranja liso para la GPU, curva en naranja punteado para la GPU en Matlab, azul liso para la CPU y azul punteado para la CPU en Matlab.

Como podemos observar en la figura 6.6, para tamaños de matrices pequeños, hasta

128, tanto la CPU como la GPU soportan bastante bien la carga, siendo siempre la GPU entre 2 y 3 veces más rápida de forma constante. Sin embargo, cuando empezamos a multiplicar matrices más grandes, como 1024 y 2048, la CPU empieza a disparar sus tiempos, creciendo notablemente. En contraposición, la GPU sigue manteniéndose muy consistente en el tiempo de ejecución, y empiezan a crecer los tiempos de ejecución a partir de matrices de tamaño 256. Comparando con el desempeño de Matlab, si es cierto que para matrices de tamaños más grandes la GPU de Matlab es mucho más consistente, manteniendo el mismo tiempo de ejecución que para las matrices más pequeñas. Sin embargo, es en la parte de la curva de rendimiento hasta tamaño 512 donde destaca el algoritmo desarrollado en GPU. Este comportamiento es similar al que se ha visto para otros algoritmos, donde en tamaños de datos pequeños Matlab resulta más lento. En cuanto a la comparativa entre la CPU y la CPU de Matlab, las curvas son similares, siendo algo mejor Matlab para tamaños grandes de matrices.

Este comportamiento es consistente para el producto de matrices de números complejos. Sin embargo, la principal diferencia está cuando en el producto de matrices grandes. La GPU tiene un aumento de los tiempos de procesamiento mucho menor y con menor pendiente, mientras que la CPU crece muchísimo más rápido en este caso a partir, sobre todo, de 512. Este crecimiento mucho más abrupto se debe a que el número de operaciones para multiplicar dos números reales es mucho más bajo que para multiplicar dos números complejos (1 producto frente a 4 productos y 2 sumas). Esto provoca que, al aumentar el tamaño, el número de operaciones crezca más rápidamente en el caso complejo, y de la misma forma crezca la carga de trabajo. Podemos ver que la GPU puede asumirla sin tanto impacto, mientras que la CPU sufre mucho más. En la comparativa con los resultado de Matlab es muy similar al visto para números reales, donde la GPU se mantiene constante y la CPU crece más despacio. Como conclusión, para tamaños hasta 512 resulta más rápido el algoritmo desarrollado, mientras que para matrices más grandes Matlab tiene mejor desempeño.

6.4 RENDIMIENTO DEL ALGORITMO DE POTENCIA DE UNA MATRIZ

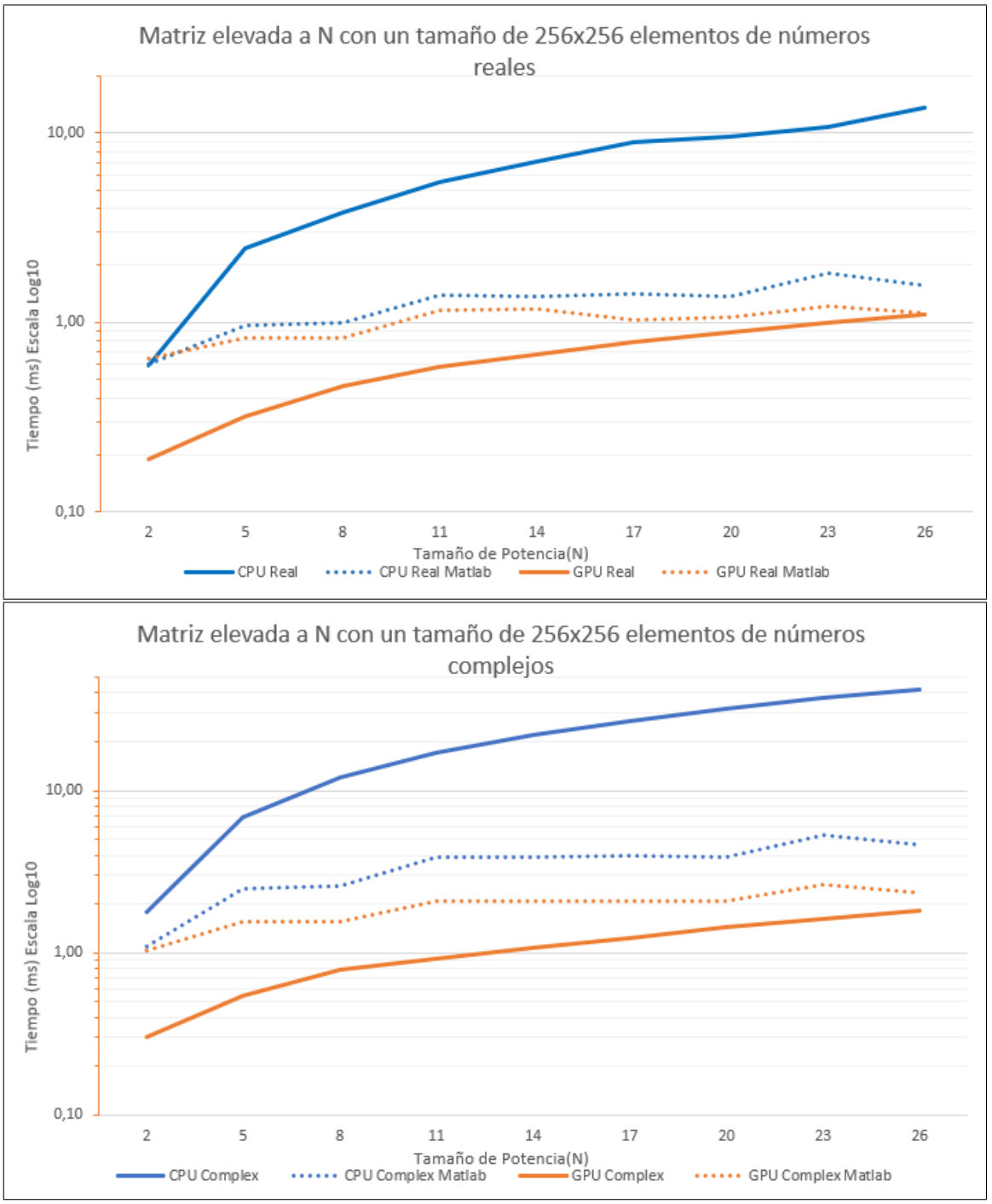

Figura 6.7: Curvas de rendimiento para el producto escalar de vectores en función de su tamaño. Arriba para elementos reales y debajo para elementos complejos. Curvas en color naranja liso para la GPU, curva en naranja punteado para la GPU en Matlab, azul liso para la CPU y azul punteado para la CPU en Matlab.

Una vez visto el rendimiento del algoritmo de multiplicación, y dado que el algoritmo que calcula la potencia de una matriz se construye a partir de llamadas sucesivas al *kernel* de multiplicación, vamos a estudiar el desempeño de nuestro algoritmo suponiendo una matriz de tamaño fijo 256x256. Vamos a ver como cambian los tiempos de ejecución en función de aumentar la potencia, y como afecta esto tanto a nuestro algoritmo en CPU y GPU, como al algoritmo de Matlab.

Como podemos observar en la figura 6.7, para la potencia 2 (que es lo mismo que el producto) los datos son consistentes con lo visto en el apartado anterior para este tamaño. Una vez que vamos aumentando la potencia, vemos como la GPU empieza aumentar el tiempo de procesamiento. Sin embargo, vemos como la tendencia es que cuanto mayor es la potencia más despacio crece sl tiempo de procesamiento. Este resultado es muy interesante,ya que si bien sigue creciendo hasta el infinito si la potencia es infinita, nos indica que se pueden llegar a calcular potencias muy altas con nuestro algoritmo sin que esto suponga un gran aumento de los tiempos de procesamiento.

En comparación con Matlab, los tiempos de ejecución aumentan de forma muy leve según aumentamos el tamaño de la potencia. Esto nos indica que Matlab tiene unos tiempos de ejecución más estables, y por tanto para potencias muy grandes el algoritmo de GPU desarrollado tendrá un peor desempeño. Estos resultados son consistentes tanto para el caso de números reales como para números complejos.

A la vista de los resultados, podemos concluir que el algoritmo del producto no tiene grandes debilidades visibles en los resultados. Tanto para el producto de matrices como para elevar una matriz a una potencia, el algoritmo se tiene un buen desempeño para matrices de tamaño 512x512, y puede realizar operaciones con matrices de mayor tamaño de forma competitiva. Por último, cabe destacar que, gracias a OpenCLIPER, se ha podido implementar el algoritmo de cálculo de la potencia de una matriz pudiendo reutilizar el *kernel* para el producto de matrices y que los resultados el algoritmo del cálculo de la potencia son competitivos.

6.5 Rendimiento del algoritmo de convolución

Para el algoritmo de convolución se van a hacer diferentes pruebas, dado que en este caso el uso de memoria local no está en todo el rango de tamaños y que en función del

uso que se le vaya a dar, el *kernel* es común que sea pequeño. De hecho, en procesado de imagen, el *kernel* no suele ser mayor de 3x3 o 5x5 [12]. Por ello, se va a hacer una primera comparativa entre GPU y CPU para una matriz de datos creciente y un *kernel* de tamaño fijo 3x3. Además, en este caso, siempre se usará memoria local ya que el *kernel* es pequeño. A continuación, se realizará una prueba donde se compare el rendimiento cuando el *kernel* de convolución tiene el mismo tamaño que la matriz de datos. Por último, para evaluar el rendimiento del *kernel* de memoria global frente al de memoria local en GPU, se llevará a cabo una última comparativa donde la matriz de datos y el *kernel* de convolución tienen el mismo tamaño, hasta el tamaño máximo del *kernel* que puede ser contiendo en memoria local para la tarjeta gráfica que se está utilizando en el proyecto.

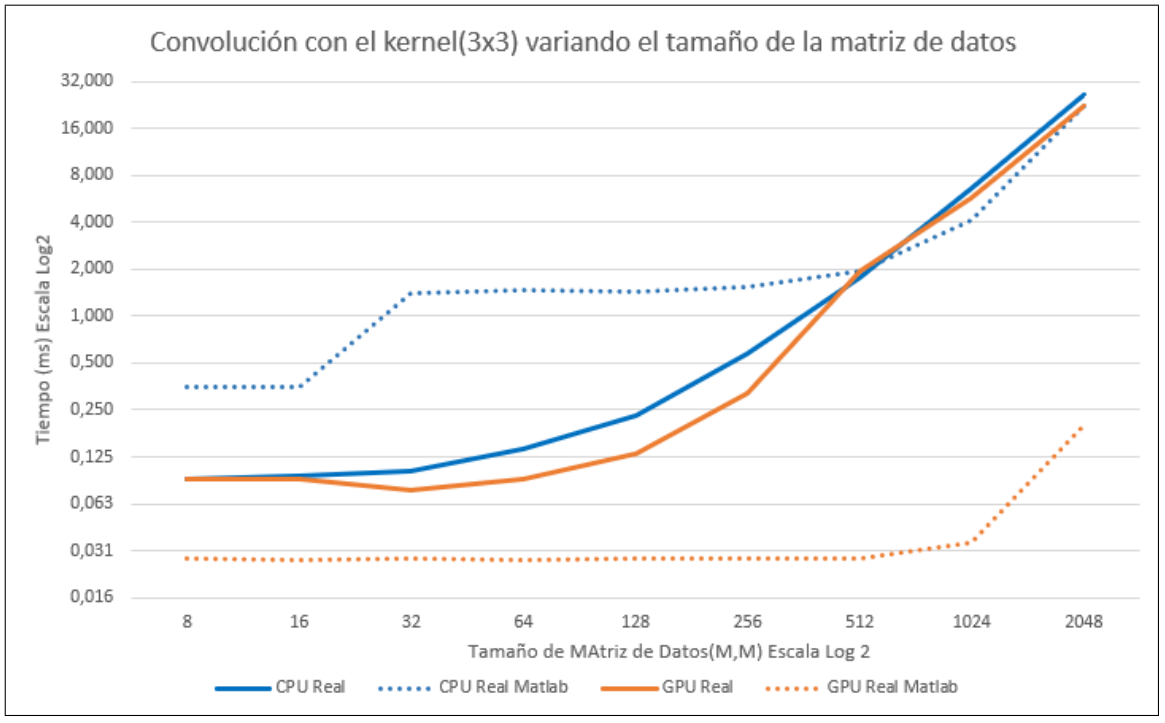

Figura 6.8: Curvas de rendimiento para la convolución de matrices con un kernel fijo 3x3 y variando el tamaño de la matriz de datos. Curvas en color naranja liso para la GPU, curva en naranja punteado para la GPU en Matlab, azul liso para la CPU y azul punteado para la CPU en Matlab.

Como podemos ver en esta figura 6.8, el rendimiento de CPU y GPU sobre todo cuando el tamaño de la matriz de datos aumenta. Además, podemos ver que con un *kernel* de convolución pequeño Matlab tiene mucho mejor desempeño de forma general. Esto se debe al mismo problema detectado en otros algoritmos como el producto escalar. Debido

al planteamiento de paralelización, la información del *kernel* de convolución tiene que ser accedida por todo el conjunto de *work-items*. Cuando este *kernel* de convolución es pequeño, las operaciones realizadas por cada hilo son muy pocas, y provoca que el tiempo requerido para la lectura de la memoria suponga un tiempo comparable al tiempo de procesamiento. Esto unido a que también se tienen que hacer lecturas por diferentes hilos a elementos comunes de la matriz de datos provoca que se forme un cuello de botella en las lecturas a memoria. Como hemos visto, esto afecta m,ás a la GPU debido a que su frecuencia de reloj es más lenta, así que la CPU consigue tiempos de ejecución similares del algoritmo con mucho menor numero de hilos. Esto nos permite concluir que la estrategia de paralelización empleada en este caso no tiene buenos resultados, al menos para escenarios en los que el *kernel* de convolución es pequeño.

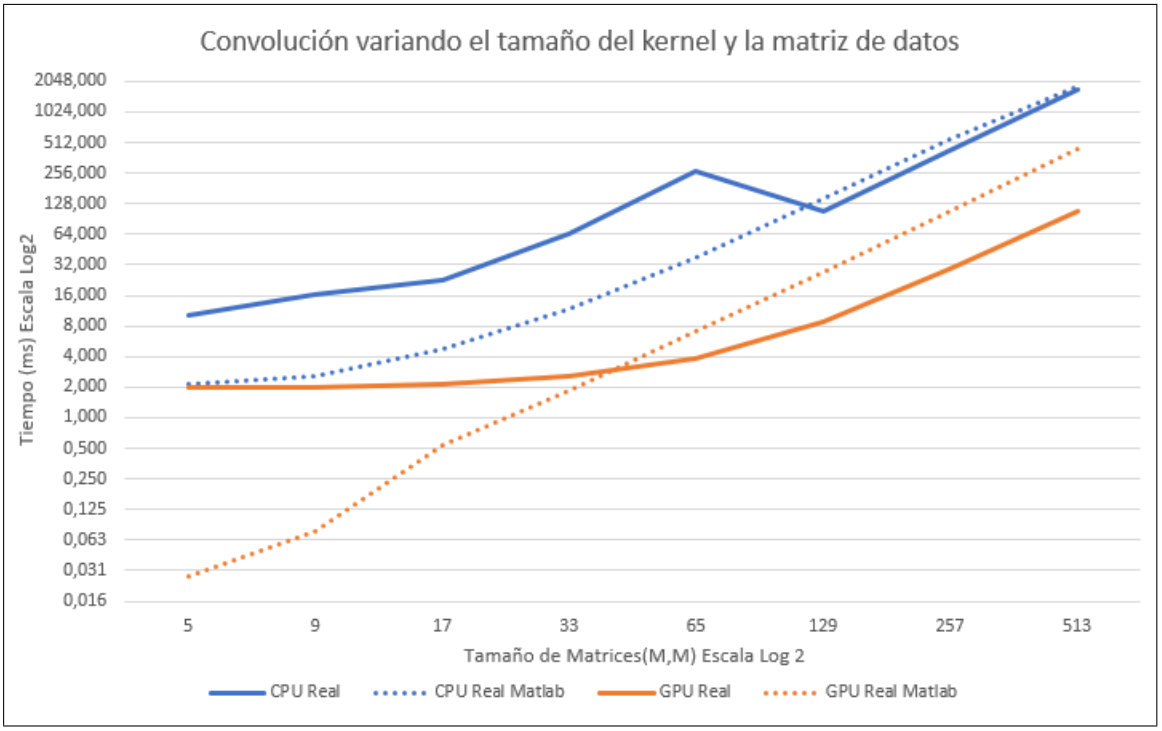

Figura 6.9: Curvas de rendimiento para la convolución de matrices con un kernel fijo 3x3 y variando el tamaño de la matriz de datos. Curvas en color naranja liso para la GPU, curva en naranja punteado para la GPU en Matlab, azul liso para la CPU y azul punteado para la CPU en Matlab.

Sin embargo, como podemos ver en la figura 6.9, vemos que el algoritmo tiene un rendimiento muy diferente cuando aumentamos el tamaño del *kernel* de convolución. Según se va aumentando el tamaño del kernel de convolución, tanto la GPU como incluso la CPU

tienen un crecimiento en los tiempos de ejecución mucho más lento que Matlab. Para la GPU vemos que a partir de un *kernel* de convolución de tamaño 40x40 aproximadamente, los tiempos de nuestro algoritmo son más competitivos que los de Matlab, llegando a alcanzar la misma pendiente para *kernels* de convolución de tamaño 129x129. Esto se debe también a que, debido como se ha planteado la solución, cuando aumentamos el tamaño del *kernel* aumentamos el número de operaciones que realiza cada hilo en relación a las lecturas en memoria, y entonces la carga computacional que suponen los cálculos es mayor. Esto provoca que el efecto del cuello de botella existente en las lecturas a memoria tenga mucho menor impacto en el rendimiento. Este comportamiento es similar al que veíamos en los resultados de las secciones 6.1 y 6.2. Como conclusión, podemos destacar que nuestro diseño tiene un buen desempeño cuando se realiza la convolución contra *kernels* de gran tamaño, pero sufre en los escenarios donde el *kernel* de convolución es pequeño.

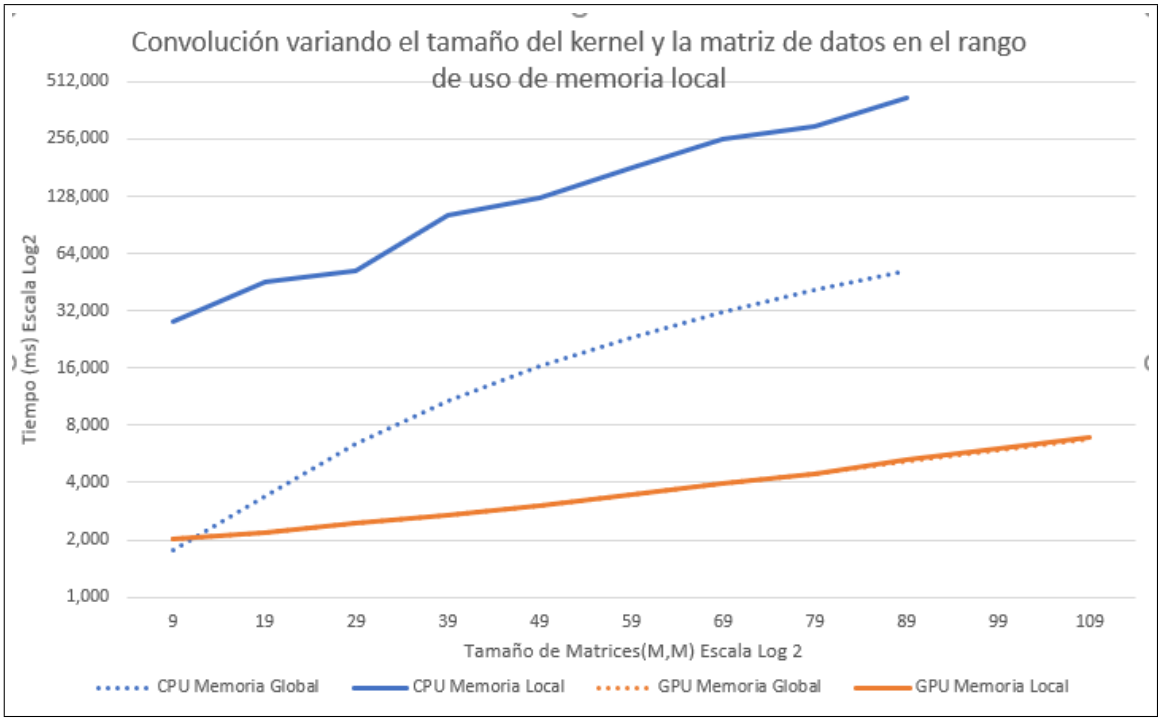

Figura 6.10: Curvas de rendimiento para la convolución de matrices con un kernel variable y variando el tamaño de la matriz de datos de la misma forma en el rango de uso del *kernel* que emplea memoria local. Curvas en color naranja liso para la GPU usando el *kernel* de memoria local, curva en naranja punteado para la GPU usando el *kernel* de memoria global, azul liso para la CPU usando el *kernel* de memoria local y azul punteado para la CPU usando el *kernel* de memoria global.

Por último, estudiando la comparativa entre la CPU y la GPU usando memoria global frente a memoria local obtenemos la figura 6.10. Comenzando por la CPU, vemos que emplear la memoria local no supone ninguna mejora. Además, el rango de uso de este *kernel* en la CPU empleada es menor dado que la memoria local tiene de tamaño 32 KB. Debido a que la CPU tiene un número de hilos pequeño y estos son relativamente rápidos, el coste de copiar el *kernel* de convolución a memoria local supone más tiempo de computación que ejecutar los cálculos directamente sobre memoria global. Si nos fijamos en la GPU, vemos que el rendimiento es exactamente el mismo leyendo el *kernel* de convolución de memoria global que copiando el mismo a memoria local antes de hacer los cálculos. Esto se debe a que, debido al tamaño pequeño de la memoria local, se ha definido copiar solo el *kernel* de convolución y no la matriz de datos, y sobre la matriz de datos se realizan la misma cantidad de lecturas solapadas en memoria, dado que la matriz de datos y el *kernel* de convolución tienen el mismo tamaño en estas simulaciones. Es por esto que si bien copiar el *kernel* de convolución a memoria local supone una mejora, esta no se ve reflejada al existir otro cuello de botella similar en la matriz de datos.

Como conclusión, la forma de paralelizar y estructurar en memoria el algoritmo de convolución implica ciertas ventajas e inconvenientes. Por un lado, supone una desventaja cuando trabajamos con un *kernel* de convolución pequeño. Además, emplear la memoria local como se plantea no resulta útil, dado que no resuelve el problema además, de admitir un rango de operación en función del tamaño del *kernel* de convolución relativamente pequeño. Sin embargo, la gran fortaleza del planteamiento realizado para este algoritmo reside en su rendimiento cuando el *kernel* de convolución y la matriz de datos son grandes. Debido en su estructura de paralelización, el peso computacional de los cálculos es mayor que el coste de los accesos a memoria en este escenario, lo que le da un gran rendimiento, incluso mejor que el ofrecido por Matlab para la misma operación. Como mejora, se podría proponer un *kernel* de OpenCL que se adaptara al escenario donde el desarrollado en el trabajo tiene peor desempeño, de forma que el proceso de OpenCLIPER escogiera cual ejecutar en función de los parámetros de los datos de entrada.

Capítulo 7

Conclusiones

Tras revisar los resultados de nuestros algoritmos, así como detallar los principales motivos de los resultados derivados de las implementaciones realizadas, vamos a revisar de forma general las conclusiones extraídas de este trabajo de fin de máster, así como revisar diversas líneas futuras a seguir para continuar el desarrollo del proyecto.

7.1 Conclusiones

En este documento se ha planteado la integración de un diversos de procesamiento de datos vectoriales y matriciales, desarrollados para su implementación de dispositivos de computación vectorial o GPU. Para ello, se ha empleado OpenCL y sobre este el framework OpenCLIPER, con el objetivo de que puedan ser ejecutados en todo tipo de dispositivos y a su vez que sean fácilmente utilizables gracias a OpenCLIPER. Los algoritmos desarrollados se pueden dividir en cinco: un algoritmo que realiza la norma-N de orden N de un vector de componentes variables, un algoritmos que realiza el producto escalar de dos vectores, un algoritmo que realiza el producto de dos matrices, un algoritmo que calcula una matriz elevada a una potencia N, y otro algoritmo que realiza la convolución de una matriz de datos con un *kernel* de convolución.

Empleando el lenguaje de programación OpenCL se ha podido evaluar la capacidad de flexibilidad del mismo así como la gran ventaja de que pueda ser portable, pudiendo ser ejecutado en numerosos dispositivos. Además, dado que también es soportado por las CPU, ha permitido evaluar pruebas de rendimiento con gran facilidad, dado que no se ha requerido modificar el algoritmo de ningún modo para las mismas. Todas estas ventajas han sido suficientes para elegirlo para este proyecto, pese a que estas virtudes implican ciertos inconvenientes, como cierta ineficiencia frente a otros lenguajes más específicos como CUDA.

Por un lado, el uso del framework OpenCLIPER ha supuesto una enorme ventaja a la hora del desarrollo de los mismos. Gracias a todas la herramientas que proporciona, como simplifica el trabajo y como permite abstraernos hasta cierto nivel de OpenCL facilita en gran medida la integración. Además, gracias a su estructura de diseño, hace que sea mucho más sencillo para un usuario el uso de la implementación, agilizando la recogida de datos, ejecución del algoritmo y almacenamiento del resultado.

Por otro lado, mediante el desarrollo del código de estos algoritmos, se ha podido comprender la potencia que supone la paralelización en la computación de datos vectoriales, así como entender un nuevo paradigma de programación. Si bien supone una forma de pensar diferente, este tipo de implementaciones permiten soluciones innovadoras a problemas que de otra forma suponen un coste mucho más elevado, además de proporcionar al usuario una herramienta más potente para todo tipo de entornos como el procesamiento de imagen médica, aplicación de filtros o aplicación sobre Big Data.

Finalmente, tras analizar el rendimiento de los algoritmos, se ha visto las fortalezas y debilidades de las diferentes propuestas implementadas. Gracias a este análisis, se ha podido estudiar puntos de mejora de los diferentes algoritmos, que permitan mejorar aún más las soluciones planteadas. las diferentes conclusiones obtenidas para cada uno de los algoritmos son la siguientes:

- **Algoritmo de cálculo de la norma-N de un vector:** tiene un buen desempeño en general. Se puede mejorar calculando el sumatorio con un flujo diferente al de un acumulador común.
- **Algoritmo de producto escalar de vectores:** tiene un desempeño mejorable debido al uso de los acumuladores para calcular el sumatorio. Se puede mejorar, al igual que el anterior, calculando el sumatorio con un flujo diferente al de un acumulador común.
- **Algoritmo de producto algebraico de matrices:** tiene un buen desempeño para tamaños pequeños de las matrices de datos. Una posible mejora sería diseñar un algoritmo diferente más eficiente para matrices de tamaños grandes de forma complementaria.
- **Algoritmo de potencia de una matriz:** tiene un gran desempeño, además de que

la curva de crecimiento de tiempos de ejecución es baja en relación con el tamaño de la potencia. Mejorar el algoritmo del producto algebraico de matrices supondría una mejora directa sobre este algoritmo.

- **Algoritmo de convolución de matrices:** tiene un buen desempeño para *kernels* de convolución grandes, pero es ineficiente para *kernels* de tamaño pequeño. Una mejora interesante sería diseñar un algoritmo complementario que aplique para tamaños del *kernel* de convolución pequeños.

7.2 Lineas futuras

Para finalizar el trabajo realizado, resulta interesante comentar diferentes líneas de mejora aplicables al proyecto, de forma que pueda incrementar tanto el rendimiento como su integración con otros procesos o facilidad de uso de cara al usuario.

Comenzando por el conjunto del programa, si bien la integración sobre OpenCLIPER y OpenCL es notablemente completa, se podrían añadir funcionalidades para hacer más amigable el uso por parte del usuario. Por ejemplo, en caso de tener ya la matriz de *padding* calculada previamente, dar la posibilidad de que la entrada ya tenga el *padding* realizado y solo deba ejecutarse la convolución. También en el algoritmo de multiplicación, permitir que el programa admita un conjunto mayor a dos matrices, realizando la multiplicación sucesiva de todas ellas y no teniendo el usuario que ir cargando la nueva entrada sobre la salida anterior de forma manual.

Comenzando por los algoritmos para el calculo de la norma-N y el cálculo del producto vectorial, una mejora interesante sería modificar la forma en la que se calcula el sumatorio. Actualmente, supone un importante cuello de botella el que todos los hilos requieran actualizar la misma posición de memoria. Un mejora sería realizar todos los cálculos en paralelo sobre un vector auxiliar, y calcular el sumatorio realizando la suma de todos los elementos de ese vector con un algoritmo de reducción eficiente.

En cuanto al algoritmo de multiplicación en sí, la implementación escogida es relativamente sencilla. Se ha imitado el algoritmo de multiplicación de matrices clásico, que tiene un acceso a memoria compartida relativamente alto. Actualmente, existen desarrollos de algoritmos de multiplicación de matrices mucho más eficientes, que podrían implementarse

modificando el *kernel*. La ventaja es que, como OpenCLIPER y OpenCL se encargan de todo lo demás, solo con sustituir el *kernel* por uno más eficiente tendríamos una nueva versión del algoritmo mejor, sin necesidad modificar el proceso.

Siguiendo por el punto más interesante, el algoritmo de convolución puede tener varias líneas de mejora diferentes. Por un lado, al implementación de diferentes algoritmos complementarios más eficientes, que cambien el enfoque para buscar mejorar el tratamiento de la memoria para tamaños pequeños del *kernel* de convolución, evitando lecturas y escrituras en zonas comunes de la memoria por un número elevado de *work-item*s. Por otro lado, desarrollar un *kernel* que no requiera el paso intermedio del cálculo de la matriz de padding, sino que realice este paso "al vuelo" mediante a desplazamientos e igualando directamente a cero aquellos elementos que queden fuera del rango de la matriz de datos y que en la implementación actual multiplican por ese marco exterior de ceros generado por el padding.

Bibliografía

[1] Aguilar, J. and Leiss, E. (2004). *Introducción a la Computación Paralela*. Universidad de los Andes, 1st edition.

[2] Dragomir, S. S. (2006). The hypo-euclidean norm of an n-tuple of vectors in inner product spaces and applications. *Research report collection*, 9(4).

[3] Joyanes Aguilar, L. (2003). *Fundamentos de programación. Algoritmos, Estructuras de datos y Objetos*. McGraw-Hill, 2nd edition.

[4] KDevelop (2022). *KDevelop*. `https://www.kdevelop.org`. *Fecha de última consulta junio de 2022*.

[5] Khokhar, A., Prasanna, V., Shaaban, M., and Wang, C.-L. (1993). *Heterogeneous computing: challenges and opportunities*. *Computer*, 26(6):18–27.

[6] Khronos Group (2018). *The OpenCL Specification. Version: 2.1, Revisión: 24*. `https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf`. *Fecha de última consulta mayo de 2022*.

[7] Laboratorio de Procesado de Imagen, UVA (2022a). *OpenCLIPER*. `http://opencliper.lpi.tel.uva.es`. *Fecha de última consulta mayo de 2022*.

[8] Laboratorio de Procesado de Imagen, UVA (2022b). *OpenCLIPER Clases*. `http://opencliper.lpi.tel.uva.es/classes/`. *Fecha de última consulta mayo de 2022*.

[9] Needham, T. (2000). *Visual Complex Analysis*. Oxford University Press.

[10] NVIDIA Corp. (2022). *CUDA Zone*. `https://developer.nvidia.com/cuda-zone`. *Fecha de última consulta mayo de 2022*.

[11] NVIDIA Corporation (2022). *Nvidia*. `https://www.nvidia.com/`. *Fecha de última consulta agosto de 2022*.

[12] Palomares, F. G., Monsoriu, J. A., and Alemany, E. (2016). Aplicación de la convolución de matrices al filtrado de imágenes. *Modelling in Science Education and Learning*, 9(1):97–108. *Fecha de última consulta mayo de 2022.*

[13] Poole, D. (2011). *Álgebra lineal. Una introducción moderna.* Cengage Learning, 3rd edition.

[14] Rivero, F. (2001). *Una introducción a los Números Complejos.* Universidad de los Andes, 1st edition.

[15] Simmross Wattenberg, F. (2018). *OpenCLIPER: Image Processing and Reconstruction in OpenCL.*

[16] Spiegel, M. R., Lipschutz, S., and Spellman, D. (2011). *Análisis vectorial.* McGRAW-HILL, 2nd edition.

[17] The Khronos Group Inc. (2022). *Khronos Group, Conecting software to silicon.* `https://www.khronos.org`. *Fecha de última consulta agosto de 2022.*

[18] The MathWorks, Inc. (2022a). *Help Center. Adtest: Anderson-Darling test.* `https://es.mathworks.com/help/stats/adtest.html`. *Fecha de última consulta septiembre de 2022.*

[19] The MathWorks, Inc. (2022b). *MathWorks. Soporte.* `https://es.mathworks.com/help/`. *Fecha de última consulta junio de 2022.*

[20] TigerVNC (2022). *TigerVNC.* `https://tigervnc.org`. *Fecha de última consulta junio de 2022.*

[21] Weisstein, E. W. (2003). *Complex Number. https://mathworld. wolfram. com/.*

[22] Zander, B., Sundermeyer, J., Braun, D., Hoffmann, T. (2022). *Welcome to TeXstudio.* `https://www.texstudio.org`. *Fecha de última consulta junio de 2022.*