Facultad de Ciencias

Herramienta para el procesamiento de audio en tiempo real mediante OpenCL

(Tool for real time audio processing with OpenCL)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Diego García Cosío

Director: Raúl Nozal Gonzalez

Co-Director: José Luis Bosque Orero

Junio - 2020

Índice general

Índice de figuras

Resumen

En los últimos años, el desarrollo de las tarjetas gráficas ha ido al alza, pero lo que no se han desarrollado tanto son las diferentes funcionalidades que se le da a este tipo de procesadores. Estos procesadores gráficos actúan muy bien en la ejecución de videojuegos, moviendo a 60 fotogramas por segundo imágenes de 1920x1080 píxeles. Pero teniendo tanta capacidad de cómputo, sorprende que no se explote el procesamiento del audio tal y como ocurre en vídeo e imagen.

En este proyecto se estudia la viabilidad del procesamiento de audio en tiempo real en sistemas heterogéneos. Además, se analizan las dificultades que se dan en el uso de procesadores gráficos discretos frente a los procesadores gráficos integrados.

Junto al desarrollo de un sistema de grabación y reproducción de audio, se presenta un nodo de cómputo que abarca desde la ejecución de algoritmos de procesamiento hasta la ejecución de kernels en OpenCL, pasando por optimizaciones realizadas con OpenMP.

Una vez se tiene el desarrollo de la aplicación con todos sus componentes funcionales se realizan una serie de experimentaciones sobre el sistema, para encontrar la configuración del sistema que ofrezca un mayor rendimiento, y sobre las optimizaciones realizadas en OpenMP.

Luego se analizan las resultados de estas experimentaciones para ver qué parámetros del sistema y qué optimizaciones realizadas ofrecen un mayor rendimiento. Una vez desarrollada la aplicación multiplataforma, falta la experimentación con OpenCL.

Por lo general, los filtros de audio son secuenciales, por lo que su paralelización es más difícil o no se aprovecha el rendimiento del procesador gráfico. De este modo, se consiguen el cálculo de la transformada rápida de Fourier, como un cómputo complejo y limitado; y un algoritmo umbral trivial, para saturar las comunicaciones y no el cómputo. Con estos cómputos se estudian las latencias de entrada y salida de los datos, así como el tiempo de procesamiento.

Palabras clave: audio, OpenCL, sistema heterogéneo, tiempo real, multiplataforma

Abstract

In the last years, the graphic card's development has increased, but the thing that hadn't been developed is the different functions that can do this type of processors. These graphic processors perform incredibly good in videogame execution, moving these games in a 1920x1080 pixels resolution at 60 frames per second. But, taking into account such computing capacity, it surprises that the audio processing is not being used as it happens with the video and image.

The project's theme is the audio processing viability on heterogeneus systems. Also, the difficulties between using a discrete GPU versus an integrated GPU are analyzed.

With the development of a multi platform audio file recorder and player, a compute node that applies from algorithm processes to OpenCL's kernels executions through optimizations done with OpenMP, it's presented.

Once the system is developed, some experimentations are done on the system in order to find the best system configurations to obtain a higher performance and in order to decide which OpenMP optimization is better.

Then, the results are analyzed so as to obtain the best parameters and the best OpenMP optimizations. The multi platform systems is ready, now it's turn for OpenCL.

In general, audio filters are sequential, so its parallelization is harder. Based on that, it's decided to work with the Fast Fourier Transform and the process that was optimizew with OpenMP, in order to study the latency of the input and output of the data as well as the processing time.

Keywords: audio, OpenCL, heterogeneus system, real time, multi platform

Capítulo 1

Introducción

1.1. Sistemas heterogéneos

Un sistema heterogéneo compuesto por CPU+GPU consiste en el uso de una tarjeta gráfica para realizar cálculos que requieran una gran cantidad de cómputo. Actualmente, los sistemas heterogéneos son utilizados por múltiples motivos: **tienen un gran rendimiento para tareas con un alto paralelismo de datos, ofrecen una buena relación energética en FLOPS/Vatio y son baratos para las prestaciones que ofrecen** (mucha gente tiene acceso a CPUs y GPUs de última generación con propósitos recreativos).

Para aprovechar esta arquitectura es necesario utilizar un nuevo modelo de programación [1]. De este modo, la parte secuencial de código se ejecutará en la CPU, mientras que la parte que requiere una gran capacidad de cómputo se ejecutará en la GPU. Esta forma de programación es llamada *host-device* e implica la introducción de sobrecarga de sincronización en la entrada y la salida de los datos de la GPU por el transporte de los datos.

Existen paradigmas de programación, como OpenCL, que permiten el desarrollo de aplicaciones capaces de utilizar los recursos del sistema de forma conjunta. OpenCL permite una programación portable, de forma que el mismo código se puede ejecutar en diferentes dispositivos, debido a que está basado en versiones antiguas de los lenguajes de programación C y C++ (C99 y C++11).

1.2. Procesamiento de Audio en Sistemas Heterogéneos

La falta de estudios o la dificultad para encontrarlos, sobre procesamiento de audio en aceleradores gráficos, no permite una conclusión clara sobre si merece la pena o no llevar este procesamiento a unidades gráficas de cómputo. La conclusión que más se repite es que no merece la pena el procesamiento en procesadores gráficos, ya que la latencia de transmisión de datos en la entrada y la salida de la tarjeta gráfica no compensan la velocidad de cómputo. También hay que destacar que para acercarse al procesamiento en tiempo real, los procesamientos tienen que ser con la menor información posible y a un ritmo muy rápido. El audio se procesa en paquetes, los cuales contienen la información de dicho audio. Antes de un procesamiento se fija un tamaño de paquete, lo cual influye en el ritmo de envío de esos paquetes, a mayor tamaño menor ritmo de envío. Teniendo en cuenta el concepto de paquete, en el procesamiento de audio es importante la latencia que conlleva el envío de estos

paquetes y el número de paquetes que se envían, porque a más paquetes más latencia.

Por otro lado, se cuestiona que estos procesadores no estén diseñados para este tipo de operaciones: si están diseñados para tratar flujos de datos de ese modo o que la optimización y paralelización del código tiene que estar adaptado a la arquitectura del procesador, sin contener errores, dado que esto desemboca en una latencia mayor.

Por lo observado en diferentes estudios[2], las unidades de procesamiento gráfico con las que se trabajan son GPUs de gama alta, lo cual requiere una conexión PCI para el funcionamiento y para la transferencia de datos. **¿Qué ocurre si se utiliza un procesador gráfico integrado en la CPU?**

En la Figura 1.1 se puede observar la arquitectura de CPU conectada a una GPU discreta. En dicha figura, se observa cómo cada procesador tiene su propia memoria y para la comunicación entre los mismos es necesaria una conexión PCI-e. Esta conexión, más el uso de diferentes memorias entre procesadores, es uno de los principales motivos por los que se genera la sobrecarga de comunicación.

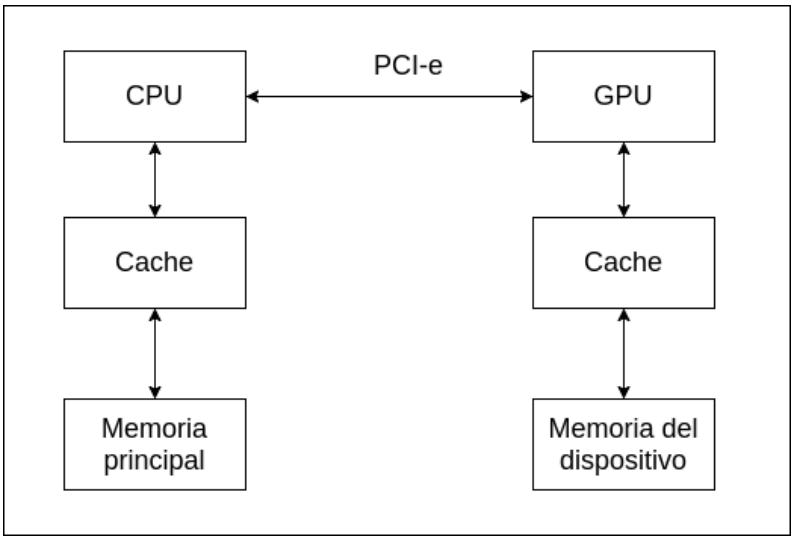

Figura 1.1: Arquitectura CPU + GPU discreta.

En el caso de la Figura 1.2, se observa la arquitectura de CPU con una GPU integrada. En esta arquitectura, CPU y GPU, comparten la memoria sobre la que trabajan, por lo que las latencias comentadas anteriormente deberían desaparecer o, al menos, verse reducidas notablemente. Esta es la arquitectura utilizada para la experimentación con sistemas heterogéneos en este trabajo.

Con este trabajo se intenta aportar un grano de arena al estudio sobre estos procesamientos en este tipo de computación.

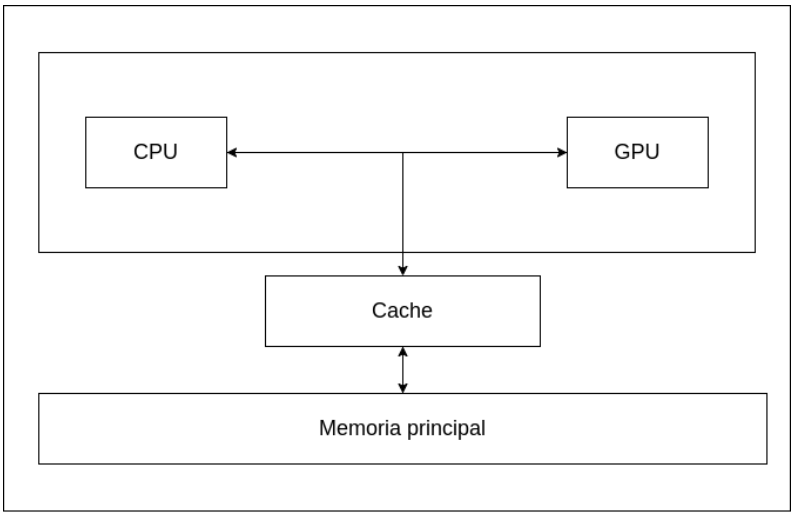

Figura 1.2: Arquitectura CPU + GPU integrada.

1.3. Descripción del Problema

El propósito del proyecto es avanzar en el estudio sobre la viabilidad del procesamiento de audio en sistemas heterogéneos, más concretamente en procesadores gráficos integrados. A la vez que se ofrece una plataforma integral, extensible y multiplataforma para la visualización y captura/reproducción de audio.

El procesamiento de audio consiste en realizar algún tipo de operación sobre los valores de las muestras del audio. Se pueden realizar una gran cantidad de procesamientos como: el cálculo de la FFT; aplicación de filtros, por ejemplo, para eliminar frecuencias o bandas de frecuencias; o aplicar efectos que modulen el sonido original, como por ejemplo un efecto de reverberación.

Se busca que este procesamiento se ejecute en tiempo real o casi en tiempo real. Para conseguirlo, independientemente del procesamiento a realizar, **se busca un sistema con una gran capacidad de cómputo y latencias bajas para la entrada y salida de los datos.** De este modo se plantea utilizar un sistema heterogéneo, formado por una CPU y una GPU, con una diferencia entre otras investigaciones encontradas: el procesador gráfico se encuentra integrado en la CPU.

El uso de un sistema heterogéneo implica un cambio en el modelo de programación, lo que quiere decir que la programación tiene que ser orientada al entorno en el que se va a ejecutar, la GPU. Esta programación difiere bastante de la programación convencional en cuanto a la gestión de hilos de cómputo.

1.4. Objetivos

El objetivo principal de este trabajo es desarrollar un sistema multiplataforma que permita la reproducción, procesamiento y grabación de audio, con un rendimiento suficiente para que

pueda hacerse en tiempo real, sobre un dispositivo heterogéneo con GPU integrada. Este objetivo principal se descompone en una serie de objetivos concretos, como son:

- Estudio de las tecnologías actuales, tanto hardware como software.
- Desarrollo de una aplicación que permite la reproducción y grabación de ficheros de audio, comunicaciones con el nodo encargado de los procesamientos y otras funcionalidades secundarias.
- Optimización de la aplicación en sistemas heterogéneos mediante técnicas de paralelización y vectorización.
- Evaluación de los diferentes desarrollos y experimentaciones realizados.

Una vez alcanzado el objetivo propuesto, se estará en disposición de contestar a la pregunta previamente planteada, y obtener una primera conclusión sobre si merece la pena o no procesar audio en procesadores gráficos, comparando tiempos, de implementaciones realizadas con diferentes optimizaciones y ejecutadas en distintos tipos de procesadores, sobre un algoritmo o cómputo específico.

1.5. Metodología y plan de trabajo

Para la realización de este trabajo se organizan los diferentes objetivos con tareas. Para conseguir un objetivo, se tienen que haber finalizado todas sus tareas. Estas tareas no tienen un tiempo límite, pero sí tienen un grado de importancia en cuanto a su realización: unas pueden ser fundamentales, con las cuales sin ellas el proyecto no podría finalizarse, y otras serán opcionales que sirven para darle puntos de calidad al proyecto. Semanalmente o cada dos semanas, se ha realizado una reunión con el director para comentar las tareas realizadas, valorar el avance del objetivo y añadir o eliminar tareas. Durante el último mes de trabajo las reuniones se realizaban cada tres o cuatro días.

En cuanto al plan de trabajo del proyecto, cabe destacar que este trabajo nace de una práctica opcional de Programación Paralela, Concurrente y de Tiempo Real, pero debido al interés por el tema en cuestión se propuso la continuación como Trabajo de Fin de Grado. Previo a la definición del plan de trabajo, que se explica a continuación, se desarrolló un prototipo durante la realización de dicha práctica y durante las primeras semanas de realización del Trabajo de Fin de Grado.

Posteriormente, se llevó a cabo el siguiente plan de trabajo:

- Implementación básica de sistema de grabación/reproducción de audio y recopilación de información sobre cómo trabajar con audio.
- Adaptación de la implementación básica a captura de paquetes de audio y envío de los mismos a los diferentes componentes del sistema.
- Evaluación del sistema de comunicaciones con el objetivo de encontrar las mejores configuraciones para el programa.

- Implementación de procesamientos de audio, y posteriormente, optimización de los mismos con OpenMP e hilos de C++. Realización de comparativas entre el procesamiento y su optimización.
- Optimización de los procesamientos previos en OpenCL.

1.6. Estructura del documento

Además del actual capítulo de introducción, el documento está formado por otros 4 capítulos adicionales estructurados de la siguiente manera:

- Capítulo 2: **Herramientas empleadas.** Se realiza una exposición de la estructura del sistema. Descripción de las herramientas utilizadas durante el desarrollo y el por qué, así como también se exponen herramientas con las que se hicieron pruebas pero que al final se descartaron.
- Capítulo 3: **Desarrollo del sistema.** Se centra en los tres componentes principales del sistema junto a sus fases de desarrollo. Se profundiza en las características de cada componente y en cómo se han utilizado las herramientas que se explican en el Capítulo 2. También se explica el desarrollo de las diferentes experimentaciones que se realizan.
- Capítulo 4: **Análisis de los resultados.** Explicación de las diferentes experimentaciones realizadas sobre el sistema junto a los objetivos de las mismas. Análisis y resumen de las experimentaciones, así como conclusiones sobre las mismas.
- Capítulo 5: **Conclusiones y trabajos futuros.** En este último capítulo se explican una serie de conclusiones sobre el desarrollo de este sistema: problemas, objetivos logrados, aprendizaje, entre otras cosas. También se indicarán unos conceptos relevantes sobre las aplicaciones que, en un futuro, podría tener el sistema, teniendo en cuenta los avances conseguidos.

Capítulo 2

Herramientas empleadas

En este capítulo se van a presentar las herramientas utilizadas en el desarrollo del programa, así como herramientas que han sido descartadas. El diseño del programa que se ha desarrollado se divide en tres grandes componentes, como se puede ver en la Figura 2.1. Estos componentes tienen funciones diferentes, desde crear la interfaz de usuario hasta analizar cada muestra de audio que se esté procesando en ese momento.

Aparte de los tres componentes que forman el sistema, también se han realizado implementaciones para el análisis de los datos y la representación gráfica de los mismos.

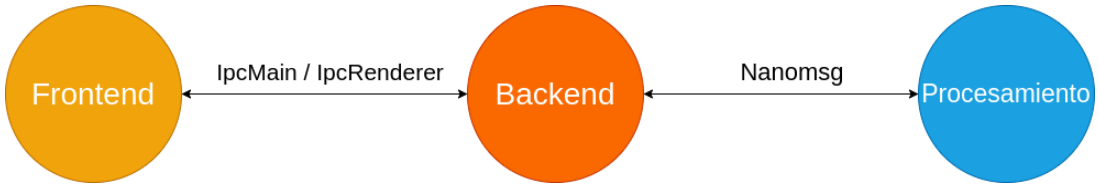

Figura 2.1: Componentes del proyecto.

2.1. Frontend: Interfaz Gráfica

El *frontend* es la interfaz gráfica de usuario, que se encarga de la grabación y reproducción de audio, de la representación gráfica del mismo y del establecimiento de parámetros para el procesamiento. Se ha optado por esta estrategia por su flexibilidad y capacidad de operar en multiplataforma, siguiendo estándares HTML5. Para su implementación se han utilizado las siguientes herramientas.

2.1.1. Electron

Electron es un *framework* de código abierto que permite el desarrollo de aplicaciones gráficas de escritorio usando componentes del lado del cliente y del servidor, originalmente desarrolladas para aplicaciones web [3]. Se usa en grandes proyectos de código abierto como el servicio de mensajería instantánea Discord y los editores de código fuente Atom y Microsoft Visual Studio Code.

En el desarrollo de aplicaciones con Electron, la interfaz funciona con un Chromium [4], navegador web de código abierto desarrollado por Google. Esto permite que el desarrollo de la

interfaz se pueda hacer con programación web (HTML, CSS y JavaScript).

La programación web permite una comunicación muy sencilla entre el lado del servidor y la interfaz mediante el uso de los módulos ipcMain[5] e ipcRenderer[6], integrados en Electron. Estos consisten en emisores de eventos que serializan los argumentos que se quiere enviar con el algoritmo de clonación estructurado [7] para poder comunicar JavaScript con Node.js.

Para la representación del audio se ha optado por una implementación propia. Previamente se había planteado utilizar wavesurfer.js[8], una librería para la representación gráfica del audio, pero se tuvo que descartar por limitaciones de la propia librería. También se descartó el uso de PyQt5[9], bindings a la librería Qt[10] para la creación de interfaces, por la complejidad de la creación de la interfaz y porque el componente de web que tiene es mucho más limitado que un navegador convencional.

2.1.2. Web Audio API

La Web Audio API[11] provee un sistema poderoso y versátil para controlar audio en la Web. Permite manejar operaciones de audio dentro de un contexto. Las operaciones básicas son realizadas con nodos de audio, que están enlazados juntos para formar un gráfico de encaminamiento.

Estos son los nodos utilizados en el desarrollo:

- ScriptProcessorNode[12]: nodo de procesamiento de audio conectado a dos *buffers*, uno que contiene los datos de audio de entrada y el otro el audio procesado de salida. **El nodo de procesamiento de audio permite definir el tamaño del bloque a procesar. Se utiliza para la extracción en bloques de los valores del audio según se va reproduciendo el mismo.** Este nodo se encuentra obsoleto debido a la baja especificación que tiene, dando lugar a diferentes implementaciones en cada navegador. A pesar de estar obsoleto, se ha usado este nodo ya que la otra alternativa, AudioWorkletNode [13], impone una restricción de 128 muestras por bloque, por lo que esa limitación no permite realizar un estudio del rendimiento de las comunicaciones entre componentes.
- AnalyserNode[14]: el nodo de análisis proporciona información sobre el bloque que se está procesando, como la frecuencia en tiempo real. Pasa el flujo de audio sin modificación desde el origen de entrada a la salida, pero permite obtener los datos generados, procesarlos y crear visualizaciones de audio.

2.2. Backend: Nodo de Comunicación

El *backend* es un nodo intermedio que se encarga de las comunicaciones entre el *frontend* y el nodo de procesamiento. Realiza tareas menores como la inicialización de parámetros del procesamiento del audio.

2.2.1. Node.js

Node.js es un entorno en tiempo de ejecución multiplataforma basado en JavaScript [15]. Al ser asíncrono y estar basado en una arquitectura orientada a eventos, permite de manera fácil y rápida el control de eventos. Por ejemplo, permite reaccionar a la finalización de la carga de un

fichero o al comienzo de la reproducción de un fichero de audio.
En este sistema, Node.js se utiliza para definir la comunicación entre el *frontend* y el nodo de procesamiento, así como la generación de archivos WAV[16] con el audio grabado, debido a que el *frontend* es un *sandbox* de navegador y no accede al sistema operativo.

2.2.2. Nanomsg

Nanomsg es una librería de *sockets* que proporciona patrones de comunicación comunes [17]. **El objetivo es hacer la capa de red más rápida, escalable y fácil de usar.** Está implementado en C y funciona en una gran variedad de sistemas operativos sin ninguna otra dependencia.

Los protocolos de escalabilidad son bloques básicos para la construcción de sistemas distribuidos. Combinándolos se obtienen una amplia gama de aplicaciones distribuidas. Los siguientes patrones son los que se encuentran disponibles:

- PAIR: comunicación simple de uno a uno.
- BUS: comunicación simple de muchos a muchos.
- REQREP: permite construir un cluster de servicios sin estados para procesar las peticiones del usuario.
- PUBSUB: distribuye mensajes a conjuntos grandes de suscriptores interesados.
- PIPELINE: Agrega mensajes de múltiples fuentes y los equilibra entre muchos destinos.
- SURVEY: permite consultar el estado de múltiples aplicaciones de una sola vez.

En este sistema solo se utiliza el protocolo de escalabilidad PAIR, ya que la topología del sistema de comunicación no es tan compleja como para usar ninguno de los otros.

Los protocolos de transporte se encargan de la transferencia libre de errores entre el emisor y el receptor y de mantener el flujo de la red. Nanomsg tiene los siguientes protocolos de transporte disponibles:

- INPROC: transporte dentro del proceso.
- IPC: transporte entre procesos de una sola máquina.
- TCP: transporte de red via TCP.
- WS: *sockets* web que funcionan sobre TCP.

El uso de INPROC no está permitido en el sistema, ya que cada componente está ejecutado en un proceso, con lo que la comunicación mediante este protocolo es imposible.

En la Sección 3.6 se presenta un estudio con IPC, TCP y WS para ver cuál de los tres ofrece un rendimiento mayor.

2.3. Nodo de procesamiento paralelo

El nodo de procesamiento se encarga de todo el procesamiento del audio: aplicación de filtros, detección de picos de audio y cálculo de FFT.

2.3.1. C++

Para la programación del nodo de procesamiento se usa C++ que es un lenguaje de programación de alto nivel orientado a objetos que expande el lenguaje de programación C con mecanismos que permiten la manipulación de objetos [18].

Para mantener una organización de todo el código, evitar repeticiones del mismo y aligerar las pruebas realizadas, se han utilizado las siguientes características de C++:

- Los ficheros de cabecera son archivos que incluyen declaraciones de variables y funciones entre otras cosas, con el objetivo de definir un estándar dentro del programa sin tener que repetir código. Estos ficheros de cabecera se añaden a los ficheros de código fuente en los que se utilicen dichas variables o funciones definidas.
- Las directivas de preprocesador son instrucciones que escribe el programador para que el compilador realice ciertas funciones, como por ejemplo: importar librerías que proporciona el propio compilador, importar ficheros de código fuente propios, definir macros que se utilizan como variables y más funciones no usadas en el desarrollo de este proyecto.
- Gracias a las directivas de preprocesador existe la compilación condicional, que permite compilar solo el código que se necesite en cada momento, usando las macros y las inclusiones condicionales.

  Las inclusiones condicionales permiten que una sección del programa sea compilada solo si la macro especificada como parámetro ha sido definida, sin importar su valor. Como se puede ver en la Figura 2.2, si la macro `TABLE_SIZE` está definida, entonces se definirá un puntero con el valor de la macro como tamaño.

```
// Codigo a compilar solo si la macro TABLE_SIZE esta definida
#ifdef TABLE_SIZE
    int table[TABLE_SIZE];
#endif
```

Figura 2.2: Ejemplo de compilación condicional.

2.3.2. OpenMP

OpenMP[19] es un modelo de programación portable y escalable que proporciona a los programadores una interfaz simple y flexible para el desarrollo de aplicaciones paralelas de memoria compartida, para plataformas que van desde las computadoras de escritorio hasta supercomputadores.

Estos programas paralelos se implementan mediantes hilos software, que son secuencias de tareas encadenadas que pertenecen a un mismo proceso. Los hilos pueden ser ejecutados a la vez en un mismo sistema de memoria compartida, donde comparten los recursos y se comunican a través del espacio de memoria asignado al proceso padre, que es de donde parten.

Un programa en OpenMP empieza con un proceso y con un hilo maestro. Si durante la compilación no se encuentra ninguna directiva de OpenMP, el programa se ejecutará secuencialmente en un solo núcleo del procesador. El programador tendrá que detectar los fragmentos de código en los que se pueda aplicar algún tipo de paralelización y deberá usar las directivas proporcionadas por OpenMP para llevar a cabo la paralelización. Las directivas de OpenMP siempre empiezan con `#pragma omp`.

Para la creación de los *threads* es necesaria la directiva `#pragma omp parallel`, que es la que indica que ese bloque se va a ejecutar por más de un *thread*. OpenMP nos permite diferentes modos de paralelización y de vectorización:

La paralelización de datos permite dividir las iteraciones de un bucle entre todos los *threads* de los que se disponga. Esto hace que la ejecución del bucle sea mucho más rápida. Para la paralelización de datos, OpenMP proporciona la directiva `for`, la cual tiene diversos parámetros, entre los que cabe destacar:

- `private()`: para definir las variables que son privadas para cada *thread*.
- `schedule()`: que se encarga de como se dividen las iteraciones del bucle entre los diferentes *threads* que lo ejecutan en función de parámetros, como pueden ser `dynamic, static` y `guided`. `Schedule()` puede llevar otro parámetro más, un número mayor que 1, que define el tamaño mínimo de las iteraciones que hace cada *thread*.

En la Figura 2.3 se puede ver un ejemplo del uso de la directiva `for`. En este ejemplo, se puede ver cómo se crea una región paralela con el número máximo de *threads* permitidos por el ordenador donde se ejecuta, `omp_get_max_threads()` devuelve el número de *cores* lógicos y físicos que tiene la plataforma donde se está realizando la ejecución del código. Cuando la ejecución llega al bucle `for`, las iteraciones de este se dividen por igual entre los *threads*.

```
#pragma omp parallel num_threads(omp_get_max_threads())
{
    #pragma omp for schedule(static)
    for(size_t i=1;i < audio_size;i += 1)
    {
        out_original[i] = audio[i];
    }
}
```

Figura 2.3: Ejemplo de código con paralelización de datos.

La paralelización de tareas permite ejecutar bloques de código, no limitados al contenido de un bucle `for` como en la paralelización de datos, sino a, por ejemplo, la ejecución de una función. Cuando la tarea es generada, no se ejecuta hasta que un *thread* es asignado a realizarla.

Para la paralelización de tareas, OpenMP proporciona la directiva `task`. Durante la ejecución de un código, cada vez que se encuentre la directiva `task`, se generará una tarea, que, en el momento en el que haya un *thread* libre, la ejecutará. Otra directiva muy importante de la paralelización de tareas es `taskwait`, que hace de barrera a los *threads* hasta que todas las tareas han sido ejecutadas.

Como se puede ver en la Figura 2.4, se crea una región paralela como en la Figura 2.3 y acto seguido se utiliza el pragma `single` para que un solo *thread* ejecute ese bloque de código. Cuando la ejecución llega al bucle `for` se encuentra el pragma `task`, con lo que se va a generar una tarea por cada iteración del bucle. Cada tarea se irá ejecutando según van quedando *threads* disponibles. Una vez se generen todas las tareas, el *thread* que las ha generado se encuentra con el pragma `taskwait`, lo que quiere decir que para poder avanzar, todas las tareas tendrán que haber acabado.

```
#pragma omp parallel num_threads(omp_get_max_threads())
{
    #pragma omp single nowait
    {
        for(int i=0; i < MAX_BUFFER; i++){
            #pragma omp task
            {
            processing((float*)ptr[next_to_process].buf);
            }
            next_to_process++;
        }
    }
}
#pragma omp taskwait
```

Figura 2.4: Ejemplo de código con paralelización de tareas.

Para la vectorización, OpenMP proporciona la directiva `simd`, la cual se puede usar sola (en bucles, en cabeceras de métodos junto a `declare`) y se puede usar junto la directiva `for`.

En la Figura 2.5 se puede ver el uso de la directiva `simd` junto a la directiva `for`. En este ejemplo, el bucle `for` también es dividido entre los *threads*, además, cada bucle parcial que ejecuta un *thread*, es vectorizado para que múltiples iteraciones de cada bucle parcial, sean ejecutadas por instrucciones SIMD.

```
#pragma omp parallel num_threads(omp_get_max_threads())
{
    #pragma omp for simd
    for(size_t i=1;i < audio_size;i += 1)
    {
        out_original[i] = audio[i];
    }
}
```

Figura 2.5: Ejemplo de código con paralelización de datos y vectorización del bucle.

En la implementación se ha utilizado OpenMP para realizar las operaciones de procesamien-

to de manera más rápida y también se ha realizado una comparativa para los diferentes modelos de paralelización y vectorización empleados.

2.3.3. OpenCL

OpenCL (*Open Computing Language*) [20] intenta aprovechar la potencia de los aceleradores para realizar operaciones intensas repartidas entre el procesador del equipo (CPU) y la GPU de cualquier tarjeta gráfica compatible. En este proyecto se usa para estudiar la viabilidad de las GPUs integradas en el procesamiento de audio.

Que sea abierto y libre permite llevar OpenCL a un entorno multiplataforma, pudiendo ser aprovechado sobre cualquier plataforma y sistema operativo. Algo muy importante, ya que los esfuerzos de los programadores para adaptar las aplicaciones y que aprovechen las ventajas de OpenCL no dependerán luego del hardware o sistema operativo que tenga la máquina del cliente.

Una de las principales características de OpenCL es su portabilidad, debido a su modelo de ejecución *host-device*. Esto permite la ejecución de código OpenCL en cualquier aplicación conforme, lo que permite una fácil reutilización de código.

Otro característica a destacar es su complejidad para realizar los procesamientos correctamente. No tanto por el desarrollo del código, sino por las diferentes optimizaciones que se pueden aplicar.

2.4. Análisis de los datos y generación de gráficas

Este apartado no pertenece en si al desarrollo de la aplicación, pero sí ha determinado los mejores parámetros para obtener el mayor rendimiento.

Para lograrlo, se ha utilizado Python, que es un lenguaje de programación multiplataforma y orientado a objetos, con el que se pueden hacer aplicaciones multiplataforma, páginas web, servidores de red e incluso, drivers. Es un lenguaje interpretado, lo que significa que no se necesita compilar el código fuente para poder ejecutarlo [21].

Para el análisis de los datos se han utilizado diferentes librerías:

- Pandas es una biblioteca para manipulación y análisis de datos. Ofrece estructuras de datos y operaciones para manipular tablas numéricas y series temporales [22]. Principalmente se ha utilizado para la manipulación de los datos, ya que te permite trabajar con ficheros CSV y también permite su generación, con lo que, fácilmente, se pueden generar las tablas que después se van a representar gráficamente.
- Seaborn es una librería basada en Matplotlib[23] que proporciona una interfaz de alto nivel para dibujar gráficas[24]. A diferencia de la integración que tiene Matplotlib, Seaborn permite una programación más sencilla para generar una gráfica más sencilla de leer que Matplotlib.

Capítulo 3

Desarrollo

En este capítulo se presentan las diferentes etapas del proceso de desarrollo de cada componente de la aplicación, así como el desarrollo de las pruebas realizadas sobre el sistema y el diseño de las mismas. El capitulo será dividido en los tres componentes del sistema, añadiendo también dos secciones iniciales sobre el objetivo de la implementación de este sistema y el primer prototipo de la aplicación, realizado en Python, y dos secciones al final sobre las pruebas realizadas del software desarrollado y las pruebas realizadas con OpenCL.

Cada componente tiene una finalidad distinta. El *frontend* tiene como objetivo interaccionar con el usuario mediante la interfaz gráfica y los dispositivos de entrada/salida. El nodo de comunicación permite la interacción de los distintos componentes, haciendo de puente entre ellos, desarrollando también múltiples funcionalidades altamente relacionadas con la interfaz gráfica. El nodo de procesamiento se encarga de la aplicación de filtros, análisis del audio y más funcionalidades relacionadas con el procesamiento de los datos de audio.

3.1. Objetivos y funcionalidades deseadas

El objetivo de la implementación es la realización de un sistema multiplataforma que permita procesar audio en diferentes tipos de aceleradores, en especial procesadores gráficos, ya que no es una rama que se haya estudiado en profundidad. Dado que el sistema tiene que ser rápido, aparte de la implementación del programa, **se necesita realizar estudios sobre diferentes aspectos de las herramientas que se utilizan, como por ejemplo, los diferentes protocolos de comunicación que se pueden utilizar y sus rendimientos o qué tamaños de paquete de audio permiten un rendimiento mayor.**

El sistema debe contener ciertas funcionalidades básicas como la grabación y reproducción de ficheros de audio, configuración modificable del sistema para diferentes situaciones, la capacidad de representar gráficamente el audio, capacidad de adaptación a otro sistema distinto de procesamiento, entre otras.

El programa requiere ciertos parámetros de entrada para las diferentes funcionalidades, como el tamaño de los paquetes a procesar para las funcionalidades de grabación y reproducción, o el valor del umbral a superar para la detección de picos de audio.

De la misma manera, se van a generar unas salidas, como los ficheros que contienen toda la

información de una prueba realizada o el fichero de audio que se ha grabado.

3.2. Prototipado en Python

En este primera sección del desarrollo se describe el origen del proyecto, cómo se abordó el problema y cuales eran los objetivos en un principio, ya que previo a la implementación con Electron, se desarrolló parte del sistema en Python.

Este proyecto viene de una práctica opcional de la asignatura Programación Paralela, Concurrente y de Tiempo Real, la cual consistía en la realización de un programa que procesase audio y lo representará en una interfaz gráfica diseñada por el alumno. Había unas pautas impuestas para la realización, tenía que ser realizada en Python, con la librería PyQt5 para interfaces gráficas y con las librerías Wave y PyAudio para la gestión del audio.

Este primer prototipo en Python, cuya interfaz gráfica se puede ver en la Figura 3.1, permite grabar y reproducir ficheros de audio, modificar los parámetros de grabación y reproducción, cambiar dispositivos de entrada y salida predeterminados y más funcionalidades relacionadas con la reproducción y grabación.

De cara al procesamiento, se encontraron limitaciones en el sistema, con lo que se decidió reconstruir el proyecto en otra tecnología distinta. Dichas limitaciones obligaban al sistema, en una etapa temprana del desarrollo, a hacer una gestión de la interfaz muy compleja. Otra limitación a destacar era la sobrecarga del sistema con el procesamiento y la representación gráfica del audio.

Esta nueva tecnología, dividida en tres componentes distintos, como se puede ver en la Figura 2.1, permite aislar la reproducción/grabación y representación del audio de su procesamiento, con lo que elimina las limitaciones dadas en Python.

El desarrollo de este prototipo ha servido para entender como funcionan los ficheros de audio y la gestión de los mismos. También destacar la comprensión de cómo se trabaja con los datos de audio y sus características. Desde las cosas más básicas, como la frecuencia de muestreo hasta cosas más complejas, como la cabecera de los ficheros WAV.

3.3. Frontend: Configuración del programa y sistema de grabación/reproducción

Este primer componente del sistema se encuentra altamente relacionado con el segundo, el nodo de comunicación, debido a que estos dos componentes se crean por el *framework* Electron, la interfaz gráfica y el nodo que actúa de servidor. Esto quiere decir que su desarrollo también está muy relacionado, ya que permiten una comunicación muy sencilla y existen ciertas funcionalidades que requieren de ambos componentes para el correcto funcionamiento de las mismas.

El desarrollo de este primer componente se encuentra dividido varias fases: pruebas iniciales del sistema, implementación del sistema de captura de paquetes, representación gráfica del audio, interacción de la interfaz y establecimiento de las comunicaciones con los distintos componentes.

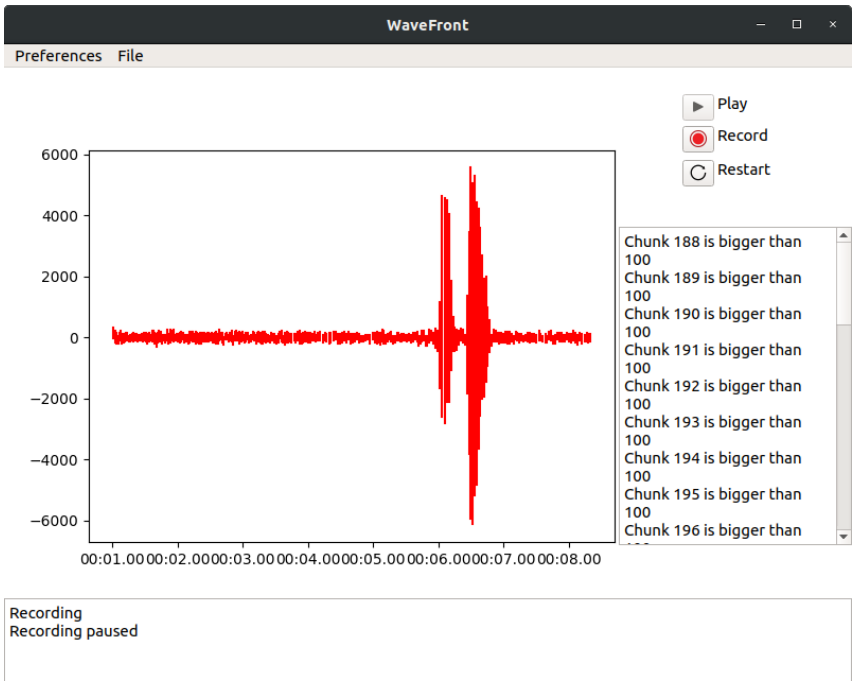

Figura 3.1: Interfaz del prototipo en Python.

3.3.1. Pruebas iniciales

Esta primera fase consiste en una toma de contacto con el sistema de Electron para la realización de interfaces gráficas con tecnología web. El objetivo es obtener una implementación mínima del sistema de grabación y reproducción, sin posibilidad aún de dividir el audio generado o reproducido en bloques para su posterior procesamiento. El sistema de grabación utilizado es MediaRecorder[25], incluido en la Web Audio API. Para la reproducción del audio se usa la implementación dada en HTML, la etiqueta `<audio>`, que permite la reproducción de ficheros y fragmentos de audio recién grabados.

3.3.2. Captura de paquetes de audio

Una vez se tiene el sistema de grabación y reproducción de audio, se empieza con la implementación del sistema de captura de paquetes de audio. Para ello se utilizan las herramientas que proporciona la Web Audio API, como el `ScriptProcessorNode`, mencionado en la Sección 2.1.2.

Para la creación del `ScriptProcessorNode` es necesario definir un tamaño de paquete, siempre en potencias de 2, entre 256 hasta 16384; el número de entradas que va a tener, que en este caso va a ser uno, y el número de salidas, que también va a ser uno. **El tamaño del paquete indica cuantas muestras de audio se van a almacenar en cada uno y por ende, cuantas muestras se procesan a la vez. Un número más grande indica tamaños de paquete más grandes y por lo tanto un menor número de paquetes.**

Con el `ScriptProcessorNode` ya definido hay que crear el contexto de trabajo para usar el resto de componentes de la Web Audio API. La creación del contexto de audio se divide en dos:

- Grabación: para la grabación es necesario crear un contexto solo para ella, ya que no se pueden compartir entre grabación y reproducción, debido a que los *streams* de datos vienen de distintas fuentes. Una vez se crea el contexto de audio con `AudioContext` [26], se genera toda la infraestructura para capturar el *stream* de datos que genera el dispositivo de entrada de audio por el que se graba. Una vez creado el input del `ScriptProcessorNode`, ya se pueden conectar.

- Reproducción: para la reproducción hay una diferencia en la implementación, ya que el *stream* de datos no viene de la misma fuente. En este caso, los datos provienen del reproductor de audio, con lo que el *stream* que hay que generar no está ligado directamente el fichero que se quiere reproducir. Por ello, la Web Audio API se utiliza de tal modo que, instantes antes de empezar la reproducción y por ende, la captura de datos, genera paquetes de datos con valor cero en todas sus muestras. Estos ceros son eliminados en las diferentes componentes del sistema una vez se realiza la generación del fichero de audio, pero no se ha podido eliminar completamente del procesamiento. Una vez se genera el *stream* de datos a partir del reproductor, la implementación y el funcionamiento son análogos a la grabación.

El `ScriptProcessorNode` y la gran mayoría de los nodos que se utilizan de Audio Web API, funcionan por medio de eventos de JavaScript. Esto quiere decir que cuando un paquete ya está listo para ser procesado, el `ScriptProcessorNode` recibirá una notificación, que por medio de los eventos de JavaScript, ejecutará una función previamente definida, que en este caso, enviará los datos del paquete al nodo de comunicación para su posterior envío al nodo de procesamiento.

3.3.3. Representación gráfica del audio

Para la representación gráfica del audio se hará uso de otro nodo de Web Audio API, llamado `AnalyserNode`.

Este nodo solo necesita una entrada, mientras que no es necesario que tenga una salida, ya que el audio no es modificado, sino que los datos son generados para poder procesarlos y crear la visualizaciones del audio.

3.3.4. Interacción con la interfaz

Cuando se tienen las funcionalidades deseadas es necesario establecer el comportamiento correcto de la interfaz, ya que el programa pasa por distintos estados durante su ejecución. De este modo, establecer un correcto funcionamiento de los botones es importante, ya que permite definir los estados fácilmente habilitando o deshabilitando botones.

En la Figura 3.2 se pueden ver las distintas transiciones entre estados, como por ejemplo, el cambio entre detener la grabación y volver al estado inicial, o de estado de reproducción de un fichero pasar al estado de pausa de la reproducción.

3.3.5. Comunicación con el resto de componentes

Todas las acciones que se realizan en la interfaz gráfica tienen que verse reflejadas en el resto de los componentes del sistema. Para ello IpcMain e IpcRenderer, como se explica en la Sección 2.1.1, permiten la comunicación entre la interfaz gráfica generada por Electron y el nodo servidor

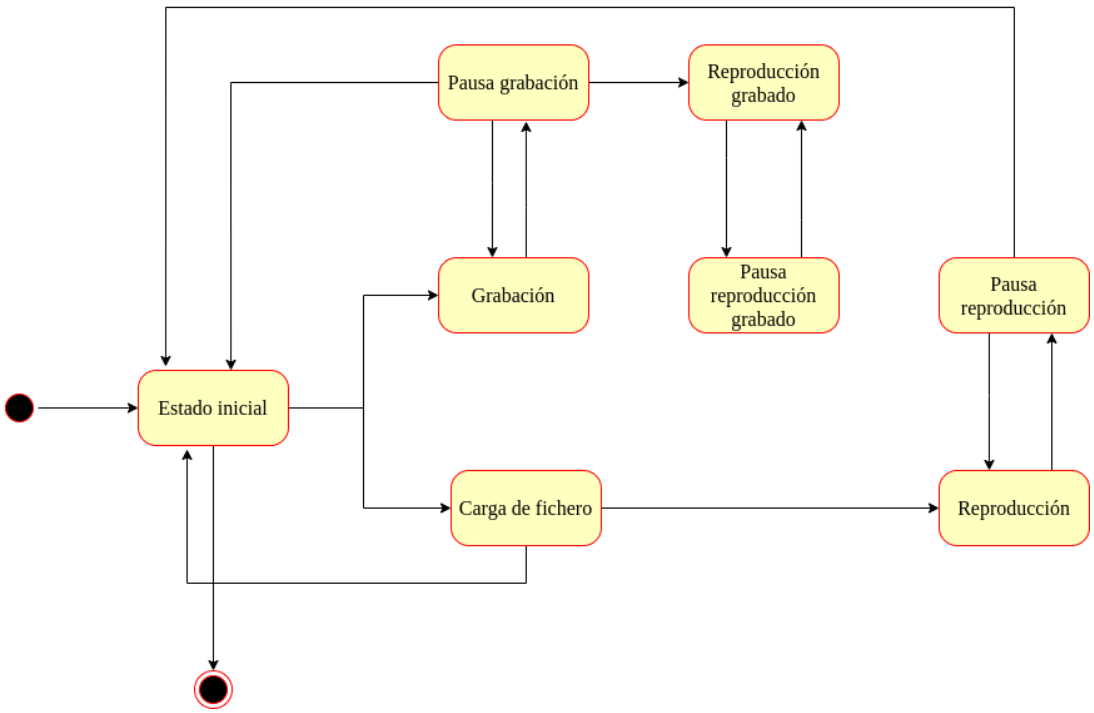

Figura 3.2: Diagrama de estados de la interfaz.

utilizado para las comunicaciones entre componentes.

Como se ha explicado previamente, JavaScript y Node.js funcionan mediante eventos. Del mismo modo funcionan sus comunicaciones. Estos emisores de eventos, IpcMain e IpcRenderer, permiten dejar definidas las acciones para todos los posibles casos de mensajes. Por ejemplo, si se quiere enviar una señal de stop para que se detenga el procesamiento, se enviará un mensaje que sea una cadena de caracteres que contenga "Stop". Una vez ese mensaje sea recibido en el otro extremo, se comprobará qué contiene el mensaje y si coincide con alguno de los patrones definidos, se realizará la acción correspondiente. En este caso notificar al nodo de procesamiento de que se ha parado el procesamiento de esa reproducción o grabación.

Esta comunicación es bidireccional, es decir, la comunicación puede ir del *frontend* al nodo de comunicaciones y viceversa.

3.4. Nodo de comunicación: gestión y unificación del sistema

El desarrollo de este componente está ligado al del *frontend*, debido a que las acciones que realiza la interfaz gráfica se tienen que notificar al nodo de procesamiento de algún modo, por lo que **el nodo de comunicaciones está hecho según el diseño del *frontend* y todas sus comunicaciones.**

Aparte de las funcionalidades de comunicación entre los otros dos nodos, también se incluyen ciertos aspectos y funcionalidades características del propio nodo, como son la generación de ficheros de audio en formato WAV [27], la inicialización de las distintas conexiones a los *soc-*

kets por medio de Nanomsg, la inicialización del sistema recogiendo los parámetros de entrada para las configuraciones iniciales y la generación de ficheros de texto con mediciones de tiempo durante la ejecución de los *benchmarks*.

El programa acepta diferentes parámetros de entrada. Estos parámetros alteran el comportamiento del programa desde el primer momento, por ejemplo, se puede arrancar el programa y que se reproduzca automáticamente un fichero de nuestra elección o se puede arrancar y que no realice ninguna acción.

Existen varios campos de valores de entrada, aunque no todos son obligatorios:

- Acción: acción que se desea realizar. Existen dos, grabación o reproducción de un fichero. Dependiendo de la acción otros valores de entrada serán necesarios o no. Este campo no es obligatorio.
- Tamaño de paquete: indica el tamaño de los paquetes de audio que se van a procesar. Siempre tiene que ser potencias de dos, empezando en 256 hasta 16384. Este campo es obligatorio siempre.
- Ruta a fichero: este campo indica la ruta y el nombre del fichero que se va a reproducir o donde se va a guardar lo grabado, dependiendo de la acción escogida. Este campo es obligatorio si se escoge una acción.
- Tiempo: tiempo en segundos que se desea grabar. Solo sirve si se ha seleccionado la acción de grabación.

Luego existen parámetros de entrada utilizados para un procesamiento determinado, como es el caso de la detección de picos de sonido en el audio. La implementación está hecha con una ventana deslizante, con lo que es necesario determinar ciertos parámetros de la ventana antes de la ejecución:

- Umbral: valor numérico de 0 a 100 que indica el valor que debe de superar el audio a procesar para que se notifique si ha superado dicho umbral.
- Tamaño de ventana: valor numérico que indica el tamaño de la ventana deslizante a la hora de analizar un audio. El tamaño de la ventana coincide con el número de muestras a examinar a la vez.
- Porcentaje de ventana: valor numérico de 0 a 100 que indica el porcentaje de ventana que tiene que superarse para que se realice un notificación por haber superado el umbral.

3.5. Nodo de procesamiento: análisis y procesamiento del audio

El desarrollo del nodo de procesamiento se encuentra dividido en dos implementaciones. La primera consiste en una implementación básica de recepción de paquetes de audio que también calcula los tiempos de envío de un paquete, mientras que la segunda contiene todos los procesamientos implementados. A su vez, los procesamientos tienen diferentes implementaciones, ya que se encuentran también paralelizados de diferentes maneras.

3.5.1. Implementación para benchmark

Como se ha comentado previamente, esta implementación es realizada con el objetivo de evaluar la velocidad del sistema y de las comunicaciones.

Consiste en la recepción y almacenamiento de los distintos paquetes de audio que se graban o se reproducen, anotando el tiempo de entrada del primer paquete y del último, para tener un tiempo de ejecución respecto al audio que se graba o reproduce. Esto permite medir la efectividad del sistema y detectar si hay cuellos de botella. El tiempo medido se envía mediante Nanomsg al nodo de comunicación, donde genera un fichero de texto con la información relevante de la prueba realizada.

3.5.2. Procesamientos y paralelizaciones

La segunda implementación del nodo de procesamiento incluye diferentes procesamientos. Dichos procesamientos tienen a su vez varias implementaciones paralelizadas con OpenMP.

Los procesamientos realizados son los siguientes:

- Aplicación de filtros de paso bajo y de paso alto: **estos filtros permiten pasar las frecuencias bajas y altas, respectivamente, atenuando las otras, lo que conlleva una modificación del sonido anterior.** En la Figura 3.3 se puede ver el código para aplicar ambos filtros a un mismo paquete de audio.

```
void filter(float* audio, unsigned long audio_size, float* out_original, float*
    low, float* high, double interval, double constant )
{
    int i = 0;
    double a_high = interval/(interval + constant);
    double a_low = interval/(interval + constant);

    high[i] = audio[i];
    low[i] = a_low * audio[i];
    out_original[i] = audio[i];

    for(i=1;i < audio_size/4;i += 1)
    {
        high[i] = a_high*high[i-1]+a_high*(audio[i]-high[i-1]);
        low[i] = low[i-1]+a_low*(audio[i]-low[i-1]);
        out_original[i] = audio[i];
    }
}
```

Figura 3.3: Ejemplo de código aplicando filtros low-pass y high-pass.

- Detección de picos de audio: **este procesamiento consiste en la búsqueda de muestras de audio que superen un determinado umbral. Dicho umbral, definido previamente, tiene que ser superado por un número mínimo de muestras dentro de una ventana.** El objetivo de este análisis es detectar picos de audio, como por ejemplo una palabra. Dicha palabra se tarda un tiempo en decir, por lo que grabando a 44.100 muestras por segundo, se necesita analizar más de una muestra para poder decir

que ese ruido pertenece a una palabra. De este modo, realizando muchas pruebas, se puede llegar a una relación entre umbral, tamaño de la ventana y porcentaje mínimo de la ventana a superar, en el que se puedan detectar todas las palabras que se dicen. En la Figura 3.4 se puede ver la estructura de datos utilizada para la implementación de detección de picos.

```
struct Window{
    int value_threshold = 0;
    int size = 0;
    int percentage=0;
    int margin_size = 0;
    int actual_size = 0;
    int counter = 0;
    int min_samples = 0;
    int thread = 0;
    int first_sample = 0;
    int first_chunk = 0;
    int last_sample = 0;
    int last_chunk = 0;
    int *buffer;
};
```

Figura 3.4: Estructura utilizada para la ventana deslizante.

3.6. Evaluación de las comunicaciones: Electron y C++

En esta sección se va a abordar el *benchmark* realizado sobre las diferentes configuraciones del sistema.

El sistema está dividido principalmente en dos modos: reproducción y grabación. Ambos funcionan de la misma manera: un *stream* de datos es generado de una fuente, un micrófono en el caso de la grabación y un fichero de audio en la reproducción. **Dicho *stream* de datos va a ser enviado desde el componente de la interfaz gráfica, pasando por el nodo de comunicaciones hasta llegar al nodo de procesamiento.**

Antes de diseñar un *benchmark* es necesario saber qué se quiere comprobar o qué preguntas se quieren responder. En este caso las preguntas tienen relación con los distintos aspectos de la implementación: ¿la representación gráfica del audio influye en el rendimiento del sistema?, ¿qué tamaño de paquete ofrece mejor rendimiento?, ¿qué protocolo de Nanomsg ofrece mejor rendimiento? y ¿existen penalizaciones al comienzo de una grabación o reproducción?

Con las preguntas ya planteadas, se tiene que pensar cómo se van a realizar las pruebas para conseguir resultados fiables y reales. Al trabajar con audio, se tienen que realizar pruebas largas, ya que no se puede reducir el tiempo de reproducción de un fichero de audio ni se puede grabar un audio de, por ejemplo 5 segundos, en menos tiempo. Por ello se han escogido tres tiempos distintos para las diferentes pruebas: 5, 10 y 20 segundos.

Una vez diseñado, las parámetros del sistema son los siguientes:

- Modo: elige entre el modo grabación y el modo reproducción.
- Dispositivo: determina el dispositivo de entrada para las pruebas cuando el modo es grabación.
- Duración del fichero: tiempo que durara la prueba, valores a elegir entre 5, 10 y 20 segundos. Para el modo reproducción se reproduce un fichero previamente grabado del tiempo elegido, mientras que en la grabación se realiza una grabación del tiempo seleccionado.
- Tamaño de los paquetes: este campo determina el número de muestras que se van a procesar a la vez. Los valores que comprende van desde 256 hasta 16384, siempre trabajando con potencias de 2.
- Protocolo de transporte: protocolo a utilizar durante la ejecución de la prueba. Como ya se vio en la Sección 2.2.2, en este sistema se pueden utilizar los protocolos de transporte IPC, TCP y WS.
- Representación gráfica: esta opción permite deshabilitar la representación gráfica del audio grabado o reproducido en la interfaz gráfica.
- Solo grabación/reproducción: con esta opción el audio no sale de la interfaz gráfica donde se graba o reproduce. Se utiliza para comprobar si la funcionalidad de la reproducción o la grabación se ven afectados por el procesamiento del audio.
- Deshabilitar procesamiento: esta opción permite deshabilitar el nodo de procesamiento para comprobar si afecta de alguna manera al resto del sistema.

Para asegurar que el *benchmark* genera resultados fiables, aparte de reproducir siempre los mismos ficheros, se realiza una comprobación más, la cual consiste en generar un fichero con los datos procesados y compararlos con el original. Si cuando se realiza la comparación, los resultados son distintos, entonces ha habido errores en el *benchmark* y los datos generados en esa prueba no son válidos.

Para conseguir buenos resultados, las pruebas se realizan diez veces, con una ejecución previa de calentamiento. De este modo, con la media de los tiempos de cada conjunto de pruebas con los mismos parámetros y con su desviación estándar, se consigue ver si los resultados obtenidos concuerdan los unos con los otros.

Durante la ejecución del *benchmark* se recogen distintos datos, como los tiempos del primer y último bloque procesados en cada uno de los tres componentes del sistema y las latencias del primer bloque desde que se genera en el frontal hasta que se procesa en los otros dos nodos. Con estos tiempos y los tamaños de los ficheros procesados, se miden los rendimientos de cada prueba para así comparar que combinación es mejor.

3.7. Aprovechamiento heterogéneo: OpenCL

En esta sección se va a abordar cómo se ha adaptado el código del procesamiento a ejecutar y cómo se va a estudiar este procesamiento; y como se ha adaptado una implementación de FFT para que funcione en el sistema.

Antes de empezar con las explicaciones sobre el código y su adaptación a OpenCL, se van a explicar unos conceptos básicos para entender mejor las explicaciones siguientes. Para empezar, en OpenCL, se ejecutan *kernels*, como en la Figura 3.5. **Los *kernels* son las funciones que se llaman desde el lado del *host*; le puedes pasar parámetros y recoger datos leyendo de la memoria del dispositivo.** El *kernel* se compila en el *host*, pero se ejecuta en el dispositivo. Otros dos terminos importantes son *work-item* y *work-group*. **Un *work-item* es una instancia del *kernel* que se ejecuta, es decir, que un *kernel* se ejecutará tantas veces como *work-items* se hayan definido.** Cada *work-item* tiene un ID global que se utiliza, en el caso de la Figura 3.5, para acceder a direcciones de memoria de varios punteros. Cada *work-item* que se define tiene que estar asignado a un *work-group*. Un *work-group* es un conjunto de *work-items*. El tamaño de los *work-groups* es el mismo para todos y no hay sincronización entre ellos, aunque comparten memoria local. Cada *work-group* tiene un identificador global, como los *work-items*, y cada *work-item* dentro de su *work-group* tiene un identificador local.

3.7.1. Adaptación de un código a OpenCL

En el ejemplo de la Figura 3.5 se ve como cada *work-item* va a utilizar su identificador global para comprobar si el valor absoluto multiplicado por cien del puntero `chunk` es mayor que el valor de la variable `value`. Este código es el utilizado para realizar la experimentación. Corresponde con el *kernel* prácticamente más simple a poder ser ejecutado.

```
__kernel void threshold(__global float *chunk, __global int *salida, const int
    value)
{
    int i = get_global_id(0);
    salida[i] = fabs(chunk[i])*100 > value;
}
```

Figura 3.5: Kernel OpenCL de detección de picos de audio.

En la Figura 4.1 se puede observar el código secuencial del procesamiento en cuestión. Para convertir el código de la Figura 4.1 en el código de la Figura 3.5 se ha eliminado el bucle, sustituyendo cada iteración por una instanciación del *kernel*. Por ejemplo, suponiendo que el tamaño de paquete es de 1024 muestras, antes se harían 1024 iteraciones en el bucle por cada paquete, pero en OpenCL se hacen 1024 instanciaciones del *kernel* por cada paquete para obtener el resultado deseado.

Para la experimentación sobre este procesamiento se hacen múltiples ejecuciones con diferentes tamaños de *work-group*.

3.7.2. Adaptación de FFT al sistema

La transformada rápida de Fourier, o FFT, es un algoritmo que sirve para calcular la transformada de Fourier discreta y su inversa. En cuanto al audio, este algoritmo permite transformar una señal de audio y mostrar el contenido frecuencial del mismo.

En el caso de la implementación encontrada, se trata del FFT de AMD APP SDK [28]. Esta implementación solo permite paquetes de 1024 muestras. Esta restricción está impuesta por

el propio *kernel*, ya que está diseñado para trabajar solo con 64 *work-items* por cada *work-group*.

Sobre esta implementación se realiza una experimentación para determinar que merece más la pena, si trabajar exclusivamente con paquetes de 1024 para ir calculando su FFT uno a uno, o, trabajar con paquetes más grandes para después dividirlos y calcular el FFT de sus diferentes divisiones de 1024 muestras. Es decir, se estudia la sobrecarga de las comunicaciones frente a la sobrecarga de las operaciones OpenCL.

Previo a la experimentación, se ha tenido que implementar el código del *host* en el sistema. Una vez hecho, se han realizado pruebas con diferentes tamaños de paquete, para comprobar que se calculaban correctamente las correspondientes transformadas. Cuando ya se ha asegurado el correcto funcionamiento, se ha probado con ficheros de audio de más de un paquete para ver como reacciona el sistema. En las primeras pruebas se observa que un fichero de 5 segundos tarda entre 5 y 40 segundos. Esto indica que el sistema, por la parte del *host*, esta teniendo problemas. **Con esto queda claro que, en la programación *host-device*, es tan importante el código del *host* como el código que se ejecuta en el dispositivo, así como la realización de optimizaciones en ambos.**

Para intentar optimizar el sistema se miden los tiempos de entrada y salida de los datos en la memoria del dispositivo. De este modo, se observa que el sistema en la entrada o salida tarda mucho más de lo esperado. Ahora se sabe que el cuello de botella se encuentra en las comunicaciones.

Después de varias búsquedas y pruebas, y alguna consulta con el director, se consigue optimizar el código *host* perteneciente al *kernel* de FFT. Esta optimización consiste en el uso de múltiples colas de comandos.

Capítulo 4

Evaluaciones y resultados

El objetivo de este capítulo es mostrar al lector tanto una validación funcional, como el rendimiento que ofrece la implementación realizada en las diferentes fases del desarrollo. Para ello se han llevado a cabo diversas evaluaciones del sistema en las que se han recogido datos relevantes para las preguntas que se han planteado previamente en este documento.

Se han realizado tres experimentos distintos, los cuales evalúan distintos aspectos del sistema:

- El primero pone a prueba todas las configuraciones posibles del sistema para así determinar cuáles son los mejores parámetros de configuración para el programa.
- El segundo compara las diferentes paralelizaciones de un caso de procesamiento determinado.
- El último evalúa diversas implementaciones sobre OpenCL.

4.1. Metodología

Las experimentaciones se han realizado sobre las siguientes configuraciones:

- Procesador AMD Ryzen 3 2200U con 2 núcleos y 4 subprocesos. Una frecuencia base de 2.5 GHz. Ejecutado sobre un sistema operativo Ubuntu 18.04 LTS.
- Para las experimentaciones con OpenCL se ha trabajado sobre un procesador AMD A10-7850K con 2 núcleos y 2 hilos por núcleo, permitiendo así 4 unidades de cómputo de OpenCL 1.2. Tiene una frecuencia base de 3.7 GHz. La gráfica integrada es una Kaveri R7 DDR3 con 512 núcleos y 8 unidades de cómputo. Ejecutado sobre un sistema operativo Linux Manjaro con una versión de kernel 4.19.122.

Debido a la complejidad de la instalación y posterior configuración de OpenCL, las experimentaciones que incluían OpenCL se han realizado en otra máquina distinta.

En total se han llevado a cabo tres evaluaciones: medición del rendimiento de las comunicaciones del sistema, evaluación del rendimiento de un procesamiento determinado y evaluación de diversas implementaciones sobre OpenCL. En el segundo se mide el rendimiento de las posibles optimizaciones respecto a ese mismo procesamiento en secuencial, destacando que para aplicar dichas optimizaciones se ha utilizado OpenMP.

En cuanto a la ejecución se preparan diferentes *scripts* de *shell*, los cuales se encargan del paso de parámetros al programa, de la comprobación de que las pruebas hayan salido bien y de la ejecución de los *scripts* de Python los cuales generan ficheros de salida con las métricas tomadas, en formato CSV, y gráficas con los resultados de las pruebas. Como requisito fundamental, es necesario que las pruebas sean realizadas en Linux.

En cuanto a la generación de ficheros, cada experimento trabaja de manera diferente:

- Para la evaluación de las distintas comunicaciones del sistema, el propio programa es el que genera esos ficheros para su recolección posterior por parte de los *scripts* creados en Python.
- La experimentación del procesamiento y sus diversas optimizaciones es distinto, ya que es el propio *script* de *shell* el que va a generar dichos ficheros para su posterior recolección.
- En las experimentaciones con OpenCL es el programa el que genera un fichero de texto con el tiempo.

En la evaluación de las distintas comunicaciones del sistema se realizan muchas mediciones de tiempos que son interesantes para el estudio de las comunicaciones. **Estas mediciones se realizan en regiones de interés**, como por ejemplo la latencia de enviar un paquete de datos desde la interfaz gráfica hasta que llega al nodo de comunicaciones.

Del mismo modo, para medir el rendimiento de las optimizaciones realizadas sobre el procesamiento del audio, interesa medir desde que se envía el primer primer paquete hasta que se termina de procesar el último. En el caso de OpenCL interesa lo mismo.

El número de veces que se ejecutan las pruebas difiere entre experimentaciones, incluso dentro de una propia experimentación:

- En el *benchmark* de evaluación de las comunicaciones, el número de ejecuciones depende de la longitud del fichero que se ejecute. Se tienen tres ficheros de 5, 10 y 20 segundos. A cada uno de ellos se asigna un número de iteraciones por prueba, debido a que la ejecución de numerosas pruebas con distintos parámetros con una duración mínima establecida por la longitud del fichero que se ejecuta, provoca una duración total de la prueba de muchas horas. De este modo, para las pruebas de 5 segundos se asignan diez iteraciones, para las de 10 segundos se asignan ocho iteraciones y para las pruebas de 20 segundos se asignan cinco iteraciones.
- Para la experimentación con el procesamiento de audio y sus optimizaciones, no se tiene en cuenta el número de iteraciones que se realizan. Esto se debe a que las posibles combinaciones son mucho más reducidas debido a que solo se prueban las mejores configuraciones del sistema, es decir, las que se han conseguido del *benchmark* de evaluación de las comunicaciones. De este modo solo se utilizan dos longitudes de audio, 5 segundos y 20 segundos, a las cuales se les asigna el mismo número de iteraciones, que son diez.
- En ambas experimentaciones con OpenCL se ejecutan el mismo número de veces cada combinación posible, ya que, como ocurre en la experimentación de las optimizaciones, el número de posibles combinaciones de parámetros no es muy grande. Se utiliza una longitud de audio de 5 segundos, asignando diez iteraciones a cada combinación posible de los parámetros de entrada.

En cuanto finaliza la ejecución de las pruebas, otros *scripts* programados en Python se encargan de recopilar y generar un fichero de extensión CSV. De este modo, otro *script* se encarga de la creación de gráficas y su almacenamiento en disco. Este *script* que genera las gráficas, se encarga del procesamiento de los datos para realizar la representación que se desea, por ejemplo, agrupa los datos en tablas según clasificaciones pertinentes para la representación gráfica. Esta clasificación de los datos consiste en agrupaciones de una o más columnas de una tabla para generar medias aritméticas y desviaciones estándar.

En las diferentes experimentaciones, este *script* es diferente, ya que no se busca calcular los mismos resultados ni utilizar las mismas métricas para comparar:

- Para la evaluación de las comunicaciones se ha utilizado el ancho de banda (kilobytes/s), es decir, la velocidad de transmisión de los datos en esas regiones de interés.
- En el caso de la comparación de optimizaciones se utilizan otras métricas, como el tiempo de respuesta, el tiempo de procesamiento y el *speed-up*, siendo este último una métrica utilizada para cuantificar como de bueno es un procesamiento respecto a otro. El tiempo de respuesta se utiliza para ver cuánto tardan las notificaciones en notificarse en el *frontend*. En el caso del *speed-up* y el tiempo de procesamiento, se usan para evaluar las mejores optimizaciones.
- En las experimentaciones sobre OpenCL se utiliza el *speed-up* y el tiempo de ejecución. En este caso, el *speed-up* se usa para comparar todos los tiempos con el peor caso.

Es importante destacar la optimización que se realiza sobre los procesamientos de OpenCL. **Esta optimización, como se comenta en la Sección 3.7.2, consiste en la creación de diferentes colas de comandos para las diferentes operaciones que se realizan en el código del *host*.** Una de las experimentaciones se ha realizado dos veces, antes de la optimización y después, de este modo se observa como influye dicha optimización y como el cuello de botella se localiza en otro punto del sistema. Para la otra experimentación con OpenCL se ejecuta sobre el código ya optimizado.

4.1.1. Evaluación de las comunicaciones

Partiendo del sistema ya finalizado pero sin ningún tipo de cómputo en el nodo de procesamiento, se procede a la evaluación de las comunicaciones y sus diferentes aspectos. Para ello se prepara un *script* que ejecutará el *benchmark* con todas las posibles combinaciones de sus parámetros de entrada.

Como se explica en la Sección 3.6, en el caso de la evaluación de la reproducción, el *script* se encarga de la comprobación de que los datos obtenidos sean fiables. **Esta comprobación consiste en generar un fichero idéntico al audio reproducido y compararlo con el fichero obtenido por la aplicación, de forma que si los dos son iguales, los datos son válidos.** De nuevo, en la Sección 3.6, se explican los diferentes parámetros de entrada que tiene el programa.

La ejecución de este *benchmark* consta de tres partes: la reproducción, la grabación con un micrófono integrado y la grabación con un micrófono externo.

Para empezar, existen un total de cincuenta combinaciones de los parámetros. A estas cincuenta combinaciones se le va a denominar prueba. Se tiene en cuenta que se está trabajando con grabación y reproducción de audio, con lo que el tiempo de cada prueba va a durar como mínimo cincuenta veces el tiempo que dure la grabación o la reproducción. De este modo se establece un número de ejecuciones a cada prueba en función de los tres tiempos definidos. Teóricamente, teniendo en cuenta todas las ejecuciones para cada parte del *benchmark*, tardaría el sistema alrededor de 10 horas en ejecutar todas las pruebas.

Para la ejecución con grabación de los distintos micrófonos se necesita un sonido constante. Para ello se ha utilizado un generador de ondas sinusoidales. De este modo se garantizan unos resultados fiables para poder compararlos entre ellos, intentando replicar lo que se hace con la reproducción.

4.1.2. Optimización de un procesamiento

En esta experimentación se ha partido de un código secuencial que, por cada paquete de audio procesado, envía una notificación a la interfaz gráfica para avisar de que se ha encontrado un pico de sonido. En la Figura 4.1 se puede ver el código secuencial de dicho procesamiento, el cual consiste en recorrer el paquete que se quiere analizar comprobando el valor de las muestras y comparando con el umbral que se tiene que superar. De cara a la evaluación de las diferentes optimizaciones se fuerza a todas las muestras a superar dicho umbral y enviar una notificación al nodo de comunicaciones por paquete, siempre al final del procesamiento del mismo.

```
void* analyse_threshold(float* buffer, int chunk_size, int socket,  int
    index_chunk){
    for(int i = 0; i<chunk_size;i++){
        if(fabsf (buffer[i]*100) > -1){
        }
    }
    // Notificacion al nodo de comunicaciones
}
```

Figura 4.1: Código secuencial del procesamiento de detección de picos.

Partiendo del código secuencial de la Figura 4.1 se han realizado las siguientes paralelizaciones y optimizaciones con OpenMP:

- Paralelización de tareas: esta paralelización permite el procesamiento de múltiples paquetes a la vez. Para ello, se ha modificado el sistema de recepción de paquetes secuencial para que la recepción sea rotativa, es decir, los paquetes se almacenan en posiciones distintas de memoria. De este modo, se reciben paquetes de audio independientemente de que el anterior haya sido procesado o no. Las pruebas se realizan con tres modificaciones en el número de paquetes que se pueden almacenar simultáneamente. Dichos valores son 5, 10 y 20.
- Vectorización con paralelización de tareas: igual que la paralelización de tareas, pero con el añadido de haber aislado el contenido del bucle `for` de la Figura 4.1 en una función, la cual se vectoriza con el pragma de OpenMP `declare simd`. A esta vectorización se añaden parámetros para mejorarla, como por ejemplo, `notinbranch`, lo que quiere decir que esa función nunca va a ser llamada de manera condicional.

- Paralelización de datos: a diferencia de la optimización con tareas, la paralelización de datos no permite ejecutar varios procesos de manera simultánea, sino que permite la ejecución de bucles de una manera más rápida. Las pruebas con paralelización de datos se realizan añadiendo el parámetro `schedule` y con los tres valores vistos en la Sección 2.3.2: `static`, `dynamic` y `guided`.
- Vectorización con paralelización de datos: esta optimización consiste en la paralelización de datos previamente explicada, pero, al igual que la vectorización con paralelización de tareas, aislando el contenido del bucle en una función, para así aplicar la vectorización.

Esta batería de pruebas busca la mejor optimización en OpenMP del procesamiento que se observa de la Figura 4.1. **De este modo, lo que se mide en estas pruebas son los tiempos de lectura del primer y el último paquete, las notificaciones en la interfaz de los mismos y el *speed-up* del programa respecto a la ejecución secuencial.**

Una vez están las optimizaciones, se piensa cómo ejecutar las diferentes pruebas. Para empezar, se decide trabajar con solo dos de los tres ficheros usados previamente, con el de 5 y con el de 20 segundos, por la simplificación de los casos y porque no se da tanta variabilidad entre unos casos y otros.

Dado que se quieren los mejores resultados posibles, se tienen que utilizar los mejores resultados del *benchmark* de las comunicaciones. De este modo, para el tamaño de los paquetes, se utilizan los valores 8192 y 16384. Se decide hacer 10 pruebas por cada combinación de optimización, fichero y tamaño de paquete.

4.1.3. Evaluación de un procesamiento OpenCL

Para esta experimentación se ha partido del procesamiento de la Figura 4.1 y se ha adaptado su código a OpenCL, como se ve en la Figura 3.5.

Partiendo de ese *kernel*, se diseñan unas pruebas para comprobar cómo influye el número de *work-groups* en el rendimiento y comprobar qué procesador obtiene un mejor rendimiento. **Dado que en el sistema donde se ejecutan estas prueba no hay interfaz gráfica, se desarrolla un lector de ficheros de audio para simular el comportamiento del sistema de reproducción desarrollado.** Dicho lector simula el comportamiento del reproductor implementado con Electron: envía los paquetes por Nanomsg respetando los tiempos de lectura teniendo en cuenta el tamaño del paquete y la frecuencia de muestreo del fichero de audio.

La experimentación se realiza con un fichero de audio; sobre la CPU y GPU del sistema; con dos tamaños de paquete distintos, 8192 y 16384; y con nueve tamaños distintos de *work-group*. El número de *work-items* coincide con el tamaño de paquete de cada prueba, ya que, como se ve en el código, un *work-item* se encarga de comprobar una muestra de audio.

4.1.4. Procesamiento de FFT en OpenCL

A la hora de diseñar las pruebas para esta experimentación se ha encontrado, en la implementación del FFT, la limitación de que solo se puede ejecutar para un tamaño determinado de paquete, 1024. Teniéndolo en cuenta, se piensa en cómo abordar la experimentación desde un punto que pueda ser interesante. Entonces se piensa, como se ha comentado en la Sección 3.7.2,

que se puede estudiar en qué afectaría trabajar con tamaños de paquete más grandes de 1024 para después dividirlos en paquetes de 1024 y poder ejecutar el *kernel*.

Con estas pautas ya definidas se diseñan las pruebas, en las que también se incluye el estudio sobre qué procesador obtendrá un mayor rendimiento. La experimentación se realizar con dos ficheros de audio; sobre GPU y CPU; y con dos tamaños de paquete distintos, 1024 y 8192.

Esta experimentación se ejecuta dos veces, una con el lector de ficheros de audio mencionado previamente y otra con el sistema completo, es decir, con la aplicación y todos sus componentes funcionando (interfaz gráfica, representación del audio,...).

4.2. Evaluación de las comunicaciones

Durante el diseño de este experimento se plantearon una serie de preguntas, como se indica en la Sección 3.6, las cuales se espera que ayuden a obtener la mejor configuración del sistema. Se recuerda que cada prueba realizada de este experimento está dividido en tres secciones, una correspondiente a la reproducción y dos correspondientes a la grabación.

Una vez obtenidos los resultados, se procede al análisis de los datos y a la contestación de las preguntas.

4.2.1. ¿La representación gráfica del audio influye en el rendimiento del sistema?

Se han realizado ejecuciones con y sin representación gráfica del audio y se han medido tiempos en los tres componentes del sistema. Para analizar si la representación gráfica impacta en el rendimiento de la reproducción de ficheros se tiene que analizar cada componente por separado.

- En la Figura 4.2 se puede ver una gráfica representando los resultados para esta pregunta en el caso de la reproducción en la interfaz gráfica. También se puede observar que en el eje de abscisas se representa el tamaño de los paquetes y en el eje de ordenadas la velocidad de transmisión medida en kilobytes/segundo. Otro dato importante es la desviación estándar, que se observa en la parte superior de cada columna con una barra negra que representa cómo de grande es la desviación de la media aritmética que se presenta. Pasando al análisis de los resultados, **se obtienen unos rendimientos muy parejos, representando gráficamente el audio y sin representarlo.** Esto se reproduce independientemente del tamaño de paquete y duración del audio, por lo que en un principio no parece estar afectando al rendimiento del primer componente. Se puede observar como, para algunos casos, el rendimiento mejora con la representación gráfica del audio y como en otros ocurre todo lo contrario. Cabe destacar que en la reproducción, debido a los problemas mencionados con la Web Audio API en la Sección 3.3.1, se observan unas desviaciones estándar muy grandes de los ficheros de menor duración. Por ello es recomendable fijarse en los ficheros de audio de una duración mayor, ya que la diferencia de un paquete en el procesamiento puede significar un aumento o perdida de rendimiento notable.

- En el nodo de comunicación, como se puede ver en la Figura 4.3, ocurre lo mismo que en la interfaz gráfica, se muestran unos resultados muy parejos, por lo que de nuevo **parece que**

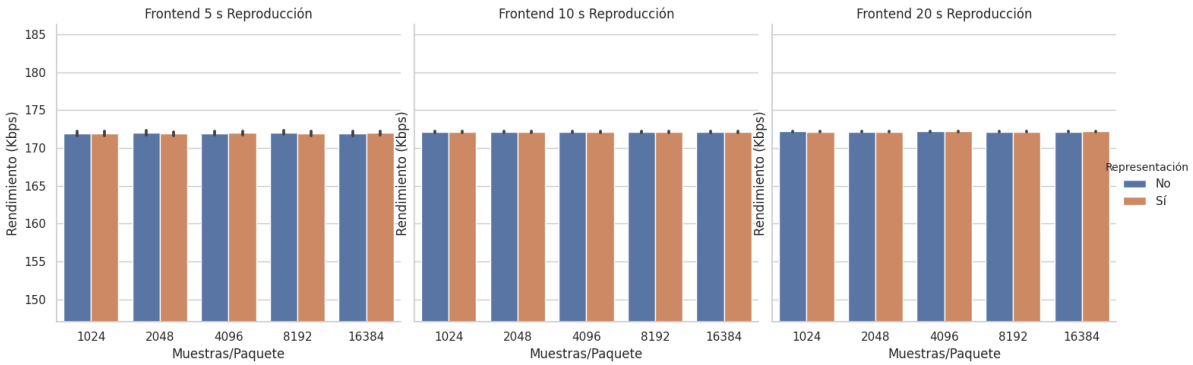

Figura 4.2: Rendimiento de la reproducción en el frontend con y sin representación del audio.

la representación gráfica no afecta al rendimiento. También se detecta algún caso en que una de las ejecuciones obtiene mayor rendimiento, como por ejemplo, la reproducción de 5 segundos y el tamaño de paquete 16384, que se obtiene un rendimiento mayor representando el audio. Como se ha comentado previamente, se aprecia una desviación estándar bastante grande respecto a los otros casos, por lo que la media aritmética no es del todo representativa en esos casos.

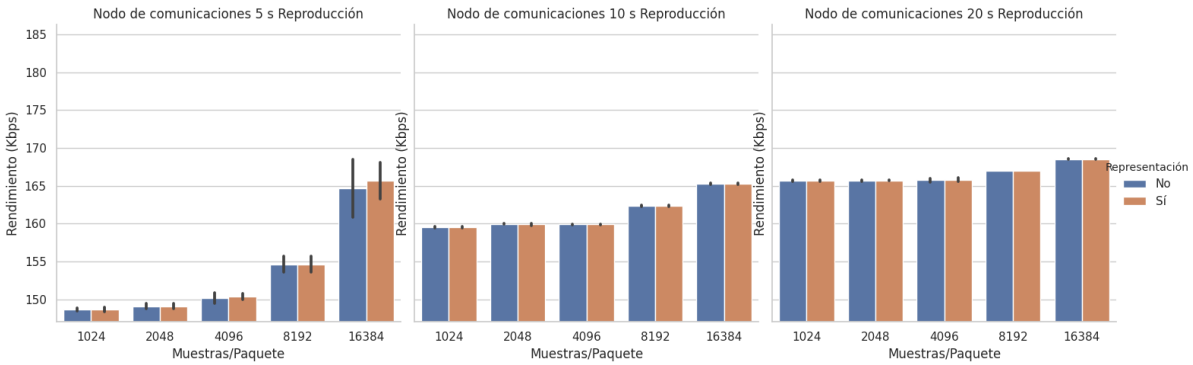

Figura 4.3: Rendimientos de la reproducción en el nodo de comunicaciones con y sin representación del audio.

- Por último, en el nodo de procesamiento, como se observa en la Figura 4.4, tampoco se ve diferencia alguna entre los resultados, como ocurre previamente en el *frontend* y en el nodo de comunicaciones.

Los resultados obtenidos en la prueba de grabación son similares. Debido a estos resultados tan parejos, se concluye que representar gráficamente el audio no afecta a ninguno de los componentes del sistema.

Respondiendo a la pregunta de si la representación gráfica influye o no en el rendimiento del resto del sistema, la respuesta es no. **Las diferencias que se encuentran entre las pruebas**

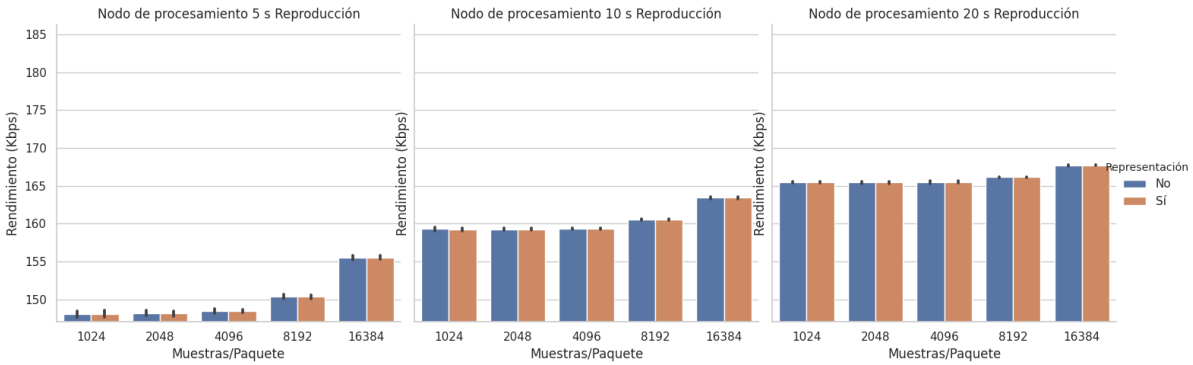

Figura 4.4: Rendimientos de la reproducción en el nodo de procesamiento con y sin representación del audio.

son mínimas y no demuestran que afecte de ninguna manera al rendimiento del resto de los componentes. Entonces se plantea la hipótesis de que la Web Audio API tiene un gran nivel de optimización y evita el consumo de recursos innecesarios, dejando así dichos recursos al resto del sistema.

4.2.2. ¿Qué tamaño de paquete ofrece mejor rendimiento?

En la configuración del sistema es posible modificar el tamaño de los paquetes a procesar. Esto permite experimentar con distintos procesamientos y ver cómo varía el rendimiento.

En estas pruebas se quiere saber cual es el tamaño que proporciona un mayor rendimiento al sistema, sin el objetivo de mejorar un procesamiento o una función. De este modo, para medir cual ofrece un mejor rendimiento, se miden los tiempos de los diferentes componentes y también se mide cuanta información ha sido transmitida en ese tiempo.

En el caso de la reproducción se obtienen los siguientes resultados:

- En el *frontend*, como se ve en la Figura 4.5, no se aprecia ninguna predominancia clara de ningún tamaño, ya que los rendimientos son muy parecidos. Lo que sí se puede apreciar es el incremento del rendimiento según aumenta la duración del fichero, la diferencia no es muy grande pero sí que se produce una subida leve.

- En la Figura 4.6, correspondiente al nodo de comunicación, **se ve un claro predominio del tamaño 16384 en todos los tiempos de fichero, seguido por el tamaño 8192, lo cual hace que se plantee la hipótesis de que a mayor tamaño de paquete, mejor es el rendimiento.** La relación comunicación/cómputo indica que al enviar muchos paquetes muy pequeños la comunicación va a generar una latencia considerable respecto a enviar pocos paquetes grandes. Hay una sobrecarga en las comunicaciones, por lo que es mejor enviar paquetes más grandes a un ritmo más lento que enviar paquetes pequeños de manera muy rápida.

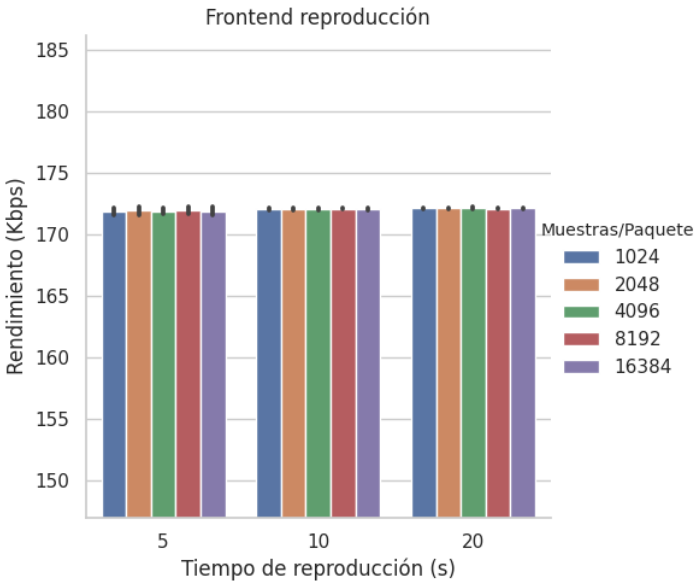

Figura 4.5: Rendimientos de la reproducción en el frontend para los diferentes tamaños de paquete.

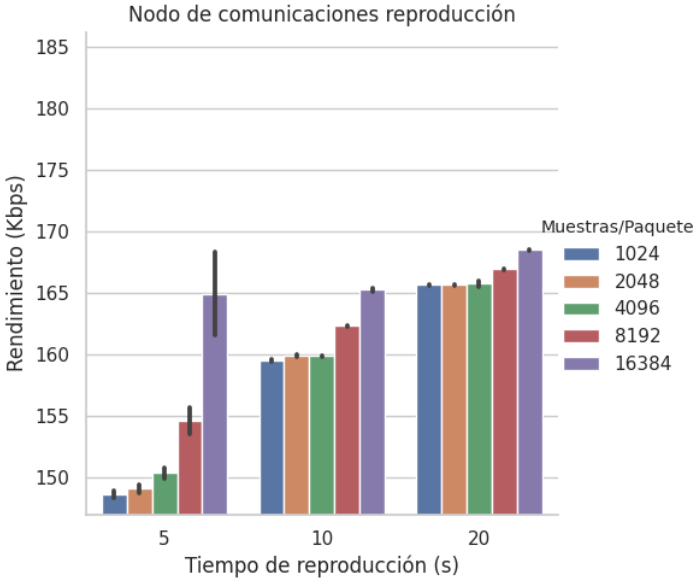

Figura 4.6: Rendimientos de la reproducción en el nodo de comunicaciones para los diferentes tamaños de paquete.

- Finalmente, en el nodo de procesamiento, reflejado en la Figura 4.7, se observa también una clara predominancia de los tamaños de paquete más grandes, manteniendo así la posibilidad de confirmar que según crecen los tamaños de paquete, el rendimiento es mayor.

Para la grabación se obtienen los siguientes resultados, mostrando ambos micrófonos en las

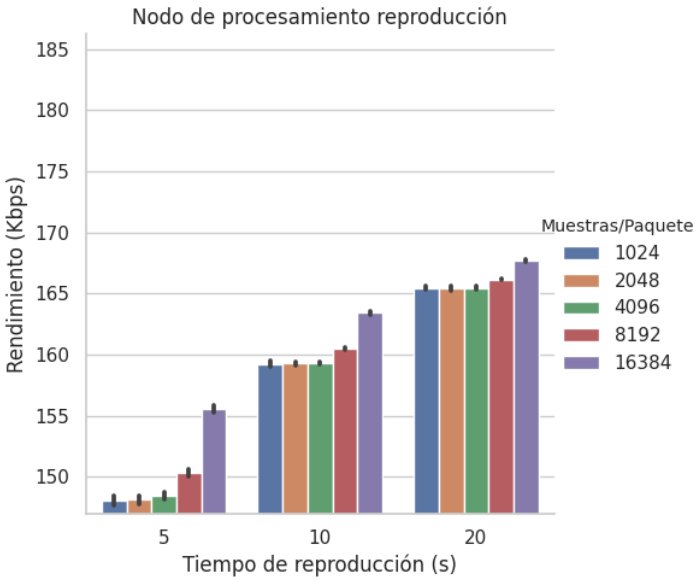

Figura 4.7: Rendimientos de la reproducción en el nodo de procesamiento para los diferentes tamaños de paquete.

mismas figuras:

- Para el componente de la interfaz gráfica, como se puede ver en la Figura 4.8, la grabación con micrófono integrado mantiene una similitud entre los diferentes tamaños de paquete para las grabaciones más largas. Sin embargo, para la grabación de 5 segundos el tamaño de paquete de 16384 tiene una gran diferencia en rendimiento con los otros tamaños de paquete. En el caso del micrófono externo se puede ver una pérdida general de rendimiento para los tamaños de paquete más grandes, destacando en el caso de la grabación de 5 segundos que el paquete de 8192 ofrece un rendimiento parejo al de los paquetes más pequeños.

- En el nodo de comunicaciones, mostrado en la Figura 4.9, se observa cómo para ambos micrófonos, **los tamaños de paquete más grandes vuelven a tomar la delantera frente a los pequeños, haciendo así una forma de escalera ascendente recordando al rendimiento de la reproducción en este mismo componente.** Cabe destacar, que a diferencia de la reproducción, donde según crecía el tiempo de reproducción aumentaba el rendimiento, aquí sería todo lo contrario, según crece el tiempo de grabación se reduce el rendimiento. Respecto a los micrófonos se ve como el micrófono externo muestra un mejor rendimiento en todos los tamaños de paquete y en todos los tiempos de grabación, respecto al micrófono integrado.

- Por último, el nodo de procesamiento, como se ve en la Figura 4.10, muestra como, **en el caso del micrófono integrado, existe cierta predominancia por los paquetes de tamaño más grande, aunque no de manera generalizada.** Destacan los paquetes de 16384 y 8192 por su mejor rendimiento respecto a los demás, aunque ya no hace una

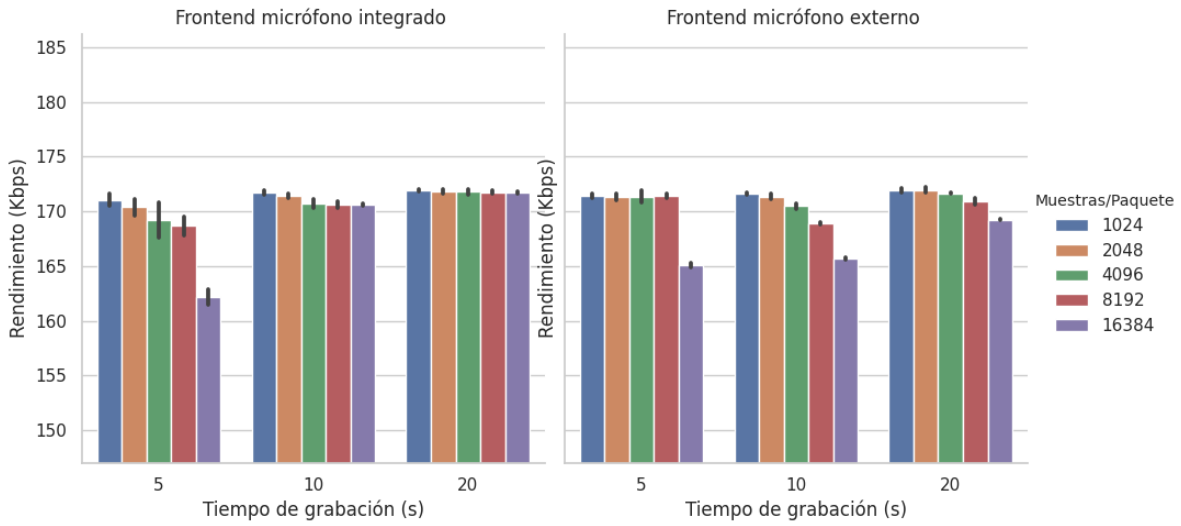

Figura 4.8: Rendimientos de la grabación en el frontend para los diferentes tamaños de paquete.

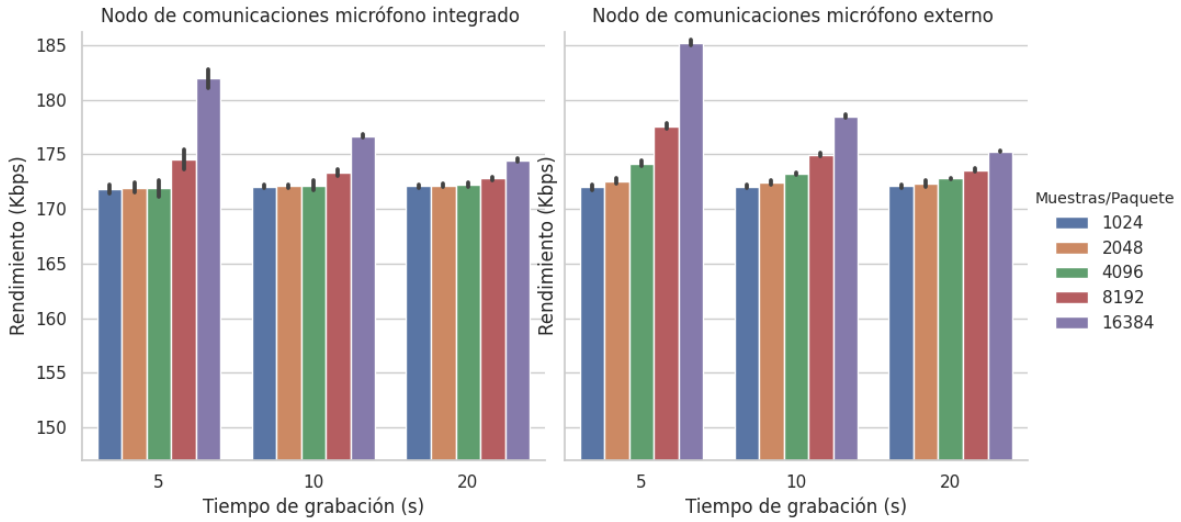

Figura 4.9: Rendimientos de la grabación en el nodo de comunicaciones para los diferentes tamaños de paquete.

forma de escalera ascendente como previamente. Se observa que en el caso de 5 segundos, el rendimiento es igual o parecido para todos los tamaños de paquete. Por otro lado, en el micrófono externo, se detecta mejor rendimiento para los tamaños de paquete grandes en el tiempo de 5 segundos, mientras que para los tiempos de 10 y 20 segundos da igual el tamaño de paquete utilizado, que el rendimiento es parecido o igual.

Después del análisis de los resultados y de la realización de comparaciones con las gráficas en cada componente del sistema, de forma global, **el mejor tamaño de paquete para el sistema es de 16384 muestras.** A esta conclusión se ha llegado por el siguiente razonamiento:

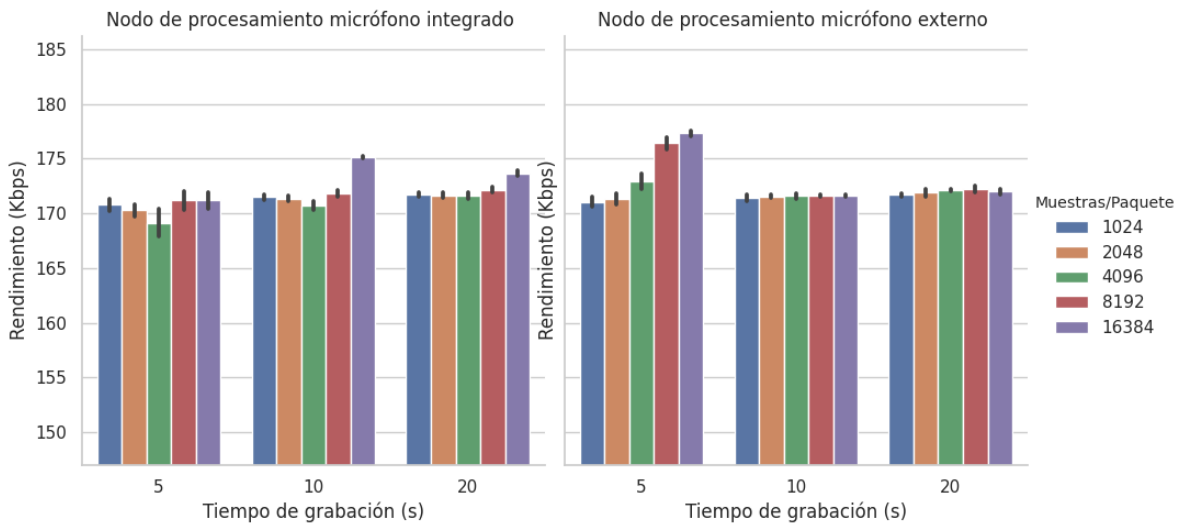

Figura 4.10: Rendimientos de la grabación en el nodo de procesamiento para los diferentes tamaños de paquete.

para la reproducción, en dos de tres componentes, los paquetes de mayor tamaño tiene un rendimiento mayor y, en el caso de ambas pruebas de grabación, haciendo balance entre los diferentes rendimientos de los componentes, compensa que la grabación en el *frontend* sea más lenta si después en los nodos de comunicaciones y de procesamiento el rendimiento de ese tamaño es mayor que el resto.

4.2.3. ¿Qué protocolo de Nanomsg ofrece mejor rendimiento?

Como ya se ha visto en la Sección 2.2.2, Nanomsg permite el uso de diferentes protocolos de transporte, pero desconocemos cual es el que da un mejor rendimiento a nuestro sistema. Para ello se van a analizar los datos obtenidos en la ejecución de las pruebas y se intentará llegar a la conclusión de esta cuestión. Los resultados se van a dividir en reproducción, grabación con micrófono integrado y en grabación con micrófono externo.

- Como se puede ver en la Figura A.1, en el caso de la reproducción se encuentran unos resultados muy parecidos para todos los protocolos. No hay ninguno que destaque por encima de los demás, por lo que para la reproducción es indiferente el protocolo que se utilice.

- En el caso de la grabación de micrófono integrado, como se puede ver en la Figura A.2 se nota una disparidad entre los resultados, también al crecimiento de las desviaciones estándar. En este caso se ven rendimientos distintos, donde a veces es mejor IPC u otras veces TCP o incluso WS. De este modo sigue sin haber un protocolo que destaque sobre los demás.

- Dados los resultados de la grabación con micrófono externo, como se pueden observar en la Figura A.3, se nota un dominio por parte de IPC en las grabaciones de 5 segundos, mientras que en el resto de tiempos hay algún pico en el que también IPC se encuentra por encima, pero no es tan dominante como en los casos de 5 segundos.

Una vez finalizado el análisis de los resultados se llega a la conclusión de que no se puede considerar un protocolo mejor que los demás en cuanto a rendimiento, ya que las pruebas no han sido concluyentes. Basándose en los resultados de las pruebas de grabación con micrófono externo, se puede llegar a la conclusión de que para grabaciones de un tiempo reducido con dicho micrófono, el mejor protocolo de transporte es IPC, siempre que se utilice un tamaño de paquete inferior a 16384. Haciendo un análisis global de los resultados, se llega a la conclusión de que ningún protocolo obtiene mejores resultados, solo en grabaciones pequeñas y con tamaños no óptimos, que no se van a utilizar para experimentaciones futuras, ya que siempre se coger el mayor tamaño posible.

Para la ejecución de futuras pruebas se usará IPC debido a los picos que ha mostrado en ciertas pruebas, como la de grabación con micrófono externo.

4.2.4. ¿Existen penalizaciones al comienzo de una grabación o reproducción?

Cuando se inicia la grabación o reproducción de un fichero de audio existe un periodo de tiempo entre pulsar el botón de grabación o reproducción y el momento en el que los datos empiezan a almacenarse. Con esta pregunta se quiere comprobar la existencia de esta penalización y de ser así, determinar cómo de grande es en función del tiempo del audio.

- En el caso de la reproducción de fichero, como se ve puede ver en la Figura B.1, se observa como se repite un patrón entre los diferentes tamaños de paquete. **Se ve como el rendimiento de la reproducción de los ficheros más pequeños se ve afectado, aparte de que en todos los casos se observa una desviación estándar bastante grande.** El rendimiento va creciendo según más tiempo dure la reproducción.

- Para la grabación, como se ve en la Figura B.2, ocurre lo mismo que con la reproducción. **Los tiempos de grabación más cortos ven su rendimiento más afectado**. Cabe destacar que en un caso concreto con el micrófono externo, el rendimiento medio del fichero más corto superó al del más largo, aunque se ve que su desviación estándar es muy grande, por lo que probablemente haya ocurrido algo inesperado durante la ejecución de las pruebas.

Una vez analizados todos los casos, se plantea la siguiente hipótesis: **el rendimiento de las pruebas más cortas se ve afectado en mayor medida por la penalización que afecta a todas las grabaciones y reproducciones, y que según aumenta el tiempo de ejecución de las pruebas, esta penalización pasa más inadvertida.**

4.3. Optimización de un procesamiento

Esta experimentación, como se comenta en la Sección 4.1.2, consiste en la optimización del procesamiento simple de la Figura 4.1 con los diferentes métodos de paralelización y optimización. Durante el análisis de los resultados se va a hablar de tres medidas que se tienen en cuenta en este experimento: el *speed-up* de las optimizaciones respecto a la ejecución secuencial, el tiempo medio de las ejecuciones y la latencia del primer y último paquete de cada prueba. Se recuerda que estás pruebas se ejecutan sobre dos ficheros de audio grabados, siendo estos los utilizados para las pruebas de 5 y 20 segundos en el estudio de las comunicaciones del sistema.

4.3.1. ¿Varían las latencias de las últimas notificaciones respecto a las primeras?

El objetivo de esta experimentación es la comprobación de que no exista ningún tipo de retraso entre las primeras notificaciones y las últimas. Esto es importante para asegurar que el sistema no sufre ningún tipo de retraso en el procesamiento, es decir, que no está tardando demasiado en procesar cada paquete.

Para empezar se van a analizar las diferentes latencias conseguidas para el primer y último paquete. **Los resultados obtenidos son muy parecidos, por lo que se van a comentar juntos los resultados del primer y último paquete.** Como se explica en la Sección 4.1.2, el sistema realiza medidas de cuándo se genera un paquete y de cuándo se notifica en la interfaz que dicho paquete ha superado el umbral. De este modo se mide el tiempo que tarda el paquete en enviarse desde que se se lee, hasta que termina el procesamiento y se notifica en la interfaz.

Con el objetivo de conseguir una comprensión correcta, se han dividido los datos de las latencias en dos grupos: uno correspondiente a las optimizaciones realizadas con paralelización de datos y otro correspondiente a las optimizaciones con paralelización de tareas.

Paralelización de datos y vectorización

- Como se puede ver en la Figura 4.11, **los resultados obtenidos para cualquier combinación de paralelismo de datos con vectorización superan con creces los obtenidos de manera secuencial**, alcanzando latencias inferiores a 10 ms. Cabe destacar que las optimizaciones con paralelización de datos, sin vectorización, también ofrecen unos buenos resultados respecto a la ejecución secuencial, pero los procesamientos vectorizados son los que mejor rinden. Respecto a los diferentes parámetros utilizados en la paralelización de datos no se obtiene ningún resultado mejor que otro, todos operan igual de bien.
- En la Figura C.1 se observa como los resultados son bastante parecidos a los obtenidos para el primer paquete con optimizaciones relacionadas con paralelismo de datos. De la misma manera que en el primer paquete, aquí también predominan las pruebas realizadas con vectorización y paralelismo de datos y después las que solo incluyen paralelismo de datos.

Paralelización de tareas y vectorización

- En la Figura 4.12 se puede ver como, independientemente del tamaño del buffer de entrada, el paralelismo de tareas consigue unos resultados parejos al secuencial. Sin embargo, añadiendo vectorización a las pruebas, la mejora es notable, alcanzando valores inferiores a los 20 ms.
- Para el caso de la paralelización de tareas, como se puede observar en la Figura C.2, los resultados también se asemejan a los obtenidos en el primer paquete con las mismas optimizaciones. **Las optimizaciones con vectorizaciones obtienen las latencias más bajas**, mientras que las que solo se limitan a la paralelización de tareas no ven afectado su rendimiento respecto al secuencial.

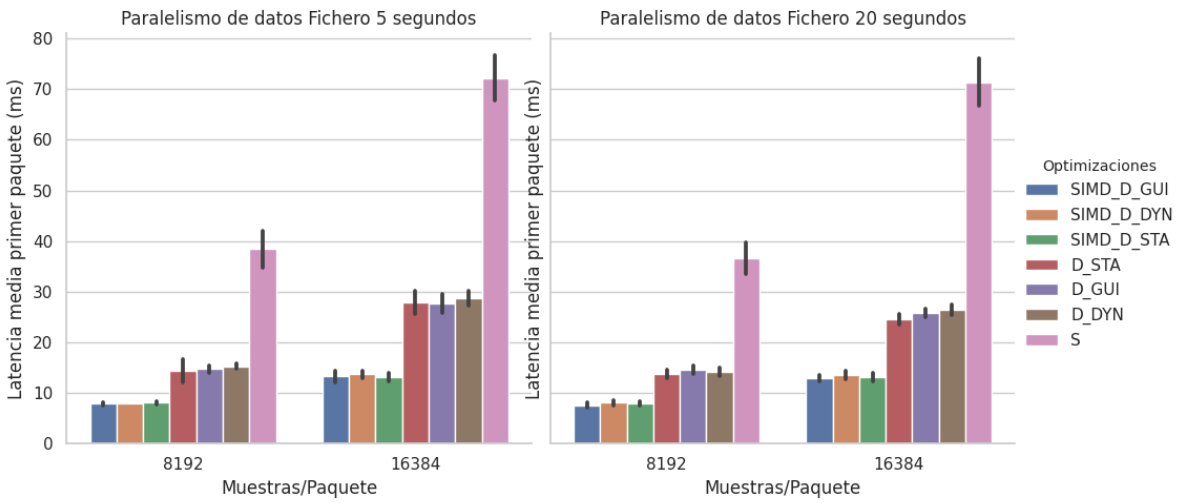

Figura 4.11: Latencias del primer paquete en las pruebas con datos.

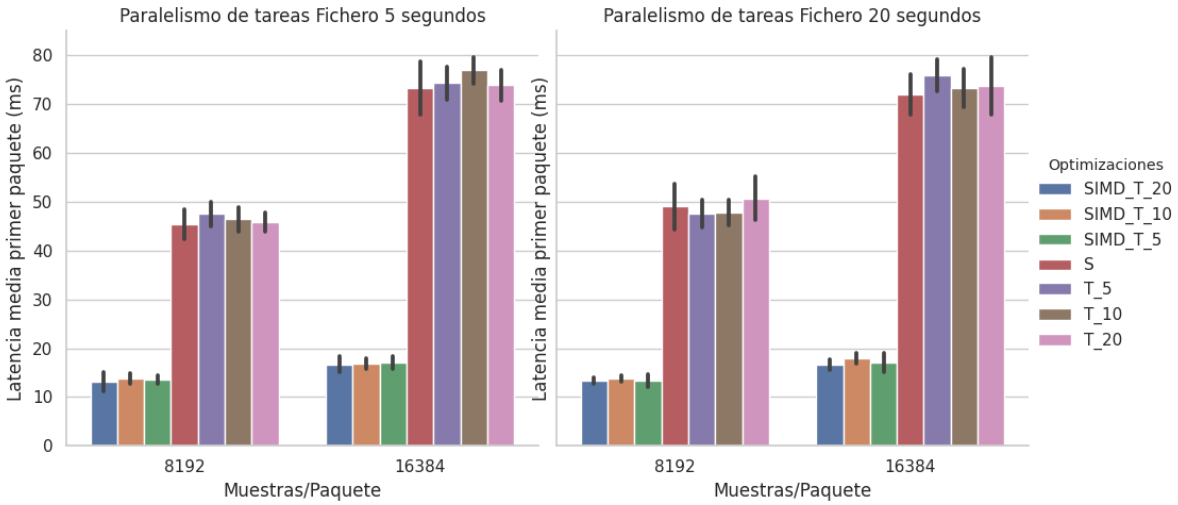

Figura 4.12: Latencias del primer paquete en las pruebas con tareas.

Visión general de los resultados

- Ahora, para ver los resultados de manera global, pudiendo comparar los distintos paralelismos empleados, en la Figura 4.13 se muestran todos los datos del primer paquete. De este modo se puede ver como las paralelizaciones con vectorización ofrecen un rendimiento muy bueno respecto a la ejecución secuencial y que son la mejor solución, mientras que la paralelización de datos ofrece también unos resultados aceptables y luego la paralelización de tareas, la cual ofrece un rendimiento como la ejecución secuencial.

- Viendo de nuevo resultados de manera global, en el caso del último paquete procesado, se puede ver en la Figura C.3 como las optimizaciones que contienen vectorizaciones mejoran el rendimiento mejor que cualquier otra combinación, al igual que ocurre en el primer

paquete.

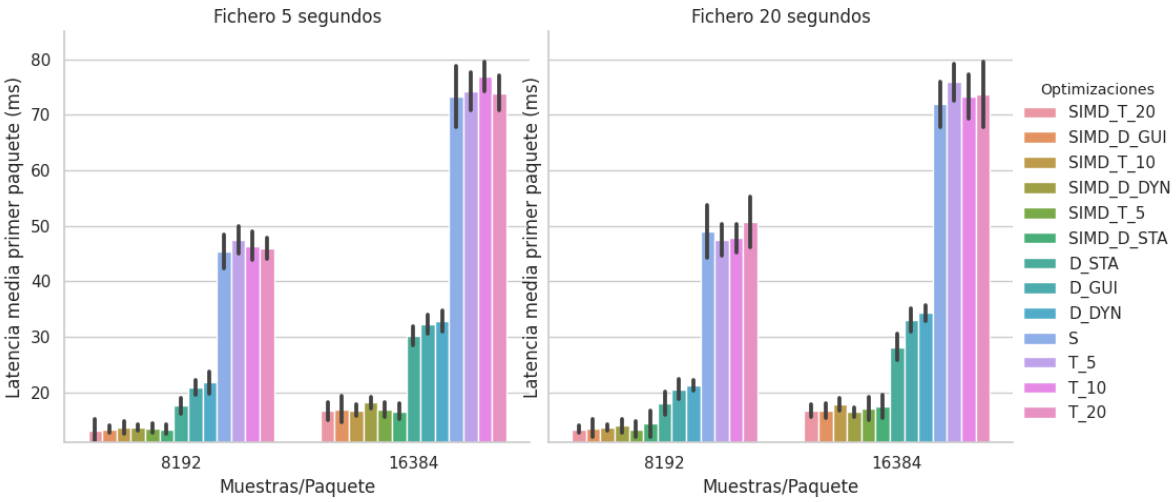

Figura 4.13: Latencias del primer paquete en las pruebas.

Como conclusión de esta experimentación, ha quedado demostrado que **las latencias de los diferentes paquetes son constantes**, teniendo en cuenta que la experimentación envía una notificación al final de cada paquete. Esto no demuestra que si se envían más notificaciones por paquete va a salir un resultado igual o parecido.

4.3.2. Tiempo medio y speed-ups

Otro aspecto importante que se quiere comprobar con la realización de estos experimentos es en que medida afectan estás optimizaciones al sistema. Para ello, se calcula el tiempo medio y el *speed-up* de las ejecuciones. Todo será contrastado con los valores obtenidos para el tiempo medio de las ejecuciones secuenciales. De este modo, se obtienen las siguientes conclusiones:

- Partiendo de la información que se proporciona en la Figura 4.14, se observa como las optimizaciones realizadas con paquetes de 8192 superan el tiempo medio esperado, que sería de 5 y 20 segundos. En el caso de ambos tamaños de paquete en la reproducción de 5 segundos, las optimizaciones realizadas solo con paralelización de tareas obtiene un resultado peor que el del secuencial, lo cual se ve reflejado en la Figura D.1 mirando su correspondiente *speed-up*, el cual es inferior a 1.
- Por otro lado, en los paquetes de tamaño 16384 se consigue un tiempo de ejecución más próximo al ideal, como se puede ver en la Figura 4.14. A pesar de que las optimizaciones realizadas solo con paralelización de tareas en el fichero de 5 segundos sigue obteniendo unos resultados nada deseables. De este modo, se consigue un *speed-up* (ver Figura D.1) parecido al obtenido en los paquetes de 8192, pero con la certeza de que el tiempo de ejecución está lo más cerca posible al límite impuesto por el tamaño del fichero.

Con esta experimentación queda demostrado que el uso de **vectorizaciones SIMD** en un sistema básico de detección de picos con tamaños de paquete de 16384 ofrece los mejores resultados, ya que **se obtienen las latencias más bajas** para la detección de picos por paquete y además completa el procesamiento del audio prácticamente en el tiempo de duración del fichero.

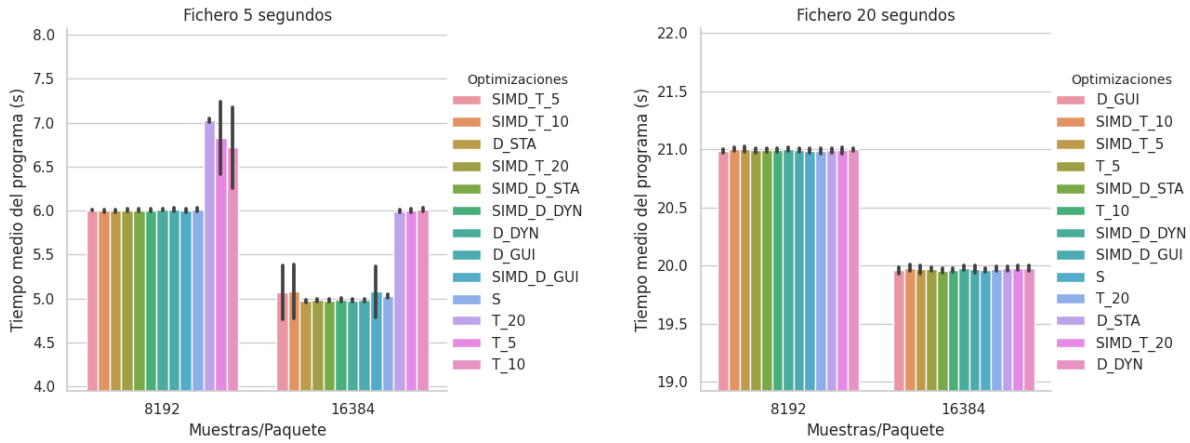

Figura 4.14: Tiempos medios de las ejecuciones para las optimizaciones.

4.4. Evaluación de procesamiento con OpenCL

Este experimento consiste en la optimización del procesamiento de la Figura 4.1, pero, a diferencia del experimento de la Sección 4.3, mediante el uso de OpenCL. El código consiste en un umbral de audio, es decir, si los valores del audio superan un valor determinado. Con este experimento también se quiere responder a las siguientes preguntas: ¿qué tamaño de *work-group* ofrece un mejor rendimiento? y ¿qué dispositivo obtiene un mejor rendimiento?.

Se recuerda que para realizar esta experimentación, se ha ejecutado dos veces: una primera vez tal como se desarrolló y una segunda vez con la optimización mencionada en la Sección 4.1.

Una vez obtenidos los resultados, se procede al análisis de los datos y a la contestación de las preguntas.

4.4.1. ¿Qué tamaño de work-group ofrece un mejor rendimiento?

Las ejecuciones se han realizado con los siguientes tamaños de *work-group*: 1, 2, 4, 8, 16, 32, 64, 128 y 256. Es un requisito de OpenCL que el tamaño del *work-group* sea divisor del número global de *work-items*.

Se introducen dos nuevos conceptos de OpenCL. LWS (local-work-size), que representa el número de *work-items* por *work-group* y el GWS (global-work-size), que corresponde con el número total de *work-items* que ejecutan el *kernel*.

En la Figura 4.15 se pueden ver los resultados obtenidos en la primera ejecución. Se puede observar como el LWS influye en el rendimiento obtenido, tanto para CPU como GPU. Esto se debe a que un LWS pequeño impide la realización de vectorizaciones necesarias. **Según crece el LWS también crece el rendimiento obtenido**, llegando a un LWS el cual no mejora más, indicando que las comunicaciones de lectura del fichero introducen una latencia que hace que no se pueda obtener un rendimiento mayor.

En el caso de la **ejecución con optimización**, como se puede ver en la Figura 4.16, **los diferentes LWS ofrecen el mismo rendimiento** para paquetes de 16384 muestras. Esto

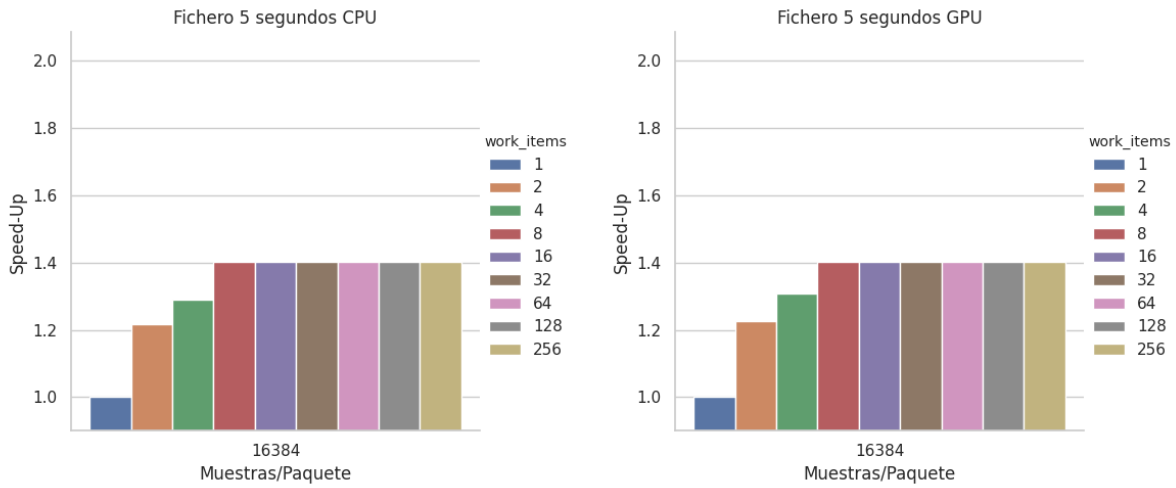

Figura 4.15: Speed-ups para fichero de 5 segundos con paquetes de 16384 ejecutado sobre la CPU y GPU sin optimización.

se debe a la optimización realizada sobre las colas de comandos. Dado que el cómputo del *kernel* no es grande, el cuello de botella se encontraba en la entrada y salida de datos del dispositivo.

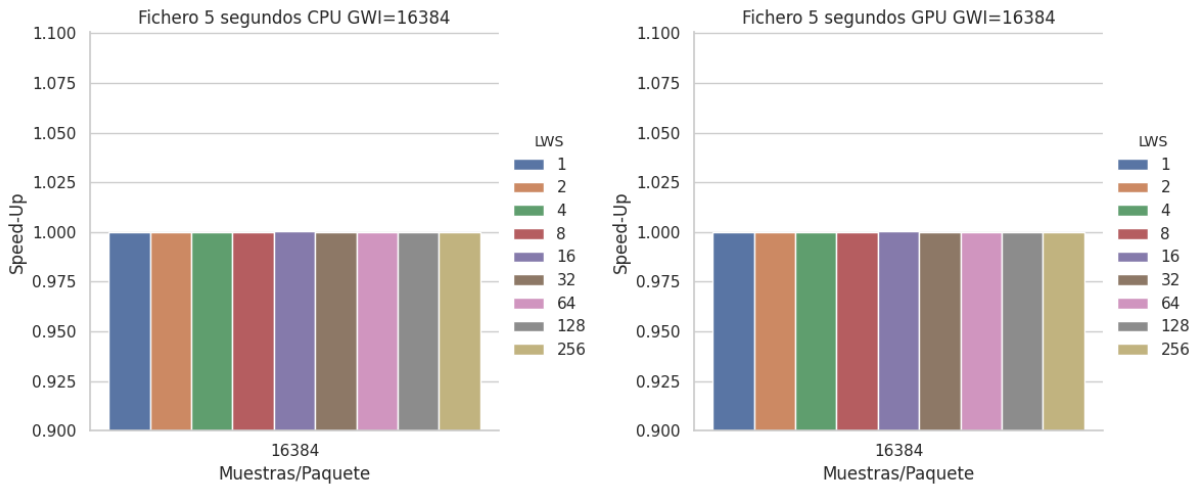

Figura 4.16: Speed-ups para fichero de 5 segundos con paquetes de 16384 ejecutado sobre la CPU y GPU con optimización.

Para el caso de los paquetes de 8192, en ambas ejecuciones se obtiene un resultados parejos a los obtenidos en el tamaño de paquete 16384.

En conclusión, no se han visto relevancias entre distintos *work-items* por *work-groups* una vez aplicadas las optimizaciones de colas. Estas optimizaciones permiten trasladar el cuello de botella, previamente situado en la entrada/salida de datos en la memoria del dispositivo, a las comunicaciones realizadas con el lector de ficheros.

4.4.2. ¿Qué dispositivo obtiene un mejor rendimiento?

Para responder a esta pregunta, se han clasificado los datos según el dispositivo y el tamaño utilizados en cada caso. Se recuerda que el procesamiento realizado es de los más simples que se puede computar a nivel de procesamiento de audio y por eso se prueba con los tamaños de paquete más grandes, para ver si la diferencia de tamaños también influye en alguno de los dispositivos.

En la Figura E.1 se observa como ambos dispositivos obtienen un rendimiento parejo en los dos tamaños de paquete. Por lo que el tamaño, en estas magnitudes, no parece influir.

Respondiendo a la pregunta, **da igual que dispositivo se utilice a la hora de ejecutar este *kernel*, ya que en ambos casos se obtiene un rendimiento parejo.** En este caso, se han probado los tamaños más grandes de paquete que permite el sistema porque son los que ofrecen un mayor rendimiento, pero es probable que al realizar la experimentación sobre un tamaño de paquete más pequeño, el rendimiento pueda bajar debido a las comunicaciones y la latencia que añaden al sistema.

4.5. Evaluación de FFT sobre OpenCL

Como se explica en la Sección 3.7.2, esta experimentación busca optimizar el uso de esta implementación del FFT mediante la división de los paquetes. Partiendo de que el *kernel* solo es capaz de ejecutar transformaciones correctamente a paquetes de 1024 muestras, se quiere estudiar si **merece la pena enviar paquetes de un tamaño mayor para después dividirlos y procesarlos como paquetes independientes de tamaño 1024.**

Se recuerda, como se explica en la Sección 3.7, que esta experimentación se ejecuta dos veces: la primera vez con un lector de ficheros de audio y la segunda vez con el sistema completo, incluyendo interfaz gráfica y todas las características de la aplicación multiplataforma.

Con esta experimentación se quieren responder dos preguntas relacionadas con los dispositivos que dispone el sistema y el trabajar con paquetes más grandes de 1024: la primera es determinar qué dispositivo actúa mejor haciendo el FFT y la segunda es comprobar si el trabajar con paquetes más grandes de 1024 y después tener que dividirlo añade mucho *overhead* al sistema.

4.5.1. ¿Qué dispositivo hace mejor el FFT para estos tamaños de problema?

Para responder correctamente a esta pregunta, los resultados se han clasificado por ficheros y tamaños de paquete, dividiendo también los resultados en las dos ejecuciones. De este modo se puede observar claramente si alguno de los dispositivos predomina sobre el otro en alguna de las diferentes configuraciones del sistema.

Simulación Como se puede ver en la Figura 4.17, para paquetes de 1024, es decir, sin división a la hora de realizar el cálculo, los tiempos de ejecución en ambos procesadores es muy parecido, difiriendo en milésimas de segundo. Cabe destacar que la CPU obtiene unos mejores resultados, pero difieren de la GPU por 5 milésimas de segundo en el mejor de los casos. Teniendo en cuenta las desviaciones estándar que se ven en las diferentes figuras, se puede ver que para las pruebas realizadas en ficheros de 5 segundos ambos procesadores operan muy parecido. En el caso

de los ficheros de 20 segundos, también fijándose en las desviaciones estándar, la CPU opera mejor pero con una desviación muy grande, llegando casi a la media que se obtiene para la GPU.

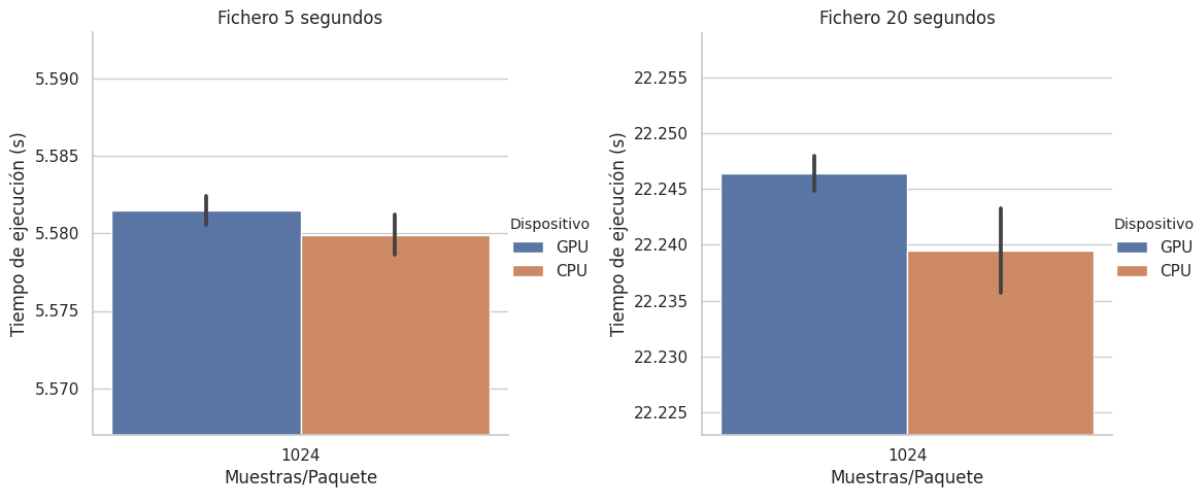

Figura 4.17: Tiempos de ejecución en CPU y GPU en paquetes de 1024 en simulación.

Para el caso de los paquetes de 8192, como se pueden ver en la Figura 4.18, los tiempos son aún más parejos que antes, aunque en este caso, la GPU obtiene unos resultados mejores que la CPU, tanto para los ficheros de 5 segundos como para los ficheros de 20 segundos. En estos casos las desviaciones estándar son bastante pequeñas, con lo que se sabe que las diferentes pruebas han obtenido unos resultados muy parecidos.

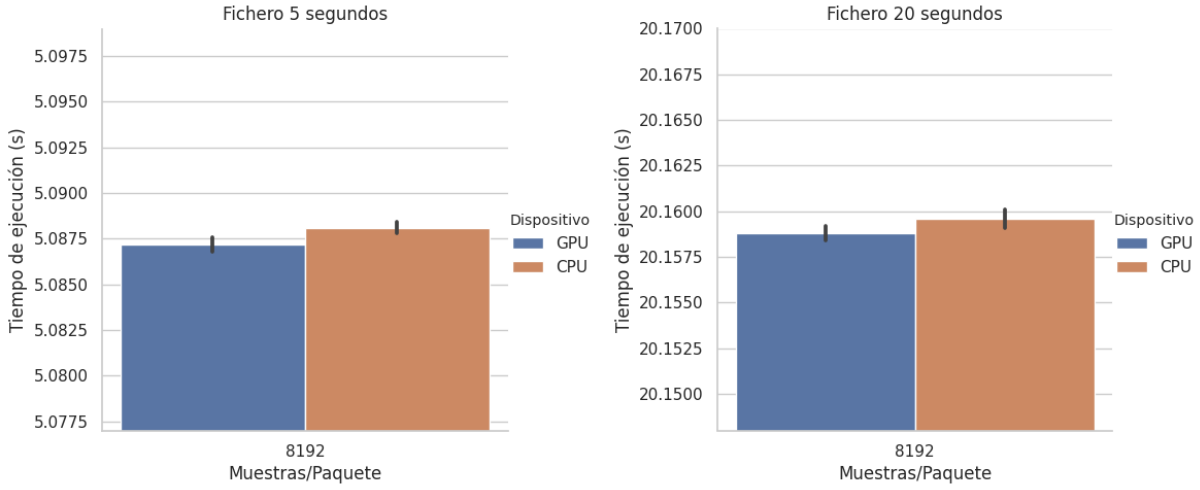

Figura 4.18: Tiempos de ejecución en CPU y GPU en paquetes de 8192 en simulación.

En base a estos resultados, **se llega a la conclusión de que ambos procesadores actúan igual ante este *kernel*.** Estos no son los resultados que se esperaban, ya que se pensaba que la GPU superaría a la CPU. Teniendo en cuenta las optimizaciones realizadas al código *host* y que el *kernel* está desarrollado por una gran empresa como AMD, se plantea la idea de que la

implementación sea muy eficiente tanto para CPU como GPU.

Sistema real En el caso de la experimentación en el sistema real, se puede ver en la Figura 4.19, que en los resultados para los paquetes de tamaño 1024 se obtienen un resultados muy parecidos entre ambos dispositivos, tanto para las pruebas con ficheros de 5 segundos como para las pruebas de ficheros de 20 segundos. Las desviaciones estándar que se observan son más grandes que las vistas en el caso de la simulación, esto puede indicar cierta inestabilidad debido a la ejecución de toda la aplicación.

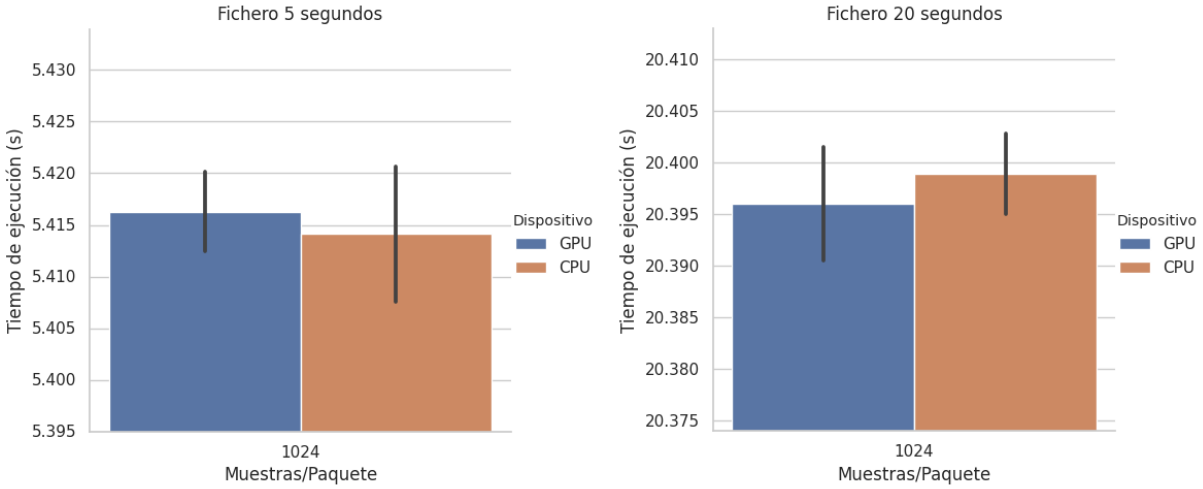

Figura 4.19: Tiempos de ejecución en CPU y GPU en paquetes de 8192 en sistema real.

Para el caso de los paquetes de tamaño 8192, como se puede observar en la Figura 4.20, ambos dispositivos vuelven a obtener unos resultados muy parejos, reafirmando la teoría de que las latencias introducidas por las comunicaciones no permiten aprovechar la capacidad de cómputo de ambos dispositivos.

En conclusión, **el sistema real tampoco deja ver ningún dispositivo como mejor para realizar esta operación.**

En cuanto a las diferencias entre la simulación y el sistema real, ambos consiguen unos resultados muy parecidos en cuanto al rendimiento de ambos dispositivos. Esto indica que la simulación ha sido muy próxima al sistema real y que si es necesario volver a hacer experimentación y no se dispone de un sistema el cual pueda ejecutar la aplicación multiplataforma, siempre se puede usar el sistema de lectura de ficheros.

4.5.2. ¿La división de paquetes para el cómputo del FFT añade mucho overhead?

Con esta pregunta se quiere comprobar qué penalizaciones conlleva dividir los paquetes y procesar sus divisiones. Para ello, se clasifican los resultados según la longitud del fichero y el dispositivo sobre el que se ejecuta.

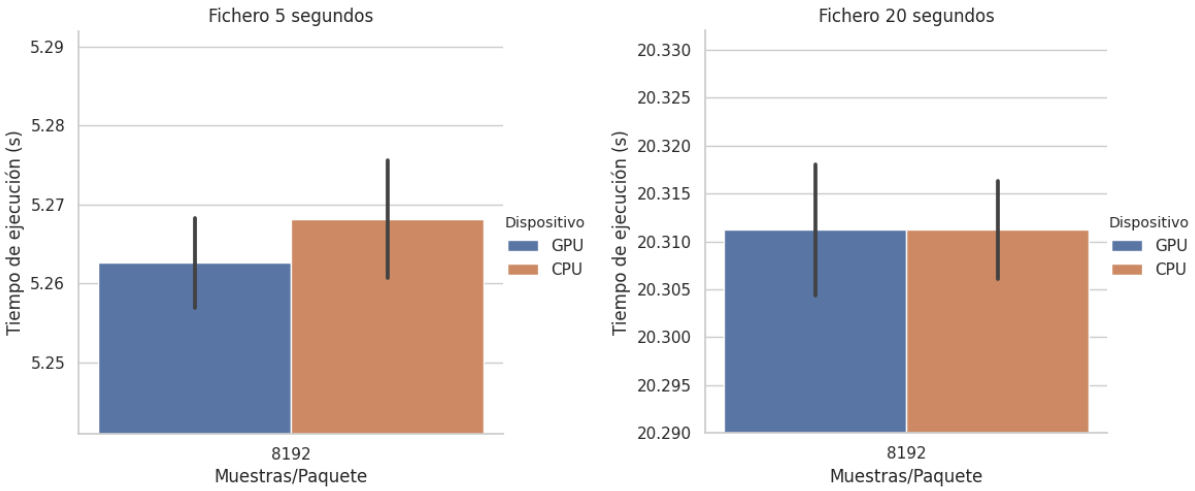

Figura 4.20: Tiempos de ejecución en CPU y GPU en paquetes de 8192 en sistema real.

Simulación Como se ve en las Figuras 4.21 y 4.22, tanto para CPU como para GPU, la división de paquetes obtiene un *speed-up* superior respecto al envío de paquetes de 1024. Con esto se plantea la teoría de que las comunicaciones están introduciendo demasiada latencia, ya que el número de mensajes, al trabajar con paquetes más grandes, es menor. También se observa como, para ambos dispositivos, el tiempo de ejecución de las pruebas con paquetes de 8192 se mantiene muy cerca del limite de los 5 segundos, rozando el tiempo real, mientras que las pruebas con paquetes de 1024 obtienen un resultado bastante peor en comparación, aumentando el tiempo de ejecución medio segundo.

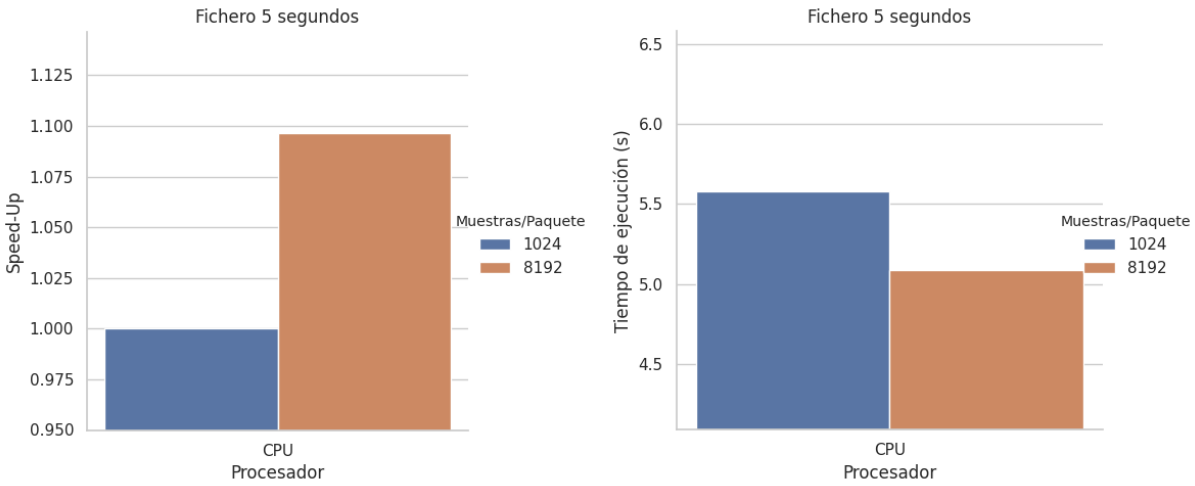

Figura 4.21: Speed-up y tiempo de ejecución en CPU para ficheros de 5 segundos en simulación.

Con los ficheros de 20 segundos ocurre lo mismo que con los ficheros de 5 segundos, como se puede ver en las Figura F.1 y F.2. Los paquetes más grandes obtienen un mejor *speed-up*, haciendo más fuerte la idea de que las comunicaciones y la latencia de lectura del fichero tengan el problema. La proporción de mensajes que se envían entre ficheros de 5 segundos y ficheros de

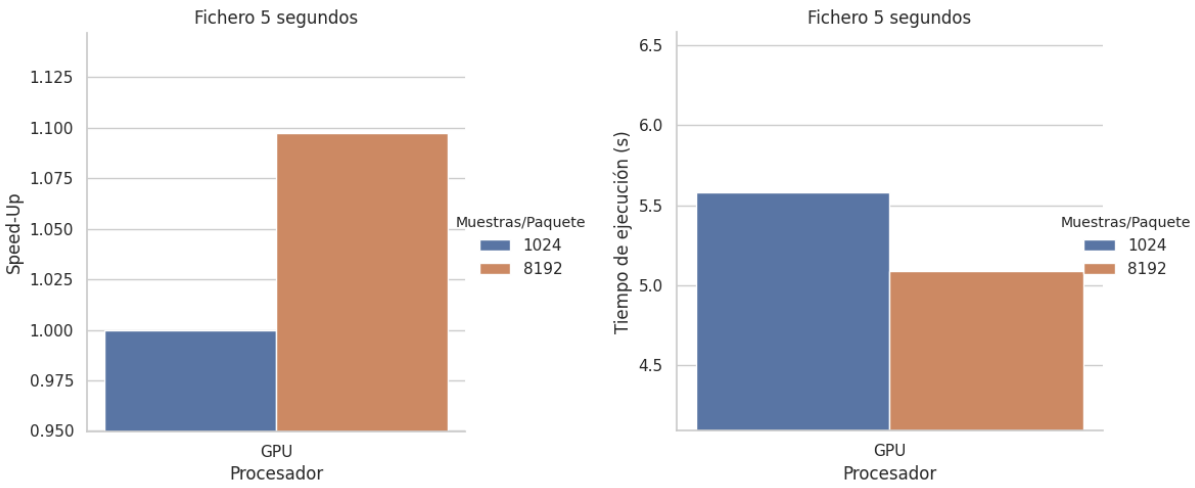

Figura 4.22: Speed-up y tiempo de ejecución en GPU para ficheros de 5 segundos en simulación.

20 segundos es la misma cuando se modifica el tamaño de los paquetes. En este caso, cuando se pasa a utilizar paquetes de 8192, el envío de mensajes se ve reducido hasta una octava parte de los que se enviaban con tamaños de 1024. Por ello, los *speed-ups* quedan muy parecidos entre ficheros.

La deducción a la que se llega es que **el envío de tantos paquetes de 1024 respecto a la agrupación de los mismos paquetes en otros de tamaño 8192 añade un *overhead* grande** debido a todas las comunicaciones que se tiene que hacer con el nodo que se encarga de la reproducción o grabación del fichero de audio.

Sistema real Como se puede ver en las Figuras 4.23 y 4.24, los resultados obtenidos en la experimentación con el sistema real dejan el mismo resultado que con la simulación. Trabajar con paquetes de 8192 permite un mayor rendimiento debido a la menor latencia introducida por la lectura del fichero y las comunicaciones. Sin embargo, el rendimiento empeora, probablemente por la ejecución de toda la interfaz gráfica y los diferentes nodos que engloban dicha interfaz.

Para las pruebas realizadas sobre ficheros de 20 segundos en el sistema real, como se puede ver en las Figuras F.3 y F.4, los resultados también se ven afectados respecto a las pruebas en simulación, pero en menor medida que las pruebas sobre ficheros de 5 segundos. Esto probablemente se deba a la inicialización del sistema.

Para las pruebas realizadas sobre el sistema real, se sacan las mismas conclusiones que para las pruebas realizadas con la simulación del sistema, pero incluyendo que la aplicación multiplataforma añade también una latencia, probablemente por las comunicaciones, ya que los paquetes tienen que atravesar los tres componentes del sistema.

Y finalmente, respondiendo a la pregunta de si trabajar con paquetes más grandes de 1024 para su posterior división añade mucho *overhead*, la respuesta es no. Con estos resultados se demuestra que **es peor utilizar paquetes más pequeños ya que se tienen que realizar más envíos de mensajes, lo que implica una mayor latencia en las comunicaciones**

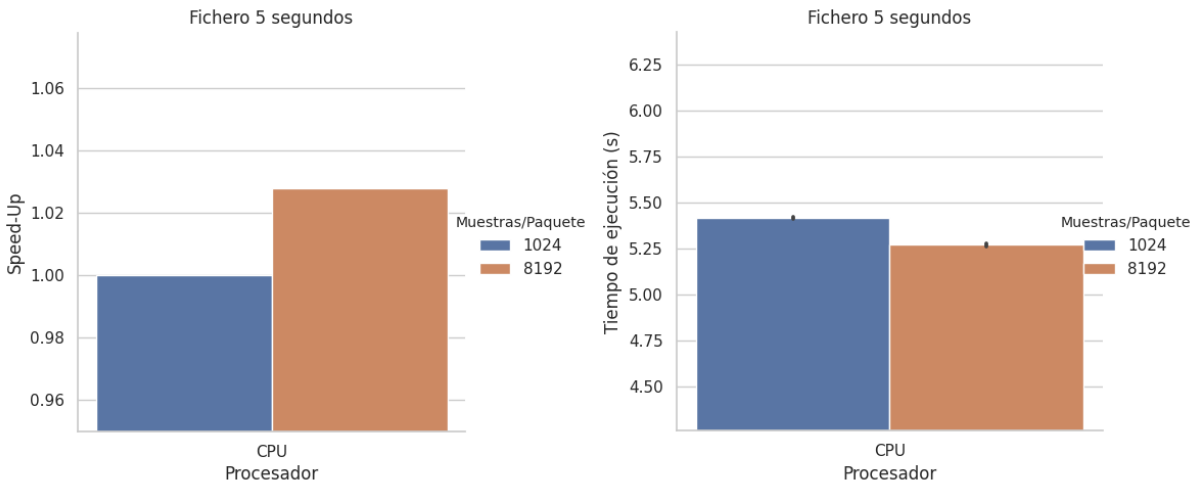

Figura 4.23: Speed-up y tiempo de ejecución en CPU para ficheros de 5 segundos en sistema real.

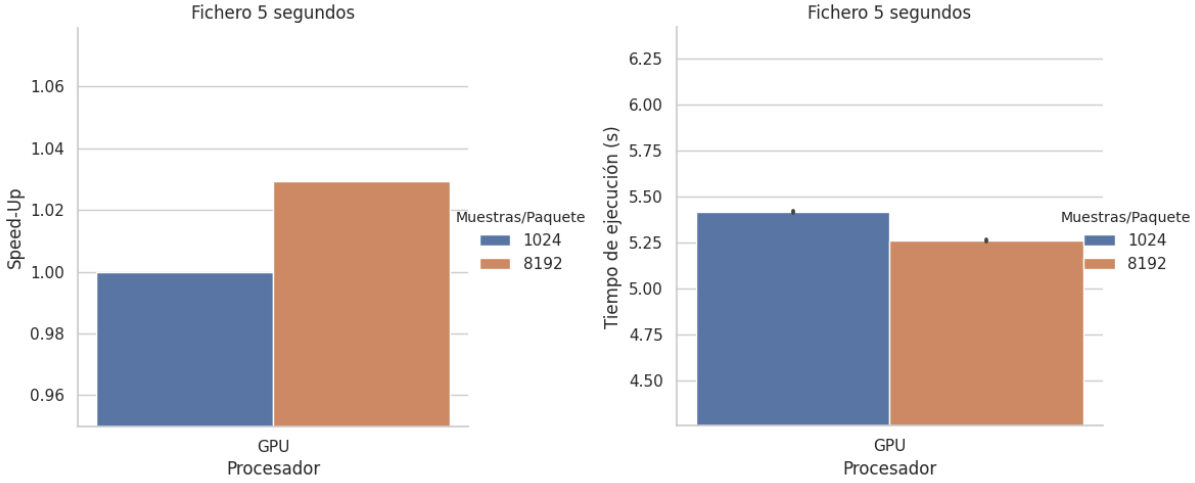

Figura 4.24: Speed-up y tiempo de ejecución en GPU para ficheros de 5 segundos en sistema real.

entre nodos. Esa diferencia se ve reducida cuando se trabaja con el sistema real, pero sigue resultando mejor operar con los tamaños de paquete más grandes.

Capítulo 5

Conclusiones y trabajos futuros

En este último capítulo se expondrán las conclusiones del proceso de desarrollo y posterior experimentación, incluyendo el cumplimiento de objetivos, el aspecto personal, las dificultades encontradas, conocimientos aprendidos y valoración del trabajo.

Además se presentarán una serie de ideas sobre posibles trabajos futuros para seguir con la realización del proyecto.

5.1. Conclusiones

Como se definen en la Sección 1.4, los objetivos del trabajo abordan diferentes estados de la implementación y sus experimentaciones. De los objetivos previamente mencionados, se va a comentar qué se ha conseguido explicando detalles que han marcado la diferencia en cuanto a dificultad:

- Como primer objetivo, el estudio sobre las tecnologías actuales, en el Capítulo 2 se han presentado las tecnologías utilizadas y las razones por las que fueron escogidas. El principal motivo por el que se escogieron las tres principales tecnologías (Electron, Node.js y C/C++) es la facilidad de comunicación entre ellas, el gran rendimiento y flexibilidad que ofrecen cada una de ellas individualmente y que las tres son multiplataforma. Según los resultados obtenidos en la Sección 4.2, tanto la Web Audio API como la comunicación implementada en Electron, permiten una manipulación rápida del audio para su posterior procesamiento. Al poder utilizarse en múltiples plataformas, el sistema puede usarse para el estudio de dichas plataformas y cómo es el comportamiento del audio en ellas. Cabe destacar la facilidad para conectar el sistema a cualquier nodo de procesamiento mediante el uso de Nanomsg.

- Otro de los objetivos ha sido el desarrollo de la aplicación multiplataforma con capacidad para grabar y reproducir ficheros de audio. En el Capítulo 3 se narra como ha sido el desarrollo de la aplicación y las funcionalidades implementadas. El diseño de la arquitectura y la implementación de esta aplicación no supuso una dificultad muy grande, dado que los problemas surgieron en las funcionalidades más específicas, como son: la captura de paquetes de audio cuando los datos provenían de un fichero o la generación de ficheros de audio con la cabecera WAV.

- En cuanto a la optimización de la aplicación, se han conseguido buenos resultados para la implementación de un procesamiento, como se puede ver en la Sección 4.1.2. **Mediante el**

uso de modelos de programación como OpenMP se ha conseguido reducir los tiempos de análisis de los paquetes y de notificación, lo que permite una mayor velocidad a la hora de detectar picos de audio. Esto es útil ya que podría utilizarse como base para un sistema de detección de palabras o un filtro de ruido. Por otro lado, las optimizaciones hechas en OpenMP han sido de gran ayuda a la hora de la realización de optimizaciones sobre el sistema heterogéneo utilizado con OpenCL. Las pruebas que se han podido realizar han sido satisfactorias, obteniendo unos tiempos buenos tanto en CPU como en GPU, aunque queda mucha investigación por delante.

- Para el objetivo de evaluación de los desarrollos y experimentos realizados, en el Capítulo 4 se explican las diferentes experimentaciones realizadas y el análisis de sus resultados mediante el uso de diversas gráficas que facilitan el entendimiento de cada experimento y de las preguntas que se quieren responder. También identificando problemáticas y conclusiones en multitud de vertientes.

A continuación se presentaran las conclusiones obtenidas de los análisis efectuados sobre los resultados globales de la evaluación experimental llevada a cabo:

- El estudio de las comunicaciones de la aplicación realizada ha dado unos resultados satisfactorios. Esto, en gran parte, se debe a que se ha trabajado con una API muy bien optimizada como es la Web Audio API. Cabe destacar que las respuestas a las diferentes preguntas también son satisfactorias, ya que **el sistema permite utilizar características secundarias, como la representación del audio, sin perjudicar al rendimiento.**

- En cuanto a la optimización del procesamiento, también ofrece unos buenos resultados. Por un lado, la experimentación sobre la latencia de notificación del primer y último paquete demuestra que el sistema es fluido, gracias a las **optimizaciones con vectorizaciones**. No se atasca durante dicho procesamiento y permite que todos los paquetes, una vez lleguen al nodo de procesamiento, se procesen nada más llegar. Por otro lado, en cuanto al tiempo de ejecución del programa y el *speed-up* de las diferentes pruebas, las versiones con vectorización en sus optimizaciones permiten una ejecución cercana al tiempo real.

- Los diferentes estudios realizados sobre implementaciones con OpenCL han permitido tener una visión más cercana sobre la complejidad de la programación *host-device* para procesamiento de audio. **Se ha estudiado la viabilidad, con resultados muy satisfactorios, pudiendo comenzar a experimentar en ambos dispositivos y viendo cómo la ejecución en una GPU integrada puede ser una opción.** Es importante destacar la complejidad de OpenCL, sobre todo el en lado del *host*, debido a las optimizaciones que se han realizado, que han sido clave para la obtención de buenos resultados así como un quebradero de cabeza para su optimización.

Respecto a la mayoría de algoritmos y filtros de audio, tienen limitaciones para el procesamiento paralelo basado en el paralelismo de datos y vectorización. Es por esto que los algoritmos escogidos hayan sido un umbral simple y un FFT.

Cabe destacar que se han podido realizar experimentaciones tanto en CPU como en GPU, que es un objetivo fundamental en este proyecto. Es algo que con los algoritmos existentes, tanto secuenciales como los optimizados con OpenMP, solo se opta a CPU.

5.2. Valoración personal

En el contexto personal y profesional, pienso que este proyecto me ha permitido realizar un trabajo con el que he estado a gusto y me ha aportado bastante como programador, siendo también el trabajo más complejo que he tenido que abordar hasta ahora. Esto es debido al uso de múltiples tecnologías y a la preparación, realización y análisis de varios experimentos manejando miles de datos.

Para empezar, he conocido tecnologías y librerías que no sabía de su existencia, las cuales me han parecido útiles y potentes como para usarlas en proyectos personales. Después, me he dado cuenta de la complejidad de hacer las cosas bien y de las veces que hay que leer todo para de verdad interiorizarlo y poder decir que lo has entendido.

Por otro lado, he desarrollado otro tipo de pensamiento a la hora de abordar problemas debido a las vueltas que he tenido que darle a ideas y planteamientos sobre ciertas dificultades que se me han ido cruzando en el desarrollo. Cuando siempre tienes una manera de abordar los problemas y por lo general ha funcionado, te sorprende cuando encuentras otra manera de hacer las cosas y además, obtienes mejores resultados. No hay que cerrarse a como has trabajado siempre, sino que hay que estar abierto a otro tipo de pensamientos y maneras de abordar los problemas, por más duro que pueda resultar.

5.3. Trabajos futuros

Ahora se van a dar unos detalles sobre el trabajo que se podría realizar en el futuro para seguir con la investigación sobre el audio y su procesamiento fuera de la CPU:

- **Diferentes procesamientos de audio**: como se ha destacado anteriormente, una de las limitaciones que se encuentra en el procesamiento del audio de manera paralela es la complejidad para realizar dichas adaptaciones a un sistema en el que el modelo de ejecución no es el mismo. Por ello, se debería dedicar un gran parte del tiempo a buscar o desarrollar un número limitado de algoritmos para tener una base desde la que partir.
- **Experimentación con procesadores gráficos dedicados e integrados**: de este modo, se podrá estudiar si la latencia de entrada y de salida de los datos se mantiene o se elimina con el uso de un procesador gráfico integrado.
- **Aplicación de estrategias de optimización**: como optimizaciones de memoria, de transferencia de datos o reutilización de mapeados de memorias de *host*.
- **Experimentación más detallada sobre el FFT**: para realizar comparaciones entre diferentes implementaciones que permitan un tamaño variable del problema. Estás implementaciones podrían ser: la utilizada ya de AMD y el proyecto clFFT.
- **Contrastar el rendimiento entre los diferentes modelos de programación**: ahora que OpenCL puede computar eficientemente en el nodo de procesamiento.
- **Optimizar el nodo de frontend a modo *headless***: es decir, con la interfaz oculta, para ahorrar recursos y aumentando así el cuello de botella en el propio procesamiento.
- **Portar el actual sistema a nodos embebidos**: como móviles, para estudiar la viabilidad de su procesamiento.

Bibliografía

[1] David Kaeli. *Heterogeneuous Computing with OpenCL 2.0*. 2015. URL: https://www.sciencedirect.com/book/9780128014141/heterogeneous-computing-with-opencl-2-0.

[2] Petr F. Kartsev. *Application of computational GPU OpenCL kernels for near-realtime audio processing*. Mayo de 2018. URL: https://dl.acm.org/doi/pdf/10.1145/3204919.3204943?download=true.

[3] Github Inc. *Electron*. 2020. URL: https://www.electronjs.org/.

[4] Google Inc. *Chromium*. 2008. URL: https://www.chromium.org/Home.

[5] Github Inc. *IPCMain*. URL: https://www.electronjs.org/docs/api/ipc-main.

[6] Github Inc. *IPCRenderer*. URL: https://www.electronjs.org/docs/api/ipc-renderer.

[7] Colaboradores de MDN. *Algoritmo de clonación estructurado*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm.

[8] *Web oficial de wavesurfer.js*. URL: https://wavesurfer-js.org/.

[9] Riverbank Computing. *PyQt*. 2018. URL: https://riverbankcomputing.com/software/pyqt/intro.

[10] Qt Project. *Qt*. 1995. URL: https://www.qt.io/.

[11] Colaboradores de MDN. *Web Audio API*. 2019. URL: https://developer.mozilla.org/es/docs/Web_Audio_API.

[12] Colaboradores de MDN. *ScriptProcessorNode*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/ScriptProcessorNode.

[13] Colaboradores de MDN. *AudioWorkletNode*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/AudioWorkletNode.

[14] Colaboradores de MDN. *AnalyserNode*. 2019. URL: https://developer.mozilla.org/es/docs/Web/API/AnalyserNode.

[15] Node.js Foundation. *Node.js*. 1995. URL: https://nodejs.org/es/.

[16] Craig Stuart Sapp. *Formato Wav*. 1991. URL: http://soundfile.sapp.org/doc/WaveFormat/.

[17] Garrett D'Amore. *Nanomsg*. 2012. URL: https://nanomsg.org/.

[18] Bjarne Stroustrup. *C++*. URL: https://isocpp.org/.

[19] OpenMP Architecture Review Board. *OpenMP*. URL: https://www.openmp.org/.

[20] Grupo Khronos. *OpenCL*. URL: https://www.khronos.org/opencl/.

[21] *Web oficial de Python*. URL: https://www.python.org/.

[22] *Web oficial de pandas*. URL: https://pandas.pydata.org/.

[23] *Web oficial de matplotlib*. URL: https://matplotlib.org/.

[24] *Web oficial de seaborn*. URL: https://seaborn.pydata.org/.

[25] *Web oficial de WebApi*. URL: https://developer.mozilla.org/en-US/docs/Web/API/MediaRecorder.

[26] *AudioContext*. URL: https://developer.mozilla.org/en-US/docs/Web/API/AudioContext.

[27] *Definición de la cabecera Wave*. URL: http://soundfile.sapp.org/doc/WaveFormat/.

[28] *Repositorio del FFT desarrollado por AMD*. URL: https://github.com/EngineCL/Benchsuite.

Anexos

Anexo A

Experimentación sobre las comunicaciones: mejor protocolo

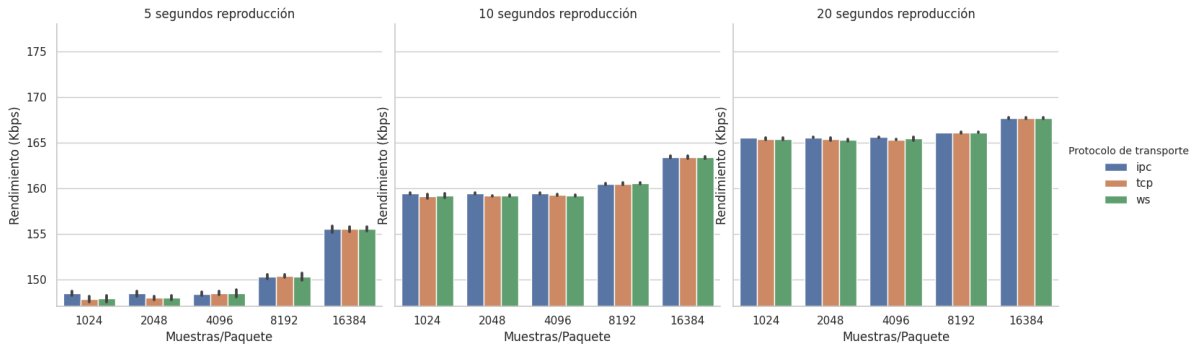

Figura A.1: Rendimientos de la reproducción con los diferentes protocolos de transporte.

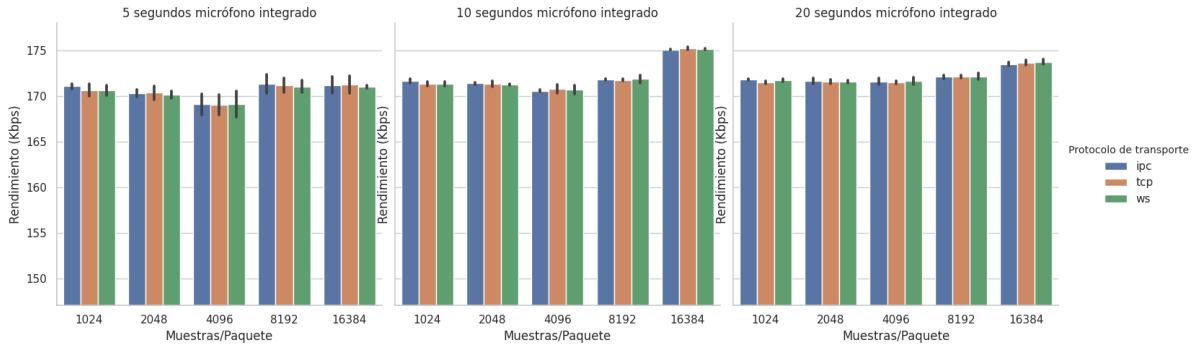

Figura A.2: Rendimientos de la grabación con micrófono integrado con los diferentes protocolos de transporte.

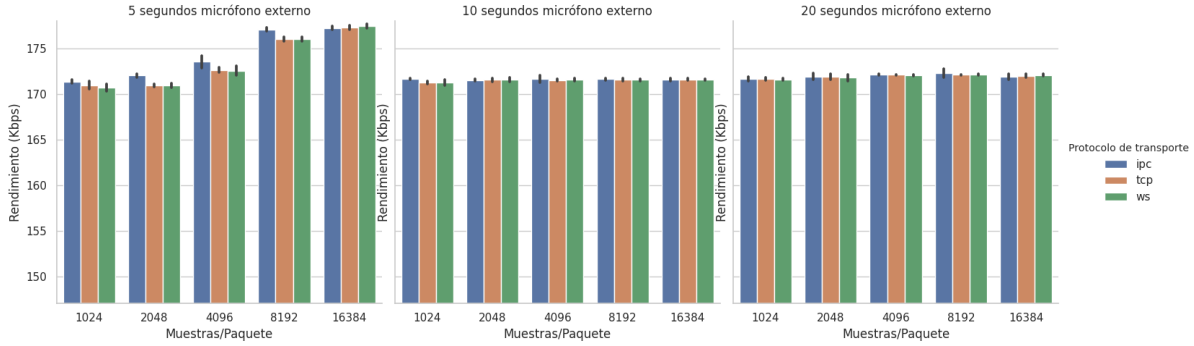

Figura A.3: Rendimientos de la grabación con micrófono externo con los diferentes protocolos de transporte.

Anexo B

Experimentación sobre las comunicaciones: penalización al grabar o reproducir

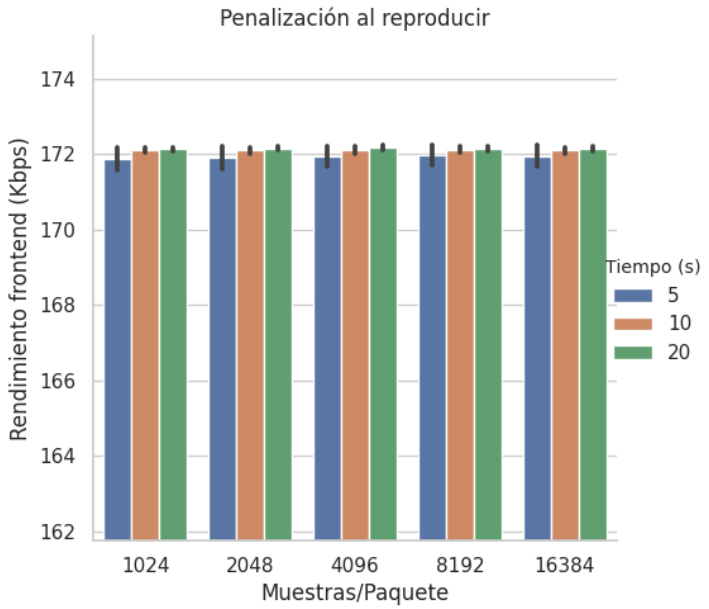

Figura B.1: Comparación de los rendimientos de la reproducción entre los distintos tiempos.

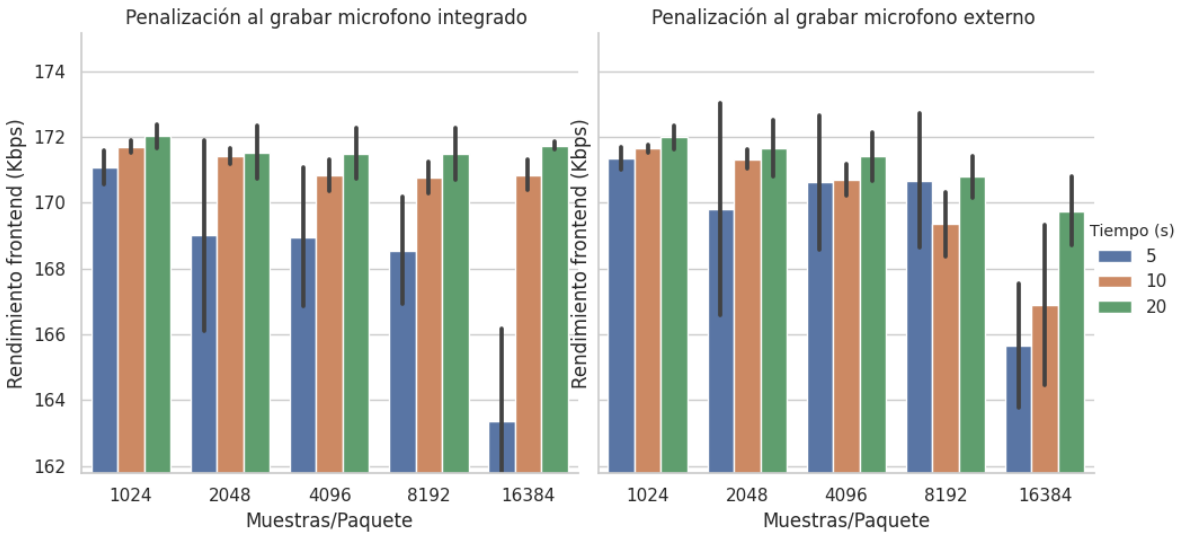

Figura B.2: Comparación de los rendimientos de la grabación entre los distintos tiempos.

Anexo C

Optimización de procesamiento: latencias del último paquete

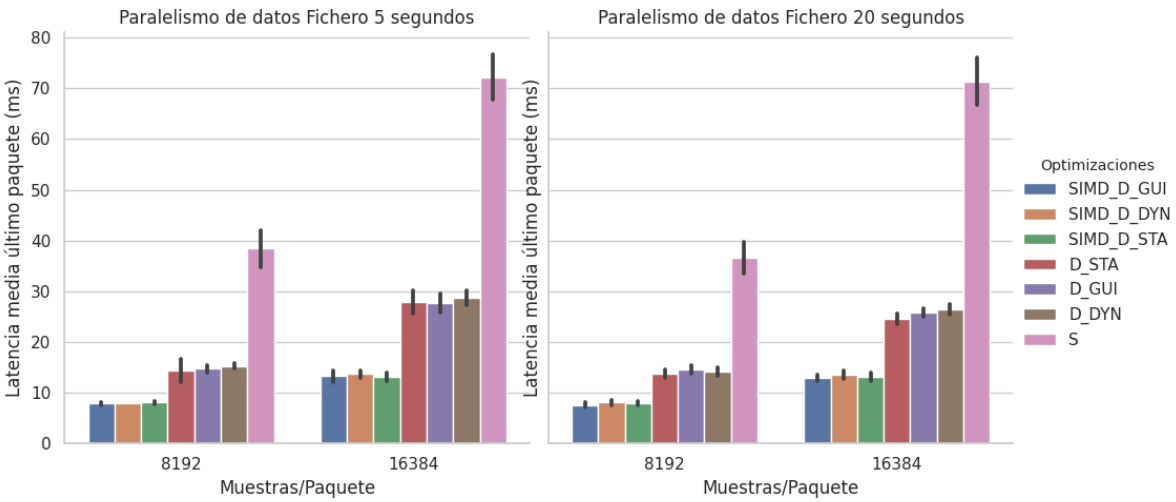

Figura C.1: Latencias del último paquete en las pruebas con datos.

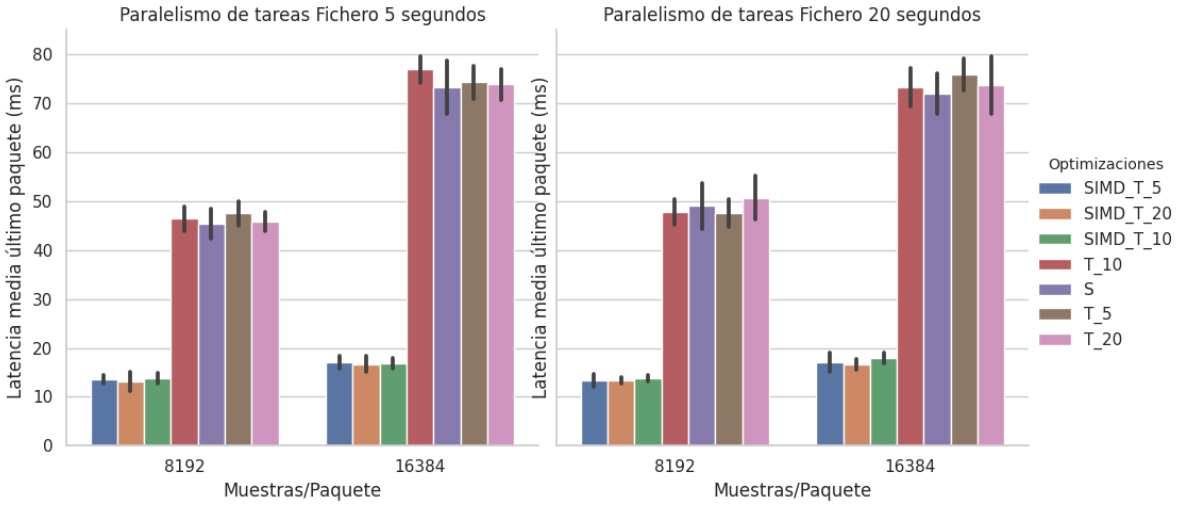

Figura C.2: Latencias del último paquete en las pruebas con tareas.

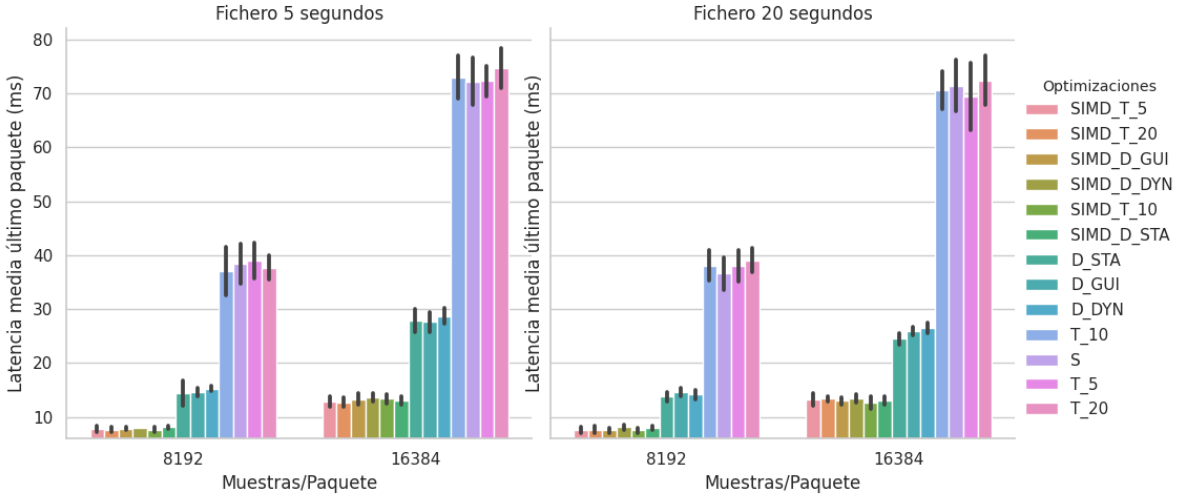

Figura C.3: Latencias del último paquete en las pruebas.

Anexo D

Optimización de procesamiento: speed-ups

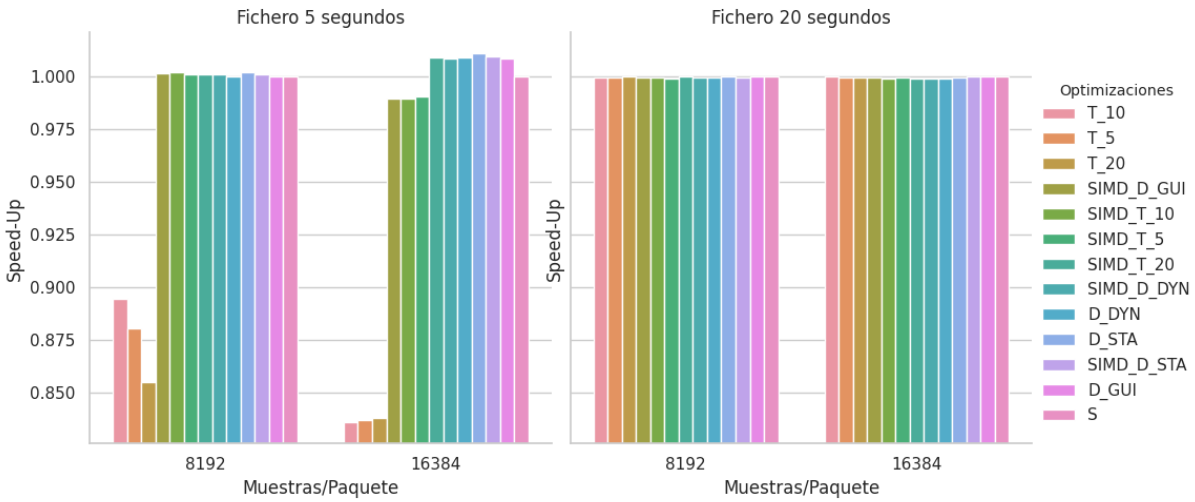

Figura D.1: Speed-ups de las optimizaciones.

Anexo E

Optimización de procesamiento en OpenCL: mejor dispositivo

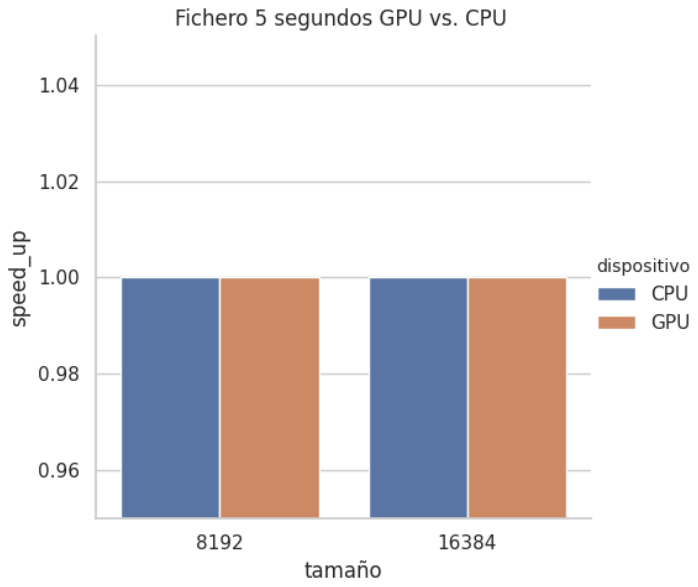

Figura E.1: Speed-ups medios para comparar CPU y GPU.

Anexo F

Experimentación con FFT en OpenCL: overhead por la división de paquetes

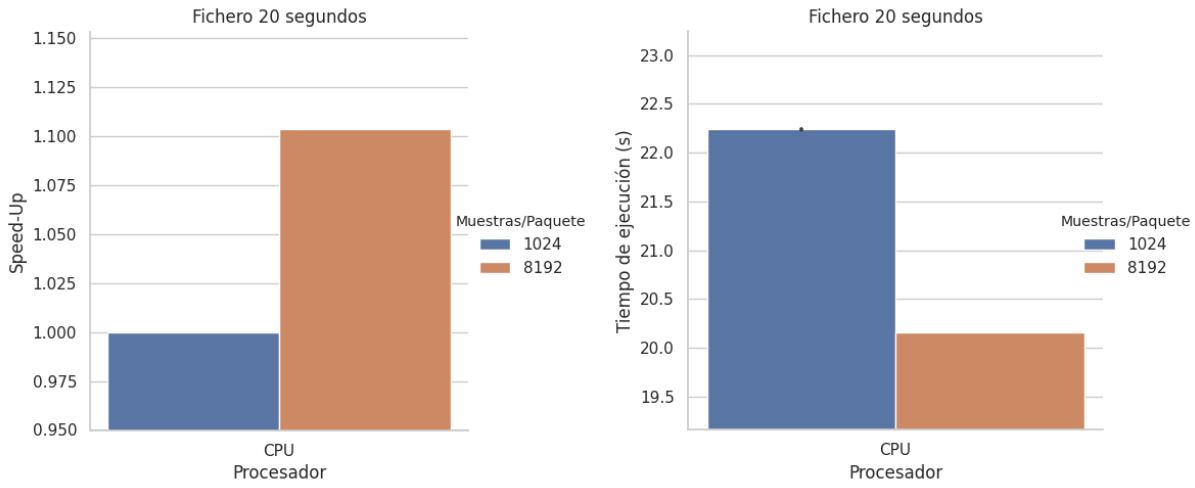

Figura F.1: Speed-up y tiempo de ejecución en CPU para ficheros de 20 segundos en simulación.

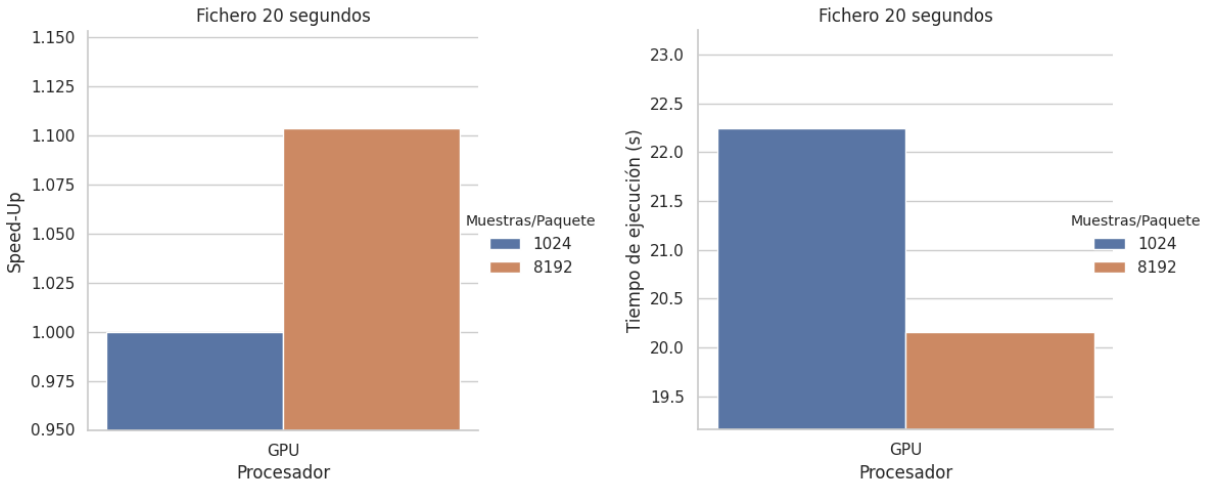

Figura F.2: Speed-up y tiempo de ejecución en GPU para ficheros de 20 segundos en simulación.

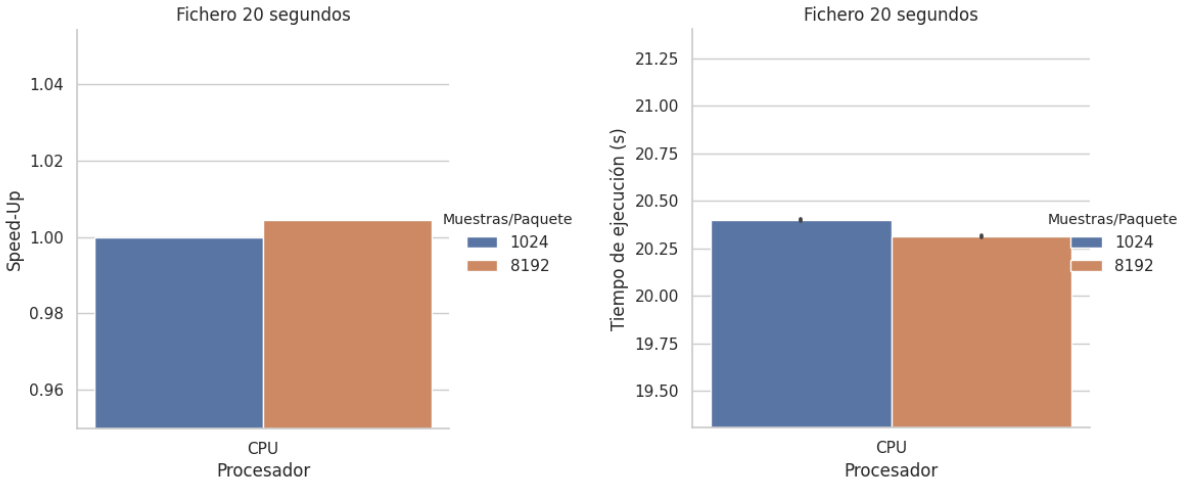

Figura F.3: Speed-up y tiempo de ejecución en CPU para ficheros de 20 segundos en sistema real.

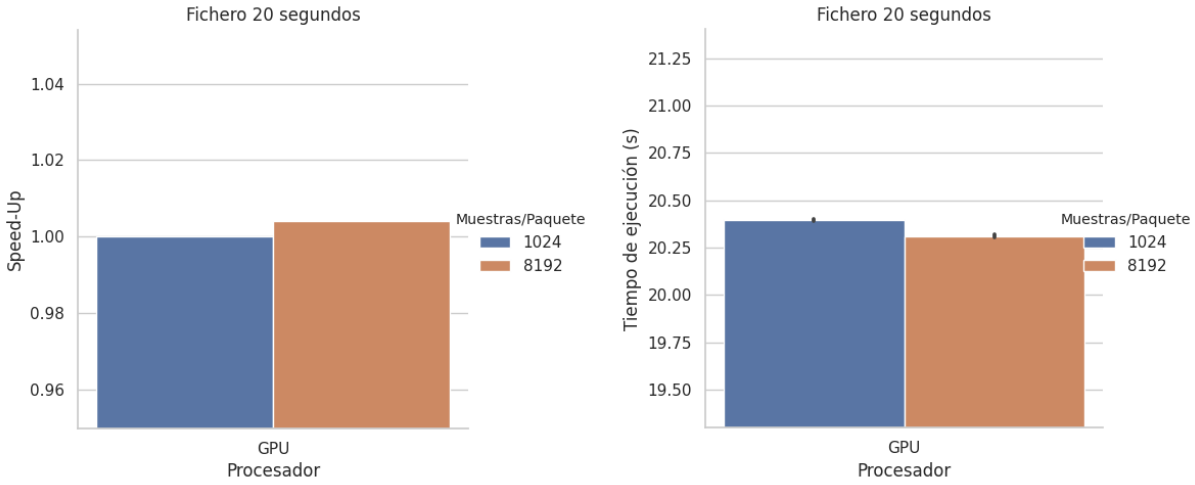

Figura F.4: Speed-up y tiempo de ejecución en GPU para ficheros de 20 segundos en sistema real.