Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

# Una aplicación para el alineamiento de partituras en tiempo real.

Trabajo Fin de Grado

**Grado en Ingeniería Informática**

**Autor**: Gómez Morillas, Carlos

**Tutor**: Alonso Jordá, Pedro

2016-2017

A mi abuelo Joaquín, que no llegó a ver cómo este trabajo terminaba después de tanto tiempo.

# Resumen

El alineamiento de partituras (Audio-to-Score) es una aplicación muy útil que requiere de cierta capacidad computacional, así como de algoritmos eficientes que permita llevarse a cabo en tiempo real. El trabajo consiste en implementar esta aplicación y en montar un sistema software (mediante herramientas tipo gmake o autoconf) sobre el que ir incorporando los algoritmos que se desarrollen. Se trabajará con los lenguajes C y CUDA sobre Linux pero también en entornos Android, ya que la aplicación final está dirigida a dispositivos móviles como los tablets.

**Palabras clave:** Programación C, CUDA, GPUs, Android, partituras de música

# Abstract

The “sheet alignment” (Audio-to-Score alignment) is a very useful application for musicians that requires some computing power to enable the application to work in real time. The work consists on implementing this application and setting up all the software system (using gmake type and Autoconf tools) on which to incorporate the current and future algorithms developed. We will work with the languages C and CUDA on Linux but also on Android environments, since the final application aims to be installed on mobile devices such as tablets

**Keywords**: C programming, CUDA, GPUs, Android, music sheets

# Tabla de contenidos

# Índice de imágenes

# 1. Introducción

## 1.1 Motivación

En la actualidad gran parte de dispositivos de consumo cotidiano, como *smartphones* o tablets cuentan con procesadores multinúcleo, lo cual les permite trabajar con aplicaciones que requieran de mayor potencia de cálculo. Un uso interesante de esta potencia se puede encontrar en el procesamiento de señales para aplicaciones musicales.

Dentro de este campo se utilizan sistemas que permitan realizar un alineamiento de partituras, es decir, procesar la señal asociada a una interpretación frente a la partitura real para poder, en primer lugar, simular un instrumento que sirva de acompañamiento a dicha interpretación ajustándose en *tempo* a la interpretación real y, por otro lado, permitir leer la partitura que está siendo interpretada por el músico. Este tipo de aplicaciones son muy interesantes para los músicos y requieren de una cantidad de cómputo considerable para poder permitir el procesamiento en tiempo real.

Para este proceso, y otros relacionados con el reconocimiento de señales acústicas, como pudiera ser de reconocimiento del habla, utilizan comúnmente un algoritmo de alineamiento temporal dinámico, o *DTW* por sus siglas en inglés, que se encarga de medir los patrones que aparecen entre señales para poder obtener una sincronización entre ambos.

Este procesamiento planteado presenta la ventaja de que es escalable a diferentes tipos de arquitectura. En este trabajo nos centramos en el desarrollo de un sistema software que permita alinear dos señales dadas utilizando todo el potencial ofrecido por los sistemas multinúcleo de las arquitecturas móviles de ARM junto con la programación de un sistema con soporte para la computación heterogénea, donde se busca obtener un paralelismo que utilice las capacidades de cálculo paralelo de un chip gráfico. En este trabajo nos centraremos en ver cómo afecta el algoritmo *DTW* mediante una implementación que explote el mayor grado de paralelismo posible para poder procesar las señales de manera que se consiga un alineamiento de la señal en tiempo real.

## 1.2 Objetivos

Con el desarrollo de este trabajo buscamos realizar una aplicación que permita, por un lado, procesar y sincronizar la señal de audio emitida por un instrumento musical para así poder transcribir dicha señal en una partitura, mientras que por otro lado también sea capaz de interpretar partituras [1].

Además, este proyecto tiene como finalidad que la aplicación funcione en diferentes arquitecturas, sistemas y dispositivos. Para ello debemos generar un paquete de software que sea modular y permita ejecutarse en la variedad de dispositivos mencionada. Además, la facilidad de uso de dicho paquete debe ser otro de los objetivos principales a desarrollar, ya que cualquier usuario debe ser capaz de entender su funcionamiento básico.

Para la consecución de estos objetivos principales se debe cumplir con los siguientes requerimientos:

- Diseñar una arquitectura de programa monolítica pero que ofrezca modularidad en su diseño para integrar diferentes elementos de manera flexible.
- Establecer una capa superior al programa que permita la instalación y configuración del mismo de acuerdo a las características del sistema donde se debe ejecutar.
- Dar soporte en un mismo paquete software a diferentes implementaciones de la misma aplicación.
- Optimizar dicho software para buscar el máximo grado de paralelismo
- Configurar el hardware utilizado para las pruebas para ofrecer compatibilidad con la aplicación.

## 1.3 Estructura de la obra

El texto está estructurado en los siguientes capítulos:

- **Introducción**

El primer capítulo de este texto presenta, además de los objetivos y la motivación y la estructura en la que se distribuye, cuáles son las herramientas y tecnologías utilizadas durante el desarrollo del sistema.

- **Estado del arte**

En el capítulo 2 se realiza un análisis del estado actual de las herramientas y métodos utilizados. En él hablaremos de cómo ha sido la evolución del procesamiento paralelo desde la perspectiva del hardware en primer lugar y desde su soporte mediante herramientas y nuevos paradigmas en el software, así como en qué consiste el alineamiento de partituras, desde su concepción original hasta las técnicas empleadas en la actualidad, y cómo todo ello puede integrarse en un paquete software para un uso y distribución para los usuarios lo más accesible posible.

- **Propuesta de trabajo**

Tras la descripción del estado en que están las herramientas del apartado anterior, en el capítulo 3 procederemos a desarrollar el plan de trabajo necesario para la consecución del mismo. Aquí describiremos cuáles serían los requisitos formales de la aplicación en primer lugar, qué debe cumplir a corto, medio y largo plazo, así como un análisis DAFO de la misma. A continuación se incluirán todos aquellos detalles necesarios para el desarrollo, tales como el calendario de trabajo, el presupuesto necesario de desarrollo asumiendo que se trata de una aplicación comercial, las tecnologías utilizadas en el proyecto, y detalles de estandarización de cara a establecer una metodología de trabajo determinada que permita el desarrollo continuado de la aplicación con el tiempo.

- **Diseño e implementación**

El capítulo 4 de este texto estará enfocado al diseño de la aplicación en sí, viendo así las diferentes etapas en las que se ha decidido dividir la aplicación y el impacto que esto tiene en aspectos como la modularidad. Además, se incluirá una detallada descripción de cómo funciona la implementación basada en el sistema de creación de paquetes *Autoconf* para así facilitar el entendimiento de la posibilidad que brinda la aplicación para la generación de ejecutables en

distintas plataformas y entornos de ejecución, ya sea en CPU[1], la cual se encarga de ejecutar las instrucciones cargadas desde memoria, para arquitecturas x86 o ARM o incluso una implementación del sistema para GPU[2] basada en la tecnología CUDA[3]. En un contexto normal se trata de un chip encargado de todas las tareas relacionadas con el dibujado de elementos en la pantalla que se conecta a un ordenador, aunque dada su capacidad de cálculo se puede utilizar para la resolución de problemas matemáticos complejos.

- **Pruebas realizadas**

En el capítulo 5 presentaremos el conjunto de pruebas realizadas sobre la plataforma desarrollada en las distintas versiones para ver si los requisitos necesarios se alcanzan y cuál es el balance de carga entre las distintas fases de la aplicación.

- **Conclusiones**

Por último cerraremos este texto en el capítulo 6 analizando los resultados obtenidos tras las pruebas, cómo este trabajo se relaciona con todo lo aprendido durante la carrera y las mejoras o modificaciones que se podrían realizar sobre el sistema en un futuro.

- **Configuración del paquete**

A título orientativo incluiremos un manual de funcionamiento de la aplicación desarrollada en el capítulo 7. Aquí el lector podrá encontrar dónde descargar el paquete desarrollado, junto con cómo debe configurarlo para su uso.

## 1.4 Tecnologías empleadas

A la hora de realizar un desarrollo de software es necesario estudiar y elegir las herramientas necesarias, puesto que permiten orientar el proyecto y el propio esfuerzo en distintas direcciones. En el caso del desarrollo de este proyecto se han realizado múltiples tecnologías, tanto durante en la propia aplicación como en el entorno de desarrollo.

Para la realización de este proyecto se ha debido de seleccionar previamente qué herramientas en cuanto a sistema operativo, donde se ha hecho uso de la distribución de Linux *Debian* junto al *hipervisor VirtualBox*, a fin de poder ofrecer mayor flexibilidad durante las pruebas.

Respecto a las herramientas de gestión y edición de código se ha optado por emplear el sistema de control de versiones *Git* junto con el editor de texto *Vim.*

En lo tocante al lenguaje de programación utilizado se ha optado por utilizar C junto con las funciones que proporcionan las herramientas de desarrollo que proporciona CUDA, así como bibliotecas y entornos de compilación.

Por último, las bibliotecas utilizadas para este proyecto han sido BLAS y *OpenBLAS*, junto con la biblioteca FFTW y el conjunto de herramientas y macros que proporcionan las *Autotools*.

---

[1] Siglas de *Central Processing Unit*. Se encarga de ejecutar las instrucciones que carga desde memoria
[2] Siglas de *Graphic Processing Unit*.
[3] Arquitectura de cálculo acelerado por GPU desarrollada por NVidia

Entrando en detalle, en lo que respecta al entorno de desarrollo se ha optado por utilizar una instalación sobre la propia máquina, evitando entornos virtualizados. El interés de esta manera de trabajar radica en que se tiene un acceso total al hardware de la máquina, lo cual permite probar todas las implementaciones desarrolladas. Sin embargo, de cara a las pruebas de la aplicación se ha optado por la virtualización mediante el cliente *VirtualBox*, el cual permite ejecutar sistemas operativos dentro de otro que puede ser radicalmente diferente. La idea tras el uso de esta herramienta se basa en que la parte de pruebas durante la fase de desarrollo es totalmente independiente de la plataforma sobre la que se ejecuta dicha virtualización. Al contar en los diferentes equipos de desarrollo de una máquina con *Debian* 8 con 4GB de RAM y todas las bibliotecas y dependencias ya resueltas se reduce el tiempo empleado en la configuración de los sistemas de desarrollo, lo que a fin de cuentas termina suponiendo un aumento en la productividad global en el desarrollo.

Otra de las herramientas configuradas en el entorno de desarrollo ha sido el Sistema de Control de Versiones (SCV) *Git*, para manejar todos los elementos del proyecto entre los diferentes equipos, así como para tener una copia de seguridad de la última de las implementaciones desarrolladas, disminuyendo así la posibilidad de pérdida de datos, puesto que dicha copia de seguridad se encuentra alojada en una Raspberry Pi 2 conectada a internet para ofrecer soporte como servidor de contenido. La implementación final de la aplicación se encuentra además alojada en *Github* para poder ser descargada por cualquier usuario. En el capítulo dedicado a la configuración de la aplicación se puede encontrar el enlace de descarga del paquete.

El último elemento dentro de las tecnologías utilizadas dentro del entorno de desarrollo ha sido el editor de código *Vim*. Este editor no sólo es ligero y potente, sino que permite trabajar en cualquier equipo de los utilizados y realizar modificaciones desde cualquiera de ellos. Así, si la implementación desarrollada presenta algún fallo menor puede ser corregido rápidamente en las máquinas de pruebas, puesto que se trata de un editor de texto incorporado en la mayoría de distribuciones de sistemas UNIX.

En lo que respecta a las tecnologías empleadas para el desarrollo en sí podemos diferenciar las que están relacionadas directamente con el desarrollo, como son los lenguajes de programación utilizados, con funciones específicas de biblioteca, o con cómo se estructura y genera el paquete final.

Así, en el lado de los lenguajes de programación utilizados se ha optado por utilizar el lenguaje de programación imperativo C. Las razones para la elección de este lenguaje son fundamentalmente dos.

En primer lugar dado que se parte de una implementación de un simulador ya existente, pero con un alto acoplamiento y con funciones incompletas, desarrollado previamente en C se ha decidido mantener dicho lenguaje frente a otros más modernos como C++, por ejemplo.

El segundo factor viene determinado por las restricciones de tiempo real previamente comentadas. Al necesitar un rendimiento elevado para poder utilizar la aplicación en dispositivos empotrados, como pueden ser teléfonos con Android o placas empotradas tipo Raspberry Pi, se busca desarrollar una aplicación que aproveche los recursos de que dispone de la manera más óptima, y C es el lenguaje idóneo para ello, si bien un lenguaje como C++ podría servir para el mismo propósito.

Por otro lado, parte de la implementación realizada aprovecha las capacidades de la arquitectura CUDA de NVidia, la cual engloba tanto al compilador[4] utilizado, como a un conjunto de herramientas para, mediante una variante del lenguaje C, desarrollar código que ejecutará una GPU.

El uso de esta tecnología permitirá aprovechar los sistemas empotrados que presenten arquitecturas desarrolladas por NVidia, lo que permite ejecutar un programa mucho más rápido que como podría ser utilizando una CPU del mismo sistema, consiguiendo así un alto grado de paralelismo [2].

Para la gestión y creación del paquete de software se ha optado por utilizar el sistema de creación *Autoconf*, así como su herramienta para la generación de *makefiles Automake*. La elección de estas herramientas viene determinada por la plataforma para la que se desarrolla el sistema de pruebas, que es principalmente los sistemas UNIX. Esta herramienta permite que, utilizando un único *script*, se resuelvan todas las dependencias del sistema así como se enlacen todas las bibliotecas que se vayan a utilizar.

Todo el desarrollo de este sistema viene englobado alrededor de generar un ejecutable con esta tecnología, puesto que la necesidad de una herramienta de pruebas de código de fácil uso y compilación es fundamental a fin de estandarizar su utilización lo máximo posible y evitar al usuario la necesidad de enlazar de forma manual múltiples bibliotecas.

En lo que respecta a *Automake*, se utiliza en el desarrollo de esta aplicación para poder simplificar y añadir fácilmente las unidades de compilación[5] que utiliza la aplicación. En secciones posteriores se entrará en detalle sobre cómo se añaden estas unidades de compilación dentro del fichero de *Automake*, pero cabe destacar que el fichero elaborado permite englobar todos los módulos de la aplicación de manera cómoda para enlazarlos a la hora de la compilación.

En lo que hace a las bibliotecas utilizadas para el desarrollo del paquete, encontramos dos tipos diferentes. El primero se trata de bibliotecas algebraicas. Para este desarrollo se ha dado soporte a las más extendidas en sistemas UNIX, que son BLAS[6] y OpenBLAS, las cuales son un conjunto de rutinas que permiten el cálculo de operaciones vectoriales y matriciales de manera eficiente.

El conjunto de subrutinas BLAS está presente en todos los sistemas Unix por defecto, puesto que es la biblioteca *de facto* para resolución de problemas algebraicos.

Por su parte, la biblioteca OpenBLAS es una biblioteca distribuida bajo licencia BSD[7] que consiste en una implementación de código abierto del conjunto de subrutinas BLAS. Está diseñada con optimizaciones para múltiples plataformas y arquitecturas y su interfaz permite enlazar de la misma manera que cualquier programa que utilice la biblioteca original.

Otra de las bibliotecas utilizadas en el desarrollo de la aplicación es FFTW. Se trata de un conjunto de subrutinas para el cálculo de transformadas discretas de Fourier de múltiples tipos

---

[4] Programa que convierte código escrito en un lenguaje de programación a código máquina
[5] Estructura de un archivo de código fuente
[6] Siglas de *Basic Linear Algebra Subprograms.*
[7] Siglas de *Berkeley Software Distribution*. Licencia de software permisiva, similar a licencias como la de *OpenSSL* o la *MIT License*

[3]. Su principal ventaja respecto a otras alternativas de software para el procesamiento de FFTs, algoritmo que se utiliza para calcular la transformada discreta de Fourier de una señal, además de poder permitir la portabilidad del código frente a otras posibles alternativas modificadas para resolver un problema en cuestión.

## 1.5 Convenciones de código

Dentro de los requisitos de este proyecto encontramos fundamental el hecho de mantener unos estándares a la hora de generar código. Para esto se deben desarrollar unas guías de estilo de programación, las cuales deben estar disponibles para cualquier usuario que en el futuro quisiese realizar modificaciones o para el propio equipo de desarrollo original, puesto que ciertos detalles de implementación pueden ser olvidados con el paso del tiempo. Además, deben indicar con detalle la manera sobre la que trabajar con el código que se vaya generando a lo largo del proceso de desarrollo de la aplicación. Así, esto incluye tanto en aspectos de nombrado de archivos, utilización de estructuras o manera de documentar cada una de las funciones desarrolladas, favoreciendo así una mayor mantenibilidad del código desarrollado.

El anexo A incluye las convenciones de código elaboradas y utilizadas para este el desarrollo de este proyecto a fin de mantener todas las cuestiones de diseño lo mejor documentadas posible.

# 2. Estado del arte

## 2.1 Procesadores multinúcleo

Dentro del campo de la informática siempre ha existido una pieza fundamental sin la cual la mayoría de las tareas que en la actualidad realizamos sería imposible. Y es que el procesador, diseñado con la finalidad de ejecutar las instrucciones que lee de memoria para obtener un resultado que presumiblemente volverá a ser guardado en otra sección de la memoria, ha sufrido cambios drásticos [4].

Así, podemos ver cómo la problemática con los procesadores siempre ha estado en la relación sobre el rendimiento obtenido en la ejecución de programas frente al calor disipado, con la consecuente pérdida de rendimiento. El problema aquí lo podemos ver en la correlación que existe entre la frecuencia de reloj como en el consumo de potencia.

Desde los primeros años de la informática se ha buscado obtener el máximo rendimiento posible del sistema que se utilizaba. En el caso de los procesadores tenemos dos planteamientos para resolver el problema del rendimiento. El primero de ellos, más clásico, consiste en aumentar la frecuencia a la que el procesador ejecuta las instrucciones, reduciendo de manera proporcional al aumento el tiempo de ejecución. Esta propuesta, que puede considerarse la más lógica para la evolución del rendimiento de los procesadores, es la que ha marcado la tendencia en el sector informático desde la década de los 80. El problema presente en esta solución lo encontramos en el consumo que presentan los procesadores mononúcleo a los que únicamente se les incrementa la frecuencia efectiva años tras año, lo que supone un aumento en el calor que emiten [5].

La siguiente figura muestra la progresión de la frecuencia de reloj de los procesadores x86 a lo largo de 30 años en relación con la potencia disipada.

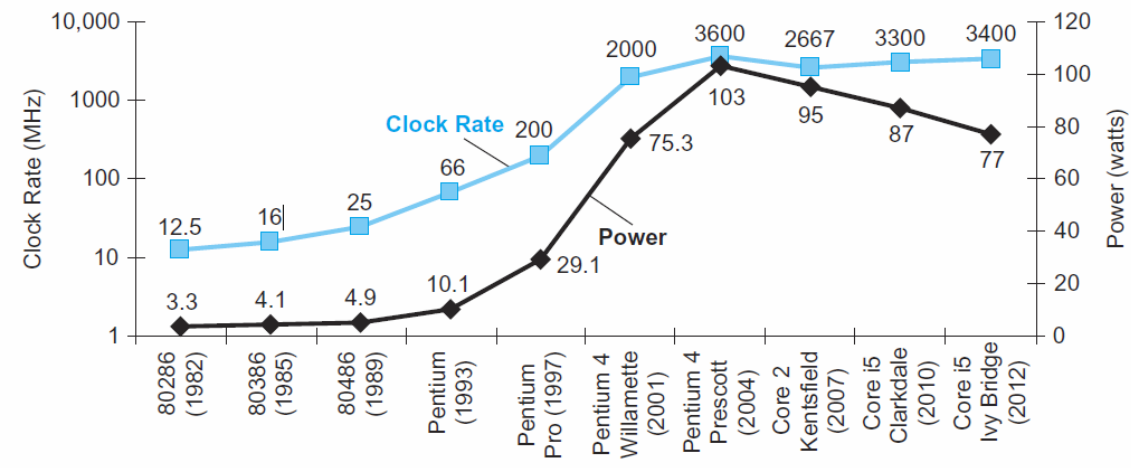



*Figura 1: Frecuencia de reloj y potencia disipada por diferentes procesadores x86 durante 30 años*

Como se puede apreciar, la evolución de los procesadores a lo largo del tiempo ha posibilitado un aumento de la frecuencia a la que trabajan en un factor 1000, aumentando la potencia

disipada en un factor de 30. Esto se debe a que los nuevos diseños van orientados a la reducción del voltaje [5].

De esta manera, el llamado “muro de la potencia” hizo que a principios de siglo se comenzasen a perfilar las primeras soluciones hardware para este problema debido no sólo a la necesidad de reducir el consumo en equipos de escritorio, sino también poder disponer de equipos con una relación potencia/consumo moderada, de manera que puedan funcionar mediante una batería con un tiempo de utilización aceptable. La respuesta a esta problemática vino dada por el desarrollo de microprocesadores con varios procesadores por chip frente a la tendencia de la época de reducir el tiempo de respuesta de un único programa en ejecución. [6] Así, el desarrollo de procesadores multinúcleo se perfila como la alternativa viable ante los problemas de rendimiento y consumo que plantean los procesadores mononúcleo, ya que, si bien no tienen por qué funcionar a mayor frecuencia que una única unidad de proceso, permiten la posibilidad de trabajar en paralelo y realizar múltiples tareas de manera simultánea.

En la actualidad el desarrollo de procesadores multinúcleo se divide en procesadores homogéneos, en los que cada una de las unidades que componen el chip son iguales y buscan la resolución de problemas mediante aproximaciones de tipo divide y vencerás [2], y en procesadores heterogéneos, los cuales son sistemas que utilizan más de un tipo de procesadores especializados para resolver tareas determinadas. Un ejemplo típico en la actualidad de procesador heterogéneo sería la combinación en un mismo System on Chip8, donde la unidad de procesamiento central y la unidad de procesamiento gráfica se combinan dentro del mismo chip para trabajar de manera conjunta [7].

## 2.2 Programación paralela

El cambio de paradigma que suponen los procesadores multinúcleo en cuanto a hardware debe estar acompañado de un cambio en cómo el software maneja estas nuevas arquitecturas de manera eficiente. Así, el desarrollo orientado al aprovechamiento y la resolución en paralelo de problemas se basa en dos propuestas diferentes que vertebran los modelos de programación paralela.

Así, podemos diferenciar claramente entre el modelo de memoria compartida y el modelo de memoria distribuida.

El modelo de memoria compartida se basa en el principio de la compartición de un espacio de direccionamiento común por cada proceso o hilo de ejecución, que puede mantener una zona de memoria privada, involucrado, que realiza operaciones de lectura o escritura de manera asíncrona. Para ello utilizan mecanismos de control de concurrencia como son los *locks* o los semáforos.

[8] Combinación de varias unidades de procesamiento diferentes (como pudiera ser una CPU y una GPU dentro del mismo espacio

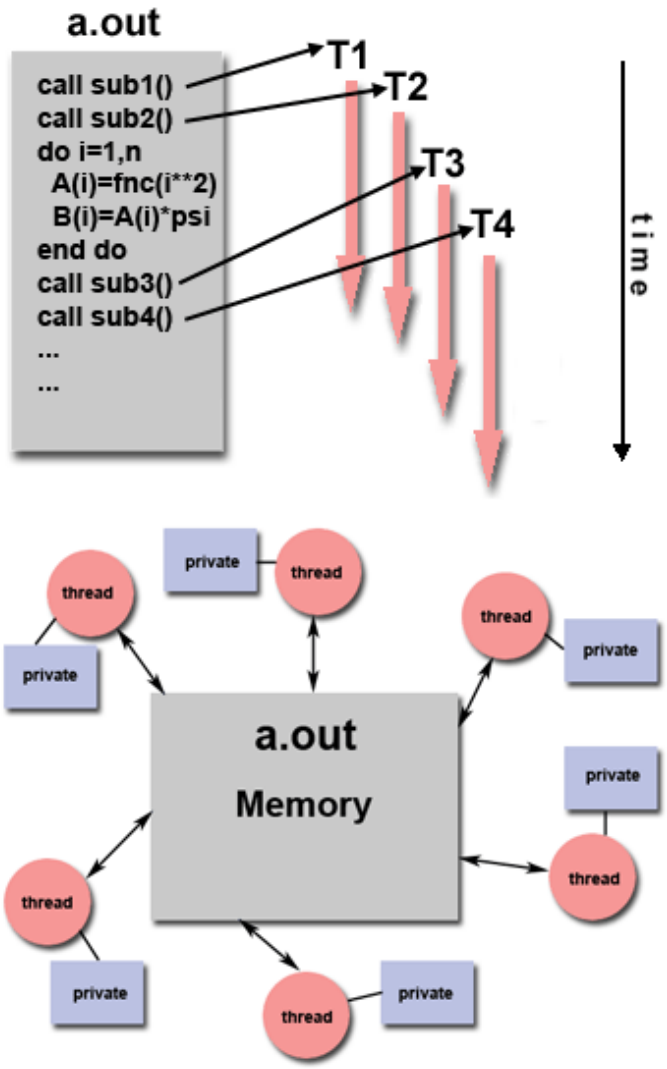



*Figura 2: Modelo de memoria compartida basado en hilos de proceso*

Se trata de un modelo ampliamente extendido gracias a su fácil comprensión, ya que su complejidad reside en conocer cómo ha de funcionar el paralelismo, no en cómo ha de distribuirse la memoria. En la actualidad las principales opciones en cuanto a este modelo de programación paralela son POSIX y OpenMP, si bien existen opciones como la librería de Intel *Threading Building Blocks* [8] para el procesamiento en memoria compartida basado en tareas.

En el caso de OpenMP, se trata de un conjunto de bibliotecas, directivas del compilador y variables de entorno para obtener paralelismo en programas escritos en Fortran, C o C++ [9]. El potencial de OpenMP no reside exclusivamente en sus mecanismos para la concurrencia, sino en el soporte que tiene, puesto que es posible compilar un programa que utilice sus características

Por otro lado, tenemos el paradigma basado en memoria distribuida o de paso de mensajes. Este sistema se caracteriza por estar formado por unidades de proceso independientes, con su memoria local, que se dedican a intercambiar datos mediante el envío o recepción de mensajes. La transferencia de datos en este proceso de paso de mensajes suele requerir la coordinación entre diferentes tareas.

Cuando se habla de memoria distribuida se hace desde un punto de vista de las bibliotecas, puesto que este modelo suele requerir llamada a subrutinas para su funcionamiento.

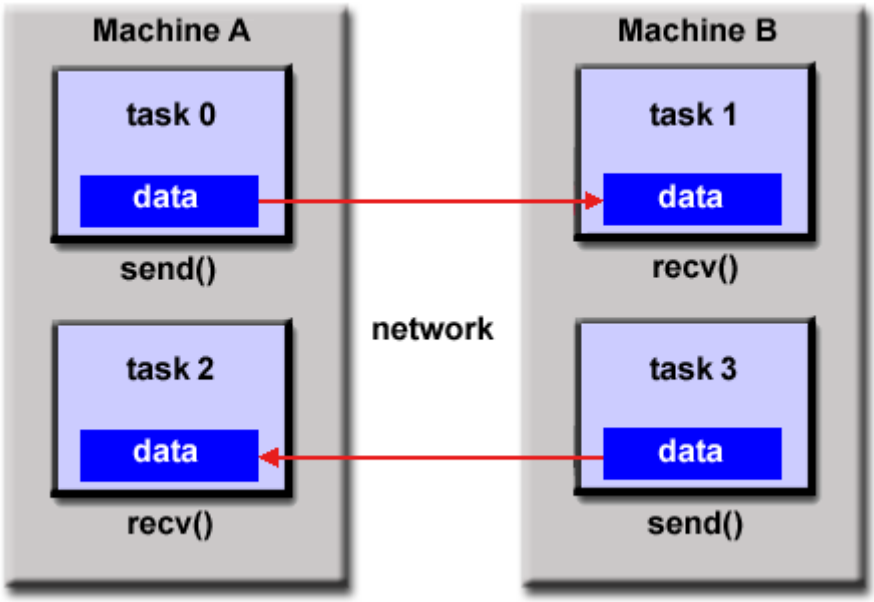



*Figura 3: Modelo de memoria distribuida*

Otra de las posibilidades de la programación paralela es la combinación de estas dos metodologías en lo que se conoce como modelo híbrido, el cual se basa en la utilización de uno de los dos modelos anteriores junto a algún modelo de programación heterogénea como podría ser CUDA u OpenCL[9], o incluso un mecanismo de memoria compartida como OpenMP junto a un mecanismo de memoria distribuida como MPI. La figura siguiente ilustra el último de estos planteamientos.

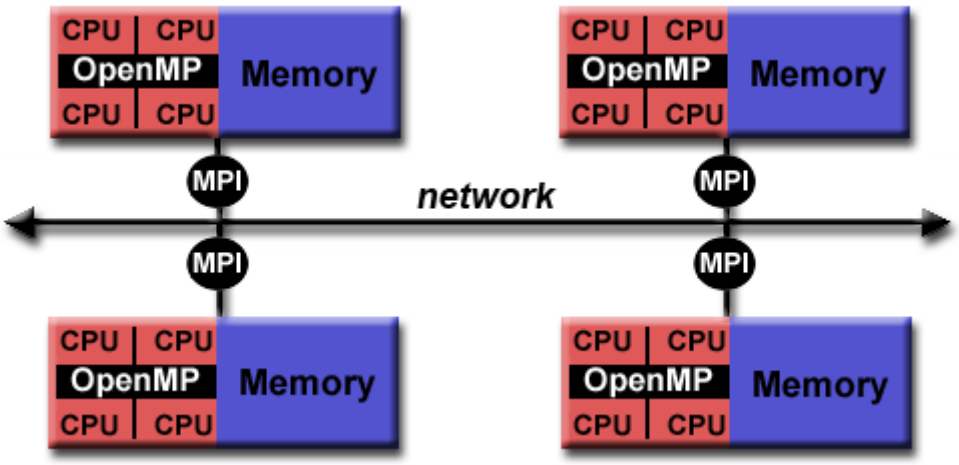



*Figura 4: Modelo híbrido basado en CPU*

[9] Siglas de *Open Computing Language*. Interfaz y lenguaje de programación para el desarrollo de aplicaciones que utilizan el paralelismo a nivel de datos

## 2.3 Alineamiento de partituras

El alineamiento de partituras (*score following*) consiste en la sincronización en tiempo real de un músico tocando una partitura con la partitura en sí misma y la decodificación de los parámetros de ésta para obtener así poder obtener la transcripción de la partitura o incluso la lectura e interpretación mediante la utilización de instrumentos virtuales. De esta manera, se permite la posibilidad de que un sistema pueda detectar las diferentes frecuencias asociadas a cada una de las notas musicales, reconocerlas dentro de una partitura e incluso generar en tiempo real un acompañamiento para la música que detecta.

El desarrollo del alineamiento de partituras se remonta a los años 80 [10]. Durante este periodo, y debido a las restricciones de cálculo de la época, se realizaban muestreos en los que la entrada recibida era monofónica, esto es, en las que sólo existe una única línea melódica, por lo que no encontramos notas simultáneas, como podría ser el caso de acordes, por ejemplo. Además de las restricciones en la capacidad de procesamiento era necesario algún tipo de sistema que incrementase el valor de salida del instrumento para que pudiese ser detectado por la aplicación. La capacidad de procesamiento de la época junto a la estandarización del formato MIDI hizo que se adaptasen los desarrollos para aceptar las entradas en formato MIDI.

El siguiente salto evolutivo en la utilización de esta tecnología estuvo vinculado en la mejora de los procesadores de audio, lo cual hizo aparecer los primeros sistemas de alineamiento basados en el tono. Posteriormente, la década de los 90 introdujo los métodos probabilísticos para el reconocimiento del habla y del audio, tales como los Modelos Ocultos de Markov[10]. Estos sistemas inteligentes tenían en consideración factores nuevos tales como la incertidumbre del rendimiento de la máquina, por lo que el uso de estos métodos probabilísticos permitía garantizar la robustez de la aplicación.

A partir de estas adiciones, las últimas variantes de los sistemas de alineamiento de partituras consisten en el alineamiento anticipado, el cual permite a los músicos realizar acciones de manera síncrona con el acompañamiento durante los conciertos. El paradigma con anticipación permitió el desarrollo de *Antescofo*, sistema de alineamiento de partituras considerado a día de hoy la plataforma estandarizada para gran parte de composiciones.

[10] Modelo estadístico que permite determinar los parámetros desconocidos de un sistema a partir de los observables.

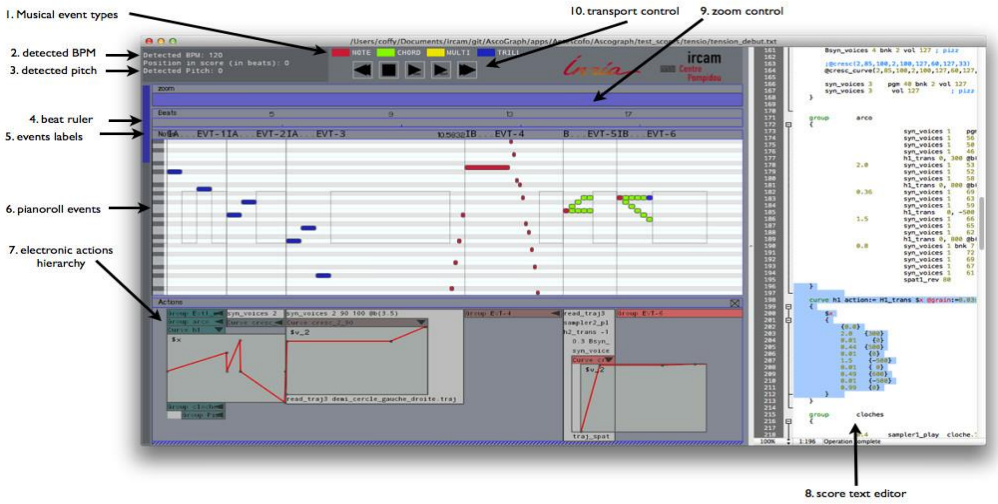



*Figura 5: Interfaz de usuario de la aplicación Antescofo*

La última innovación en estos sistemas consiste en la programación síncrona. El alineamiento de partituras basado en programación síncrona permite introducir técnicas de concurrencia y paralelismo y añadir polifonía, más de una línea melódica al mismo tiempo, a la detección de las partituras [10].

El gráfico siguiente muestra la evolución la que ha pasado el alineamiento de partituras desde su concepción inicial.

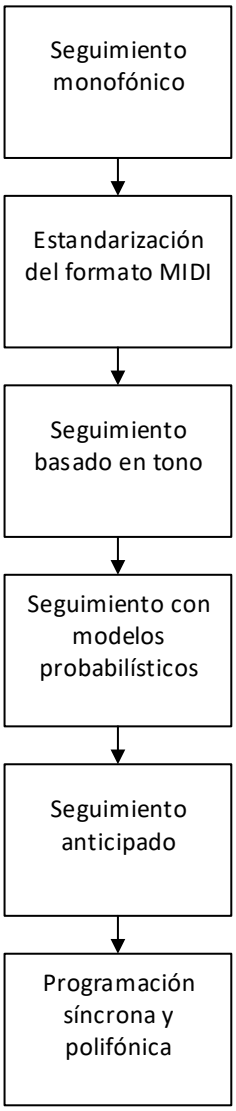



*Figura 6: Evolución del alineamiento de partituras*

## 2.4 Creación de paquetes de software

Dentro del desarrollo de software suele ser normal la generación de los ejecutables de la aplicación de manera automática, a fin evitar que el usuario deba lidiar con todas las posibles dependencias que el programa tener, así como poder extender la distribución de dicho software para que el público objetivo sea el más amplio posible.

Así, dentro de los sistemas Unix encontramos gran variedad de sistemas de automatización y generación de paquetes. Las tareas que estos sistemas están relacionadas con la generación de *scripts*[11] que permitan resolver todas las dependencias anteriormente comentadas, generación de un paquete compilado y realizar pruebas de funcionamiento para comprobar que se ha generado correctamente. Estas herramientas están enfocadas a que los usuarios no tengan que realizar todas estas áreas, que pueden llegar a ser tediosas, manualmente.

[11] Archivo de texto que es leído por un intérprete y ejecuta órdenes para realizar algún tipo de interacción con el sistema

Actualmente dentro de los sistemas de generación de paquetes encontramos gran variedad y con soporte a diferentes tecnologías. A continuación comentaremos la trayectoria de los más extendidos: *CMake* y *Autotools* (*Autoconf*, *Automake*, *Libtool*), si bien existen más opciones tales como Boo*st* o SCons.

El primero a analizar es *CMake*. Se trata de un sistema extensible e independiente del compilador pensado para generar paquetes en diferentes plataformas con una configuración única. Para dicha configuración utiliza archivos de texto llamados *CMakeLists*, los cuales se utilizan para generar archivos de compilación para las diferentes plataformas.

CMake fue creado para responder a la necesidad de una herramienta que permitiese el desarrollo multiplataforma durante el año 2000 para el proyecto *Insight Segmentation and Registration Toolkit (ITK)*, partiendo de la base de intentar adoptar parte de la funcionalidad que presenta la herramienta *configure* [11].

La segunda herramienta que vemos es el conjunto de las *Autotools*, formado por el sistema de generación de paquetes *Autoconf*, el sistema de compilación automática *Automake* y el sistema de creación de bibliotecas *Libtool.* Estas herramientas fueron desarrolladas por el proyecto GNU[12], el cual busca dotar a los usuarios de libertad y control en el uso de sus ordenadores mediante la utilización y desarrollo de herramientas libres. La idea detrás de este sistema reside en la generación de *makefiles*[13] de manera automática para garantizar el funcionamiento en diferentes máquinas.

En 1991, David MacKenzie desarrolló un *Shell script*[14] llamado *configure* que se encargaba de la tarea de ajustar su *makefile* en función de la máquina en cuestión, simplificando el proceso de compilación a ejecutar "*./configure && make*" [12].

Estas herramientas fueron incorporadas como parte del estándar del proyecto GNU y en la actualidad están ampliamente extendidas para ser utilizadas en el desarrollo de gran cantidad de paquetes software, dado que permiten la instalación de software mediante tres simples comandos: "*./configure", "make" "make install*".

*Autoconf* consiste en un paquete de macros M4[15] que se encargan de producir *Shell scripts* que configuran paquetes de código fuente de manera automáticas. Estos *script*s pueden adaptar los paquetes de múltiples formas sin la necesidad de modificación por parte del usuario, si bien éste puede introducir cambios como para aspectos como podrían ser la compilación cruzada, un proceso que consiste en la creación de un ejecutable para una plataforma diferente sobre la que se ejecuta el compilador. La principal ventaja de este paquete es que es autocontenido, por lo que no necesita utilizar

En lo que respecta a *Automake,* se trata de una herramienta que permite generar ficheros *makefile* de manera automática que cumplen con los estándares de código de GNU. El uso de *Automake* requiere en primer lugar la utilización de *Autoconf* y su funcionamiento se basa en ficheros similares a los *makefile* y que pueden contener cierto tipo de variables que se encargan

[12] Siglas de "*GNU is Not UNIX*"
[13] Archivo de texto que contiene un conjunto de directivas que lee el programa de compilación *make*
[14] Archivo de texto que contiene instrucciones a ejecutar por el intérprete de órdenes de UNIX
[15] Procesador de macros incluido en los sistemas UNIX

de definir determinadas reglas de compilación, tales como el nombre de los archivos binarios a compilar o todos los ficheros de código fuente que hacen falta [13].

Por último, el paquete *Libtool* se trata de un *script* que permite el soporte de bibliotecas compartidas mediante la utilización de una interfaz adaptable y transparente al usuario. La idea detrás de *Libtool* es conseguir modularidad y código reutilizable en los programas de GNU, lo cual exige que la manera en la que las bibliotecas son generadas, puesto que no existe un procedimiento estándar a la hora de crear bibliotecas compartidas [14].

# 3. Propuesta de trabajo

## 3.1 Requisitos del proyecto

A la hora de afrontar el desarrollo de esta aplicación debemos tener en cuenta todas las limitaciones y requisitos que debe cumplir para su correcto funcionamiento, puesto que es necesario adaptarse a estas condiciones a fin de desarrollar software que cumpla con las expectativas de los clientes. Dado que esta aplicación es un proyecto de final de titulación asumimos que el cliente en cuestión es la entidad que gestiona el título en este caso, que es la Escuela Técnica Superior de Ingeniería Informática y para el caso que nos ocupa los requisitos del proyecto asignado son las siguientes:

- El desarrollo debe estar enfocado a la obtención de un simulador que procese ficheros equivalentes a las partituras obtenidas mediante instrumentos MIDI. Para el caso de la aplicación desarrollada se devuelven los tiempos asociados a cada una de las etapas importantes dentro de la aplicación, ya que durante este proyecto se busca obtener una optimización correcta de la aplicación. Sin embargo sería sencillo modelar la aplicación de manera que devolviese un vector con el camino mínimo asociado a los estados alineados por los que se ha pasado en la partitura, como podría ser en el caso de querer hacer un seguimiento de la partitura o un acompañamiento musical.
- Por otro lado, dicho simulador debe funcionar de manera óptima en una amplia variedad de entornos y configuraciones. Así, la aplicación debe ejecutarse utilizando la CPU en entornos Unix bajo las arquitecturas x86 y ARM, mediante ciertas modificaciones sobre el código original. Además, se añade la posibilidad de ejecutar una versión optimizada para el cálculo mediante GPU de dicho simulador. Para dicho funcionamiento será necesario que el equipo sobre el que se quiere ejecutar la versión de GPU cuente con un procesador de NVidia[16] con capacidad de ejecutar programas que utilicen la tecnología CUDA, de la cual hablaremos más adelante. Además, se propone también la utilización de dispositivos Android para hacer uso de ambas tecnologías, en caso de disponer de esa posibilidad, o de utilizar exclusivamente la implementación para CPU del simulador. La compilación del código requerirá cierto tipo de herramientas y/o de tecnologías que detallaremos más adelante, pero al margen de eso se trata de un proceso transparente para el usuario de la aplicación, el cual debe indicar únicamente si quiere compilar para para una versión diferente de CPU (tanto en arquitectura x86 o ARM) y el soporte de la precisión de los números de coma flotante a utilizar.
- Otro factor importante está vinculado a las restricciones de tiempo real de la aplicación. Dado que el funcionamiento de los algoritmos está orientado a la sincronización con instrumentos utilizados por músicos o a la generación de música mediante instrumentos virtuales la aplicación debe ser capaz de procesar cada una de las tramas en un tiempo

[16] Empresa dedicada en el desarrollo de unidades de procesamiento gráfico

de promedio de 12.8ms de acuerdo al funcionamiento del tratamiento de las tramas, el cual detallaremos más adelante.

- El último de los requisitos que debe cumplir la aplicación está relacionado con el diseño modular que debe tener la aplicación. Así, el interés de este simulador está en la capacidad de adaptarse con facilidad a cambios como podría ser la ejecución de funciones de bibliotecas externas o el tratamiento independiente de cada una de las etapas. Así, se busca que el usuario pueda indicar a la hora de generar un ejecutable para la aplicación qué bibliotecas utilizar, dentro siempre de una lista de posibilidades, o que cualquier desarrollador pueda realizar cambios en cierta parte de la aplicación sin afectar al resto de elementos de la misma, buscando reducir los tiempos de desarrollo de nuevos elementos de la aplicación y un acoplamiento bajo entre las diferentes unidades de compilación.

## 3.2 Análisis DAFO

Analizar los requisitos de una aplicación nos permite ver cómo puede ser su situación en el momento del despliegue de la misma. A la hora de desarrollar una especificación sobre las características que tiene cualquier aplicación es importante analizar el entorno en el que se encuentra, utilizado para ello una visión tanto interna como externa a la misma. De esta manera podremos ver en qué situación se encuentra la aplicación frente a otras similares y cómo poder buscar el valor diferenciador frente al resto.

Desde la perspectiva más externa de la aplicación debemos centrarnos en la situación del mercado, y la competencia que podemos encontrar, así como las posibles tendencias del mercado que se pueden aprovechar para potenciar la aplicación. Así pues, si nos centramos en el tipo de aplicaciones similares presentes en el mercado vemos, como hemos comentado anteriormente, algunas con una larga trayectoria dentro del este campo, como puede ser el caso de Antescofo o Tonara [15]. Estas aplicaciones tienen una gran utilización debido tanto al tiempo que llevan en mercado, lo cual les ha permitido evolucionar en gran medida, como en su facilidad de uso, dado que en el caso de Tonara al tratarse de una aplicación para dispositivos móviles, está enfocada para cualquier usuario no técnico que tenga conocimientos musicales. En el caso del sistema que se ha desarrollado para este proyecto se propone una alternativa similar a la que ofrece Antescofo, puesto que se podría buscar la integración con otras aplicaciones y entornos de composición musical como *PureData*, si bien ofrecerles a los equipos de desarrollo una plataforma modular sobre la que puedan trabajar, modificar e implementar algoritmos y optimizaciones realizadas para comprobar su correcto funcionamiento puede ser un sector del mercado interesante de cara a la explotación de la aplicación.

Sin embargo, desde una perspectiva empresarial interna debemos ver cómo es la capacidad de producción, de organización y del personal encargado del desarrollo de este sistema. Así, en lo que se refiere a las capacidades organizativas y productivas, podemos ver que al contar con un único desarrollador la innovación tecnológica será limitada, puesto que las herramientas más modernas requerirían de mayor tiempo de desarrollo e investigación y ello supondría un mayor coste de un presupuesto que de por sí ya es ajustado. Aquí las alternativas de Tonara y Antescofo destacan enormemente, puesto que ambas disponen de equipos de desarrollo grandes y de presupuestos varios órdenes de magnitud superiores a los disponibles para este paquete. Sin embargo, el proceso de control sí sería conocido puesto que un único desarrollador debería

desarrollar todo el plan de trabajo, por lo que conocería en detalle desde la especificación de los requisitos hasta qué módulos deben ser probados para comprobar el correcto funcionamiento del paquete desarrollado de cara a su publicación en un repositorio online. Por otro lado, en lo que respecta al personal encargado del desarrollo, el hecho de contar con una única persona limita la capacidad y complejidad en que se puede realizar el proyecto.

Por último, y una vez vistas todas estas características, en lo que respecta a la estrategia a tomar para este proyecto se podría optar por una aproximación debería ser de supervivencia, puesto que la situación respecto a otras empresas del mismo sector es inferior y la situación interna del equipo de desarrollo está muy limitada en términos financieros y de innovación, a pesar de que la aplicación pueda tener un público claro y que pueda presentar un interés en su uso.

Como último aspecto de este análisis, se adjunta la matriz DAFO asociada al desarrollo de las fortalezas internas y externas de este proyecto:

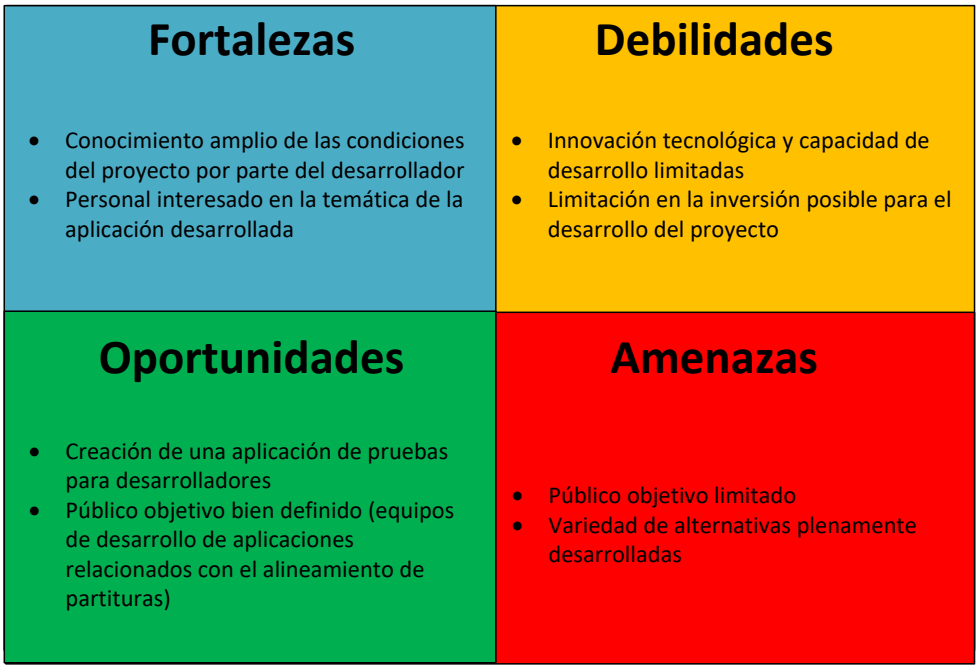



*Figura 7: Matriz DAFO asociada al desarrollo de este proyecto*

## 3.3 Plan de trabajo

### 3.3.1 Calendario de trabajo

El desarrollo de software debe ser una actividad que, como cualquier otro proyecto de ingeniería, debe disponer de un plan de trabajo claro y realista para poder estudiar la viabilidad del proyecto con posibles inversores y para poder establecer las fases necesarias para poder terminar con éxito. Así, a continuación presentamos un plan de trabajo propuesto al inicio del proyecto separando las fases lo máximo posible y realizando la mayor parte del trabajo de investigación durante el primer mes de desarrollo. Además, el diseño de la aplicación se desarrolla desde el principio de manera modular para que la posterior implementación sea mucho más rápida. Además, la realización durante el segundo mes de la implementación del simulador permitiría dedicar todo un mes al análisis de la herramienta *Autoconf* y *Automake* para así poder

generar un paquete instalable del simulador totalmente funcional y compatible con la especificación, dejando casi un mes completo a la redacción final y a la corrección de errores dentro del programa. Esta planificación permitía además añadir funcionalidades nuevas con casi dos meses de margen en caso de que la especificación cambiase, ya que la complejidad de añadir elementos nuevos dentro del paquete funcional es menor, puesto que el resto de elementos están desacoplados del módulo nuevo. La siguiente figura muestra el diagrama de Gantt asociado a la planificación original.

Sin embargo, el plan original se ha visto modificado por diferentes factores que han retra-

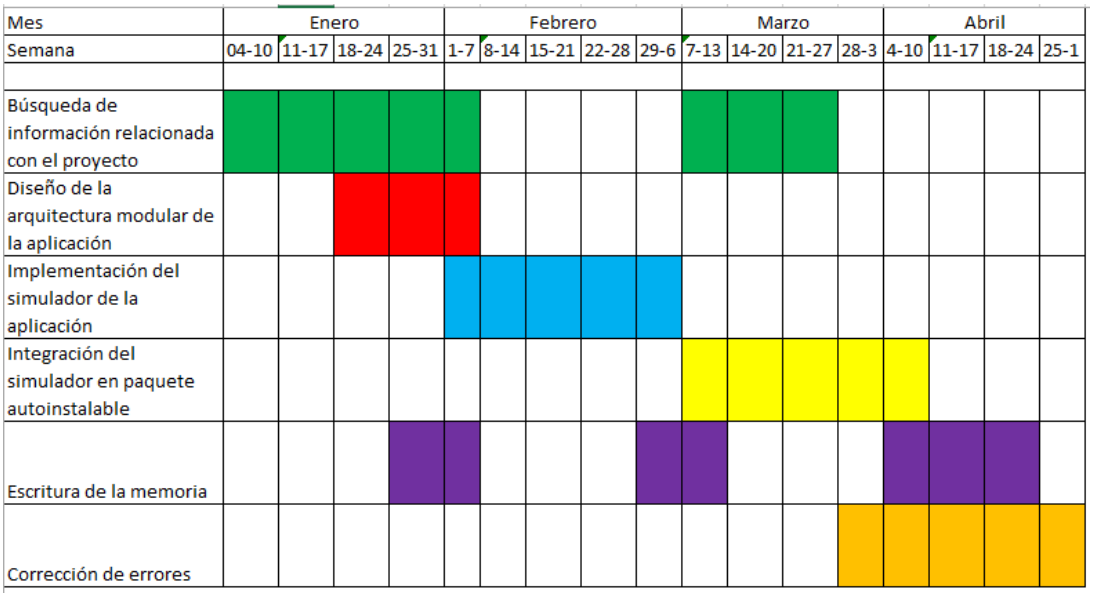



*Figura 8: Calendario de trabajo propuesto al inicio*

sado el desarrollo de manera considerable. El primer problema lo encontramos en la búsqueda de información relacionada con las herramientas utilizadas. En primer lugar el uso de las herramientas *Autoconf* y *Automake* es mucho más complejo de lo que parece. Posteriormente entraremos a desarrollar el sistema de funcionamiento de estas herramientas para clarificar el porqué de los problemas encontrados, pero estas herramientas han retrasado el desarrollo prácticamente un mes, con el consiguiente retraso en el calendario propuesto originalmente.

Otro de los problemas ha estado relacionado con la entrega de trabajos dentro de las últimas asignaturas de la carrera, lo que ha desplazado el inicio del proyecto un mes completo.

El siguiente problema encontrado durante el desarrollo de la aplicación ha estado relacionado con las versiones de los algoritmos. Puesto que en este trabajo se trabaja de manera conjunta con la Universidad de Jaén y la de Gijón, los cambios que realizaban en aspectos como la representación de las matrices, la implementación de las estructuras utilizadas, el cambio en el funcionamiento de algunos algoritmos o el soporte de nuevas características no previstas durante el inicio han hecho que se haya tenido que rediseñar tantas veces como cambios se han tenido sobre los algoritmos originales, puesto que la máxima de este proyecto es crear una aplicación lo más desacoplada entre sus elementos dentro de lo posible.

El último problema encontrado está relacionado con la implementación, dado que al buscar el acoplamiento mínimo dentro de la aplicación aparecen ciertos problemas de indirección que,

al reimplementar los nuevos algoritmos dentro del sistema llevaban a corrupción de datos e incluso a violaciones de segmento en más de una ocasión, por lo que más de una vez ha sido necesario parar el desarrollo para utilizar el depurador de GNU y arreglar los problemas.

Todos estos problemas han reducido tiempo para la corrección de errores, contando con poco más de una semana para solucionarlos. En la figura siguiente se puede ver cómo es el calendario de trabajo final.

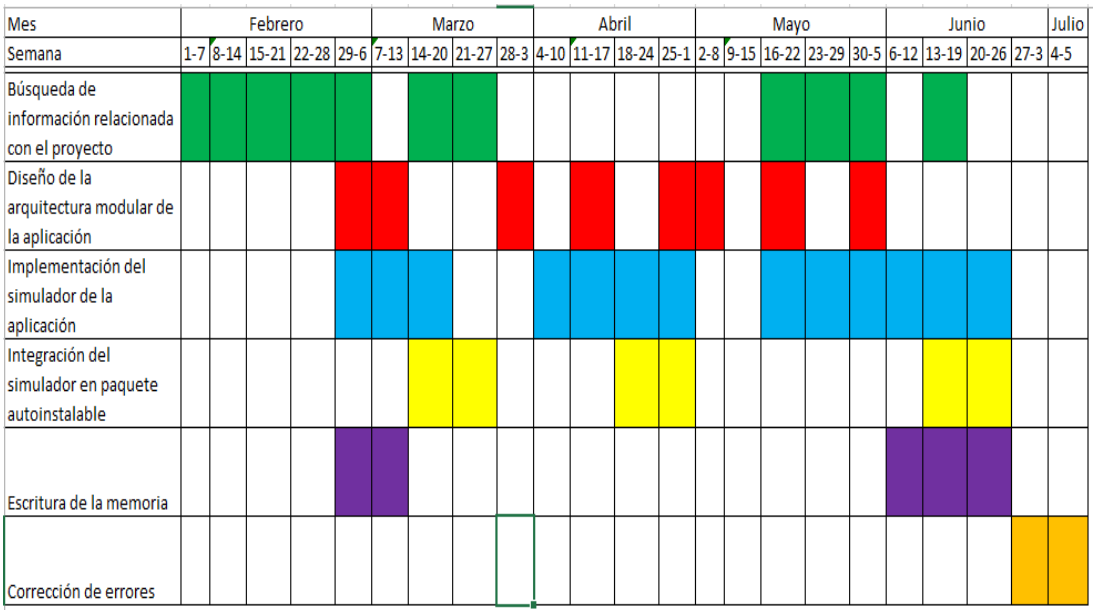



*Figura 9: Calendario de trabajo real*

### 3.3.2 Presupuesto del proyecto

Uno de los aspectos interesantes durante la realización de un proyecto es una estimación de los costes asociados al mismo, puesto que todo desarrollo real requiere de una financiación y unos recursos que dependen en gran medida del dinero, y es el factor diferenciante entre la realización de un proyecto o su cancelación por falta de fondos.

Así, para este proyecto asumiremos costes fijos como el equipo de desarrollo y alguno de los equipos de pruebas utilizados y los costes variables como podría ser el alquiler de una oficina con todos los costes asociados a la misma.

En lo que respecta a los costes fijos lo separamos en ordenador de escritorio para trabajo diario y ordenador portátil para trabajo descentralizado. El equipo de escritorio para el trabajo habitual es un ordenador montado a piezas y actualizados a medida que iba siendo necesario y dispone de las siguientes características principales:

- Procesador Intel Core i5-3450 a una velocidad de 3.1GHz con modo Turbo a 3.5GHz
- Memoria RAM DDR3 de 24 GB de capacidad
- Tarjeta gráfica NVidia GeForce GTX 970 de 4GB de capacidad y 1664 núcleos CUDA.
- Sistema operativo Debian 8

Por su parte, el equipo portátil es un ordenador de cuatro años de antigüedad que permite el desarrollo de la mayor parte de la aplicación y cuyas características principales son:

- Procesador Intel Core i5-2430m a una velocidad de 2.4GHz con modo Turbo a 3GHz
- Memoria RAM DDR3 de 8GB de capacidad
- Tarjeta gráfica Intel HD 3000 con 256MB de memoria compartida
- Sistema operativo Arch Linux 4.8.4-1

El equipo de escritorio, sumado a las pantallas utilizadas y demás periféricos, tiene un coste asociado de unos 1250€, mientras que el ordenador portátil cuesta 500€. Así, el presupuesto asociado a equipos de desarrollo asciende a 1750€.

Además, debemos considerar también el coste del dispositivo de pruebas Raspberry Pi 2 Model B, con un coste de unos 41€ [16] y las siguientes características básicas:

- Procesador ARM Cortex-A7 a una velocidad de 900MHz
- Memoria RAM DDR2 de 1GB de capacidad
- Chip gráfico VideoCore IV 3D con 128MB de memoria compartida
- Sistema operativo Raspbian

Por otro lado, se dispone, además, de acceso remoto a equipos cedidos por el Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia para las pruebas de rendimiento de la aplicación. Estos equipos incluyen un kit de desarrollo NVidia *Jetson* con las siguientes características principales:

- Procesador ARM Cortex-A15 a una velocidad de hasta 2.2GHz
- Memoria RAM DDR3 de 2GB de capacidad
- Chip gráfico NVidia Kepler con 256 núcleos CUDA

El otro equipo del que se dispone consiste en un equipo con gran capacidad de procesamiento para aplicaciones GPGPU[17] con las siguientes características.

- Procesador Intel Core i7-3820 a una velocidad de 3.6GHz con modo Turbo a 3.8GHz
- Memoria RAM DDR3 de 16GB de capacidad
- Tarjeta gráfica (para visualización) ATi Radeon X1550 de 1 GB capacidad
- 2 tarjetas gráficas (para procesamiento) NVidia Tesla K20c de 5GB de capacidad y 2496 núcleos CUDA
- Sistema operativo Red Hat 4.4.7-11

En lo que respecta al software al utilizar Linux se ha optado por utilizar software libre durante todo el proceso de desarrollo, por lo que el coste asociado al software es de 0€. En el apartado 3.3.4 entraremos en detalle sobre qué herramientas y tecnologías se han utilizado,

[17] Siglas de *General Purpose Graphic Processing Unit*. Consiste en la utilización de un procesador gráfico para realizar operaciones matemáticas complejas

pero era una consideración necesaria el aclarar que los no existe coste relativo al software utilizado.

Respecto a los gastos variables se ha decidido buscar una oficina en un edificio de Co-Working [17] por 15€ al mes a fin de poder reducir al máximo posible los costes variables y poder sacar adelante este proyecto con un presupuesto económico. Así, y de acuerdo con el calendario de trabajo propuesto, el coste de la oficina sería de unos 75€, puesto que además de los meses propuestos en el calendario de desarrollo se debe pagar una fianza el primer mes para cubrir gastos. Todos los costes de mantenimiento asociados al proyecto están ya incluidos en la mensualidad del alquiler.

El último coste que debemos considerar es el del salario del programador. Para este desarrollo se cuenta únicamente con un programador que realiza desde la fase de especificación de requisitos hasta la fase de pruebas. Suponiendo que se trata del caso de un recién graduado su salario, por convenio [18], sería de unos 1050€ al mes, por lo que la suma total durante todo el desarrollo alcanzaría los 4200€

La siguiente tabla muestra el coste requerido para el desarrollo del proyecto según el plan de trabajo propuesto al inicio del mismo:

| | |
|---|---|
| Coste de material | 1791€ |
| Coste de alquiler | 75€ |
| Coste de mantenimiento de la oficina | Incluido |
| Coste del personal | 4200€ |
| **Total** | **6066€** |

*Tabla 1: Desglose del presupuesto original del proyecto*

Sin embargo, como se ha visto en la fase final del desarrollo la duración del proyecto ha sido mayor que la propuesta en un primer momento. Como se ha visto en el apartado anterior el tiempo propuesto para el desarrollo de este trabajo era de 4 meses, pero el resultado final ha sido de 5 meses. Esto supone tener que pagar un mes más de alquiler de oficinas y de salario del programador, mientras que como no ha habido incidencias con los equipos de desarrollo y pruebas no se ha tenido que sumar ningún coste extra de material.

La siguiente tabla muestra el presupuesto con las modificaciones temporales que corresponden al resultado final del desarrollo, junto a la diferencia de coste respecto a la planificación original.

| | |
|---|---|
| Coste de material | 1791€ |
| Coste de alquiler | 90€ |
| Coste de mantenimiento de la oficina | Incluido |
| Coste del personal | 5250€ |
| **Total** | **7136€** |
| | |
| Total proyectado | 6066€ |
| Diferencia de costes | 1070€ |
| **Desviación presupuestaria** | **17.64%** |

*Tabla 2: Desglose del presupuesto final y desviación respecto al original*

# 4. Diseño e implementación

## 4.1 Diseño de la solución

Si entramos a analizar cómo hemos desarrollado este proyecto podemos ver varios factores relevantes para su funcionamiento que condicionan de manera fundamental cómo se concibe la aplicación.

Así, en lo que respecta al diseño hemos de analizar la aproximación que se ha tomado en el diseño de la aplicación. Esto nos permitirá ver cómo el diseño offline facilita la implementación para los dispositivos objetivos y permite compartimentar los procesos que sigue la aplicación.

Posteriormente veremos las etapas involucradas en la aplicación, y cómo pueden ampliarse con facilidad gracias a la modularidad que existe entre ellas. Para cada etapa se explicará las funciones que realiza, así como las posibles estructuras que puede utilizar y los accesos a otros elementos externos a la aplicación que haga.

Dentro de estas etapas se han utilizado varios algoritmos y soluciones de diseño que merecen la pena destacar de forma separada. De esta manera, veremos cómo funciona el algoritmo de *Dynamic Time Warping (DTW)* en el sistema, así como la distribución de datos en el tiempo y posterior utilización o la forma en que se gestionan las matrices de datos necesarias.

Por último veremos cómo este diseño resuelve ciertos problemas presentes en la aplicación de la que se parte y las diferencias más significativas respecto a las estructuras empleadas y forma de ejecutarse.

El planteamiento del que se parte al desarrollar esta aplicación consiste en un banco de pruebas para sistemas de alineamiento de partituras. Este sistema se caracteriza por permitir integrar las soluciones algorítmicas necesarias en el sistema para poder probarlas y así, según el rendimiento que se obtenga, integrarlas en una posible aplicación real o realizar modificaciones para probar nuevas soluciones para el problema que trata, el alineamiento de partituras con requisitos de tiempo real para su uso en dispositivos empotrados como tablets o teléfonos móviles.

El funcionamiento de la aplicación es el siguiente: El usuario introduce un fichero de configuración que contiene archivos binarios que representan muestras de audio de una partitura interpretada por un instrumento musical. Así, la aplicación recoge esta información y comienza a realizar un alineamiento sobre qué tipo de notas deberían sonar en el instrumento que se está interpretando, si bien también se podría pasar un fichero de configuración que estableciese un alineamiento entre la melodía interpretada por un músico, con la posible desviación que pudiese existir entre lo que el músico interpreta y lo que el instrumento hace sonar, y la partitura que se interpreta, generando una señal de audio intermedia que corrige la posible variación en el tono y determinar qué nota es la que se está haciendo sonar.

El simulador desarrollado parte de una aplicación con unas características base similares, pero una aproximación a la resolución del problema diferente de cara a su despliegue. Por lo que respecta a este proyecto, se ha desarrollado un simulador cuyo funcionamiento se limita al ámbito local. Es decir, la aplicación se instala y se ejecuta dentro de la misma máquina. Esto es

lo que se ha denominado un simulador para *DTW monolítico*, puesto que se busca la ejecución en una máquina local de los algoritmos desarrollados.

Cabe destacar que la concepción monolítica del simulador no debe confundirse con el tipo de implementación de los mecanismos de *DTW*, donde encontramos tanto una ejecución *offline* como *online,* los cuales hacen referencia a la manera de procesar los datos que hace la aplicación.

Una aproximación *offline* es la utilizada para el desarrollo del simulador, basada en la interpretación de archivos que representan la interpretación de partituras durante un tiempo determinado, que en el caso analizado van desde los 30 segundos de composición hasta la media hora. Este tipo de aproximación se presenta como la más eficiente, puesto que no requiere de ningún otro mecanismo para la obtención y extracción de características de la partitura. La relevancia de este tipo de implementación es útil para ciertas herramientas de recuperación de música.

Por su parte, la ejecución *online* es la que está constreñida por aspectos de tiempo real para ofrecer soporte en sistemas en los que la información es conocida a medida que pasa el tiempo. Este es el tipo de aproximación empleada en aplicaciones que realizan un paso de página automático o un acompañamiento musical, entre otros usos. Además, el uso de un modelo *online* de este paquete es el que plantea mayor interés de cara a su uso en dispositivos móviles [19].

Dentro de la capa de lógica desarrollada se ha analizado en profundidad el flujo de la aplicación original con el fin de determinar posibles dependencias y relaciones entre las diferentes funciones que realiza el programa para poder diseñar un flujo basado en etapas. En el apartado siguiente entraremos a desarrollar en detalle el flujo del programa, pero cabe destacar el diseño de las etapas desarrolladas.

El simulador está separado actualmente en cuatro etapas principales: Lectura de datos, gestión de memoria, precálculo e iniciación del proceso y proceso *DTW.* A continuación detallaremos las funciones principales de cada una de las etapas.

La etapa de lectura de datos se encarga de gestionar la entrada de datos de los ficheros de configuración que se pasan a la aplicación para así poder delimitar aspectos como el tipo de partitura, el tipo de instrumento, el tamaño de muestras, el número de estados a desarrollar por el algoritmo o el tamaño de tramas, entre otros aspectos. Todos los datos leídos desde fichero son leídos y cargados en *structs* para su posterior manejo en toda la aplicación.

La etapa de gestión de memoria realiza todas las operaciones relacionadas con la reserva (*malloc*) y liberación de memoria (*free*) de todos los vectores y estructuras de los que la aplicación hace uso. Esta etapa es una de las más básicas, puesto que se encarga de realizar la reserva de memoria de todos los vectores y estructuras al principio para liberarlos justo antes de cerrar la aplicación.

La tercera etapa se encarga del precálculo e iniciación del cálculo para la etapa de *DTW.* Así, en esta etapa se realizan operaciones como la aplicación de una *ventana de Hanning*, que se trata de una operación para reducir los efectos laterales asociados al posterior cálculo de una *FFT* con un número de ciclos no entero, de manera que se reduce la amplitud de las discontinuidades en los extremos de la secuencia obtenida [20].

Posteriormente en esta etapa se calcula la *FFT* asociada a la trama que se está tratando para obtener un valor normalizado al que aplicarle un tipo de distorsión según la función $1 - e^{(-1 \times \|v(i) \times norma\|)}$ , siendo "*v*" el vector de distorsión sobre el que trabajar posteriormente.

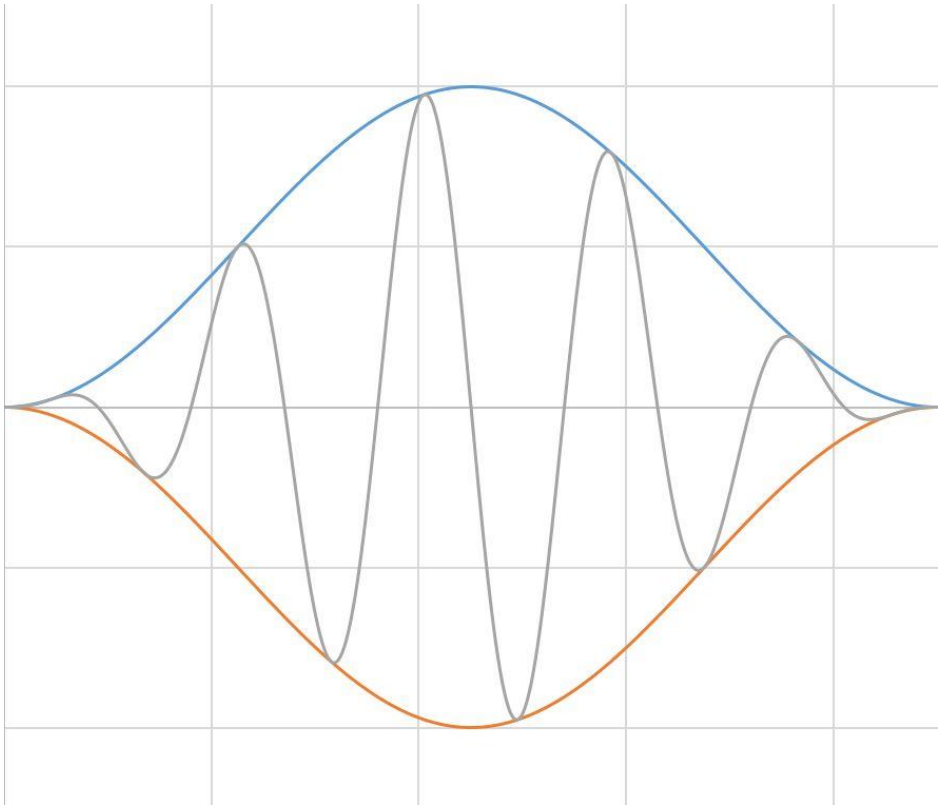


*Figura 10: Aplicación de una ventana sobre una señal*

La última etapa desarrollada se encarga de aplicar una implementación del algoritmo *DTW* sobre los vectores. Antes de explicar cómo funciona es interesante mostrar los vectores más relevantes involucrados en el proceso y qué utilidad tienen para resolver este problema.

- pS: Vector que almacena el número de saltos en la partitura
- pD: Vector que almacena la distancia entre diferentes estados
- pV: Vector que almacena el cociente entre la distancia entre estados y los saltos
- costes: Vector que almacena el coste de cada estado.

Una vez analizado el comportamiento asociado a los principales vectores de la aplicación podemos entrar en detalle sobre el algoritmo empleado. El algoritmo *DTW* desarrollado inicialmente para el reconocimiento del habla. Su objetivo principal es alinear dos secuencias de vectores de características combinando el eje temporal hasta encontrar una coincidencia óptima entre ambas secuencias.

Este algoritmo garantiza la no repetición de características durante el alineamiento, la no omisión de características importantes o una consideración total de ambas secuencias entre otros aspectos [21].

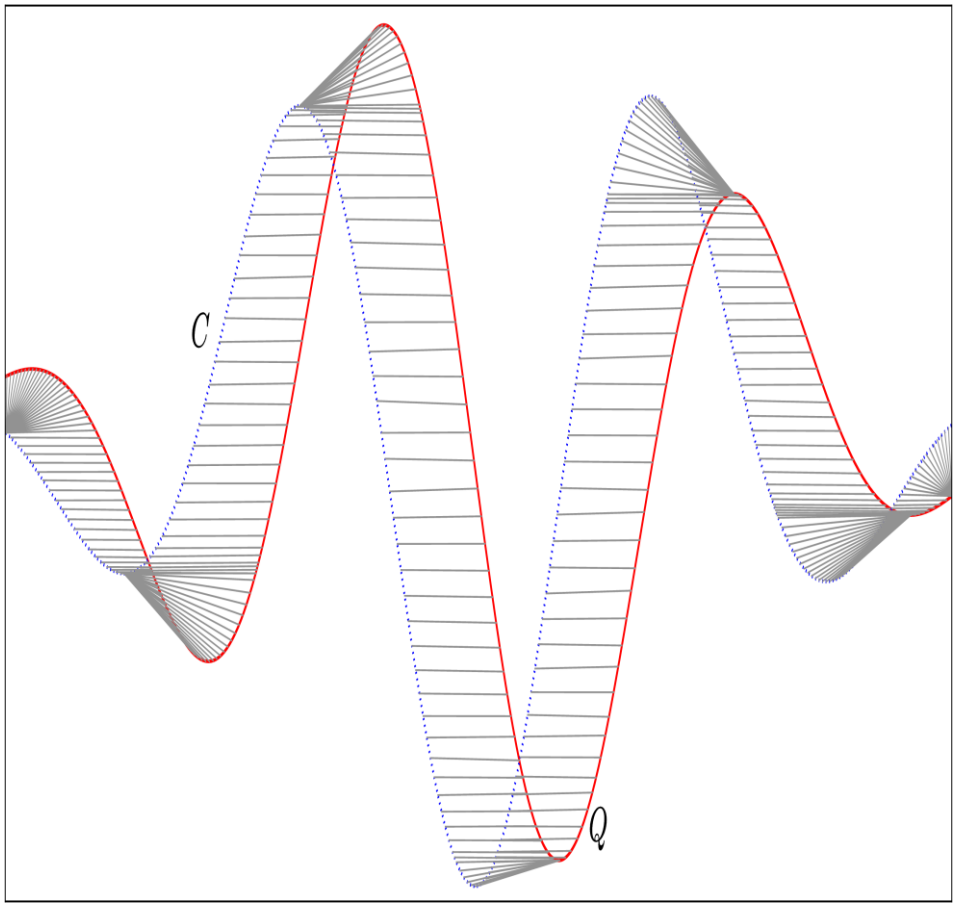



*Figura 11: Alineamiento de dos señales*

La figura siguiente muestra el diseño de la aplicación en función de las etapas que se han establecido.

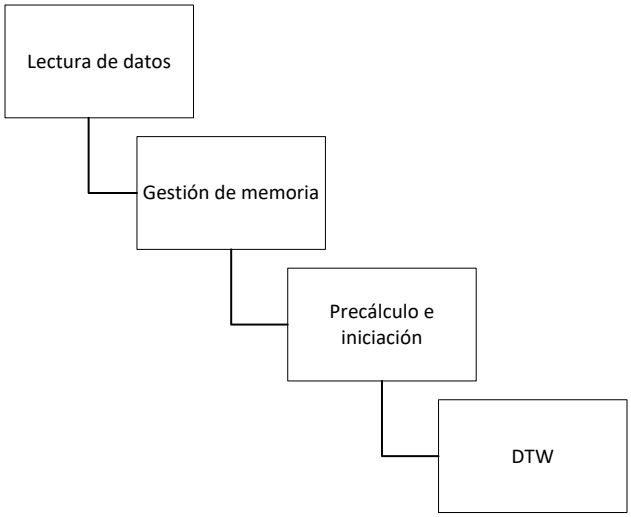



*Figura 12: Evolución del proceso de alineamiento de partituras*

Como último aspecto importante en el diseño de la aplicación se debe considerar cómo afectan a las arquitecturas el tratamiento de las diferentes tramas que se deben procesar. Esto

nos lleva a dos aspectos clave: la distribución de datos a lo largo del tiempo y a un concepto de distribución espacial de los valores de una matriz.

El primero de los aspectos viene dado por la restricción de tiempo real que se busca y por la cantidad global de datos a procesar. Dado que cada una de las tramas que se procesan se forma como conjuntos de 5700 datos, los cuales equivalen a 12,8ms de música, la carga en el sistema debe ser consciente de los estados previos. Esto se debe a que cada una de las tramas que entran en el sistema están subdivididas en 4 muestras y la primera trama que entra en el sistema tiene un tiempo de proceso superior, ya que supone cargar 4 muestras nuevas, lo cual tiene un coste computacional de 12,8ms x 4 = 51,2ms. Así, una vez cargada dicha trama el resto de tramas deberían ser capaces de llegar cambiando la muestra que corresponda cada 12,8ms. El siguiente gráfico ilustra cómo evolucionan las tramas durante los primeros instantes de funcionamiento de la aplicación y cómo se forman las tramas que se procesan.

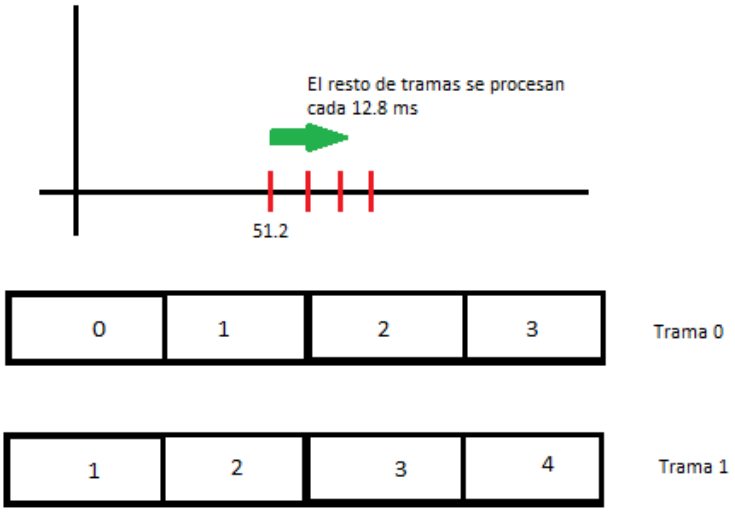



*Figura 13: Tiempo de cálculo de las tramas y evolución de las mismas*

El otro aspecto a considerar durante el diseño del simulador es cómo se contemplan las tramas a lo largo del tiempo. Es en este aspecto donde se introduce el concepto de *padding*, donde la matriz que contiene la distancia entre saltos guarda los valores asociados a 4 instantes previos. Así, al mantener 4 instantes actuales y 4 instantes previos el alineamiento es más preciso. La figura siguiente muestra cómo se reparte el vector pD al introducir el uso de *padding.*

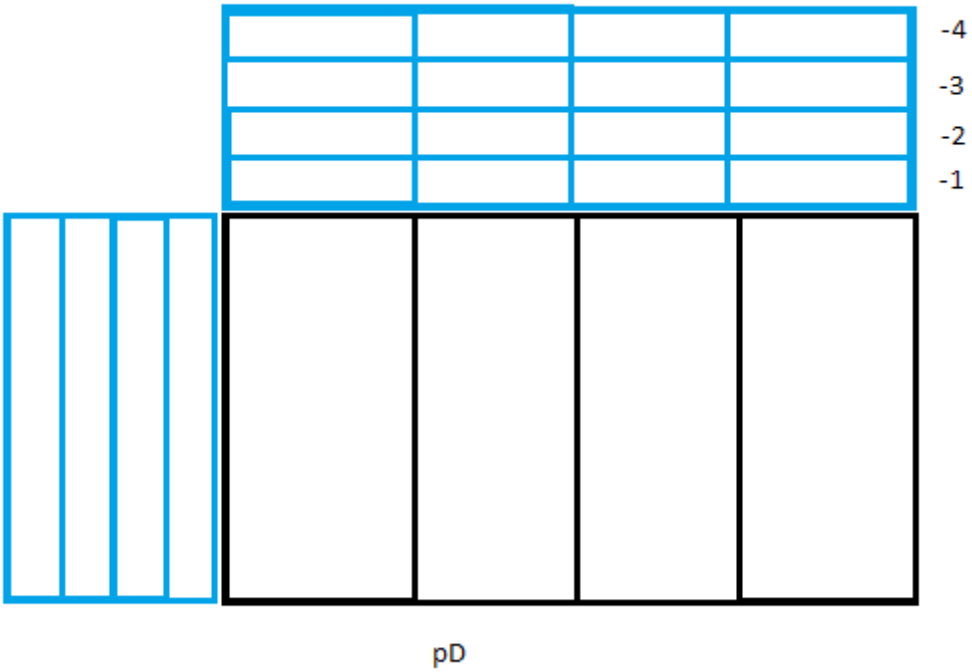



*Figura 14: Padding para la representación de los estados en el tiempo del vector pD*

Como se puede ver, se busca que el diseño planteado para la aplicación sea robusto, preciso, modular y consiga en la medida de lo posible el requisito de tiempo real necesario para el alineamiento. En el siguiente apartado entraremos a analizar a nivel de lógica de negocio cómo todo esto se desarrolla en el sistema.

## 4.2 Estructura del proyecto

Una vez visto el diseño de la solución debemos ver cómo afecta a la implementación del sistema. En este apartado veremos cómo se diseña el paquete que forma el *testbench*, desde lo que ve el usuario hasta el enlazado con las funciones de biblioteca utilizadas.

Posteriormente entraremos en ver cómo se ha desarrollado cada uno de los módulos, observando la interfaz de programación establecida para facilitar a los usuarios su manejo en otras aplicaciones si fuera necesario. Dentro de cada módulo veremos el flujo de ejecución que tiene para, finalmente, poder analizar el flujo de la aplicación en su totalidad.

El tercer aspecto a destacar en los detalles de implementación estará en cómo el sistema ofrece modularidad en la utilización de bibliotecas para ciertos aspectos, y cómo se pueden añadir cambios en bibliotecas o bibliotecas nuevas sin requerir que el usuario deba modificar el grueso de la aplicación.

Finalmente veremos cómo es el ciclo de vida de las estructuras de la aplicación y sentaremos las bases de una posible alternativa al diseño planteado y desarrollado.

Desde el punto de vista estructural, cada uno de los módulos diseñados está desarrollado como unidades de compilación independientes. Así, se puede modificar un aspecto concreto de la aplicación sin la necesidad de tener que recompilarla en su totalidad. Esto permite fundamentalmente dos cosas. La primera de ellas es permitir el mantenimiento de la aplicación de cara a

futuras ampliaciones, dado que se puede ampliar con nuevas características, como pudiera ser un módulo de entrada de audio mediante micrófono, nuevos algoritmos para el alineamiento, reestructuración de la gestión de memoria mediante alguna biblioteca de terceros, etc.

El otro aspecto que permite resolver este diseño de la lógica de negocio es el diseño de una interfaz simple para que cualquier usuario que desarrolle su aplicación utilizando este *testbench* tenga ocultación de código en las partes relevantes a la vez que mantiene toda la flexibilidad que permite el lenguaje de programación.

Así, cada uno de los módulos desarrollados presenta un modelo de funciones donde se busca el encapsulamiento [18] de las funciones. Si bien este concepto es propio los lenguajes orientados a objetos, en el lenguaje C podemos simular un comportamiento similar mediante el uso de la palabra reservada *static*. Si dentro de una unidad de compilación declaramos las funciones como estáticas estamos limitando el alcance de dichas funciones con lo que se conoce como *enlazado interno*[19], haciendo que sólo sean visibles dentro de ese mismo archivo. De esta manera ocultamos al usuario funciones que realizan algún tipo de cálculo complejo para darle un acceso exclusivamente al conjunto de funciones mediante una única llamada. La siguiente figura ilustra este comportamiento mediante la interfaz que presenta el módulo de gestión de datos, donde se puede apreciar qué permite la función de acceso público respecto al resto de funciones del módulo.

```
int Config_DTW_data( DTW_const_st *DTW_const, DTW_files_st *DTW_files,
DTW_validation_files_st *DTW_verify, char *file_name, FILE *fp );
int Write_vector_in_text_file_d( char *file_name, precission_type *v, const int
size );
int Write_vector_in_text_file_i( char *file_name, int *v, const int size );
int DTW_set_vectors( DTW_files_st *DTW_files, DTW_const_st *DTW_const );
void Read_frame( FILE *fp, precission_type *frame, const int FRAME_SIZE );
int Compare_output( const precission_type *v, const char *file_name, const int
size );

static int Read_vector( const char *file_name, precission_type *vector, const int
size );
static int Read_s_fk( precission_type *s_fk, const int N_MIDI, const int N_BASES,
const char *file_name );
static int Read_vector_int_64( char *file_name, int *vector, int size );
static int Read_int64_vectors( DTW_files_st *DTW_files, DTW_const_st *DTW_const
);
static int Read_structure( DTW_const_st *DTW_const, char *file_name, FILE *fp );
static int Read_data( DTW_files_st *data, DTW_validation_files_st *verify_data,
FILE *fp );
```

*Figura 15: Interfaz de llamadas a función del módulo DTWIO*

Entrando en detalle sobre la interfaz de llamadas a función, a continuación analizaremos todos los módulos presentes dentro de la aplicación, y cómo estos ofrecen por un lado soporte a diferentes arquitecturas, en el caso de la aplicación desarrollada arquitecturas x86 y ARM para

[18] Ocultamiento del estado de un miembro de manera que sólo se pueda cambiar mediante operaciones definidas
[19] Forma de referenciar sólo a componentes dentro del ámbito de la unidad de compilación

cálculo por CPU y mediante GPU, y por otro cómo posibilitan el uso de diferentes bibliotecas de manera flexible.

El diagrama siguiente muestra el diseño de los diferentes módulos que forman la aplicación. En él podemos ver cómo para cada una de las etapas diseñadas se ha desarrollado una unidad de compilación independiente al resto y sobre ellas gobierna un módulo lanzador que es al que se llama desde el programa principal. Además, todos estos módulos utilizan de forma global un fichero de cabecera que contiene las definiciones de las estructuras y vectores utilizados en la aplicación.

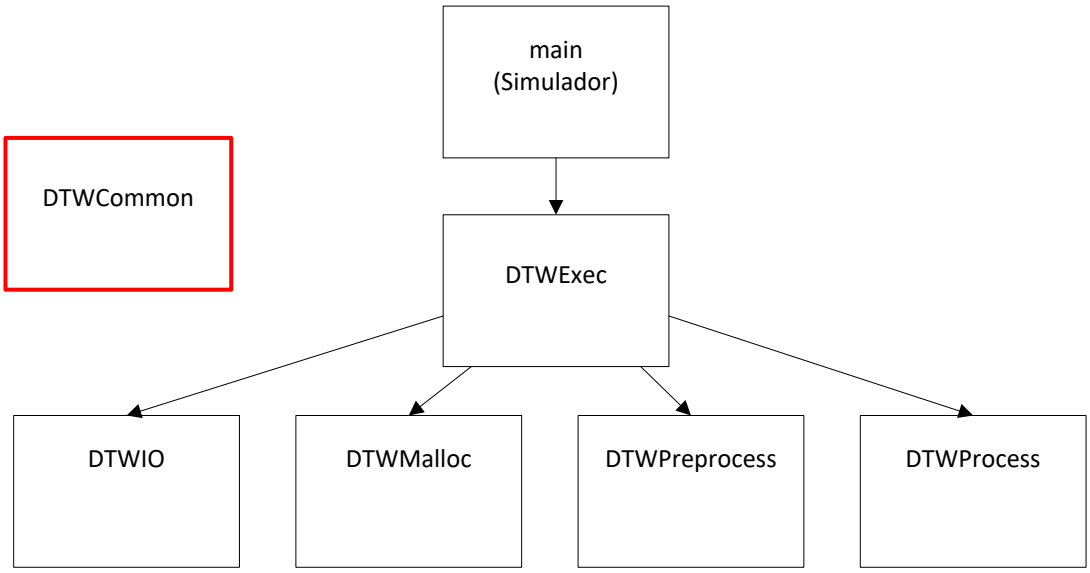



*Figura 16: Módulos desarrollados del simulador de alineamiento de partituras*

Si vemos cada uno de los módulos desarrollados, la ejecución de la aplicación comienza en el archivo *main.c*, el cual se dedica exclusivamente a enlazar en tiempo de compilación con el programa lanzador para la arquitectura objetivo. Posteriormente entraremos a analizar los parámetros que hacen variar la forma de compilación, pero cabe destacar que mediante directivas de preprocesador se enlaza al módulo adecuado. Cada uno de los módulos, con independencia de la arquitectura objetivo, presenta una interfaz genérica en el lanzador. Puesto que dicho concepto es propio de los lenguajes de programación orientados a objetos, el uso de directivas de preprocesador permite definir en cada uno de los módulos una llamada a función con el mismo nombre y misma interfaz para así, en función del archivo enlazado durante la compilación, ejecutar el programa de manera transparente para el usuario.

La figura siguiente ilustra este funcionamiento del programa mediante la definición genérica de la función *DTWExec*, la cual es común a todas las arquitecturas del programa.

```
#if CPU
#include "../launcher/DTWExec_CPU.h"
#elif GPU
#include "../launcher/DTWExec_GPU.h"
#endif


int main( int argc, char *argv[] )
{
	DTWExec( argc, argv );
	return 0;
}
```

*Figura 17: Definición de funciones genéricas en función de la cabecera*

Así pues, el siguiente módulo que se ejecuta, como parte de la aplicación es el simulador de la aplicación para la arquitectura determinada. En este documento haremos mención principalmente a la versión para CPU desarrollada, puesto que el flujo de ejecución es el mismo para ambas versiones. En el caso de la implementación para GPU se comentarán los cambios importantes en los módulos que correspondan.

El módulo del simulador se encarga de realizar la ejecución del alineamiento de partituras utilizando el resto de módulos. Así, se trata del único módulo del programa cuya interfaz es totalmente accesible por el usuario, debido a que se verifica que el resto de módulos funcionan adecuadamente y devuelve el tiempo de ejecución total de la aplicación. El flujo de funcionamiento del simulador consiste en las llamadas a las funciones visibles del resto de módulos, siguiendo el orden de llamadas a módulos del diagrama siguiente.

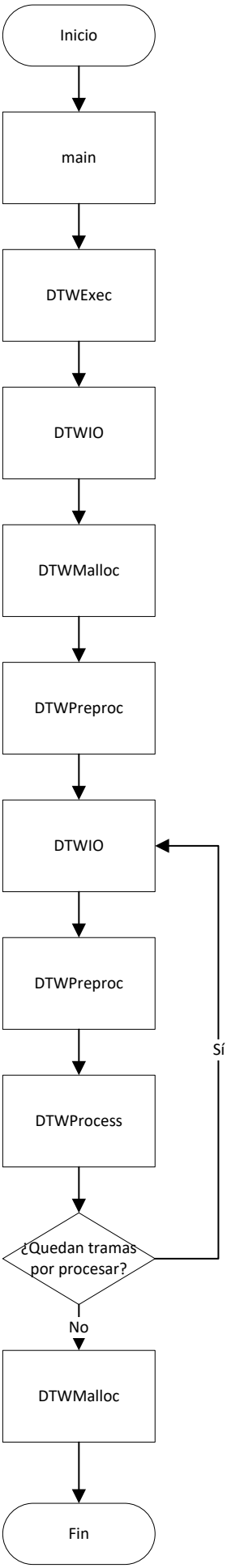

Inicio
main
DTWExec
DTWIO
DTWMalloc
DTWPreproc
DTWIO
DTWPreproc
DTWProcess
¿Quedan tramas por procesar?
Sí
No
DTWMalloc
Fin

*Figura 18: Diagrama de flujo del simulador*

Tras la ejecución tantas veces como tramas se tengan en la partitura se obtienen los tiempos asociados al cálculo total, al tiempo por trama, a la aplicación de la *FFT,* a la aplicación de la distorsión y al proceso *DTW*.

En lo que respecta al resto de módulos, haremos su análisis viendo las funciones accesibles desde cualquier punto del programa para luego entrar en detalle sobre cuál es la función que realizan dentro del simulador.

El primer módulo con esta estructura en iniciar su ejecución es el de lectura de datos. En él encontramos en la parte visible del programa una función de configuración de los datos de la aplicación, junto con otra función de lectura de los vectores necesarios para la aplicación.

Durante la configuración del sistema se leen datos desde un archivo *dat* en el que se leen el número de muestras MIDI que hay, lo cual indica el tipo de instrumento para el que se simula el alineamiento, el número de bases que hay en la partitura para poder separar en estados posteriormente, el número de estados que existe en toda la partitura, junto con el número de tramas. Además, para cada una de las tramas también se debe leer el tamaño que tienen, junto con el tamaño de las muestras que forman cada trama.

Por otro lado, el módulo debe leer el tamaño que tendrá la matriz que se utilizará para realizar la *FFT* sobre el sistema. Por motivos de eficiencia el tamaño de la matriz siempre deberá ser una potencia de 2. Los dos últimos parámetros a leer por el módulo son el número de costes por dimensión para cada uno de los vectores y una constante (alfa) para poder utilizar durante la aplicación de las distorsiones de los vectores.

La siguiente tabla muestra los valores por defecto que la aplicación utiliza para cada uno de los valores, siendo estos de acuerdo a una experimentación de valores de simulación para los dispositivos objetivo.

| | |
|---|---|
| Número de muestras MIDI | 114 |
| Tamaño de la trama | 5700 |
| Tamaño de cada muestra | 5700 |
| Tamaño de la matriz para la FFT | 16384 |
| Número de costes por dimensión | 4 |

*Tabla 3: Variables utilizadas con valor predefinido y su valor*

Los valores previamente leídos desde fichero son almacenados en una estructura denominada *DTW_const*, la cual es utilizada a lo largo de toda la aplicación para simplificar el acceso a las variables comunes de tamaño y cantidad de muestras.

Por otro lado, la aplicación también lee diferentes archivos binarios determinados en el mismo fichero *dat,* con la diferencia de que en este caso se hace referencia a las matrices que contienen los valores asociados a la partitura, a los vectores de distorsión y a los conjuntos de estados.

Como aspecto final, se ha dejado acceso global a la función que permite leer tramas desde archivo, puesto que la lectura se debe hacer continuamente cada vez que una trama es procesada.

En la parte privada del módulo encontramos funciones de lectura de vectores genéricos para la mayoría de vectores y una función para la lectura del vector asociado a la distorsión de la trama, en la cual a diferencia de la lectura estándar se debe tener en consideración el tamaño de muestras leídas, que debe ser de $bases \times midi$. La escritura de vectores en un fichero es un aspecto considerado inicialmente pero descartado finalmente, ya que no era relevante para el análisis que se realiza en esta aplicación.

El siguiente módulo que veremos será el relacionado con la gestión de memoria de la aplicación. La parte accesible por el usuario permite únicamente realizar la reserva (*malloc*) y posterior liberación (*free*) globales de memoria del sistema, evitando así que el usuario deba conocer cada una de las estructuras involucradas en el proceso, el tamaño en memoria que requieren y las bibliotecas que utilizan. Con la utilización de las llamadas *DTW_alloc* y *DTW_dealloc* el usuario estaría gestionando todas las matrices, vectores y estructuras de datos implicadas.

El ámbito privado del módulo incluye únicamente las reservas de memoria de cada una de las variables, además de una función para la memoria utilizada por la matriz que se utiliza para realizar la *FFT*, la cual depende de una biblioteca utilizada para conseguir realizar las operaciones necesarias. Para eso la función de gestión de memoria para la *FFT* tiene definida dentro de su cuerpo una macro con la intención de realizar una compilación condicional de las funciones según la biblioteca utilizada. Como en el caso elaborado la biblioteca para este tipo de operaciones ha sido *FFTW* se ha definido únicamente una llamada a función para esta biblioteca pero, como comentaremos más en detalle en la siguiente sección de este documento, la modificación para cualquier tipo de biblioteca diferente sería bastante trivial.

Cabe destacar también cómo la versión para GPU gestiona la memoria. El simulador desarrollado para esta versión distingue si el tipo de memoria es unificada y, en función del parámetro de entrada que se le pase a la llamada al simulador, se reserva la memoria siguiendo este modelo o el modelo de memoria clásico.

El siguiente módulo dentro del simulador es el que realiza las tareas de precálculo e iniciación del proceso. Este módulo es el más complejo de todos los elaborados y el que más tareas realiza y el que más tiempo de cálculo utiliza, puesto que es donde se encuentran las operaciones de distorsión, que son las más costosas dentro del proceso de alineamiento.

En lo que respecta a las funciones accesibles de este módulo podemos distinguir entre la de inicio de proceso de alineamiento y la de preproceso del sistema.

El inicio del alineamiento es una llamada que se realiza una única vez en la que se calcula el vector auxiliar para la distorsión de las tramas junto con el cálculo de las normas de dicho vector. La razón de esto es una cuestión de eficiencia, ya que se trata de guardar valores en cache para así tener un acceso más rápido que si hubiese que calcularlo por cada una de las tramas procesadas.

Por otro lado, el preproceso del sistema consiste en realizar una serie de operaciones para cada una de las tramas que lee el sistema. Para cada una de ellas aplica una ventana de Hanning para suavizar los límites de la *FFT* y así reducir el error por la longitud de las ondas. Posteriormente se aplica la *DFT* sobre la trama que se está procesando. Este punto de la aplicación depende de bibliotecas de terceros y, de la misma manera que se realizan operaciones de gestión de memoria se llama a una función externa, en la implementación realizada de la biblioteca

*FFTW*, para crear y aplicar un plan de resolución y obtener la solución de la operación. En el mismo caso que con la gestión de memoria, la aplicación es modificable para que no dependa estrechamente de una biblioteca y, llegado el momento, pudiera realizarse un cambio prácticamente transparente para el usuario.

Una vez realizada la *FFT* sobre la trama que se está procesando se realiza el cálculo del vector de distorsión. En esta etapa se utiliza la implementación de las subrutinas de BLAS para acelerar el proceso de los vectores. Dado que las pruebas realizadas han sido compiladas mediante *gcc*[20], sólo se ha podido probar el soporte que se ofrece para estas subrutinas, si bien podría existir compatibilidad con el conjunto *mkl* [21] de Intel, el cual aceleraría el tiempo de ejecución.

El diagrama siguiente muestra el flujo de ejecución que presenta el preprocesado de la señal para cada una de las tramas tratadas.

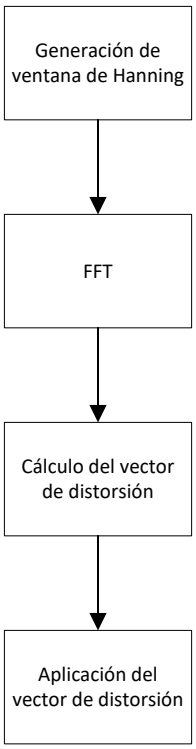



*Figura 19: Flujo de ejecución del preproceso*

Es tanto en esta etapa como en la de proceso de la señal donde aparecen las diferencias de implementación entre la versión para CPU y la de GPU. Mientras la primera sigue el funcionamiento descrito anteriormente, el simulador para GPU utiliza llamadas a *kernels* para la resolución del problema.

El último módulo desarrollado durante este proyecto consiste en el que se encarga de realizar el proceso de alineamiento mediante el algoritmo *DTW* para poder encontrar el camino de menor coste entre las dos señales obtenidas.

En lo que respecta  al diseño de las funciones accesibles al usuario podemos ver como únicamente existe una función *DTW_process* que, dada una trama a la que se le ha aplicado una

[20] Compilador desarrollado por el proyecto GNU
[21] Biblioteca de alto rendimiento desarrollada por Intel para cálculo de funciones matemáticas

distorsión aplica el algoritmo de alineamiento para obtener un valor mínimo de la posición entre la señal real y la ideal.

En primer lugar se hace una búsqueda sencilla del camino mínimo sobre el vector de distorsión y, a partir del valor obtenido se procede a ejecutar el algoritmo para obtener, esta vez sí, una posición mínima real para alinear las señales del sistema.

En el caso de la implementación para GPU, y de igual manera a como pasaba con el preproceso mediante GPU, se lanza un *kernel* que ejecuta el algoritmo por bloques para su resolución.

## 4.3 Optimizaciones realizadas

Una vez visto cómo se ha desarrollado la aplicación podemos ver qué tipo de optimizaciones se han utilizado para intentar obtener ese requisito de tiempo real que establecimos previamente.

En primer lugar veremos los buffers utilizados en la aplicación, y cómo éstos permiten reducir los costes de tiempo y memoria utilizada según el caso.

Por último, veremos las soluciones propuestas para la obtención de código genérico y adaptable a los diferentes sistemas objetivo con la mayor sencillez posible, así como el funcionamiento del sistema de gestión de paquetes para poder compilar de manera correcta.

Dado que el simulador desarrollado parte de una implementación basado en una arquitectura radicalmente diferente a lo propuesto a lo largo de este texto, las primeras versiones incluían un diseño basado en un buffer circular, el cual permite ahorrar el espacio utilizado y permite un posicionamiento correcto en operaciones simples, las características de este simulador, y de este tipo de aplicación, hacen que no sea viable tratar ciertas cuestiones, como pudiera ser el paralelismo a nivel de datos mediante una arquitectura *SIMD* , entre otras cuestiones.

Así, la optimización del diseño pasa por desarrollar un buffer que, si bien evita frontalmente el uso de un buffer circular, no propone tampoco hacer un consumo de memoria directo, puesto que los dispositivos objetivo pueden no tener tal cantidad de memoria *RAM* disponible. La idea pasa por utilizar un buffer de tamaño ampliado respecto a la aproximación original pero con operaciones de movimiento de bloques de memoria. Esta aproximación lineal busca, a costa de aumentar el tamaño ocupado en memoria, aumentar el rendimiento y el grado de paralelismo que se puede conseguir. Por contra, parte de la implementación ha mantenido también los buffers circulares para ver cómo afecta al rendimiento este cambio.

El otro aspecto sobre el que se ha buscado optimizar ha sido en la genericidad del código de cara al uso de tipos básicos. Para esta aplicación, y como se ha comentado en repetidas ocasiones, el objetivo son los dispositivos móviles como teléfonos y tablets, se ha decidido crear tipos de datos genéricos que, en tiempo de compilación, se modifiquen para satisfacer las necesidades de la aplicación.

Este aspecto se ha tratado principalmente en el uso de números de coma flotante. En lugar de usar un tipo como *float* o *double* por defecto, se ha optado por definir una macro denominada *precission_type*. Así, a la hora de compilar se puede establecer si *precission_type* es de precisión simple o doble en función de la macro *SIMPLE_PRECISSION*. La siguiente figura muestra esta declaración en el fichero de datos común.

```
#if SIMPLE_PRECISSION
	#define precission_type float
```

*Figura 20: Definición del tipo precission_type*

Sin embargo, en lo que respecta al tipo de número en la aplicación, el uso de un tipo de precisión u otro condiciona cierto tipo de operaciones. Es por ello que a lo largo de toda la aplicación se analiza si el tipo de datos de la aplicación es de precisión simple o doble para realizar llamadas a función determinadas o declarar tipos compatibles. Esto lo podemos ver en el uso de la biblioteca *FFTW*, donde en función de la precisión empleamos los tipos *fftw* para precisión doble o *fftwf* para precisión simple.

La figura siguiente muestra este aspecto declarando una macro en función del tipo de precisión.

```
#if SIMPLE_PRECISSION
	#define precission_type float
#if FFTW
	#define FFTW_plan_type fftwf_plan
#endif
#else
	#define precission_type    double
#if FFTW
	#define FFTW_plan_type fftw_plan
#endif
#endif
```

*Figura 21: Definición de tipos en función de precission_type*

El último aspecto a destacar en esta implementación es la integración de todo el paquete con el sistema de gestión de paquetes *Autoconf*. Este sistema permite definir un *script* que permita detectar las características de la máquina y realizar una instalación de la versión compatible con ese equipo, si bien se puede forzar la compilación de una determinada versión, interesante para casos de compilación cruzada, siempre que se definan a mano los símbolos que se utilizan.

El simulador desarrollado presenta varios prerrequisitos a fin de hacerlo funcionar correctamente. Entre ellos tenemos la necesidad de tener instalado el paquete *fftw3* y las subrutinas de *BLAS/LAPACK*. Si el usuario desea compilar para GPU necesita tener instalado el entorno de desarrollo de *NVidia* para *CUDA,* además de un dispositivo compatible con esta tecnología. Como posibilidad, el paquete también busca una instalación de *Doxygen* para poder generar la documentación del proyecto, pero la ausencia de este componente no limita el funcionamiento del simulador.

El flujo de funcionamiento de Autoconf determina en primer lugar la plataforma sobre la que se ejecuta el *script* de configuración, con lo cual puede ver si se trata de un entorno *Linux* bajo arquitectura x86_64 o se trata de un entorno para procesadores *ARM.* Además, si el usuario lo especifica se puede buscar directamente la compilación para GPU, puesto que mediante la variable de entorno *VERSION* realiza estos cambios. A partir de estos parámetros el sistema determina si debe compilar *DTW_GPU o DTW_CPU,* con la diferencia de que esta última versión se

modifica en los flags que se le pasan al compilador para optimizar el rendimiento en función de la plataforma.

Para la detección de las funciones que cumplen con la interfaz de BLAS, sean de la implementación incluida en el sistema como de bibliotecas de terceros, mediante el uso de macros de *Autoconf* se ha tenido que incluir en el paquete un archivo M4 disponible en la web para poder detectar si existe alguna biblioteca que cumpla dicha interfaz.

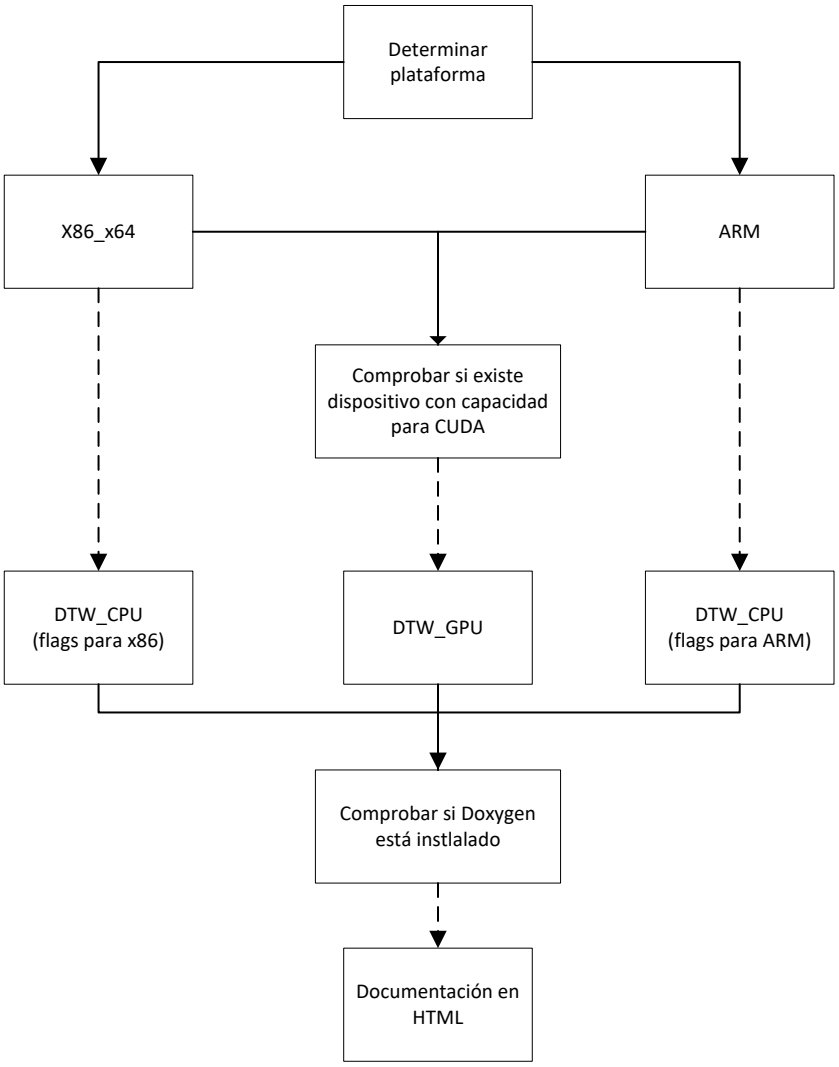



*Figura 22: Flujo de comprobación de Autoconf*

# 5. Pruebas realizadas

## 5.1 Entorno de pruebas

El paquete desarrollado busca corresponder con una implementación acorde a la que otros paquetes que utilizan herramientas de generación similares ofrecen. Así, la instalación del paquete, de la cual hablaremos en el capítulo 7 de este texto, debe ser lo más sencilla para el usuario. Además, el código fuente de la aplicación debe estar disponible dentro del directorio del paquete para poder realizar su compilación. Además, suele ser conveniente adjuntar una serie de tests sobre la aplicación para comprobar que la instalación ha sido correcta, las dependencias que tenga con el sistema o con las bibliotecas sean adecuadas y que su funcionamiento es el esperado.

Así, el subdirectorio del paquete *test* proporciona una serie de *script*s para poder probar todos estos aspectos para las implementaciones de CPU y GPU desarrolladas. Para poder realizar las pruebas de rendimiento en primer lugar se deben generar los datos que utilizará el programa, puesto que debido al tamaño de estos no es conveniente incluirlos en el paquete, pero sí ofrecer un mecanismo que los genere.

Los *script*s empleados para el test están desarrollados para ser lanzados en entornos UNIX. Además, en el caso de analizar prestaciones la ejecución cambia el valor asociado al parámetro *beta,* en ambas implementaciones, y el tamaño de los hilos de ejecución por bloque en el caso de la versión GPU, para ver qué rango de valores hacen que se obtenga el mayor rendimiento.

El *script* desarrollado para comprobar el correcto funcionamiento del simulador realiza una ejecución con la menor carga computacional para poder comparar su resultado con el de una ejecución que se conoce que es correcta. Así, el resultado de la comparación debe devolver como única diferencia la línea asociada al tiempo de ejecución en caso de que la implementación del simulador sea correcta. Este tipo de análisis es interesante en el caso de realizar modificaciones sobre el código disponible, puesto que permite ver si la nueva implementación sigue siendo correcta en su ejecución. Otro de los usos relevantes de este tipo de *scripts* aparece durante la instalación del paquete en múltiples máquinas. Puesto que las pruebas de rendimiento del simulador desarrollado son *offline* se requiere generar una serie de archivos de datos para probar el rendimiento o validar que el funcionamiento es el correcto.

Existen cuatro *scripts* de validación del simulador, uno para la versión de CPU y otra para la de GPU, además de sus respectivas versiones en simple precisión, por lo que se pueden probar todas las posibilidades de la aplicación.

Un ejemplo de este *script* de validación es el que aparece en la siguiente figura:

```
#!/bin/sh

# Determine path
AppPATH=$(dirname `pwd`)

# Append data folders
BinPATH=$AppPATH/bin
DataPATH=$AppPATH/data
RawDataPATH=$DataPATH/Datos
ResDataPATH=$DataPATH/res

# Define test parameters
TestParam=$RawDataPATH/30sec/parameters.dat
Beta=1.5

Time=$(date +"%Y-%m-%d-%H-%M-%S")
FileName=execution_$Time

$BinPATH/DTWAppCPU $TestParam $Beta >> $ResDataPATH/$FileName


diff $ResDataPATH/$FileName $ResDataPATH/benchmark/execution_double >>
$ResDataPATH/diff_res

lined=$(head -n 1 $ResDataPATH/diff_res)

if [ "$lined" = "1001c1001" ]; then
    echo "(Double) Correct application execution"
else
    echo "(Double) Wrong application execution. There is some error
with the processing code"
fi

rm $ResDataPATH/$FileName
rm $ResDataPATH/diff_res
```

*Figura 23: Script de comprobación de la instalación*

Por otro lado, el simulador incluye dentro del módulo de pruebas *scripts* encargados de hacer un análisis del rendimiento que ofrece la aplicación dentro la plataforma sobre la que se ejecuta. El funcionamiento de estos programas se basa en la ejecución con diferentes parámetros y esquemas de ejecución durante varios ciclos. La idea de este *script* se centra en obtener un tiempo promedio de ejecución y tiempo por trama promedio que permita analizar la viabilidad del dispositivo con las restricciones de tiempo real impuestas.

De igual manera a como el *script* de validación presenta diferentes variantes en función del tipo de ejecución o del tipo de datos, el test de rendimiento del simulador se puede realizar para las implementaciones de CPU y GPU, además de para datos de simple o doble precisión.

Podemos ver un ejemplo de este tipo de *script* en la siguiente figura.

```
#!/bin/bash

AppPATH=$(dirname `pwd`)

DataPATH=$AppPATH/data
RawDataPATH=$DataPATH/Datos
ResDataPATH=$DataPATH/res

Time=$(date +"%Y-%m-%d-%H-%M-%S")

TestFolder=$ResDataPATH/perfomance_test_$Time
mkdir $TestFolder

BETA="0.0 1.0 1.5 1.7 2.0"
TIME="150sec 5min 10min 15min 30min"
N=20

# Execute the application.
# First using all the cores
# Then using only one core
until [ $N -lt 0 ];do
    echo "Iteration $N" >> $TestFolder/one_core_$i-$j
    for i in $BETA
    do
        for j in $TIME
        do
            echo "(All cores) Beta $i with time $j"
            DTWAppCPU $i $RawDataPATH/$j/parameters.dat >>
$TestFolder/all_cores_$i-$j-execut
            line=$(tail -n1 $TestFolder/all_cores_$i-$j-execut)
            echo $line >> $TestFolder/all_cores_$i-$j
            rm $TestFolder/all_cores_$i-$j-execut
        done
    done
    let N-=1
done

export OMP_NUM_THREADS=1

N=20
until [ $N -lt 0 ]; do
    echo "Iteration $N" >> $TestFolder/one_core_$i-$j
    for i in $BETA
    do
        for j in $TIME
        do
            echo "(One core) Beta $i with time $j"
            DTWAppCPU $i $RawDataPATH/$j/parameters.dat >>
$TestFolder/one_core_$i-$j-execut
            line=$(tail -n1 $TestFolder/one_core_$i-$j-execut)
            echo $line >> $TestFolder/one_core_$i-$j
            rm $TestFolder/one_core_$i-$j-execut
        done
    done
    let N-=1
done

echo "Performance test completed. Your files are stored in
$TestFolder"
```

*Figura 24: Script de análisis de rendimiento*

## 5.2 Pruebas sobre arquitecturas x86

Si bien no es un objetivo de los que se busca con la aplicación, el sistema ha sido probado en equipos de escritorio para probar que su funcionamiento es el correcto y que demuestran que el requisito de tiempo real. Los tiempos medios de proceso de tramas obtenidos para la ejecución de la aplicación paralelizada por parte del *script* de rendimiento son los siguientes:

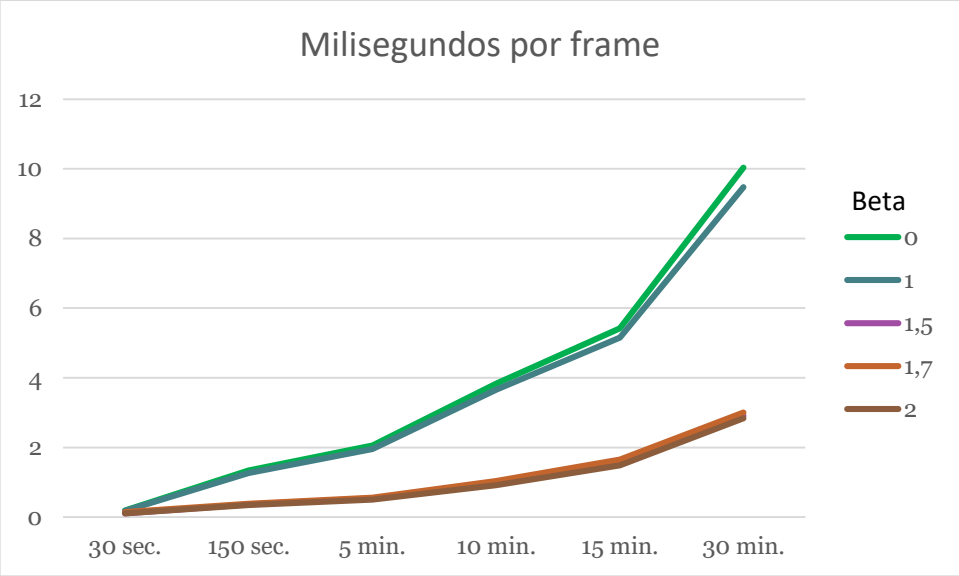



*Figura 25: Tiempo de ejecución para diferentes simulaciones*

Como se puede apreciar, todas las ejecuciones, con sus diferentes cantidades de música procesada, cumplen los requisitos de tiempo real, por lo que este tipo de procesador sería válido para una ejecución *online*. Sin embargo, el factor clave para la ejecución óptima lo encontramos en el valor asociado a la constante *beta.* La figura 26 muestra en detalle las ejecuciones para valores de *beta* mayores que uno. En ella podemos ver como los valores que llevan la ejecución a su valor mínimo está con 1.5 o con 2, puesto que la diferencia entre ambos en este caso es despreciable.

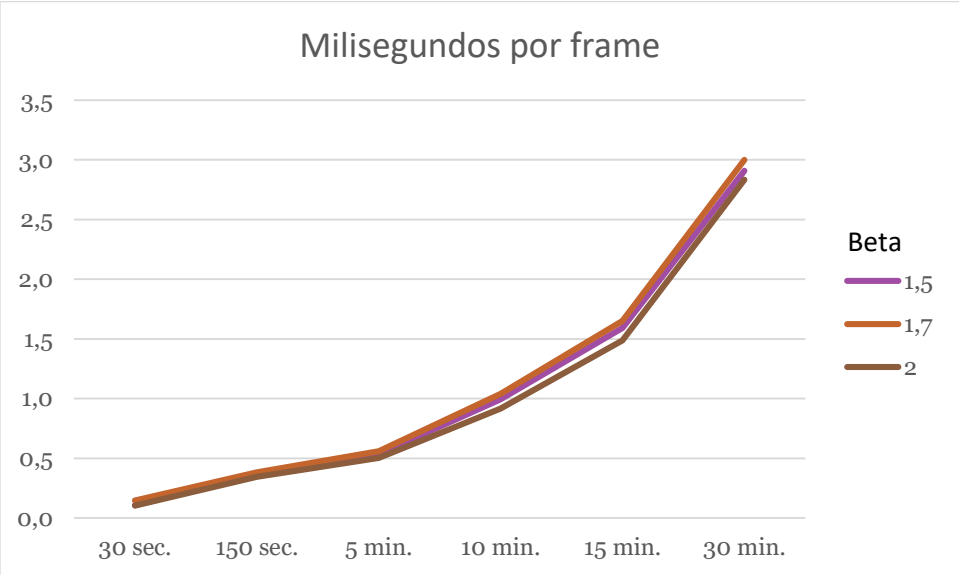



*Figura 26: Detalle de la ejecución para β > 1*

Sumado a esto, es relevante ver cómo afecta la implementación paralela al sistema en términos relativos al SpeedUp y a la eficiencia del sistema:

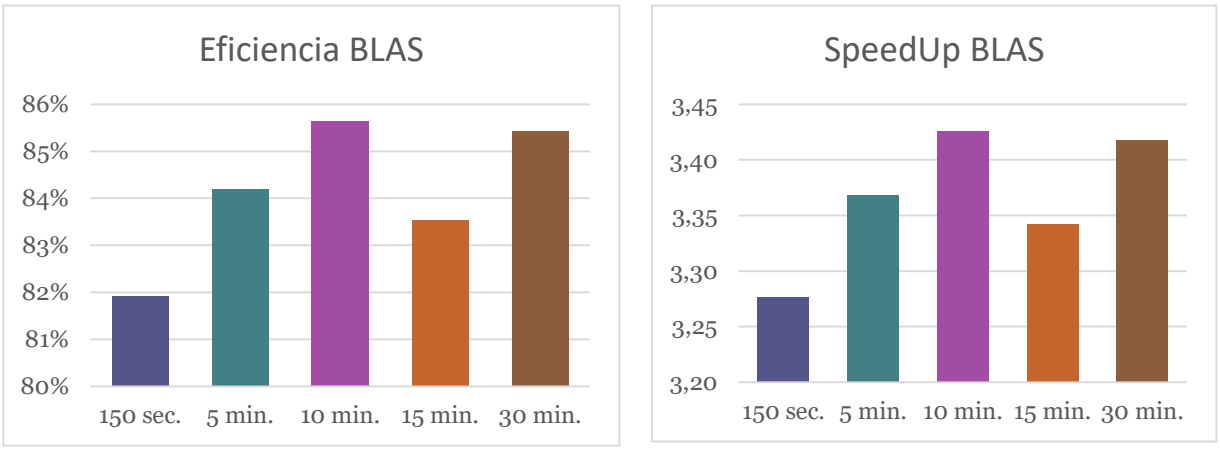



*Figura 27: Eficiencia y aceleración del sistema en arquitecturas x86*

Como se puede ver, la paralelización del sistema ofrece una aceleración muy próxima, en los casos con mayor carga computacional, al número de núcleos disponibles en el sistema. Sumado a esto también se puede ver como prácticamente la utilización de la CPU es total y no se tiene un hardware sobredimensionado que no es aprovechado.

Cabe destacar un último aspecto. Aunque, como se puede apreciar, el tiempo de ejecución y alineamiento es mínimo para un valor de *beta* igual a dos, la ejecución óptima, mostrada en las gráficas de eficiencia y aceleración anteriores, corresponde a un valor de *beta* de 1.5 puesto que es con este valor con el que se obtiene una mayor utilización de los recursos disponibles, ya que ha sido uno de las consideraciones a tener en cuenta a la hora de realizar este proyecto, y la diferencia en sus valores está en el intervalo $[0.016, 0.104]$.

## 5.3 Pruebas sobre arquitecturas ARM bajo Linux

El último de los análisis se centra en la ejecución de la aplicación en los dispositivos basados en arquitectura ARM. Dado que este tipo de dispositivos y arquitecturas están muy extendidas en la actualidad, debido a que la mayoría de teléfonos y tablets los utilizan, es interesante ver hasta dónde se pueden rebajar los costes del dispositivo para obtener un alineamiento *online* efectivo.

En este apartado nos centraremos en los resultados obtenidos para la ejecución de la versión para CPU únicamente, y en el siguiente apartado analizaremos el resultado que el kit *Jetson* obtiene utilizando la GPU.

El primer parámetro que analizaremos será el tiempo de proceso para cada una de las tramas, buscando que alcancen los 12.8ms de objetivo. La siguiente figura muestra los tiempos asociados al kit *Jetson* y a la *Raspberry Pi 2* para una compilación con datos en doble precisión.

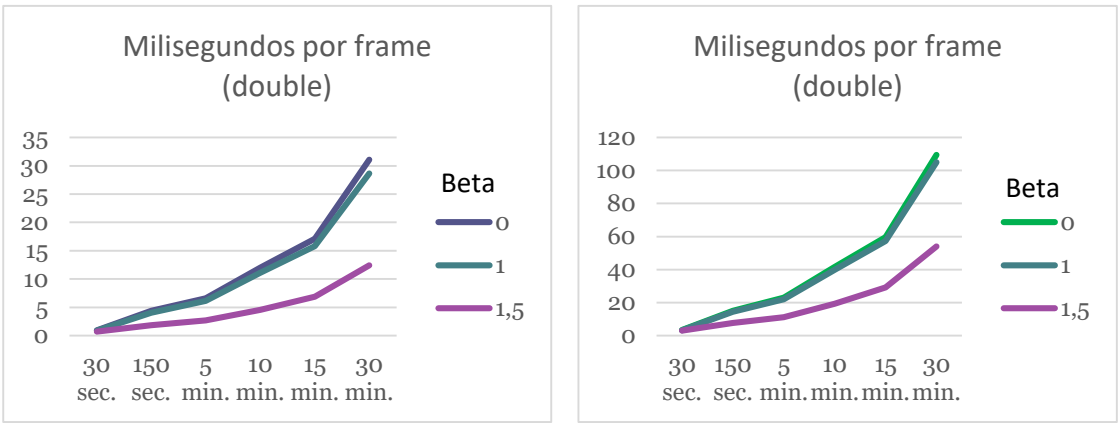



*Figura 28: Tiempo por trama para double de NVidia Jetson (izquierda) y Raspberry Pi 2 (derecha)*

Como se puede observar, el uso de la constante *beta* altera de manera efectiva el tiempo de proceso por trama igual que en la implementación para CPU basada en arquitecturas x86. Sin embargo, los datos obtenidos no son muy positivos de cara a la utilización de estos dispositivos para el alineamiento de partituras con doble precisión, puesto que la tendencia es que las composiciones de mayor duración no garantizan mantener el umbral por debajo del requisito de 12.8ms para las ejecuciones de más de 10 minutos en ambos dispositivos con $\beta < 1.5$. Sin embargo, la potencia extra del procesador del kit *Jetson* sí permite procesar en doble precisión composiciones de cinco minutos con un valor de la constante menor que el mencionado anteriormente. El cambio notable lo vemos al utilizar un valor de la constante de 1.5, puesto que vemos que uno de los dos dispositivos objetivo, el kit *Jetson*, sí garantiza el alineamiento online en cinco de los seis casos analizados. Por otro lado, la *Raspberry Pi 2* a pesar de la mejora en tiempo de procesamiento continúa siendo incapaz de procesar en tiempo real composiciones de más de cinco minutos en doble precisión.

Por otro lado, si utilizamos valores de precisión simple podemos ver cómo la carga computacional del sistema se libera y la situación mejora en ambos dispositivos.

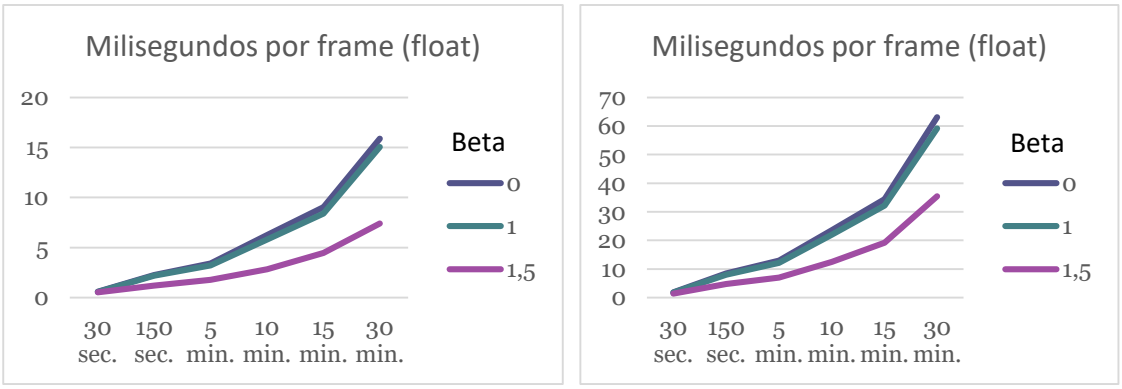



*Figura 29: Tiempo por trama para float de NVidia Jetson (izquierda) y Raspberry Pi 2 (derecha)*

Como se puede apreciar, el uso de datos de simple precisión permite reducir considerablemente los tiempos obtenidos. El caso más notable es el de la Raspberry Pi 2 que, si bien sigue sin alcanzar el nivel de rendimiento necesario para ejecuciones de gran carga en tiempo real pero abre la puerta a la posibilidad de que con valores mayores de la constante *beta* se pueda conseguir un rendimiento superior, ya que actualmente la mejora conseguida en el peor dispositivo está entre el 34% y el 40%.

## 5.4 Pruebas sobre arquitecturas CUDA

Para las pruebas de la versión de GPU se han realizado comprobaciones únicamente en un entorno relevante, como es el caso del Kit NVidia Jetson, puesto que de igual manera a como pasa con la implementación para CPU, las tarjetas gráficas de escritorio de las que se disponen obtienen unos resultados que en todos los casos cumplen los requisitos para un alineamiento *online* sobremanera. Así, la prueba realizada sobre el kit NVidia Jetson, el cual ofrece un hardware similar al que se puede encontrar en dispositivos móviles, permite determinar la viabilidad de esta aplicación para una ejecución en tiempo real.

El tiempo de proceso por trama promedio (ejecución de las cuatro diferentes posibilidades) asociado a la ejecución mediante GPU con datos en doble precisión es la siguiente:

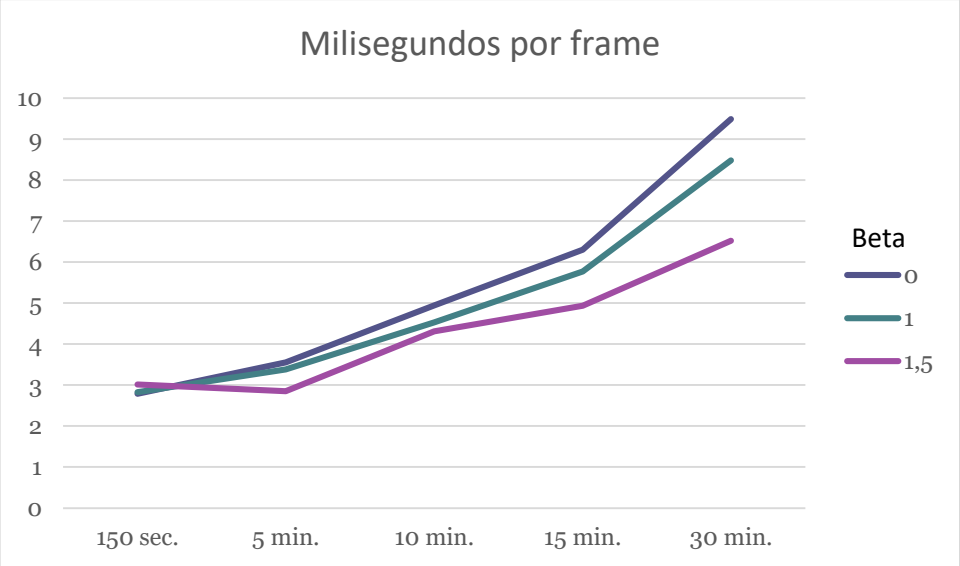



*Figura 30: Tiempo de ejecución para diferentes simulaciones mediante GPU*

Al contrario de lo que ocurría en la implementación para CPU, la versión basada en CUDA obtiene en todos los casos, al margen del valor de la constante, un tiempo de procesamiento inferior al máximo exigido para tiempo real.

Como se ha comentado anteriormente, la figura anterior muestra los resultados de la ejecución en promedio, esto es, utilizando una implementación de *cuBLAS*, que es una implementación de las funciones presentes en *BLAS* pensado para el modelo de cálculo que utiliza *CUDA.* Se ha utilizado memoria unificada, que es un modelo mediante el cual la *CPU* y la *GPU* acceden por igual a la memoria principal del sistema, evitando así la necesidad de copiar datos entre dispositivos, tal como sucede en sistemas tradicionales en los que la GPU está conectada al host mediante un bus *PCI.* Las pruebas realizadas permiten observar el comportamiento siguiendo estas dos variantes de esquema de memoria. Si entramos a analizar el caso más óptimo (β>= 1.5) podemos comparar cuál de estas versiones ofrece un mejor rendimiento.

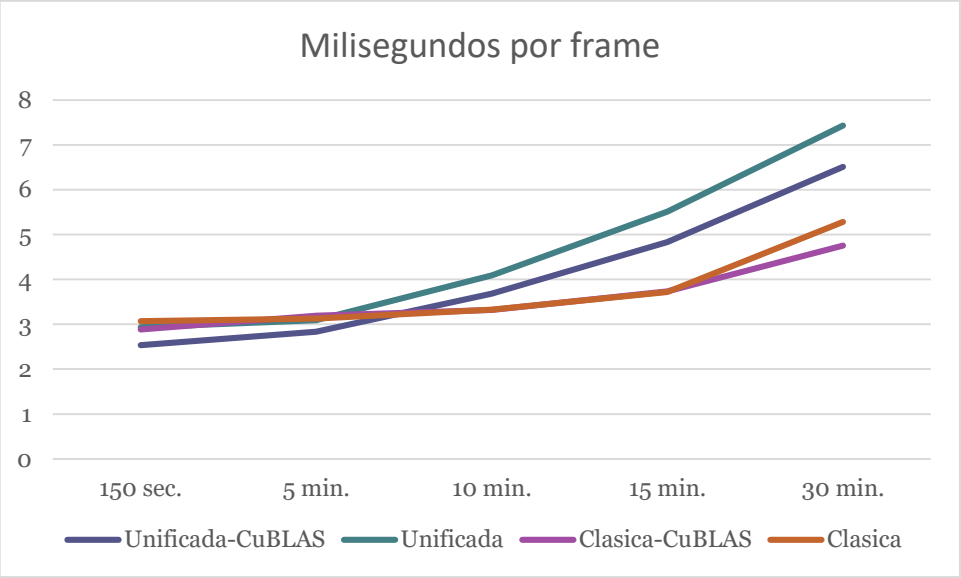
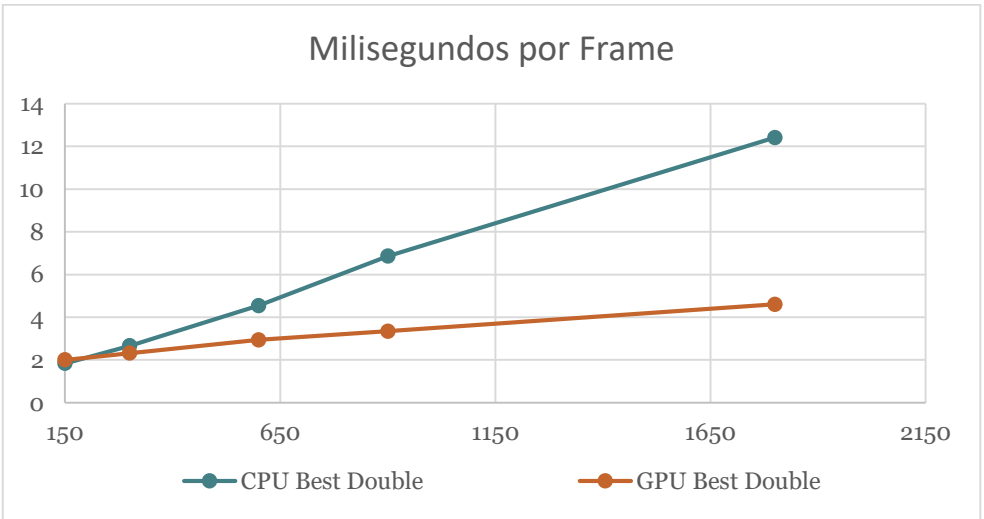



*Figura 31: Tiempo de proceso en función del tipo de ejecución*

Es interesante ver la variación que existe entre las diferentes implementaciones, puesto que con ello se puede ver cómo en función de la carga a procesar se puede optar por una versión u otra. Para cargas reducidas, siendo aquellas de menos de diez minutos, la aproximación basada en *CuBLAS* y esquemas de memoria unificada el tiempo que se consigue es el mínimo, mientras que para un procesamiento mayor se puede optar por una aproximación basada en *CuBLAS* y esquemas de memoria clásicos. Otro aspecto que puede resultar relevante en algunas situaciones es el que podemos observar en el rango de los diez minutos, donde la diferencia entre las implementaciones basadas en *CuBLAS* y en *kernels* diseñados para este tipo de problemas apenas se aprecia. Esto nos lleva a pensar que si el dispositivo va a trabajar con cargas equivalentes puede ser interesante no utilizar la implementación de *CuBLAS*, puesto que así las dependencias del sistema con otras bibliotecas se reducirían.

Como último punto del análisis podemos ver la diferencia entre el rendimiento obtenido entre la versión para CPU y GPU en la implementación para el kit *Jetson*, puesto que un dispositivo equivalente en rendimiento lo podríamos encontrar en el *NVidia Shield TV,* cuyo precio ronda los 200€. Los resultados obtenidos para el mejor caso analizado, con *beta* 1.5 y datos en doble precisión, aparecen en la figura siguiente.



Como se podía intuir por el rendimiento obtenido en los análisis previos la implementación basada en la tecnología *CUDA* permite obtener un rendimiento por trama procesada hasta un 63% más rápido que frente a la implementación para CPU. Es, por tanto, la versión para GPU la que ofrece un mayor rendimiento sin requerir de un dispositivo con gran cantidad de multiprocesadores, puesto que el caso analizado se basa en una tecnología con 256 núcleos CUDA frente a los varios miles que puede tener una tarjeta gráfica de escritorio en la actualidad.

# 6. Resultados y conclusiones

## 6.1 Resultados relevantes

Como podemos ver, hemos obtenido un paquete software que persigue un diseño ampliamente modular y abierto, puesto que permite a los usuarios analizar nuevos algoritmos, añadir nuevas etapas al flujo de la aplicación sin realizar un cambio drástico en el diseño de la misma o incluso incluir funciones de biblioteca nuevas con sencillez. Además, vemos que a partir de los resultados obtenidos, si bien se podría buscar la mejora del sistema para alcanzar el requisito de tiempo real en los casos en los que la CPU no proporciona el tiempo de proceso deseado.

Sin embargo, el rendimiento general del paquete es satisfactorio, puesto que proporciona una estructura abierta para el análisis de algoritmos cuyo objetivo sea el alineamiento de partituras en tiempo real y se ofrece una documentación detallada para que cualquier usuario pueda instalar, analizar y modificar la aplicación según sus necesidades.

## 6.2 Relación con la carrera

Después de todo lo desarrollado, es interesante analizar cuál ha sido el impacto que han tenido los estudios cursados sobre el desarrollo de este proyecto durante todos estos meses. Así, vemos que destacan ciertas metodologías en algunas de las asignaturas cursadas durante estos años.

La primera de estas asignaturas sería *Ingeniería del Software,* dado que en ella se enseña que el mejor diseño de una aplicación es aquel que tiene un acoplamiento bajo y permite distinguir todos los elementos que forman el conjunto de una aplicación con claridad y cómo estos deben estar correctamente diseñados y documentados para permitir que desarrolladores externos a la implementación original puedan trabajar sobre ese código. Podemos ver el impacto de esta asignatura en el diseño modular de la aplicación, que busca diferenciar claramente en módulos cada una de las etapas que requiere el funcionamiento de la aplicación para facilitar su uso, además de estar realizadas con una documentación rigurosa que permite a desarrolladores que deseen añadir nuevos módulos comprender cómo funciona cada módulo y cuál es la interfaz de llamadas a funciones de la que dispone.

Las siguientes asignaturas a las que podemos mencionar son *Gestión de proyectos,* por un lado, y *Análisis de los requisitos de negocio,* de la especialidad de Sistemas de la Información, por el otro. En estas asignaturas se ha destacado la importancia que tiene la correcta elaboración de un plan de trabajo, desde la especificación de los requisitos que debe cumplir la aplicación, su presupuesto, su calendario de trabajo o un análisis de las herramientas a utilizar durante el desarrollo. Esto facilita tener unas metodologías de trabajo sistematizadas, con lo que se optimiza el rendimiento de cara a las fases de diseño de la aplicación, implementación y depuración puesto que sienta unas bases sólidas sobre las que trabajar, ya sea una única persona como en el caso que nos ocupa con este proyecto o con un equipo de desarrollo formado por varias personas.

También tienen importancia las asignaturas *Computación Paralela* y *Lenguajes y Entornos de Programación Paralela* puesto que este proyecto busca sacar el máximo rendimiento de los

dispositivos *hardware* utilizado, kits de desarrollo *NVidia Jetson* y placas empotradas *Raspberry Pi*. En estas asignaturas se instruye en la utilización de herramientas para el desarrollo de programas paralelos y metodologías y soluciones de diseño para resolución de problemas paralelos. Dado que en este proyecto se utiliza de manera extensiva la biblioteca OpenMP para el desarrollo de funciones paralelas, así como el uso de técnicas como la programación dinámica para la implementación del algoritmo de *DTW* implementado basado en una solución paralela.

Las últimas asignaturas que podemos destacar para el desarrollo de este proyecto son *Diseño de Sistemas Operativos* y *Sistemas de Tiempo Real,* ambas de la especialidad de Ingeniería de Computadores. La primera de estas dos nos ha enseñado a utilizar con mayor destreza el lenguaje C por un lado, además de a comprender cómo funcionan ciertos aspectos de los sistemas Unix con el fin de poder aprovecharlos al máximo. En lo que respecta a la asignatura *Sistemas de Tiempo Real,* si bien se enfoca en el desarrollo de aplicaciones para sistemas empotrados con requisitos de tiempo real, ha permitido conocer la problemática que estos sistemas tienen asociada y cómo resolver los principales puntos que pueden aparecer al tener requisitos de tiempo real que surgen en el desarrollo de paquetes como el realizado durante este proyecto.

## 6.3 Ampliaciones futuras

La finalización de éste proyecto no supone la finalización del desarrollo de esta aplicación, puesto que existen diferentes mejoras que serían interesantes de realizar pero que en el momento del desarrollo inicial no se estudiaron o se descartaron durante la planificación.

La primera ampliación que se podría realizar sobre el paquete de alineamiento de partituras desarrollado consistiría en cambiar la implementación del paquete instalable desarrollado mediante las *Autotools* a un paquete que ofrezca soporte para más sistemas operativos. Por ello, una implementación de este paquete utilizando un sistema de gestión de paquetes como Cmake permitiría generar ejecutables tanto para los actuales entornos Unix como incluso para sistemas con Windows, lo que permitiría ampliar el número de usuarios que podrían utilizar la aplicación.

Otra ampliación interesante sería ofrecer soporte para más bibliotecas, dado que además de la inclusión de las subrutinas de la biblioteca BLAS se podría buscar algún tipo de implementación de las operaciones matriciales, ya sea para entornos Windows o para entornos que utilicen otras bibliotecas como pudiera ser Armadillo [22].

Relacionado con otras funciones de biblioteca podríamos también considerar la modificación del ciclo de vida que tiene la aplicación. Como hemos podido ver, se presenta una interfaz de estructuras y datos común para cada uno de los módulos que forman el programa. Podría estudiarse la posible mejora que presentaría el sistema portándolo a un lenguaje orientado a objetos como C++ para comprobar si la mejora en el ciclo de vida ofrece mejor rendimiento, además de permitir reducir las dependencias que tiene la implementación actual con un archivo de datos común.

Dado que el desarrollo de este paquete está orientado al tratamiento y alineamiento de partituras en tiempo real se podría proponer el desarrollo de módulos de tratamiento de las señalas de audio en tiempo real en contraposición a los archivos utilizados durante el desarrollo de este proyecto que simulan el comportamiento de una entrada de audio dada. Podría ser interesante analizar si dispositivos empotrados a los que está dirigido esta aplicación son capaces

de obtener la señal de audio, procesarla y obtener la salida corregida en tiempo real con los requisitos establecidos para su correcto funcionamiento.

La última optimización que se podría hacer estaría más centrada en la arquitectura ARM, puesto que se podría utilizar el conjunto de instrucciones NEON de ARM para el tratamiento de todas las tramas de la aplicación, optimizando el rendimiento de la aplicación en estas arquitecturas.

## 6.4 Conclusiones finales

Para cerrar este documento podemos hacer una vista general a lo que ha supuesto y extraer cierta información de valor. En primer lugar, se ha desarrollado una aplicación software cuyo propósito y utilidad puede estar lejos de proyectos más académicos, en los que la funcionalidad puede quedar más eclipsada puesto que no es lo que se busca demostrar. En este caso se dispone de un paquete software instalable, que se encarga de gestionar las dependencias de la aplicación con el sistema y con bibliotecas de terceros, con una utilidad práctica importante en un rango amplio de herramientas y con soporte a diversas tecnologías y entornos de ejecución. Sumado a esto, el conjunto presenta una *suite* de opciones de prueba que permiten que cualquier usuario analice su dispositivo para probar si cumple con las características necesarias para el alineamiento de partituras *online,* basado en tiempo real, o si, por el contrario, sólo serviría para un servicio *offline,* a partir de un fichero que contenga la información a procesar.

Sin embargo, no todas las conclusiones que se pueden extraer son positivas. El desarrollo de este proyecto y de esta memoria ha sido duro debido a multitud de problemas que, en último lugar, han reducido el tiempo dedicado. Ello ha hecho que alternativas interesantes, como la generación de un ejecutable para dispositivos Android, haya tenido que descartarse. El alineamiento de partituras es un problema interesante, y este tipo de dispositivos pueden beneficiarse de disponer de un software abierto que posibilite la creación de aplicaciones más complejas que resuelvan este problema.

Aun así, es la sensación de completitud la que queda al comprobar el resultado obtenido, al margen de no cumplir restricciones de tiempo real en algún dispositivo analizado, y presenta un entorno de trabajo que, en un futuro, se puede continuar mejorando.

# 7. Configuración del paquete

## 7.1 Obtención del paquete

El paquete desarrollado *DTWApp* está disponible para su descarga en el repositorio público de *Github* https://github.com/flyingonion93/ETSInf_DTWApp

## 7.2 Requisitos preliminares

Para poder ejecutar e instalar el paquete, puesto que la instalación compila los archivos de código fuente descargados, es necesario tener instaladas ciertas bibliotecas de funciones para que durante la fase de enlazado no aparezcan problemas. Las bibliotecas requeridas para el funcionamiento de la aplicación son:

- OpenBLAS (https://github.com/xianyi/OpenBLAS) o cualquier otra implementación de la interfaz de las subrutinas del BLAS desarrollada para C. Actualmente el paquete ha sido probado con la biblioteca *MKL* de Intel, además del OpenBLAS anteriormente mencionado.
- FFTW3 (http://www.fftw.org/) para la realización de las *FFTs* del sistema. Dado que se trata de la biblioteca para el cálculo de este tipo de operaciones más extendida el paquete actualmente sólo ofrece soporte para ésta, si bien se podría realizar una modificación sobre el módulo de preproceso con facilidad para poder añadir nuevas bibliotecas.

Por otro lado, el código del simulador se ha documentado para poder obtener una API de las funciones de las que dispone el programa de manera clara y sencilla. Para ello se incluye un fichero Doxyfile que, en caso de disponer de la aplicación Doxygen (http://www.stack.nl/~dimitri/doxygen/) hará que durante el proceso de instalación haga que se genere la documentación del proyecto.

## 7.3 Instalación del paquete

La forma de instalar el paquete es la utilizada por los sistemas desarrollados mediante las Autotools. El usuario debe ejecutar desde la consola la secuencia

*"./configure"* Para analizar el sistema objetivo y establecer las dependencias y los parámetros de compilación

"*make"* Para compilar de acuerdo a los parámetros establecidos previamente

"*sudo make install"* Para poder añadir el ejecutable de la aplicación al directorio */usr/bin/* y poder hacer así uso de ella como en el caso de cualquier aplicación UNIX estándar.

Una vez realizados estos tres pasos la aplicación ya está lista para ser ejecutada mediante línea de comandos, puesto que podrá ser ejecutada al encontrarse dentro de la variable de entorno *PATH*.

La realización de los pasos anteriores instalaría el paquete en su versión para CPU con las características por defecto, que han sido definidas como arquitectura x86, con el uso de BLAS y FFTW y con análisis de datos de doble precisión. Sin embargo, el paquete tiene definidas cuatro

variables de entorno que permitirían al usuario instalar una versión para una arquitectura diferente, o incluso una versión basada en CUDA para su ejecución en GPU, con diferentes funciones de biblioteca o con análisis de datos de simple precisión. Las variables definidas actualmente son:

- *VERSION:* Indica la versión que se va a compilar de la aplicación. Las opciones posibles actualmente son “x86”, “gpu” y “arm” y su valor por defecto es “x86”. El uso de esta variable permite realizar compilación cruzada del paquete para poder enviar el ejecutable al dispositivo objetivo, en lugar de tener que compilar desde el dispositivo, lo cual podría incluso ser más lento. Dado que existe multiplicidad de arquitecturas y tecnologías es amplia, ciertos parámetros más específicos del objetivo, como el tamaño de palabra del procesador, pueden definirse en los *CFLAGS* durante la ejecución del *Makefile*.
- *FFT:* Especifica la biblioteca encargada de realizar las *FFT* que el sistema debe resolver. Actualmente el único valor válido es “FFTW”, puesto que no se han hecho más implementaciones de este tipo de operaciones. Si un usuario añadiese nueva funcionalidad al paquete y desease utilizar su propia biblioteca debería incluir en el fichero *configure.ac* una comprobación del valor para vincular su alternativa a la aplicación.
- *ALG:* Especifica la biblioteca encarga de realizar las operaciones de álgebra numérico. Actualmente el único valor válido es “BLAS”, puesto que al ser la biblioteca estándar se ha optado por ofrecer soporte a esta. De la misma manera que pasa en el caso de la biblioteca para las *FFT,* cualquier usuario podría dar soporte a bibliotecas nuevas que satisfagan sus necesidades mejor.
- *PRECISSION:* Especifica el tipo de datos a analizar. Las opciones posibles son “simple” y “double”. El resultado de elegir una opción u otra hace, como se comentó en el capítulo de diseño, que se defina una macro en el programa y que el tipo de datos genérico creado pase a estar definido en tiempo de compilación.

## 7.4 Esquema de directorios

Como cualquier otro paquete que haga uso de las *Autotools*, el esquema de directorios es bastante claro, con la finalidad de que el usuario pueda tener acceso total a los elementos que contiene. El paquete se divide principalmente en un directorio que contiene el código fuente de la aplicación, uno otro con los datos de la aplicación en el caso de alineamiento *offline* y un último directorio que contiene los elementos de prueba. Una vez compilado, además, aparece un directorio donde se genera el ejecutable de la aplicación. Por último, y si el usuario lo desea y dispone de la herramienta *Doxygen*, durante el proceso de compilación se puede generar un directorio con la documentación de cada uno de los módulos desarrollados, junto con las variables que define y los tipos de datos genéricos empleados.

## 7.5 Uso del paquete

Para utilizar el paquete primero se deben generar los datos a alinear, puesto que como se comentaba en capítulos anteriores la implementación está basada en el modelo *offline* del sistema. Así, en el directorio *data* encontramos los *scripts build_data.sh* y *build_verify.sh*, los cuales permiten generar los archivos que faltan para que la ejecución del sistema sea correcta. Una vez generados los datos del problema la aplicación se puede ejecutar desde línea de comandos para

probar una ejecución concreta o bien se pueden realizar las pruebas de funcionamiento y rendimiento disponibles en la carpeta *tests* de la aplicación. A la aplicación se le debe pasar como parámetro de entrada un fichero que contiene los elementos a procesar junto a un valor para la constante *beta*, en el caso de la versión para CPU, o esos mismos parámetros junto con un indicador para determinar si se utiliza un esquema de memoria unificada o si por el contrario se emplea la memoria que CUDA permite utilizar por defecto, además de otro valor para determinar el tamaño de bloque y si se debe utilizar el *kernel* propietario o el de *CuBLAS.* Si se ejecuta cualquiera de las dos aplicaciones sin parámetros se puede ver un mensaje de ayuda que indica el tipo de entrada que requiere la aplicación.

Una vez ejecutada la aplicación esta mostrará por pantalla la trama que se ha procesado junto con la posición mínima para alinear la señal. En último lugar la aplicación muestra una serie de estadísticas como el tiempo de ejecución total y el tiempo que tarda en procesar una única trama.

# 8. Bibliografía

[1] IRCAM - CNRS - UPMC, «Music Representation Team,» 13 10 2014. [En línea]. Available: http://repmus.ircam.fr/score-following. [Último acceso: 25 06 2016].

[2] NVidia, «Procesamiento Paralelo CUDA,» 2016. [En línea]. Available: http://www.nvidia.es/object/cuda-parallel-computing-es.html. [Último acceso: 30 6 2016].

[3] FFTW, «FFTW,» [En línea]. [Último acceso: 15 4 2016].

[4] B. Venu, «Multi core processors - An overview,» 2011.

[5] D. A. Patterson y J. L. Hennessy, «El muro de la potencia,» de *Estructura y diseño de computadores. La interfaz Hardware / Software*, Barcelona, Reverté, 2011, pp. 39-40.

[6] D. A. Patterson y J. L. Hennessy, «El gran cambio: el paso de monoprocesadores a multiprocesadores,» de *Estructura y diseño de computadores. La interfaz Hardware / Software*, Barcelona, Reverté, 2011, pp. 41-42.

[7] AMD, «AMD Developer Central,» 2012. [En línea]. Available: http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-computing/. [Último acceso: 7 6 2016].

[8] Intel, «Threading Building Blocks,» 2016. [En línea]. Available: https://www.threadingbuildingblocks.org/. [Último acceso: 7 6 2016].

[9] OpenMP ARB, «Frequently asked question on OpenMP,» 2013. [En línea]. Available: http://openmp.org/openmp-faq.html. [Último acceso: 29 6 2016].

[10] IRCAM - Music Representation Team, «Score Following History in Video,» 12 1 2016. [En línea]. Available: http://repmus.ircam.fr/antescofo/videos#score-following-history-in-video. [Último acceso: 27 6 2016].

[11] CMake, «CMake,» [En línea]. Available: https://cmake.org/overview/. [Último acceso: 30 6 2016].

[12] Free Software Foundation, «Introducing the GNU Build System,» 2012. [En línea]. Available: http://www.gnu.org/software/automake/manual/html_node/GNU-Build-System.html#GNU-Build-System. [Último acceso: 30 6 2016].

[13] Free Software Foundation, «Automake Documentation,» 2014. [En línea]. Available: http://www.gnu.org/software/automake/manual/automake.html. [Último acceso: 30 6 2016].

[14] Free Software Foundation, «Libtool manual,» 16 1 2015. [En línea]. Available: http://www.gnu.org/software/libtool/manual/libtool.pdf. [Último acceso: 30 1 2016].

[15] Tonara, «Tonara,» 2016. [En línea]. Available: http://tonara.com/. [Último acceso: 30 6 2016].

[16] «Raspberry Pi 2 Model B - Placa base,» [En línea]. Available: https://www.amazon.es/Raspberry-Pi-Placa-Quad-Core/dp/B00T2U7R7I. [Último acceso: 7 6 2016].

[17] Idealista.com, «Idealista.com,» [En línea]. Available: http://www.idealista.com/inmueble/33156694. [Último acceso: 2016 6 20].

[18] «Cálculo del sueldo mínimo según convenio,» [En línea]. Available: http://www.elpicador.org/Archivos/General/CalculoSueldoConvenio.xls. [Último acceso: 7 6 2016].

[19] P. Alonso y otros, «Parallel online time warping for real-time audio-to-score alignment in multi-core systems,» pp. 2-3, 2016.

[20] National Instruments, «National Instruments,» 04 05 2015. [En línea]. Available: http://www.ni.com/white-paper/4844/en/#toc2. [Último acceso: 16 08 2016].

[21] Computational Biology, «Computational Biology,» [En línea]. Available: http://www.psb.ugent.be/cbd/papers/gentxwarper/DTWalgorithm.htm. [Último acceso: 15 7 2016].

[22] «Armadillo - C++ linear algebra library,» 2016. [En línea]. [Último acceso: 10 7 2016].

# Anexo A. Convenciones de estilo

**NOMBRES**

Al declarar variables se usarán las minúsculas. En el caso de ser variables con más de una palabra se utiliza '_' como el separador.

```
int variable;
int multi_word_variable;
```

Cuando se declaren funciones se utilizará la misma convención que para las variables, con la diferencia de que la primera palabra empezará por mayúscula.

```
void Set_ammount( int n )
{
        ...
}
```

Toda variable que haga referencia a algún tipo de magnitud debe indicar el tipo de unidades que representa. Por ejemplo:

```
int time_seconds;
int pressure_mbar;
```

En el caso de declarar un puntero a una variable se debe colocar el * lo más cerca posible de la variable.

```
int *pointer_var;       //CORRECTO

int* pointer_var;       //INCORRECTO
```

Al declarar una constante se utilizará un nombrado en mayúsculas, usando '_' como separador en caso de tener más de una palabra. Esto se aplica de igual manera para las macros que se definan.

```
const int MY_CONSTANT = 27;
#define MAX( a, b ) (  ( a ) > ( b )  ? ( a ) : ( b ) )
```

Todas las estructuras que se usen deben declararse siguiendo el mismo estilo de nombrado que para las variables, añadiendo el sufijo '_st'.

```
struct some_struct_st
{
        int var;
        size_t *buffer_size;
};
```

Las funciones recursivas van acompañadas del sufijo '_r'

```
void Factorial_r( int n )
{
        ...
}
```

De manera análoga, al declarar un tipo *enum* se debe añadir el sufijo 'nm'. Los elementos dentro de dicho *enum* deberán estar en mayúscula.

```
enum some_enum_nm
{
        FIRST,
        SECOND,
        LAST
};
```

**FORMATO**

Las llaves se colocan justo después de la sentencia, en la primera columna.

```
if ( ERROR == value )
{
        ...
}
```

Al utilizar los paréntesis se debe poner una separación entre los mismos y el elemento que envuelven.

```
x = ( y * 0.5f );
```

En el caso de condicionales se debe colocar la sentencia *else* después de la llave que cierra al *if*.

```
if ( x )
{
        ...
} else
{
        ...
}
```

Los bloques (*if-else*, *while*, *for*, etc.) que sólo contengan una instrucción no necesitan estar envueltos por llaves, pero deben dejar un espacio de separación entre el cuerpo del bloque y la siguiente instrucción.

```
for( i = 0; i < value; i++ )
        v[i] = 0;

int next_value = 100;
```

Con el fin de aumentar la legibilidad del código las variables se declaran una por línea.

```
char **a, *x;    //INCORRECTO

char **a;
char *x;         //CORRECTO
```

Cada fichero de cabecera creado contendrá guardas para proteger ante la inclusión múltiple. Para mejorar la interoperabilidad con C++ no se utiliza '_' al principio de la declaración

```
#ifndef my_file_h
        #define my_file_h
#endif
```

**DOCUMENTACIÓN**

Los comentarios deben seguir el formato establecido por Doxygen. Toda función debe tener en su documentación los siguientes campos:

1. Un resumen de las operaciones de dicha función.
2. La lista de argumentos que recibe, indicando si se tratan de valores de entrada, de salida o de entrada-salida.
3. El valor de retorno de la función
4. Si la función tiene problemas en su funcionamiento, junto con la fecha en la que se observa dicho fallo.

**MISCELÁNEA**

Los bloques condicionales deben indicar en primer lugar el valor que se quiere comprobar. Esto permite reducir el número de errores inesperados en la aplicación puesto que en caso de olvidar un operador igual el compilador lo detectaría como error, no como asignación.

```
if( 27 == condition )          //CORRECTO
{
        ...
}

if( condition == 27 )          //INCORRECTO
{
        ...
}
```

Está prohibido utilizar valores literales en el código. Todo valor que se utilice debe estar asociado a una variable.

```
for ( i = 0; i < 10; i++ )     //INCORRECTO
{
        ...
}

for ( i = 0; i < value; i++ )  //CORRECTO
{
        ...
}
```