Escuela Politécnica Superior de Linares

Grado en Ingeniería de Tecnologías de Telecomunicación

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Linares

Trabajo Fin de Grado

TECTEL-15

PROCESADO DE SEÑAL EN TIEMPO REAL EN DISPOSITIVOS MULTICORE

Alumno: **Fernando Parra Moya**

Tutor: Prof. D. Pedro Vera Candeas
Depto.: Ingeniería de Telecomunicación

Octubre de 2016

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Linares

Trabajo Fin de Grado

TECTEL-15

PROCESADO DE SEÑAL EN TIEMPO REAL EN DISPOSITIVOS MULTICORE

Alumno: Fernando Parra Moya

Tutor: Prof. D. Pedro Vera Candeas
Depto.: Ingeniería de Telecomunicación

Octubre de 2016

ÍNDICE:

1. INTRODUCCIÓN 6
2. ESTADO DEL ARTE 7
3. OBJETIVOS 12
4. MATERIALES Y MÉTODOS 13
   4.1 KIT DE DESARROLLO NVIDIA JETSON TK1 13
   4.2 ORDENADOR PORTÁTIL LENOVO THINKPAD 11E 14
5. DESARROLLO 16
   5.1 ECUALIZADOR MATLAB 16
   5.2 ECUALIZADOR EN C 22
   5.3 ACOMPAÑAMIENTO MUSICAL EN PARTITURA 30
6. RESULTADOS Y DISCUSIÓN 41
   6.1 FUNCIONAMIENTO EN MODO READFRAME 41
   6.2 FUNCIONAMIENTO EN MODO READMIC 41
   6.3 USO DE COMPILADOR GCC VS ICC 43
   6.4 DIFERENTES FUENTES DE ENTRADA 44
   6.5 PROCESADOR EN ESTADO DE ESTRÉS 45
7. CONCLUSIONES 47
8. ANEXOS 49
   8.1 FLASHEO DEL KIT JETSON TK1 49
   8.2 PREPARACIÓN DEL SOFTWARE 51
   8.3 MANUAL TÉCNICO DE LA APLICACIÓN DE ACOMPAÑAMIENTO MUSICAL 53
   8.4 FUNCIONAMIENTO DTW 55
9. REFERENCIAS BIBLIOGRÁFICAS 58

ÍNDICE DE FIGURAS:

Fig.1 - Interfaz de Antescofo (arriba la señal que se sintetiza) 8
Fig.2 - Interfaz de Tonara (la línea roja marca la posición de avance en partitura)..... 9
Fig.3 – Espectrograma de la señal original y las dos de salida 10
Fig.4 – Kit de desarrollo NVIDIA Jetson TK1 12
Fig.5 – Lenovo ThinkPad 11E 14
Fig.6 – Diagrama de flujo de datos de ecualizador.m..... 16
Fig.7 – Método de transformada DFT con overlap-add (solapamiento-suma) 18
Fig.8 – Se observa que se ha eliminado el tono a 1,5kHz 20
Fig.9 – Se observa que el tono a 2,5kHz se ha conservado 21
Fig.10 – Primera parte del diagrama de bloques del Ecualizador en C..... 22
Fig.11 – Segunda parte del diagrama de bloques del Ecualizador en C..... 23
Fig.12 – Buffer circular 50%..... 24
Fig.13 – Ejecución del filtro finalizada..... 25
Fig.14 – Arriba la señal de entrada, abajo la señal de salida..... 25
Fig.15 – Espectro de la señal de entrada 26
Fig.16 – Espectro de la señal de salida (filtro paso banda 400-4400Hz aplicado)..... 26
Fig.17 – Primera parte del diagrama de flujo de datos de Preproceso_CPU 28
Fig.18 – Segunda parte del diagrama de flujo de datos de Preproceso_CPU 28
Fig.19 – Diagrama de flujo de datos de ReadMic..... 30
Fig.20 – Lista de dispositivos del portátil 33
Fig.21 – Buffer circular 90%..... 35
Fig.22 – Finalización de Preproceso_CPU para 5min ReadMic..... 36
Fig.23 – Ralentizado un 8%..... 41
Fig.24 – Acelerado un 8% 42
Fig.25 – Variables double, 5min, β=0 (peor caso) 42
Fig.26 – Variables float, 150sec, β=2 (mejor caso) 42
Fig.27 – Variables double, 5min, β=0 (peor caso) 43
Fig.28 – Variables float, 150sec, β=2 (mejor caso) 43
Fig.29 – Topología flasheo Jetson 46
Fig.30 – Árbol de archivos de la aplicación..... 51
Fig.31 – Diagrama de flujo de DTW..... 54

1. INTRODUCCIÓN

Hace poco más de 50 años, Gordon E. Moore, cofundador de Intel, dijo que cada dos años se duplicaría el número de transistores en un Circuito Integrado. A esto se le denominó la Ley de Moore, la cual se ha llevado cumpliendo a lo largo de todos estos años. El cumplimiento de esta afirmación ha hecho posible la rápida evolución de las nuevas tecnologías, aunque este constante decrecimiento en el tamaño de los transistores parece que se está frenando cada vez con más fuerza [1].

Actualmente se comercializa en masa una litografía con transistores de 14nm, y el siguiente paso serán los 10nm (aunque ya se ha logrado fabricar con 7nm). Esto supone un incremento importante en la potencia de procesado de datos, pero debido a las limitaciones físicas (velocidad de la luz y tamaño de los átomos) cada vez resulta más complicada la disminución de esta litografía. Esto se está intentando remediar con la integración de varios núcleos en los procesadores (los cuales se podría decir que trabajan codo con codo).

En este trabajo vamos a aprovechar este incremento de potencia de procesado para implementar un sistema de seguimiento de partitura de una pieza musical. Actualmente ya existen aplicaciones realmente optimizadas que consiguen acompañar musicalmente a un músico, o incluso a varios. Un claro ejemplo es Tonara [2], una aplicación gratuita para iOS (sistema operativo de teléfonos y tablets de la marca Californiana Apple). A pesar de ser gratuita, las piezas musicales que queramos ir añadiendo a nuestra colección de partituras tendremos que pagarlas. Esta aplicación utiliza el acompañamiento musical gracias a la transformada DTW (de la cual hablaremos más adelante). Su funcionamiento consiste en que el músico, mientras visualiza la partitura en el dispositivo, vaya tocando la pieza musical, y de forma automática, en la partitura vaya viendo una línea la cual indica la nota por la que va. Además de esto, también cambia de página de forma automática, algo realmente útil, ya que con la mayoría de instrumentos se tendrán las dos manos ocupadas.

Además del acompañamiento musical, también vamos a analizar el procesado en tiempo real que ello conlleva (considerando tiempo real como un tiempo de procesado muy corto en relación al ritmo en que la pieza musical avanza).

Una aplicación muy interesante respecto al procesado en tiempo real es Antescofo [3], la cual analiza la voz de un cantante y sintetiza la música que la acompaña. Antescofo funciona tanto en Mac OS X como Linux, y además, sus mismos creadores, han creado una aplicación (AscoGraph) con la que poder crear nuevas partituras para que sean procesadas acompañando musicalmente al cantante.

2. ESTADO DEL ARTE

El acompañamiento musical es la sincronización de un músico tocando una pieza musical conocida por el dispositivo con el que se sincroniza. A día de hoy el IRCAM (Institut de Recherche et Coordination Acoustique/Musique) lo sigue investigando, contando ya con más de 20 años de desarrollo [4].

El método que se emplea para realizar el acompañamiento de partitura está basado en el Modelo oculto de Márkov (Hidden Markov Model), modelo estadístico cuyo objetivo es, mediante probabilidad y parámetros observables (señal acústica), determinar cuáles serán los parámetros no conocidos (posición en partitura) u ocultos, de ahí su nombre.

La investigación sobre el acompañamiento musical se inició por iniciativa de Barry Vercoe y Lawrence Beauregard en 1983, y tomaron el relevo Miller Puckette y Philippe Manoury. Desde 1999 hasta hoy se sigue trabajando en conseguir que las aplicaciones de acompañamiento musical en tiempo real sean posibles con un cierto grado de robustez.

El principal objetivo de esta investigación es simular que un músico está tocando junto a otro. Uno de ellos, el músico real, tocará su instrumento mientras que su "acompañante virtual" va tocando junto a él una melodía que ambos conocen. De esta manera se puede cubrir la necesidad de un ensayo en grupo de forma individual, ya que con un simple audio de fondo existiría el problema de tener que adaptarse al tempo de ese audio. Con el acompañamiento musical es el propio músico el que, en su ensayo individual, marca el tempo con el que sus compañeros virtuales le acompañan.

Uno de los principales problemas que actualmente se plantean es la posibilidad de que el músico toque mal una nota, por lo que no puede basarse únicamente en la detección de notas para que el acompañante virtual también avance. El software tampoco podría basarse únicamente en el tempo con el que el músico real está tocando, ya que sería bastante complicado que a lo largo de la pieza musical, éste no varíe el tempo con el que toca.

Otro problema muy presente es la detección de varios instrumentos, o incluso, en algunos casos, instrumentos con características muy polifónicas que confunden al software. Para resolver esto, aparte de analizar el tempo, la detección de notas debería ir precedida por un análisis/corrección espectral para saber mejor con qué nota debería acompañar el acompañante virtual.

Actualmente y desde 2012, en el IRCAM, se lleva desarrollando Antescofo, con la colaboración del INRIA (Institute for Research in Computer Science and

Automation). Esta aplicación está disponible para Linux y OS X, aunque por el momento en versión beta, cuya última versión (v0.9) fue lanzada en noviembre de 2015. Antescofo es capaz de sintetizar una melodía que almacena en una partitura interna al mismo ritmo y acompañando a un cantante, cubriendo la necesidad antes mencionada. Sus creadores también han desarrollado una aplicación (AscoGraph) con la que poder crear nuevas partituras. Antescofo consiguió el Premio de Industria del ministerio de Industria de Francia en mayo de 2013, además de conseguir otro premio otorgado por una conocida revista francesa en 2011 (La Recherche).

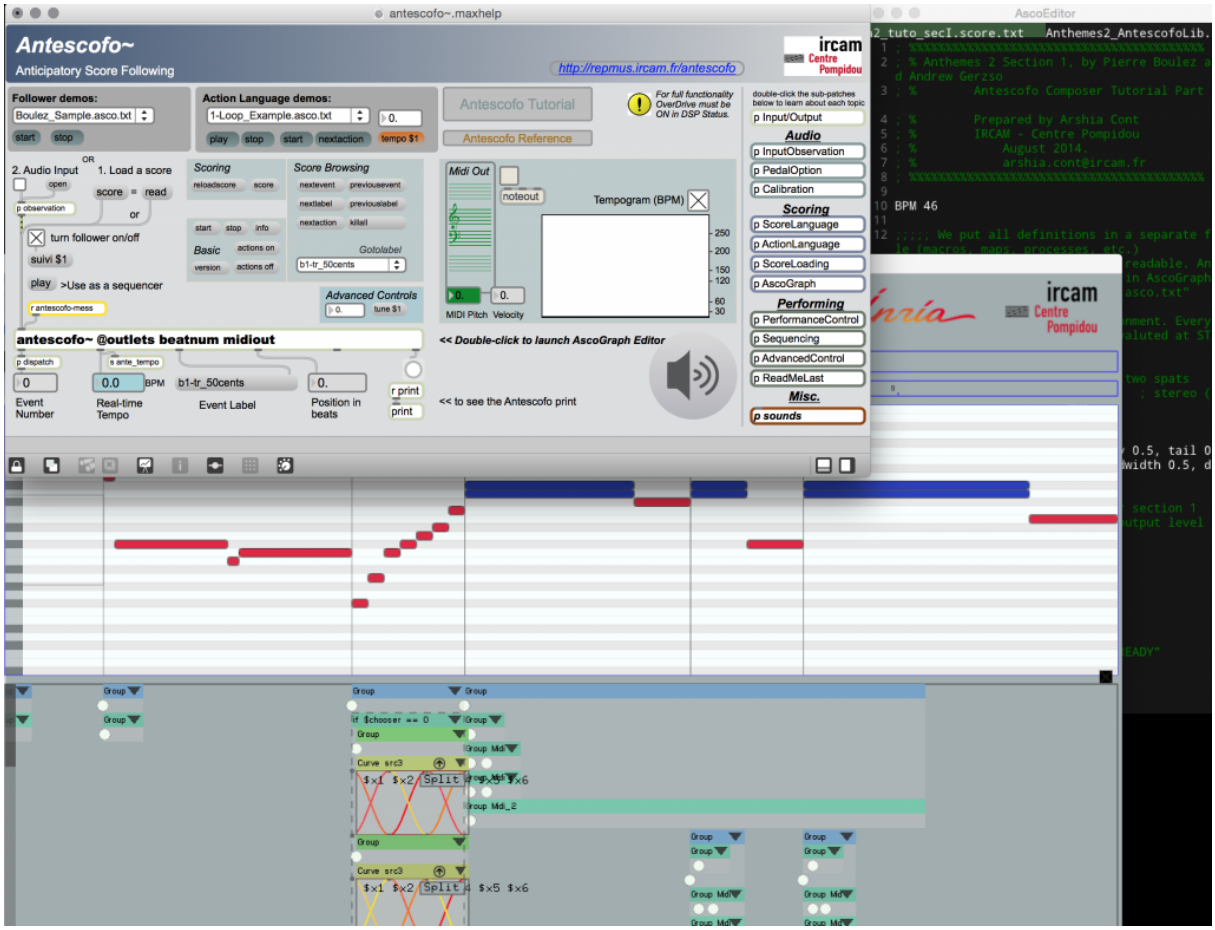

Fig.1 – Interfaz de Antescofo (arriba la señal que se sintetiza)

Otra aplicación que hace uso del acompañamiento musical (sin llegar a usar la parte de sintetizado) es Tonara, exclusiva de iOS (requiere versión 7.0 o superior). Esta aplicación toma el nombre de su equipo homólogo, el cual se fundó en 2008 en Israel por Yair Lavi y Evgeni Begelfor. Actualmente tienen disponibles su aplicación principal (Tonara) y una variante a modo de juego llamada Wolfie (requiere iOS 8.0 o superior), ambas disponibles en la App Store de Apple para iPad. La aplicación Tonara tiene un largo repertorio en su tienda virtual, en la cual se pueden encontrar piezas de Beethoven, Chopin, J.S. Bach, etc, cuyos precios oscilan entre los 0,99 y los 2,99 euros. En noviembre de 2011 pusieron en práctica su primera versión de Tonara en

Square Park de Washington, donde dos pianistas y un violinista tocaron un largo repertorio de piezas cuyas partituras están disponibles en la aplicación. Esta aplicación fue muy aclamada por los medios, y fue “Aplicación de la semana” en la App Store en octubre de 2011 en países como China, Alemania, Austria y Suiza.

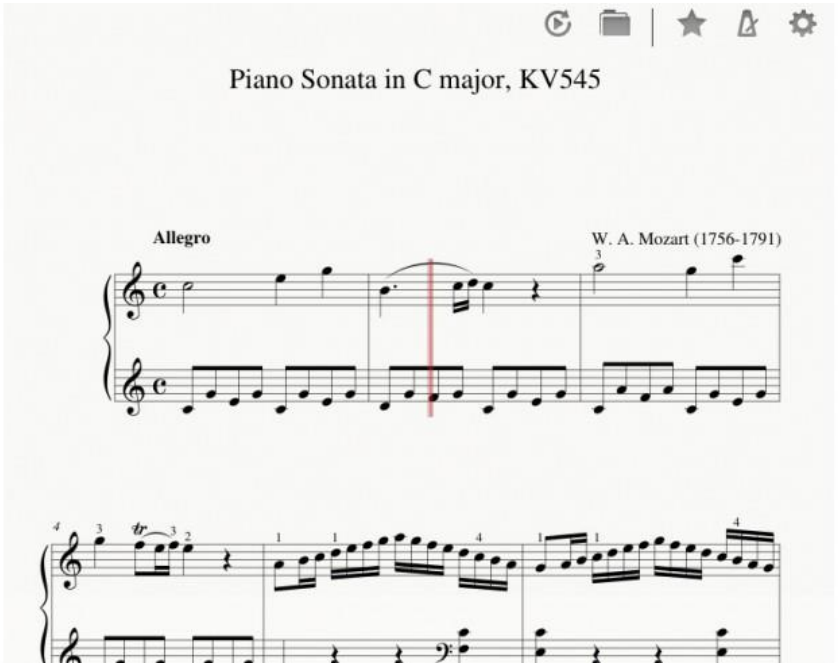

Fig.2 – Interfaz de Tonara (la línea roja marca la posición de avance en partitura)

En cuanto a la separación espectral de fuentes sonoras, un referente en el mercado es ADX TRAX (by Audionamix) [5]. Esta aplicación es capaz de separar la melodía de la voz en la mayoría de pistas musicales que existen, realmente útil para el ámbito musical actual. Este tipo de separación es vital para versionar cualquier tipo de canción, sobre todo si lo único que se tiene es la versión final de esa canción. De esta forma se puede tener de forma separada por un lado la música, y por otro lado la voz a capela. A partir de esas pistas individuales, el productor musical puede utilizarlas, por ejemplo, para poner la voz de ese cantante a otra música y viceversa. El problema con el que se puede encontrar un productor a la hora de emplear estas pistas, aparte de los derechos de autor, es la incorrecta separación espectral, o que al hacerlo, la calidad no sea del todo adecuada, requiriendo un procesado posterior para conseguir una separación totalmente limpia.

Actualmente tienen disponibles la versión estándar (ADX TRAX 3) por un precio total de 299 dólares o 19,99 dólares al mes y la versión pro (ADX TRAX PRO 3) cuyo precio es algo mayor (499 dólares o 32,99 dólares al mes). Ambas son exclusivas para OS X con versión entre 10.9 y 10.12 (aunque incluyen la versión 2.5, compatible con OS X 10.7 y 10.8), y sus requisitos son una CPU de dos núcleos a 2.3GHz y, en cuanto a memoria RAM, la versión estándar necesita 2GB, y la pro 4GB.

En su página web se pueden visualizar vídeos de ejemplo de algunas canciones conocidas (http://audionamix.com/separation-demos/). A continuación se muestra el espectro de la separación de la voz y la melodía de la canción *House of the rising sung* (1964) del grupo *The Animals*:

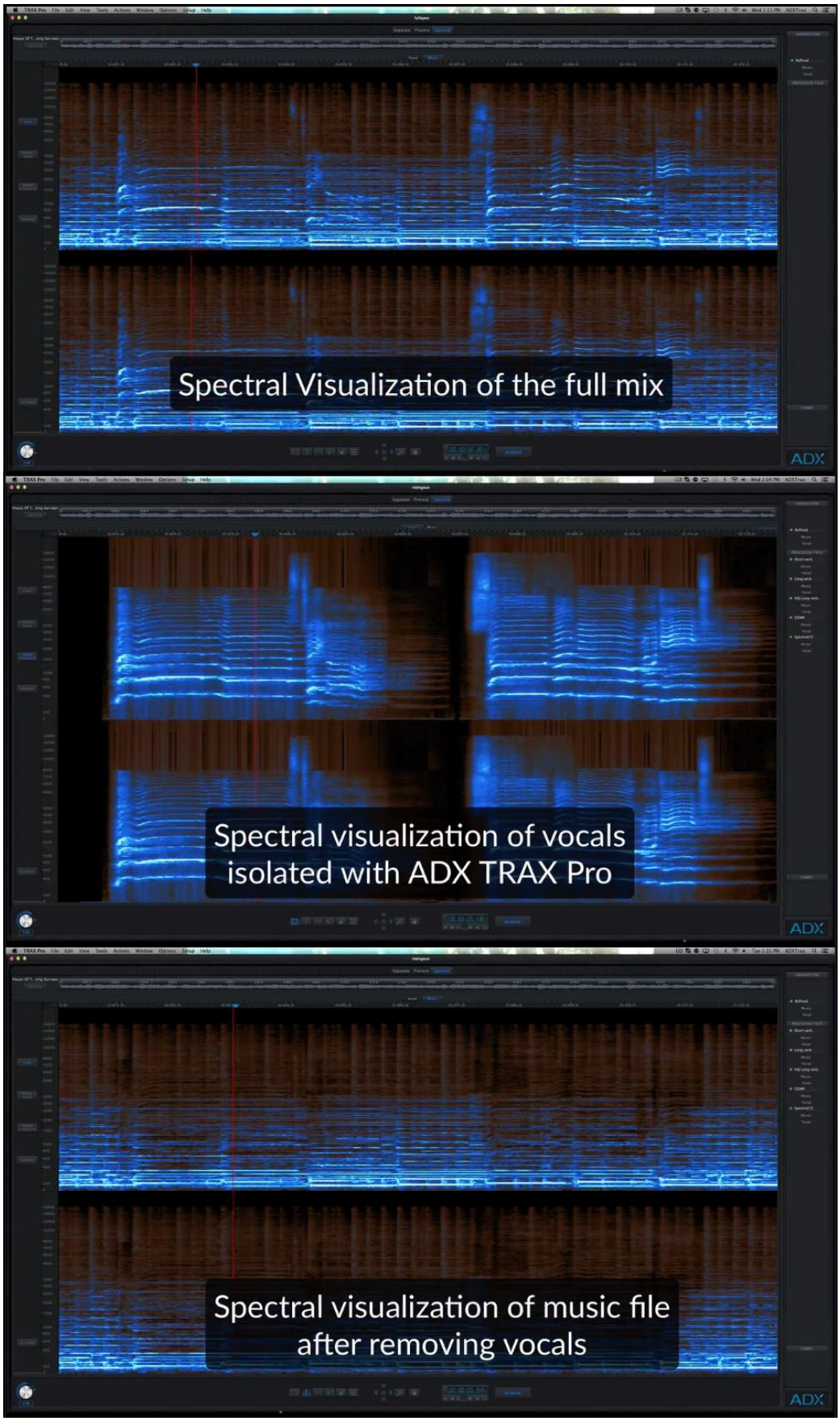

Fig.3 – Espectrograma de la señal original y las dos de salida

3. OBJETIVOS

El objetivo principal marcado por este Trabajo de Fin de Grado es una aplicación que procese audio en tiempo real para alineamiento musical o separación de fuentes sonoras y que se ejecute en varios dispositivos con procesador multicore.

La metodología a seguir es la siguiente:

1. Uso en Matlab de herramientas de procesado de señal orientadas a separación y/o alineamiento.
2. Creación de una librería de procesado en tiempo real en C.
3. Programación en C sobre GPU de la librería.
4. Instalación de diferentes dispositivos para dar respuesta en tiempo real.
5. Pruebas y promoción de la aplicación final.

Dada la complejidad de la programación sobre GPU, el punto número 3 de la metodología no ha podido ser cumplido.

Para satisfacer el resto de puntos se ha empezado creando una función en Matlab que aplicaba un filtrado de audio (captura-filtra-reproduce), el cual no ha podido considerarse tiempo real.

A continuación se ha adaptado este código de Matlab a C para aprovechar su potencial y conseguir un ecualizado en tiempo real de lo capturado por micrófono.

La siguiente etapa fue la modificación de una aplicación proporcionada por el departamento de Software de la Universidad de Oviedo para añadir nuevas opciones que proporcionen acompañamiento musical en tiempo real recibido por entrada microfónica (de forma similar a lo que hace la aplicación de iOS Tonara).

Esta aplicación se ha ejecutado en dos dispositivos diferentes; un kit de desarrollo NVIDIA Jetson TK1 y un ordenador portátil. En ambos dispositivos se han ejecutado una batería de pruebas, haciendo una comparación en la utilización de un compilador con optimización en el paralelizado de procesos (ICC – Intel C++ Compiler) y el tradicional de Linux (GCC – GNU Compiler Collection) en el caso del ordenador portátil.

Para compensar el tercer objetivo se realizarán pruebas adicionales a la aplicación principal de este Trabajo (acompañamiento musical). Estas pruebas consistirán en la realización de pruebas con el procesador en estado de estrés e introduciendo como fuente sonora la pieza musical original tanto acelerada como ralentizada.

De estas pruebas hemos obtenido unas conclusiones las cuales han sido planteadas en su apartado correspondiente.

4. MATERIALES Y MÉTODOS

Para la ejecución de la aplicación de procesado de audio se han utilizado dos dispositivos diferentes:

4.1 Kit de desarrollo NVIDIA Jetson TK1

Este dispositivo cuenta con un SoC (System on-Chip) formado por una CPU de 4+1 núcleos a 2.3GHz con arquitectura ARM Cortex A15 y una GPU de 192 núcleos CUDA. El SoC completo es capaz de alcanzar los 326 GFLOPS (Giga flops – floating point operations per second), mientras que la CPU solo es capaz de lograr los 76 MFLOPS. En cuanto a memoria, consta de 2GB de RAM de dos canales de 32-bit y 16GB de almacenamiento interno (tipo eMMC). Este kit incluye varios puertos que lo convierten en un mini-ordenador, pero su punto fuerte (y lo cual lo convierte en un kit de desarrollo) son los puertos de expansión que proporcionan señales del tipo LVDS, GPIO, UART, i2c, HSIC y Touch SPI/CSI-2 [6] [7] [8]. Para el propósito para el cual usaremos este kit nos interesa sobre todo la salida HDMI y la entrada de señal microfónica de 3,5mm (mini-jack). La competencia de este kit de desarrollo es muy amplio, sobre todo por el precio y disponibilidad, ya que dispositivos como Raspberri o Arduino tienen una accesibilidad mucho mayor. El coste de este Jetson ronda los 200 dólares americanos (en el mercado europeo se podía encontrar por unos 250 euros). Sus competidores tienen un precio mucho menor, pero ninguno de ellos combina la potencia de la CPU+GPU que el SoC de NVIDIA posee.

Fig.4 – Kit de desarrollo NVIDIA Jetson TK1

4.2 Ordenador portátil Lenovo ThinkPad 11E

Este pequeño ordenador portátil posee un procesador Intel Celeron N2920 de cuatro núcleos a 1.86GHz con arquitectura de 64bit. Posee una tarjeta gráfica integrada en el procesador Intel HD Graphics, 4GB de memoria RAM a 1600MHz y un disco duro mecánico de 320GB, el sistema completo es capaz de alcanzar los 25,6 GFLOPS, y su precio ronda los 380 euros. En cuanto a la competencia de este dispositivo es infinita, ya que este modelo en concreto no destaca por su potencia de procesado a pesar de poseer un procesador de cuatro núcleos (destaca por su robustez y compactibilidad) [8] [9]. Por el mismo precio pueden encontrarse algunos modelos de portátiles con tamaños más estandarizados (15,6 pulgadas de pantalla) con procesadores Intel Core i3 e i5, los cuales también tienen series de bajo consumo pero que a pesar de ello logran potencias mucho mayores a las de este Intel Celeron.

Ambos dispositivos funcionan bajo un sistema operativo basado en Linux. La versión del kit de desarrollo Jetson es una versión escueta de Ubuntu para poder ser ejecutada por procesadores ARM. Esta versión se instala al mismo tiempo que se *flashea* (método de *flasheo* detallado en el Anexo 8.1). La versión utilizada en el ordenador portátil es Xubuntu, una versión más ligera que Ubuntu y con un entorno gráfico diferente (mientras que Ubuntu usa Unity, Xubuntu usa XFCE, ambos basados en GNOME).

En cuanto al software adicional que se necesita (librerías fft, aplicación de reproducción de sonido en bruto, aplicación de acompañamiento musical, función de estrés, compilador ICC y API de Alsa), en el Anexo 8.2 se explica con detalle cómo se instala en ambos dispositivos.

Para la aplicación principal de este Trabajo se han aplicado dos métodos diferentes para su evaluación en los dos dispositivos:

La primera es la ejecución que simula el propósito fundamental de esta aplicación, es decir, el acompañamiento musical de un músico (de ahora en adelante **ReadMic**). Para ello se va a realizar una serie de ejecuciones que simularán que un músico toca la melodía almacenada en partitura de la aplicación. Esto se simula reproduciendo esta melodía con un dispositivo externo como puede ser un teléfono móvil con reproductor de música. Esta melodía la captará el dispositivo procesador (Jetson o portátil) y la procesará para ir indicando la posición de la partitura por la que detecta que el músico (reproductor externo) lleva su progreso. Estas pruebas serán realizadas con unos tiempos de escucha de 150 y 300 segundos, para valores de β igual a 0, 1 y 2 los cuales caracterizan una función de coste distinta aplicada a la DTW (más detalles en el Anexo 8.4), y tratando con variables tipo *float* (4 bytes) y *double* (8 bytes). El valor de salida será la posición final que ha alcanzado el músico.

La segunda ejecución (de ahora en adelante **ReadFrame**) simulará el concepto anterior pero de forma interna, es decir, en vez de captar la señal sonora por micrófono durante un tiempo determinado (150 y 300 segundos) ahora se procesarán todas las tramas correspondientes a esos tiempos de forma continua, cogiendo la señal de entrada directamente de fichero. El número de tramas correspondientes a cada tiempo se indica en el apartado de Desarrollo. La idea de esta batería de pruebas (con los mismos valores de β, y los mismos tipos de variables que en el caso anterior) es comprobar el tiempo total que se emplea para procesar toda la partitura correspondiente al tiempo indicado, por lo que el valor de salida de estas ejecuciones va a ser el tiempo en segundos que ha tardado la aplicación en procesar esta partitura.

Estas dos ejecuciones se realizan ambas bajo el compilado GCC nativo de Linux (GNU Compiler Collection). En el caso del portátil también se van a hacer estas ejecuciones con el compilador **ICC** (Intel C++ Compiler) [10], el cual optimiza el paralelizado de procesos indicados en el código con las sentencias prefijadas por `#pragma omp` cuando se compila con alguno de los flags de OpenMP (conjunto de directivas del compilador ICC que habilitan el multiproceso paralelo [11]). En el apartado de Resultados se comentarán las ventajas de su uso.

Aparte de la batería de ejecuciones de la aplicación principal, se van a realizar dos pruebas adicionales que hacen conocer mejor el funcionamiento y los límites de esta aplicación. Estas dos nuevas pruebas consisten en reproducir el archivo de audio de forma **acelerada** y **ralentizada** para el método ReadMic, y en someter al procesador a un **estado de estrés** mientras ejecuta la aplicación (tanto ReadMic como ReadFrame). Se observará la comparación en las muestras totales alcanzadas en la partitura para el primer caso y veremos qué ocurre cuando el procesador trabaja a pleno rendimiento y no dispone de toda esa potencia para el procesado. Todo esto se explica con detalle en el apartado de Resultados.

Fig.5 – Lenovo ThinkPad 11E

5. DESARROLLO

Un **filtro de audio** consiste en la atenuación de ciertas bandas de frecuencias para obtener una señal con unas características espectrales deseadas. Se podría considerar que existen 4 tipos de filtros; los paso-bajo (LPF – Low-Pass Filter) los que permiten el paso de la zona grave del espectro hasta cierta frecuencia de corte (fc), los filtros paso-alto (HPF – High-Pass Filter) los que, al contrario que los LPF, permiten el paso de la zona aguda del espectro a partir de la frecuencia de corte, los paso-banda (PB – Pass Band) cuya salida está comprendida entre un determinado ancho de banda (fmin - fmax), y por último los elimina-banda (BSF – Band-Stop Filter) los cuales son el caso inverso a los paso-banda, permitiendo el paso de todo el espectro exceptuando el ancho de banda comprendido entre las dos frecuencias especificadas.

A diferencia del filtrado de audio, el **acompañamiento musical en partitura** es algo mucho más reciente. Consiste en, mediante comparación de muestras de audio y una partitura virtual, ir hallando la posición en ésta por la que el músico va tocando esa misma melodía. Esto es capaz gracias al Alineamiento Dinámico Temporal (DTW – Dynamic Time Warping), cuya explicación se puede consultar en el Anexo 8.4. A grandes rasgos, este alineamiento compara la señal de entrada con una señal interna cuyo aprendizaje ha debido hacerse previamente a la etapa de acompañamiento [12] [13].

5.1 Ecualizador Matlab

En primer lugar vamos a explicar el desarrollo de una aplicación (código adjunto a la memoria) que satisface el primero de los objetivos de este Trabajo. La primera idea de esta aplicación hecha con Matlab consiste en un ecualizador de audio en tiempo real, el cual aplica un filtrado paso bajo, baso banda, elimina banda o paso alto permitiéndonos seleccionar las frecuencias de corte. Desarrollando un poco más el código se podría lograr plantear unos ecualizados predefinidos para realzar o atenuar las frecuencias deseadas, pero como veremos a continuación, este tipo de aplicación no es viable con Matlab, por lo que posteriormente se va a plantear un sistema similar programado en C.

Fig.6 – Diagrama de flujo de datos de ecualizador.m

Antes que nada hay que conocer cómo se consigue realizar ese filtrado. La transformada DFT (Discrete Fourier Transform) es crucial para lograr obtener la señal en frecuencia, al igual que la IDFT (Inverse DFT) lo es para volver a transformar la señal a dominio temporal.

La transformada DFT de N puntos (ecuación 1), convierte la secuencia en dominio temporal discreto x[n] en otra secuencia discreta X[k] en dominio de la frecuencia, ambas de N muestras.

$$X[k] = \sum_{n=0}^{N-1} x[n] * e^{-j\frac{2\pi}{N}kn}, \;\; k = 0,1, \dots (N-1) \tag{1}$$

El cálculo de su inversa (IDFT, ecuación 2) revierte la transformación DFT, es decir, siendo x[n] una señal discreta de N muestras, si a la $DFT_N(x[n])$ se le aplica la $IDFT_N$ (ecuación 3), volvemos a obtener la señal original x[n] (de longitud N).

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k) * e^{j\frac{2\pi}{N}kn}, \;\; n = 0,1, \dots (N-1) \tag{2}$$

$$x(n) = IDFT_N(DFT_N(x[n])) \tag{3}$$

Si a la señal de entrada x[n] le añadiésemos al final M ceros (teniendo en este caso un total de N+M muestras), su transformada DFT_{N+M} tendría una mejor interpolación entre muestras (no añade información real, pero la señal en dominio de la frecuencia se suaviza).

Debido a que para aplicar la DFT a una señal debemos conocerla por completo (existiría un retardo como mínimo de la duración de la señal de entrada), lo que vamos a hacer será dividirla en bloques e irlas procesando por separado (siendo esta vez el retardo mucho menor y reduciendo la complejidad de la transformada). A este segmento de señal, la primera transformación que se le hará será un enventanado con la función de Hann (también conocida como Hanning) para suavizar los extremos, a continuación se le aplicará la FFT (Fast Fourier Transform, algoritmo que permite el cálculo de la DFT de forma muy eficiente), se hará una multiplicación en frecuencia con el filtro que queramos aplicar y por último se aplicará la IFFT (FFT inversa), obteniendo una copia de la señal de entrada pero ecualizada a razón del filtro aplicado. Este proceso se realizará para todos los segmentos en que se divida la señal de entrada, y no habría límite de tiempo total para la señal de entrada completa. En el caso de que se añadan ceros (hasta la siguiente potencia de 2) a cada bloque para mejorar la interpolación habría que realizar un solapamiento-suma (overlap-add) para que la señal de salida siga teniendo la misma longitud que la de entrada y además se complete la información obtenida por la FFT. En caso de no haber rellenado con ceros se habría utilizado el método overlap-save (más eficiente al no necesitar sumar fragmentos de tramas).

El siguiente esquema da más detalles sobre el método overlap-add:

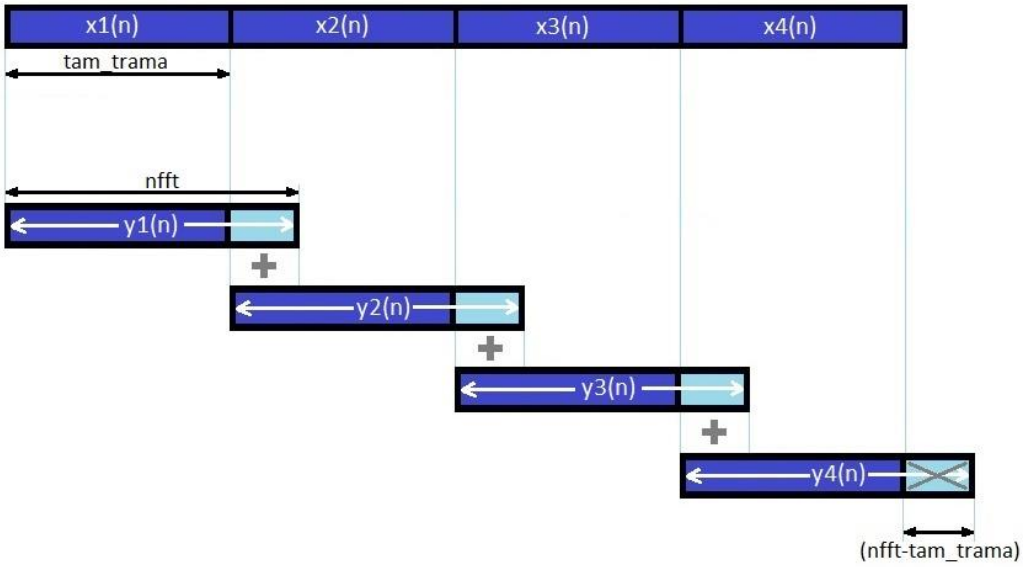

Fig.7 – Método de transformada DFT con overlap-add (solapamiento-suma)

Correspondiendo *tam_trama* al número de muestras que forman un bloque en dominio temporal, y *nfft* el tamaño del mismo bloque completado con ceros (coincidiendo con el tamaño de la transformada FFT), las últimas *nfft-tam_trama* muestras de cada bloque de salida se sumarían con las *nfft-tam_trama* primeras muestras del bloque siguiente, a excepción de las del último bloque que se desecharían.

Para facilitar su comprensión se puede consultar el diagrama de flujo que aparece al inicio de este apartado (Figura 6).

La función en Matlab se ha nombrado como "`ecualizador.m`" y su modo de uso es el siguiente:

```
[x_rec_wav] = ecualizador(tiempo, min_f, max_f, tipo);
```

Siendo la salida `x_rec_wav` las muestras en forma temporal de la señal filtrada, la entrada tiempo la duración de la captura de audio, `min_f` la frecuencia mínima del filtro, `max_f` la frecuencia máxima, y tipo (cuyo valor puede ser 1 o 0) seleccionará el tipo de filtrado que haremos a la banda de frecuencias comprendida entre `min_f` y `max_f`, correspondiendo al valor seleccionado. Por ejemplo, si seleccionamos `tipo=1` el filtrado eliminará todas las frecuencias a excepción de las comprendidas entre `min_f` y `max_f` (paso bajo, paso alto o paso banda). En caso de que seleccionemos `tipo=0` el filtro dejará intacto todo el espectro a excepción de la banda comprendida entre `min_f` y `max_f` (elimina banda). Para el uso del filtrado paso bajo deberá seleccionarse la frecuencia mínima como 0, la máxima como la frecuencia de corte y el tipo 1. Para el filtro paso alto pasa al contrario, debiendo seleccionar la frecuencia mínima como frecuencia de corte, la máxima como 20kHz y tipo 1. A continuación se muestran algunos ejemplos de ejecución:

```
[salida]=ecualizador(5,0,4000,1); % Filtro paso bajo con fc=4kHz y 5s.

[salida]=ecualizador(5,16000,20000,1); % Filtro paso alto con fc=16kHz.

[salida]=ecualizador(5,400,4000,1); % Filtro paso banda entre 400Hz y 4kHz.

[salida]=ecualizador(5,15500,16500,0); % Filtro elimina banda 15,5 a 16,5kHz.
```

Los parámetros utilizados internamente en esta función son una frecuencia de muestreo (fs) de 44100Hz, ya que según el teorema de Nyquist-Shannon (o Teorema del muestreo) el ancho de banda debe ser mayor o igual al doble de la frecuencia máxima para que pueda recuperarse correctamente. La variable L (número de muestras por trama) se ha establecido en 2940 (una quinceava parte de fs, para facilitar el procesado). Se ha capturado audio de un solo canal por estar utilizando un micrófono mono. La longitud de la FFT debería haberse asignado a la siguiente potencia de 2, pero por facilidad de implementación se ha establecido de la misma longitud de cada trama (la siguiente potencia de 2 ascendía casi al doble), y para que de esta forma el espectro no tenga demasiadas muestras que representar.

El problema que nos encontramos ante esta aplicación es su inviabilidad de uso debido a que Matlab ejecuta su código de forma secuencial, es decir, hasta que no ha terminado de ejecutar una línea no pasa a ejecutar la siguiente, y esto hace que se reproduzcan una sucesión de silencios que hacen totalmente incomprensible el fichero de salida. Esto se debe a que mientras la trama capturada por micrófono se está procesando, el micrófono está parado. Aun así, esta función resulta útil para analizar, al menos, de forma visual el espectro que capta el micrófono, la forma de su señal, y comprobar si efectivamente realiza un filtrado seleccionado.

A continuación se muestran dos casos de ejecución en los que se aplica un filtro elimina banda entre 1 y 2kHz y en los cuales se reproduce un tono de 1,5kHz (Figura 8) y 2,5kHz (Figura 9).

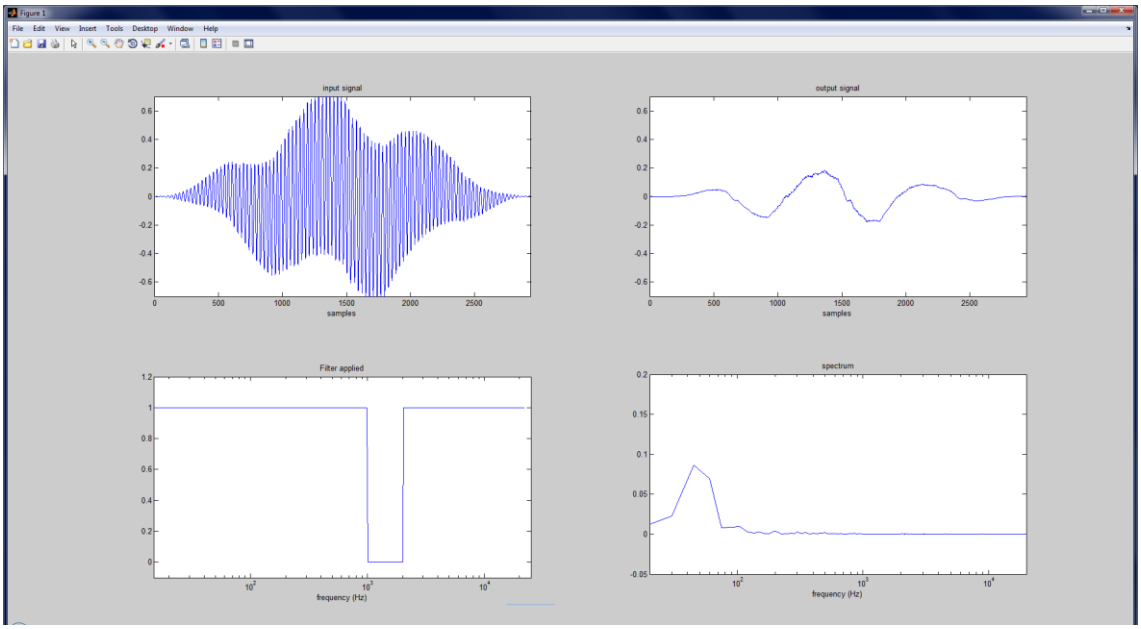

Fig.8 – Se observa que se ha eliminado el tono a 1,5kHz

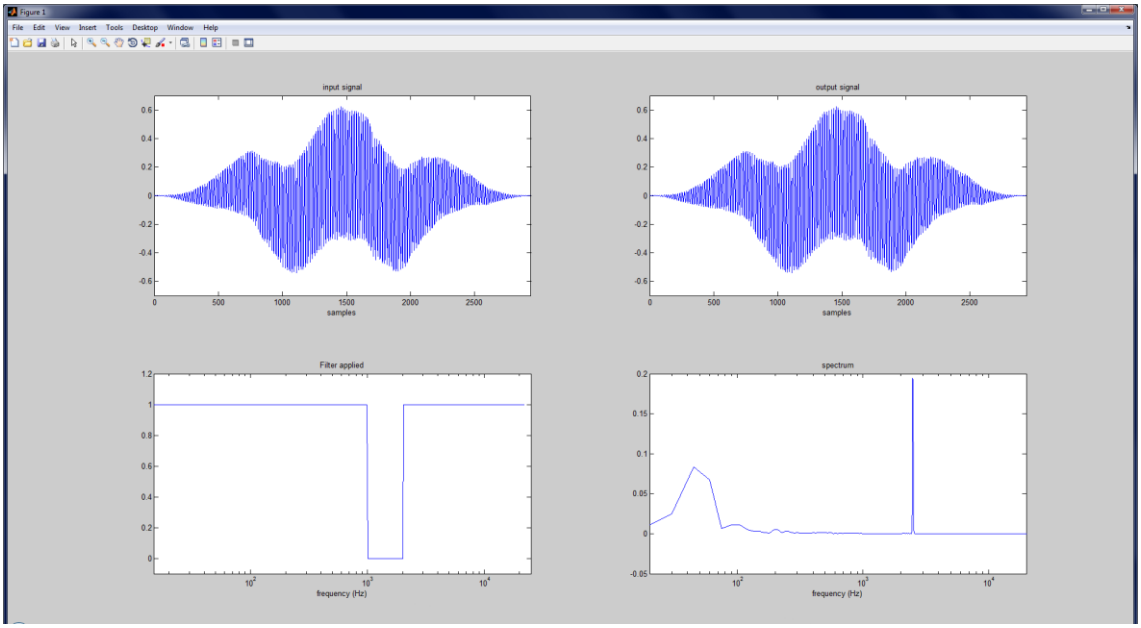

Fig.9 – Se observa que el tono a 2,5kHz se ha conservado

Ahora se muestra un diagrama de bloques de la ejecución de la función con las entradas y salidas de cada bloque y las funciones principales utilizadas.

5.2 Ecualizador en C

La idea de esta aplicación es una adaptación a C del código del apartado anterior, para aprovechar la potencia de este lenguaje de programación y conseguir solventar los problemas que hemos tenido en Matlab, y ya de paso conseguir cumplir el segundo objetivo de este Trabajo. Su código se puede hallar adjunto a la memoria.

Para ello, antes vamos hablar sobre la API (Application Programming Interface) ALSA (Advanced Linux Sound Architecture) [14], gracias a la cual esto es posible. Su instalación está detallada en el Anexo 8.2.3.

Unas de las principales características de esta potente herramienta son el soporte multiprocesador y operación full-dúplex (en teoría), aunque una peculiaridad que en cierto momento del desarrollo de este Trabajo causó varios quebraderos de cabeza es que solo captura audio en estéreo (como vamos a procesar audio en mono la solución fue convertirla a mono una vez capturada en estéreo).

En su página web (www.alsa-project.org) podemos encontrar toda la información necesaria para trabajar con esta API. Nosotros nos vamos a centrar en el módulo PCM Interface, cuyas funciones nos van a permitir controlar la interfaz de grabación/reproducción de la tarjeta de sonido del dispositivo con el que estemos trabajando (Jetson o portátil, esto hace que se cumpla el cuarto objetivo; dar respuesta en tiempo real en dispositivos diferentes). Todas las funciones utilizadas pertenecientes a esta API se anotan y se detalla su funcionamiento aparecerán más adelante, cuando expliquemos el flujo de entrada y salida de cada uno de los bloques de procesado de esta aplicación de acompañamiento musical.

Esta aplicación se encarga de capturar audio por el micrófono y reproducirlo por la salida digital (conector HDMI) habiéndole aplicado el filtro especificado por el usuario.

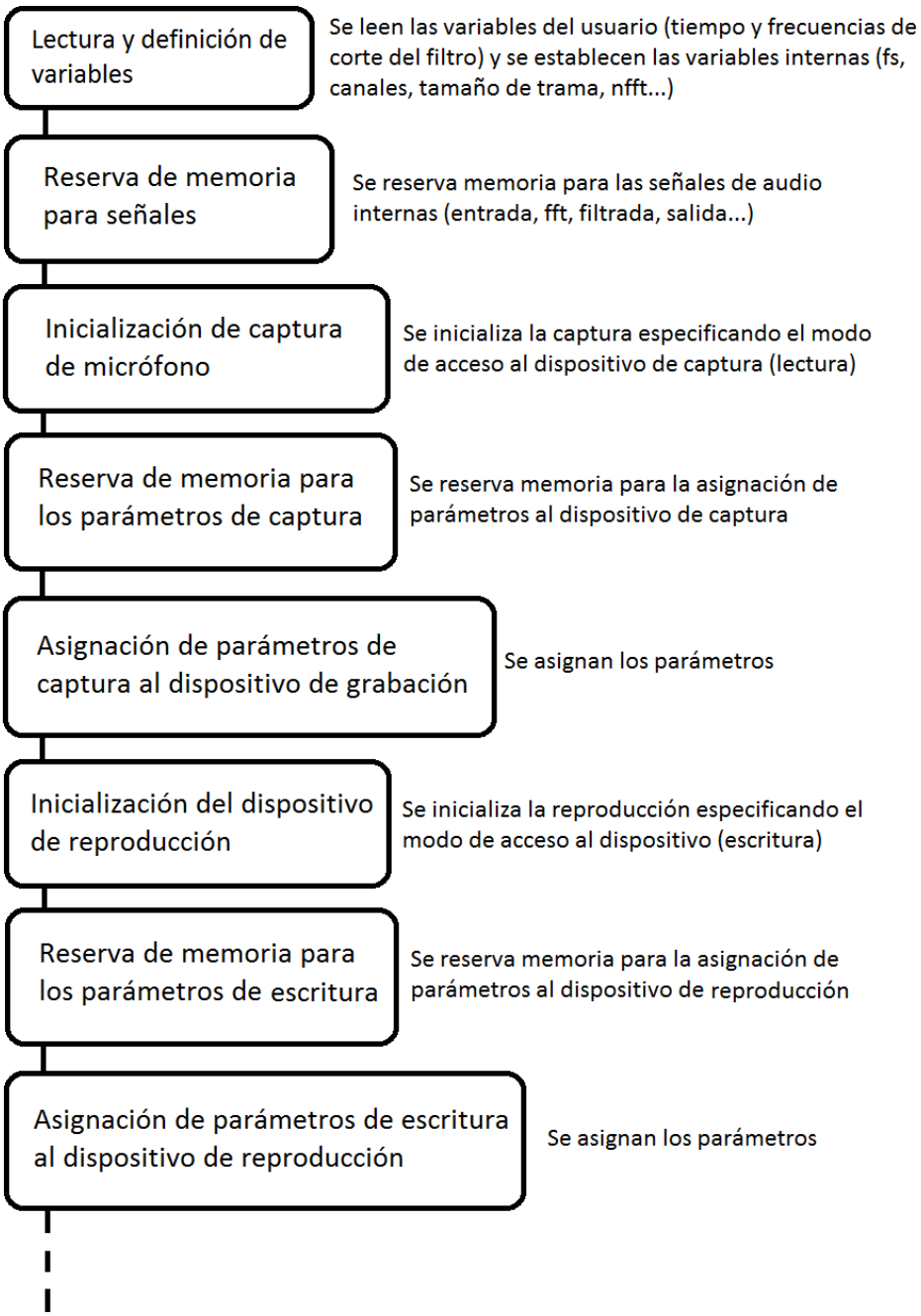

Fig.10 – Primera parte del diagrama de bloques del Ecualizador en C

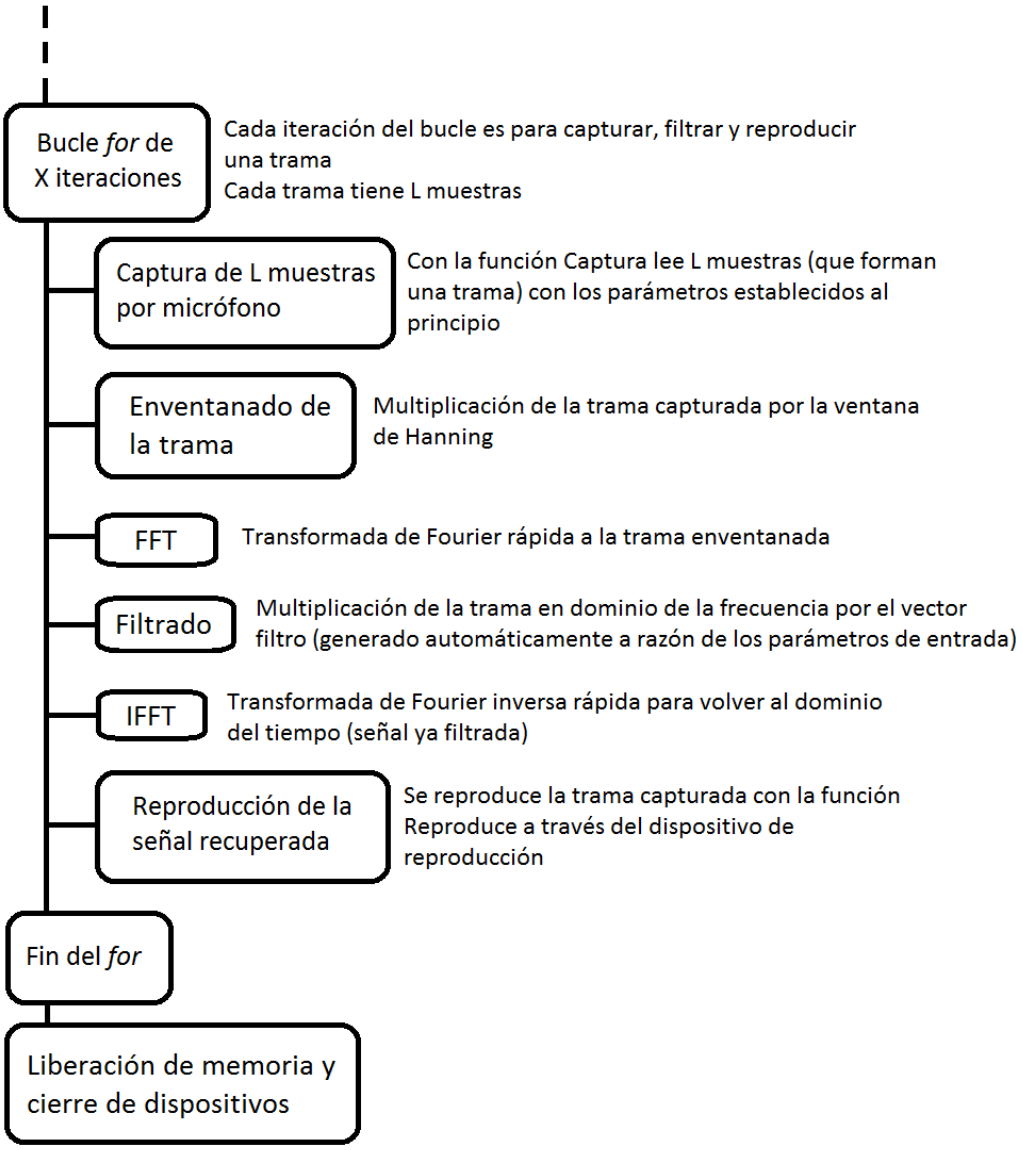

Fig.11 – Segunda parte del diagrama de bloques del Ecualizador en C

Al igual que en el caso anterior, se va a utilizar el método overlap-add (Figura 7) y en la captura y escritura de tramas se va a implementar un buffer circular. La primera trama será de 2048 muestras nuevas, y en las siguientes iteraciones se van a capturar 1024 muestras nuevas y se irán concatenando con las 1024 anteriores (teniendo el total de 2048 que forman una trama completa):

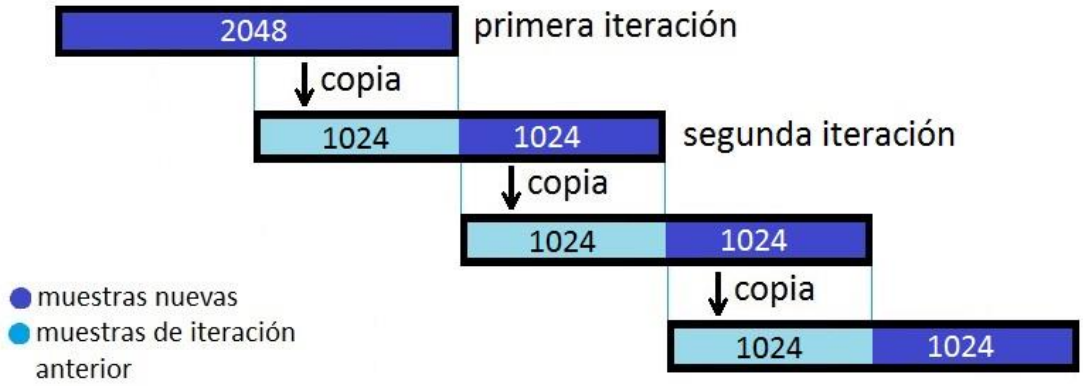

Fig.12 – Buffer circular 50%

A continuación se muestra un diagrama de bloques que representa el flujo de datos de la aplicación:

Para ejecutar la aplicación, en primer lugar debemos estar ubicados en la carpeta donde se encuentran los ficheros de la aplicación (haciendo uso del comando cd en el Terminal de Linux). Una vez estemos en la carpeta que contiene los ficheros filtro.c y funciones.h compilamos la aplicación con una de las siguientes sentencias, dependiendo del hardware desde el que la estemos compilando:

Jetson TK1:

```
gcc filtro.c -o filtro -I./ -I/opt/fftwf/include -L/opt/fftwf/lib
-lfftw3f -lasound –lm
```

Lenovo 11E:

```
gcc filtro.c -o filtro -lfftw3f -lasound -lm
```

El flag `-lasound` es para hacer uso de la API de Alsa, `-lm` es para la utilización de funciones matemáticas, y `-lfftw3f` para la utilización de la librería FFTW con variables float [16]. En el caso del Jetson TK1 hay que especificar de forma forzada la ubicación de esta última librería, ya que al tratarse de una instalación manual de la FFTW, el Sistema Operativo no conoce su ruta.

El modo de lanzar la aplicación es muy similar a la versión de Matlab. Se especifica el valor en segundos que queremos estar haciendo uso del filtro en tiempo real, frecuencia mínima y máxima y el tipo de filtrado que se desea aplicar. Para utilizar un filtrado elimina banda se usará el tipo 0, lo que hará que se elimine la información de señal entre las dos frecuencias especificadas. Si se usa el tipo 1 eliminará todas las frecuencias excepto las comprendidas entre estas dos frecuencias. Un ejemplo de ejecución (para un funcionamiento de 30 segundos y un filtrado paso banda entre 400 y 4400 Hz) es:

```
./filtro 30 400 4400 1
```

En la siguiente figura vamos a observar un ejemplo de la pantalla que se ve cuando la aplicación está en ejecución ha finalizado:

Fig.13 – Ejecución del filtro finalizada

En este ejemplo se puede comprobar que se han capturado y reproducido un total de 1291 tramas (desde la 0 a la 1290), correspondiendo con los 30 segundos introducidos de tiempo total:

$$num\_tramas = \frac{fs * t\_en\_segundos}{muestras\_por\_trama} = \frac{44100 * 30}{1024} = 1291\ tramas \quad (4)$$

La aplicación además incorpora unas líneas de código para exportar la señal de entrada y la de salida a un fichero .pcm. Con el Audacity [15] podemos observar las dos señales en dominio temporal:

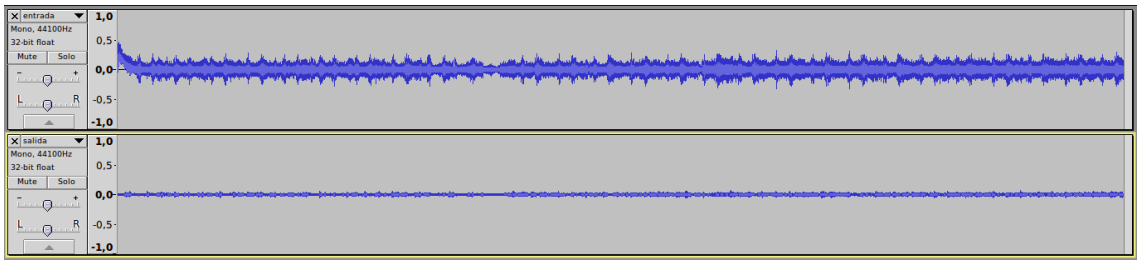

Fig.14 – Arriba la señal de entrada, abajo la señal de salida

El audio capturado se trata de música de fondo mientras además se reproducía un tono de 5kHz a un alto nivel. En el espectro de ambas señales veremos la diferencia:

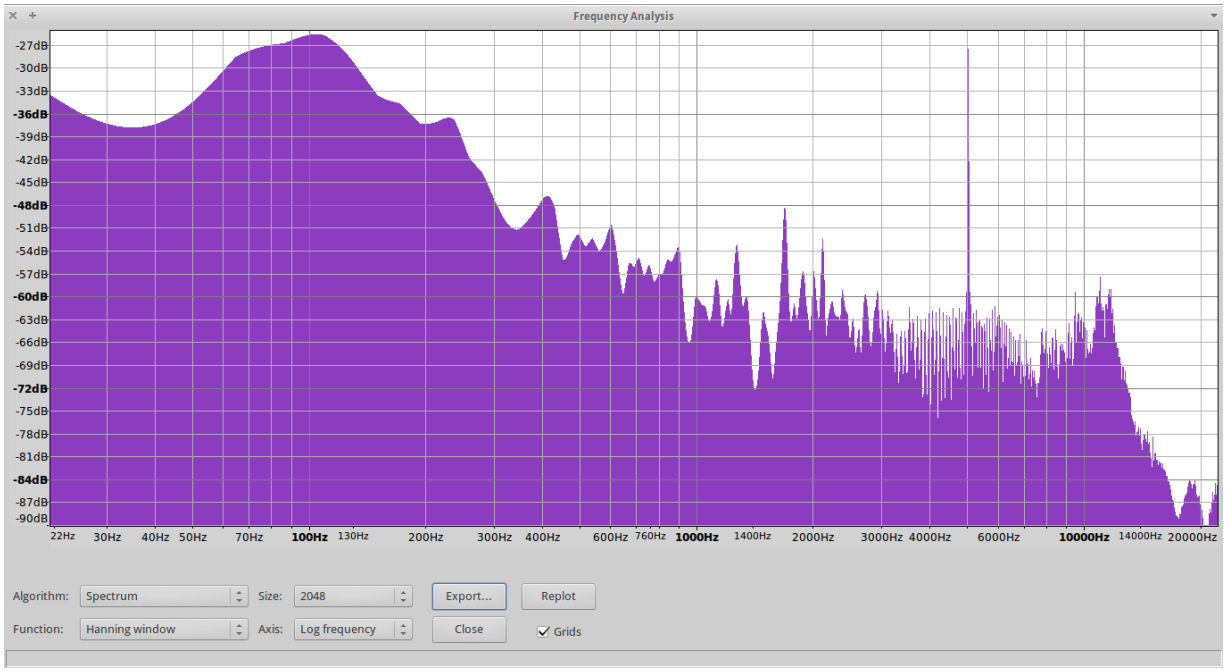

Fig.15 – Espectro de la señal de entrada

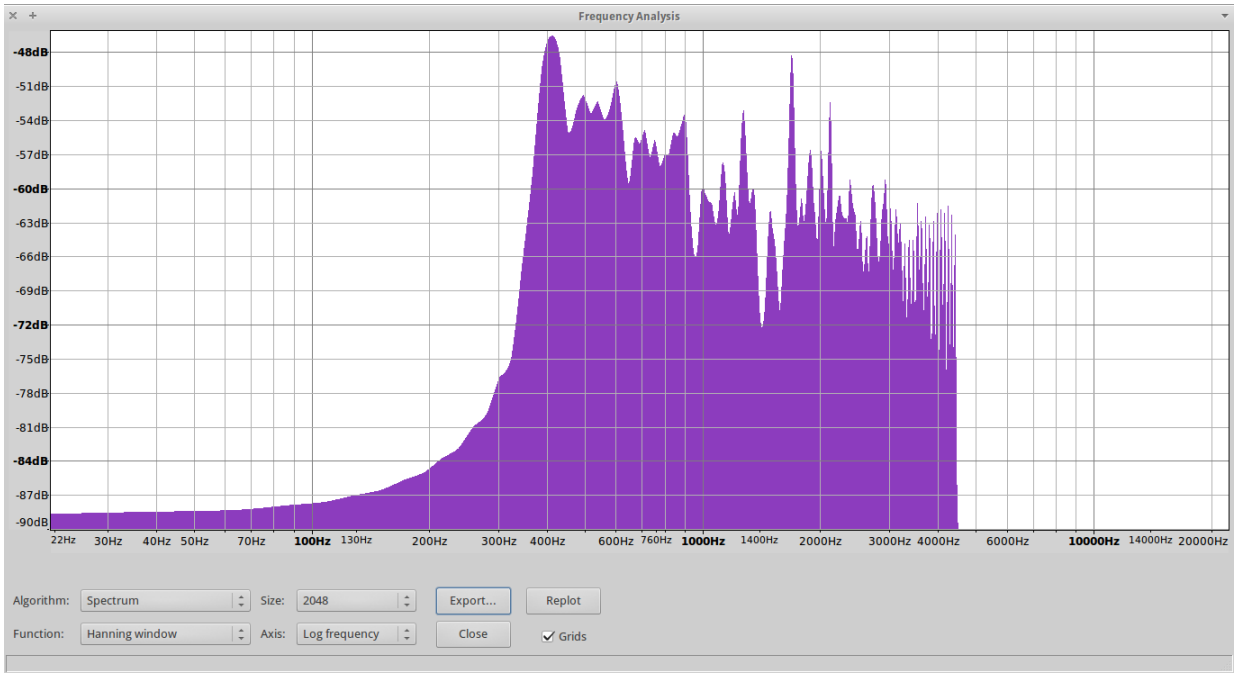

Fig.16 – Espectro de la señal de salida (filtro paso banda 400-4400Hz aplicado)

En la Figura 15 se puede apreciar el tono a 5kHz, y en la Figura 16 cómo se han eliminado las frecuencias que quedan fuera del rango especificado (por haber utilizado el tipo 1).

Los resultados obtenidos en varias pruebas que se han realizado son satisfactorios, teniendo como única limitación la caída del filtro y su ancho de banda mínimo (Δf). Esta ancho de banda mínimo está limitado por el tamaño utilizado en la FFT, ya que el vector de frecuencias tiene la mitad del tamaño de la FFT, siendo 2048 y 1024 muestras respectivamente. Esto hace que entre muestra y muestra del vector de frecuencias haya un total de 21,5 Hz, siendo el ancho de banda mínimo de 43 Hz. Este Δf se puede reducir aumentando el tamaño de muestras de la FFT (nfft), pero aumentaría la complejidad de la transformada FFT haciendo que sea menos eficiente. Para un filtrado cuyo ancho de banda sea mayor de unos 100 Hz, la utilización nfft=2048 es suficiente, mientras que, en caso de que se quiera eliminar un ruido, o tono a una frecuencia en concreto, se debería aumentar el tamaño de la FFT para así reducir Δf (duplicando nfft, Δf se reduce a la mitad). Aun así, la aparición de armónicos de un tono a una frecuencia cualquiera es inevitable.

Cabe destacar que siempre es recomendable la utilización de un tamaño de la FFT que sea potencia de 2, para que ésta sea más eficiente.

5.3 Acompañamiento musical en partitura

Ahora vamos a detallar el funcionamiento de la aplicación principal de este Trabajo, la cual tiene como objetivo realizar el acompañamiento musical en partitura de la melodía captada por un micrófono (esto satisface el tercer objetivo propuesto, ya que como veremos en los resultados se va a considerar procesado en tiempo real).

En primer lugar se presenta un diagrama de bloques de la estructura general de la aplicación y de sus funciones desarrolladas:

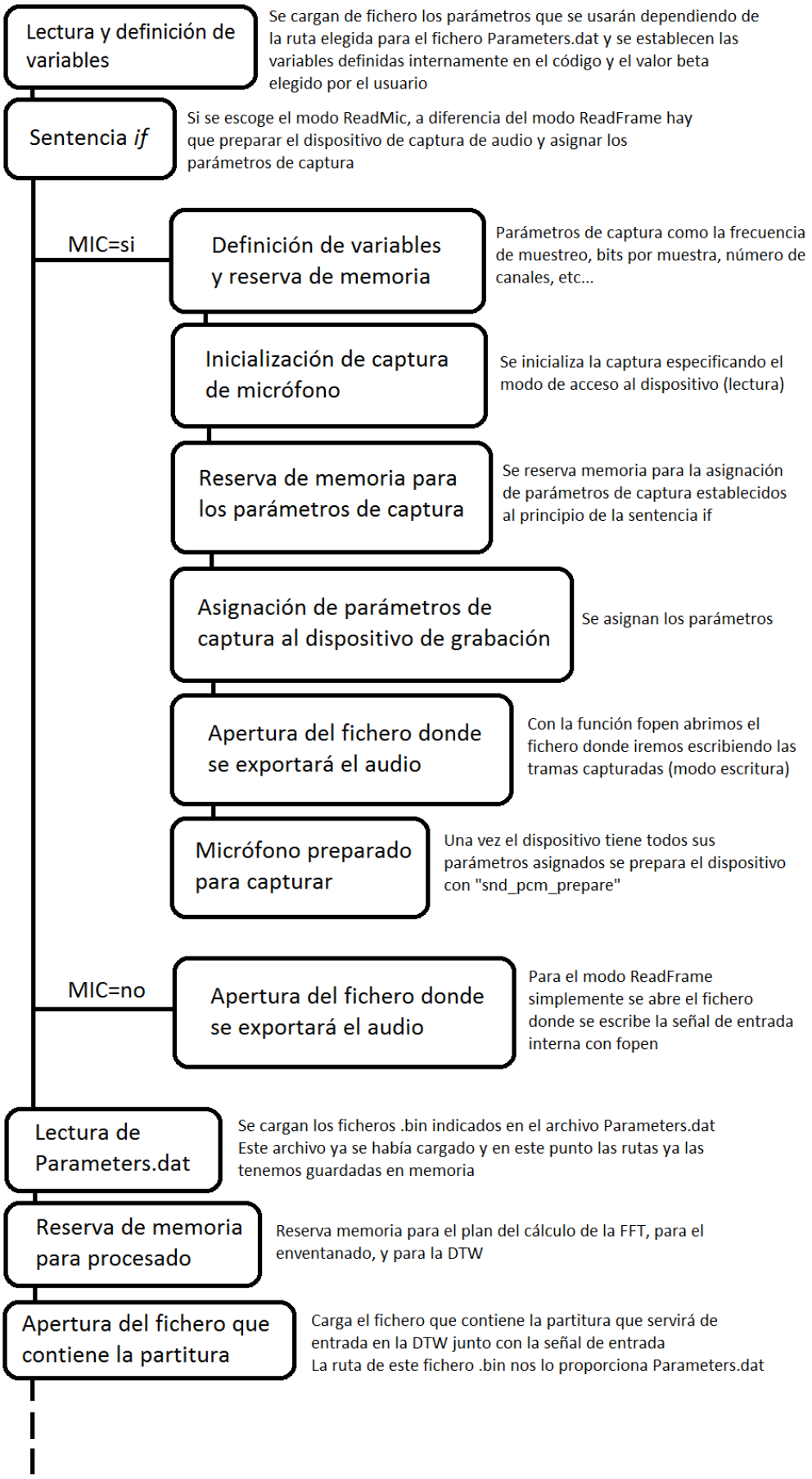

Fig.17 – Primera parte del diagrama de flujo de datos de Preproceso_CPU

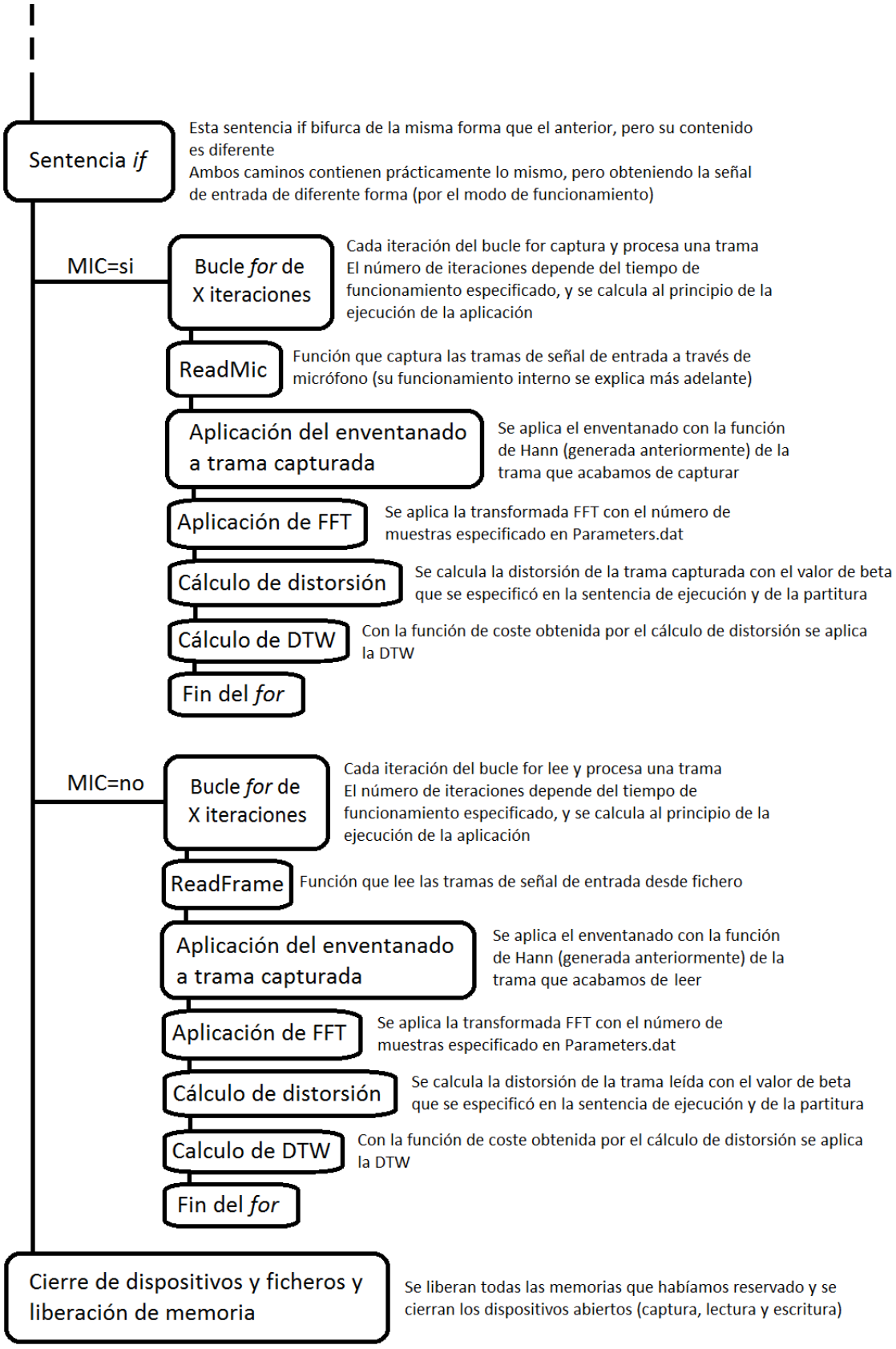

Fig.18 – Segunda parte del diagrama de flujo de datos de Preproceso_CPU

A continuación el diagrama de bloques de la función ReadMic, teniendo como entrada el ID del dispositivo de captura y como salida la trama capturada (como se observa en el diagrama de bloques a nivel superior):

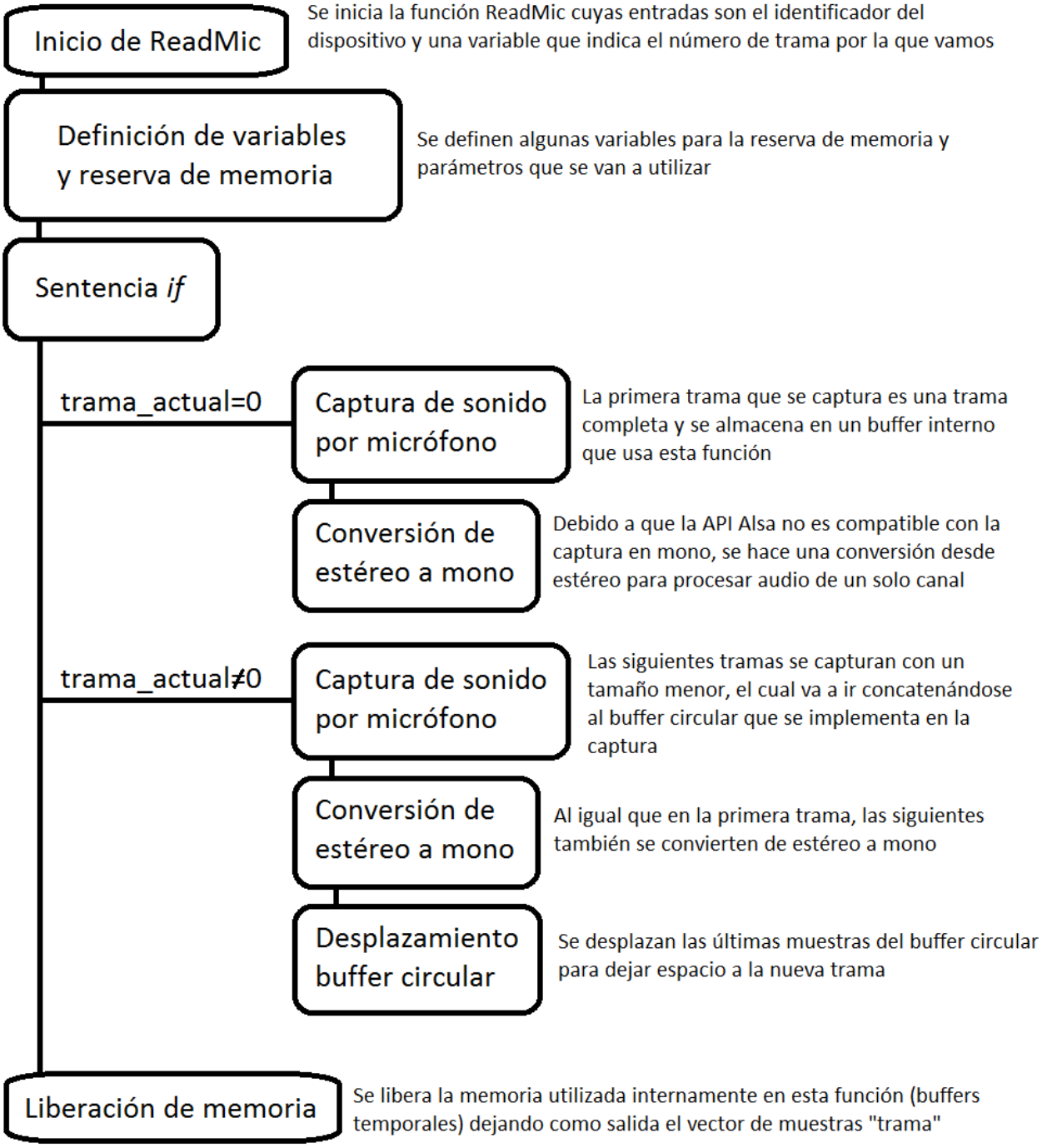

Fig.19 – Diagrama de flujo de datos de ReadMic

Una vez tengamos la aplicación Preproceso_CPU correctamente instalada (ver Anexo 8.2.2) primero vamos a explicar algunos de los parámetros que podemos modificar de los cuales depende su modo de funcionamiento (la explicación de lo que contiene cada fichero que forma la aplicación se encuentra señalado en el manual técnico correspondiente a la aplicación, en el Anexo 8.3).

El primero de todos lo vamos a localizar en la línea 61 del archivo *main.c* (nombre de la variable `tiempoMic`). Este valor representa la duración de la señal de entrada en segundos, y aunque en realidad puede ser cualquier número entero, se va a limitar a 300 segundos de duración, ya que la partitura tiene una duración finita. Las pruebas realizadas para el análisis de resultados han sido establecidas con una duración de 150 y 300 segundos. El resto de variables no se van a modificar, y se justificará su valor más adelante.

En segundo lugar, en el archivo *Makefile* podemos modificar más valores. Uno de ellos es el compilador que se utilizará al pasarle el `make` al *Makefile*, teniendo como posibles opciones de uso el compilador `gcc` (Linux) y el `icc` (Intel, solo en el portátil). En este mismo fichero puede seleccionarse el modo de funcionamiento de la aplicación asignando el valor `si` o `no` a `MIC`. En caso de que `MIC=si` la aplicación funcionará en modo ReadMic, mientras que si por el contrario `MIC=no` funcionará en modo ReadFrame. El último de los valores que podemos modificar en este fichero va a ser el tipo de variable utilizada la señal de entrada memorizada en caso de funcionar en modo ReadFrame (en modo ReadMic siempre usará el tipo *short*, cuya decisión se justifica más adelante). En caso de que `SIMPLE=si` la aplicación utilizará variables tipo *float*, mientras que si `SIMPLE=no` la aplicación usará variables *double*. Estas variables están definidas en el código como tipo *MyType*, el cual tendrá el valor *float* o *double* según se escoja.

Una vez tengamos los ficheros guardados con las variables correspondientes al modo de funcionamiento deseado procedemos a compilar el programa con la sentencia:

```
make -f Makefile
```

Con esto se nos generará el archivo ejecutable `main`, el cual se puede ejecutar siendo precedido de `./` de la siguiente forma:

```
./main B {ruta_parametros}
```

Siendo `B` el valor de β deseado (más detalles en el Anexo 8.4) y `{ruta_parametros}` la ruta en la que está ubicado el fichero de parámetros para la partitura (*Parameters.dat*). Este fichero suele estar dentro de la carpeta correspondiente al tiempo de la carpeta cuyo nombre corresponde a la duración de la señal de entrada seleccionada (`tiempoMic`) la cual se encuentra dentro de `Datos` o `Datosf` (depende de si lo seleccionado en `SIMPLE`). Este árbol de carpetas se debe haber generado si hemos instalado correctamente la aplicación, quedando a las carpetas `Datos` y `Datosf` al mismo nivel que la carpeta *Preproceso_CPU* (la cual contiene los ficheros de aplicación), por lo que el archivo main quedaría a un nivel inferior y debiendo preceder la ruta con `..` (indica directorio superior). Un ejemplo de sentencia de ejecución de la aplicación puede ser:

```
./main 1 ../Datos/5min/Parameters.dat
```

En este caso `{ruta_parametros}` es `../Datos/5min/Parameters.dat`, habiendo seleccionado β=1, duración de 300 segundos y variables *double* (el compilador y modo de funcionamiento utilizado podría ser cualquiera).

En primer lugar la aplicación comienza estableciendo los parámetros introducidos en la sentencia de ejecución e inicializando variables, algunas de las cuales dependen del tipo de directiva de compilación empleada (tipo *float* o *double* para *MyType*). Tras esto, el flujo de procesado de divide en dos dependiendo de una de otra de las directivas de compilación. En este caso de si se utiliza micrófono o no (`MIC`).

En caso de que no se utilice micrófono lo único que hará será abrir el fichero al que se exportará la señal de entrada proveniente de fichero (`fpF=fopen("grabadoFrame.pcm","w");`) y continuar, mientras que si por el contrario se utiliza el micrófono se ejecutarán varias funciones antes de llegar al mismo punto donde hemos dejado el flujo de datos en caso de no usar entrada microfónica.

Se comenzará con la definición de las variables para la utilización de la API Alsa y se reservará memoria para la utilización de vectores que representan las tramas de datos.

En el código de la aplicación se encuentra asignado el dispositivo de captura correspondiente al hardware que se esté utilizando (Jetson o portátil). En caso de que se quiera ejecutar en otro equipo habría que averiguar el identificador de su dispositivo de captura introduciendo el comando `aplay -l` en el Terminal de Linux:

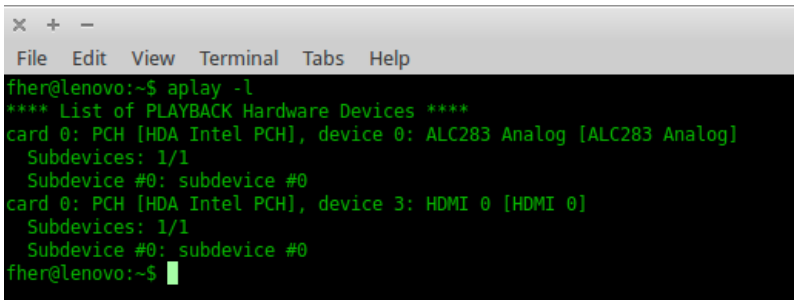

Fig.20 – Lista de dispositivos del portátil

En la Figura 20 se observa la lista de todos los dispositivos de audio del ordenador portátil Lenovo. Para que la API Alsa reconozca qué dispositivo es con el que se va a trabajar hay que indicarle los dos valores numéricos que representan la tarjeta (card) y el dispositivo dentro de esa tarjeta (device), siendo en este caso el 0,0 (card 0, device 0, la tarjeta de sonido Intel HDA). El dispositivo 0,3 corresponde al interfaz de audio digital utilizada por el conector HDMI. El formato para indicarle a la API este identificador es: **"`hw:0,0`".**

Una vez se tiene inicializado el dispositivo en modo captura (especificado por la entrada `SND_PCM_STREAM_CAPTURE` a la función `snd_pcm_open`), se continúa reservando memoria para los parámetros de funcionamiento de este dispositivo (`snd_pcm_hw_params_malloc`), a continuación estableciéndose unos predeterminados por si alguno de los deseados no se consigue aplicar (`snd_pcm_hw_params_any`) y por último asignándose los que hemos decidido establecer:

- `snd_pcm_hw_params_set_access`: modo de acceso, lectura/escritura en este caso.
- `snd_pcm_hw_params_set_format`: formato de las muestras capturadas, 16bits con signo.
- `snd_pcm_hw_params_set_channels`: número de canales, estéreo, ya que en mono no funciona (se convierte a mono al capturar las muestras en la función ReadMic).
- `snd_pcm_hw_set_rate_near`: frecuencia de muestreo, 44100Hz.
- `snd_pcm_hw_params`: aplica todos los parámetros establecidos del dispositivo.
- `snd_pcm_hw_params_free`: libera todos los parámetros reservados en memoria debido a que ya están aplicados en el dispositivo y no los necesitamos más.

Tras esto se abre el fichero donde se exportará el archivo de audio capturado en modo escritura `fpm=fopen("grabadoMic.pcm","w");` y continúa por el mismo sitio por el que podría haber continuado en caso de no haberse utilizado micrófono.

Una vez ha terminado la sentencia *if* condicionada por la utilización del modo ReadMic o ReadFrame se continúa con la lectura del archivo Parameters.dat que contiene los parámetros correspondientes a la partitura (`ReadParameters`). A continuación reserva memoria para todas las variables que se van a utilizar para el procesado de audio (`AllocDTW, AllocS_fk, AllocFFT y AllocAuxi`).

Ahora abre el archivo de partitura con `fp=fopen("NameFiles.file_trama", "rb");` siendo `NameFiles.file_trama` la ruta donde se almacena la partitura establecida por *Parameters.dat*.

A continuación volvemos a tener otra sentencia *if* que bifurcará el flujo en dos. Ambos flujos comparten la estructura principal de procesado (formado por un bucle *for* para *N_tramas*). La única diferencia es la obtención de datos de la señal de entrada, siendo en caso de ReadFrame una simple lectura de tramas desde fichero, y en caso de ReadMic una captura de señal de micrófono que detallamos a continuación:

-En primer lugar se establecen variables como el tamaño de trama, tamaño de muestra, contador interno y punteros para la utilización del buffer circular.

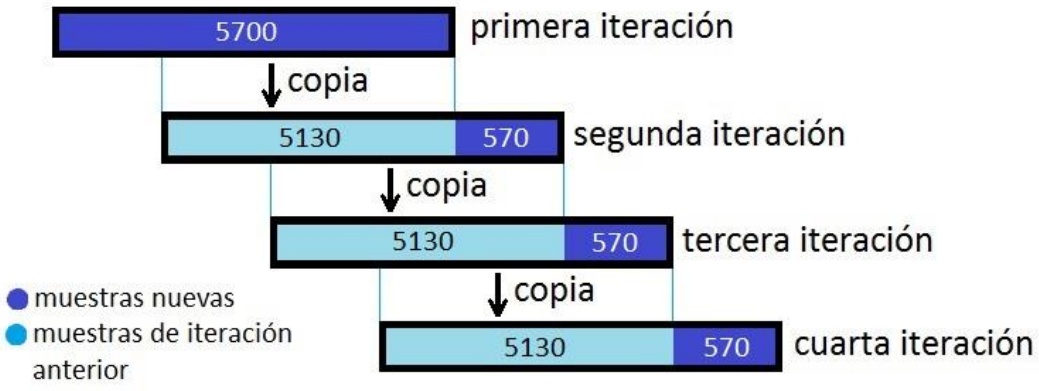

Fig.21 – Buffer circular 90%

-Una vez establecidas las variables y punteros se reserva memoria para éstos con la función `malloc`.

-Para la primera iteración del bucle *for* (primera trama) se captura una trama de 5700 muestras para rellenar el buffer al completo. De esta forma en las siguientes iteraciones se podrá implementar el buffer circular. Esta captura se lleva a cabo con la función `snd_pcm_readi`, e inmediatamente después se convierte la trama a mono (sumando la mitad de potencia de las muestras pares y las impares).

-Para el resto de iteraciones se van a capturar 570 muestras nuevas y se irán añadiendo al buffer circular (Figura 21). Una vez concatenadas las 570 nuevas muestras se convierte a mono como en la primera iteración a las 5700 últimas muestras (5130 anteriores + 570 nuevas).

-Como las tramas se van almacenando en el puntero `trama_mic`, se libera la memoria de `buffer` para la siguiente iteración.

Una vez se ha completado ReadMic o ReadFrame, el siguiente paso en su flujo de procesado es la aplicación de enventanado a la señal de entrada (ApplyWindow), después se calcula la FFT (FFT) incluyendo el método de solapamiento suma (rellenado con ceros hasta la siguiente potencia de 2 a la señal de entrada), se calcula y aplica la distorsión (ComputeDist y ApplyDist) y por último se hace la DTW como tal (ParIdamin y DTWProc) obteniendo como resultado la posición estimada de la trama de entrada en la partitura. Esta estimación se calcula combinando la DTW con unos estados MIDI que representan las diferentes notas que hay en la partitura. La DTW comienza con el cálculo de la distancia entre la señal de entrada y cada uno de los estados MIDI (ComputeDist, ApplyDist ParIdamin). Con este vector de distancias (función de coste) se selecciona la de coste mínimo y se estima que ese estado MIDI es el correspondiente al de la trama actual de la señal de entrada (DTWProc y ParIdamin). El funcionamiento de la DTW se detalla en el Anexo 8.5.

Todo este procesado se agiliza de forma notable en caso de utilizar el compilador ICC y la directiva OpenMP, ya que las sentencias `#pragma omp` añaden concurrencia al código y hacen que la ejecución de los bucles *for* se hagan de forma paralela o no se tenga que esperar a que finalice por completo para continuar (estas son algunas de las optimizaciones de OpenMP).

Como paso final se libera la memoria del resto de punteros y se cierran las interfaces de captura y ficheros que hemos abierto a lo largo del proceso.

Mientras la aplicación se está ejecutando deberán aparecer mensajes que representan el progreso de la aplicación:

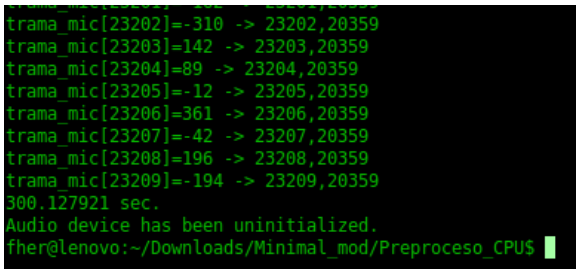

Fig.22 – Finalización de Preproceso_CPU para 5min ReadMic

Debemos observar principalmente en el tiempo total de ejecución (300,13s) y en los dos últimos números que aparecen en cada una de las líneas que aparecen para cada trama. Estos números representan la trama de la señal de entrada y su posición en la partitura respectivamente. En este caso la señal de entrada ha tenido un total de 23209 tramas y en la partitura se ha llegado a la trama 20359. El número de tramas podemos comprobar que corresponde a lo especificado:

$$\text{N\_tramas} \cong \frac{(300\ \text{s} * 44100\ \text{muestras/s})}{570\ \text{muestras/trama}}) \cong 23210\ \text{tramas} \quad (5)$$

Esta leve variación respecto al cálculo se debe a que hemos supuesto todas las tramas de 570 muestras, pero la primera de ellas es de 5700. Esto es para poder implementar el buffer circular utilizado.

Este buffer circular es de 5700 muestras divididas en 10 bloques de 570 muestras cada uno, por lo que entre trama y trama hay un salto de 570 muestras, lo cual equivale a unos 12,9ms (a 44100Hz de frecuencia de muestreo, la cual es idónea para el espectro audible). Este salto de 12,9ms (valor aproximado respecto al mencionado en [13] para redondear el número de muestras) es vital para la correcta implementación de la aplicación, ya que es durante este tiempo (en el que se están capturando 570 muestras nuevas) cuando se deben llevar a cabo de forma simultánea todos los cálculos de las 570 muestras anteriores. En caso de que se tarde más de esos 12,9ms, la aplicación dejará de funcionar y no finalizará correctamente, ya que se habría intentado capturar nuevas muestras y asignarlas a una sección de memoria que se está utilizando en ese preciso instante.

La utilización de variables tipo *short* para la captura de señal microfónica (ReadMic) está justificada por el tipo de audio que se pretende capturar (16bits por muestra, que son 2Bytes, justo el tamaño que tiene una variable tipo *short*).

El resto de parámetros escogidos para la captura de audio son 44100Hz de frecuencia de muestreo (como ya hemos dicho idóneo para el espectro audible y que cumple con el Teorema del muestreo), y un solo canal para reducir la carga de procesado (recordemos que en modo ReadMic debe hacer muchas más operaciones por muestra que en modo ReadFrame).

6. RESULTADOS Y DISCUSIÓN

Como se mencionó en el apartado de Materiales y métodos, se han realizado una sucesión de ejecuciones para comprobar las ventajas de la utilización de ciertos parámetros de entrada, para conocer los límites considerados como procesado en tiempo real y para comprobar el correcto funcionamiento de la aplicación de acompañamiento musical. Con este apartado se consigue cumplir el quinto y último de los objetivos propuestos (pruebas y promoción de la aplicación).

6.1 Funcionamiento en modo ReadFrame

		Jetson		Intel	
muestras	**beta**	**float**	**double**	**float**	**double**
11604trm (150sec)	*0*	106	174	40	63
	1	120	162	37	59
	2	48	68	18	27
23209trm (5min)	*0*	332	566	124	191
	1	306	520	120	179
	2	157	220	56	78

Tabla.1 – Tiempo de procesado en segundos

En esta tabla de resultados se muestran los tiempos que han tardado en procesarse el total de tramas correspondientes a los tiempos de captura. A simple vista se observa que el caso de β igual a 2 es el más rápido, mientras que el caso de β igual a 0 el más lento. También se observa la evidente diferencia de tiempos en el procesado de muestras tipo *float* y *double*, además de la diferente potencia de los dos procesadores.

6.2 Funcionamiento en modo ReadMic

		Jetson		Intel	
tiempo	**beta**	**float**	**double**	**float**	**double**
150sec (11604trm)	*0*	8337	CRASH(20)	8348	8345
	1	8328	CRASH(47)	8350	8349
	2	8342	8385	8349	8346
5min (23209trm)	*0*	CRASH(33)	CRASH(5)	20359	20359
	1	CRASH(403)	CRASH(6)	18934	18919
	2	18839	18845	18921	18919

Tabla.2 – Posiciones alcanzadas en partitura

En este caso lo primero que observamos es que en el caso del kit de desarrollo Jetson TK1, con variables tipo *double* (8 bytes) suele *crashear* la captura para β igual a 0 y 1, ya que mientras captura las 570 nuevas muestras se están procesando las 570 anteriores. Este *crasheo* se debe a que el tiempo de procesado es superior al tiempo de captura (570 muestras=12,9ms), por lo que cuando el micrófono va a capturar nuevas muestras aún se están procesando las anteriores, devolviendo un error de captura de señal y dejándose de ejecutar. Para β igual a 2 vemos que no hay ningún problema.

En el caso de variables float (4 bytes), para tiempo de captura de 150 segundos consigue completarse, pero en el caso de capturar el doble de tiempo no. Esto se debe a que la partitura es mayor y el procesador no puede procesar a tiempo la partitura.

En el caso del portátil con procesador Intel no hay ningún problema y todas las ejecuciones las hace con resultado satisfactorio.

Si observamos simultáneamente las dos tablas anteriores nos damos cuenta de una cosa. El tiempo total de procesado de todas las tramas tiene una relación directa con el hecho de que la captura de señal microfónica falle o no en el kit de desarrollo Jetson. Mirando primero en la Tabla.1, para las 11604 tramas, las cuales equivalen a 150 segundos, vemos que las únicas ejecuciones que han tardado más del tiempo equivalente a ese número de tramas coincide con los valores de β igual a 0 e igual a 1 con las variables *double* y el procesador TK1, las cuales, observando ahora la Tabla.2, son las únicas que han *crasheado*. Esta observación se cumple también para el caso de las 23209 tramas y los 300 segundos equivalentes (incluso en un caso se supera por tan solo 6 segundos, un 2%). Con esto se confirma que el tiempo de procesado de cada trama supera los 12,9ms de máximo que tiene permitido.

Podría pensarse en aumentar este límite de tiempo (más muestras por captura manteniendo el tamaño de trama, ya que la complejidad de la FFT depende del tamaño de la señal) para que mientras se capturan más muestras tengamos más tiempo para procesar las anteriores. De esta forma estaríamos reduciendo el tamaño del buffer circular, lo cual nos limita el número de muestras útiles de iteraciones anteriores, debiéndose de procesar más rápidamente antes de que se pierdan.

6.3 Uso de compilador GCC vs ICC

		ReadFrame (float)		ReadMic	
tiempo	**beta**	**GCC**	**ICC**	**GCC**	**ICC**
150sec 11604trm	*0*	40	15,4	8348	8375
	1	37	15,0	8350	8325
	2	18	13,5	8349	8351
5min 23209trm	*0*	124	45,8	20359	18538
	1	120	43,9	18934	18517
	2	56	38,9	18921	18527

Tabla.3 – Resultados GCC vs ICC

Los resultados son claramente mejores en el caso de la utilización del compilador ICC debido al paralelizado de procesos entre los distintos núcleos del procesador. Nótese que en los resultados del modo ReadMic no notamos diferencia, ya que su avance está directamente relacionado con el número de muestras capturadas (aunque tengamos mejor procesado, se va a demorar siempre el mismo tiempo, ya que tiene que estar un tiempo constante capturando por micrófono). Si el procesador TK1 del kit de desarrollo Jetson fuese compatible con el compilador ICC seguramente no presentaría problemas de procesado en el modo ReadMic al menos con variables *float*, ya que la diferencia de tiempo de ejecución por trama se ve notablemente mejorada y estos fallos se presentan por un margen muy pequeño (entre un 2 y un 10%). Otro detalle a tener en cuenta es la consideración de procesado en tiempo real como tal en modo ReadMic, y eso lo podemos justificar observando el tiempo total extra en milisegundos que la aplicación tarda en capturar, procesar y reproducir todas las tramas respecto a lo especificado:

		GCC		ICC	
tiempo	**beta**	**t_{out}-t_{in}**	**%diferencia**	**t_{out}-t_{in}**	**%diferencia**
150sec 11604trm	*0*	121,1	0,08	119,7	0,08
	1	123,9	0,08	118,6	0,08
	2	120,2	0,08	118,7	0,08
5min 23209trm	*0*	123,3	0,04	120,1	0,04
	1	124,1	0,04	121,7	0,04
	2	119,8	0,04	119,8	0,04

Tabla.4 – Consideración de procesado en tiempo real

Observando que en ningún caso apenas se supera el 0,1% del tiempo total (124ms máximo en ambos casos) y dividiéndolo entre el número total de muestras se obtiene un retraso medio de 0,0107ms en el caso del tiempo de ejecución de 150

segundos y de 0,0053ms en el caso de 5 minutos. El propio sonido tarda unos 3ms en recorrer un metro de distancia, por lo que dicho retardo podría considerarse nulo.

$$v_{prop} = 340m/s; \quad t_s = \frac{1}{340m/s} = 2{,}9ms/m; \tag{6}$$

$$A\ 1\ metro = 2{,}9ms \gg 0{,}0053ms \tag{7}$$

6.4 Diferentes fuentes de entrada

En la Tabla.2 se ha podido observar que las ejecuciones de 150 segundos rondan las 8300 muestras de partitura, mientras que las de 5 minutos rondan las 18000-20000. Todo ello se ha obtenido con el mismo audio captado por el micrófono (grabadoFrame.flac). Pero, ¿qué pasaría si acelerásemos o ralentizásemos la pieza musical que capta el micrófono? (sin cambiar el pitch). Sería una equivalencia a que el músico tocase más deprisa o más despacio, y recordemos que la DTW es la solución a las variaciones de tempo de dos señales similares.

Para este caso hemos acelerado y ralentizado un 8% la pieza musical, y los resultados eran de esperar. En el caso de capturar por micrófono la señal ralentizada (grabadoFrameRalentizado_8.flac) para un mismo tiempo de 150 segundos no se llega a avanzar tanto en la partitura, como es lógico. En este caso se llegan hasta las casi 7700 muestras de partitura, lo que aproximadamente es un 8% menos que se alcanzaba con el audio original. En el caso del audio acelerado un 8% (grabadoFrameAcelerado_8.flac) ocurre justamente lo contrario, se alcanzan aproximadamente un 8% más de muestras en la partitura (unas 9100). Los ficheros de audio se pueden hallar adjuntos a la memoria.

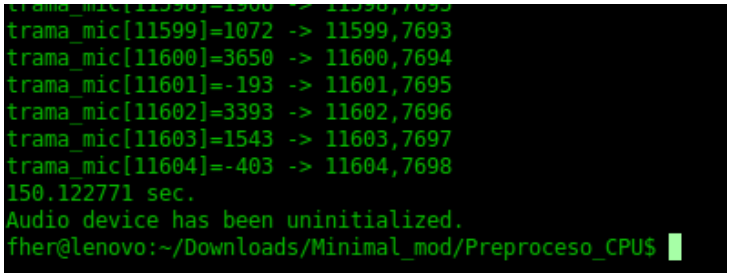

Fig.23 – Ralentizado un 8%

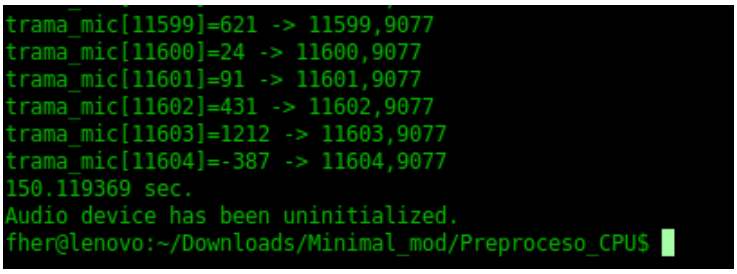

Fig.24 – Acelerado un 8%

6.5 Procesador en estado de estrés

Como vimos en el apartado de resultados, con el Jetson tuvimos problemas con el modo captura, ya que el procesador no trataba las tramas a tiempo, pero en cambio en el portátil no hubo ningún problema. En este caso, someteremos el procesador del portátil a un estado de estrés mientras se ejecuta la aplicación para observar qué ocurre. Lo haremos para cuatro casos distintos; para el modo ReadFrame y modo ReadMic, el mejor y peor caso (que recordemos que coincidían con los tipos de variable *double* y *float* y valores de β igual a 0 e igual a 2 respectivamente).

-Modo ReadFrame:

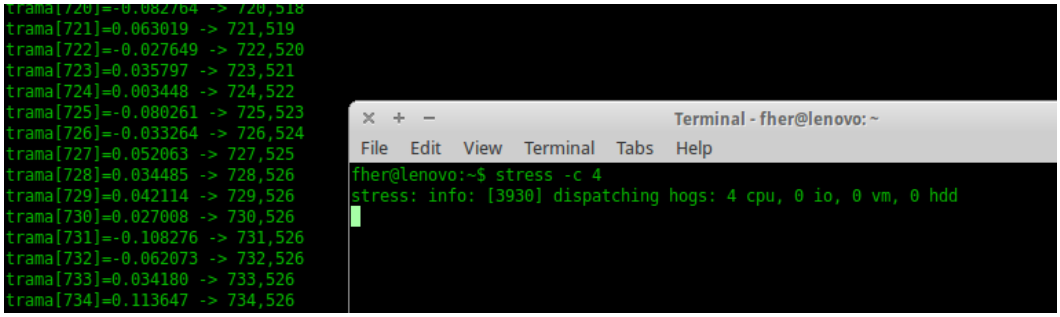

Fig.25 – Variables double, 5min, β=0 (peor caso)

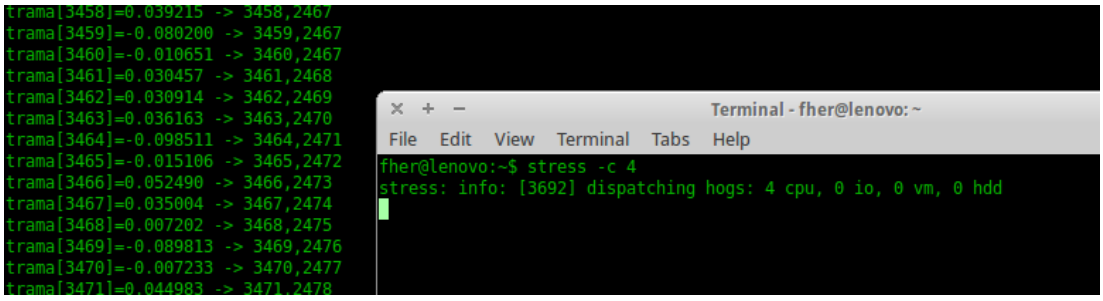

Fig.26 – Variables float, 150sec, β=2 (mejor caso)

Incluso saturando el procesador, en ambos casos la lectura de datos del fichero no se interrumpe (la lectura de datos del fichero sigue tardando menos de 12,9ms).

-Modo ReadMic:

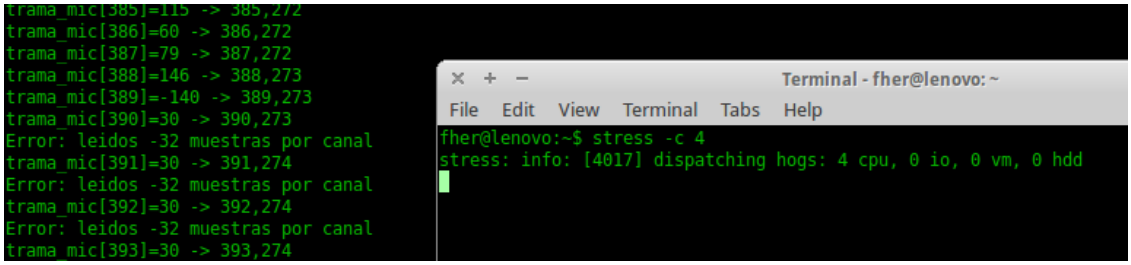

Fig.27 – Variables double, 5min, β=0 (peor caso)

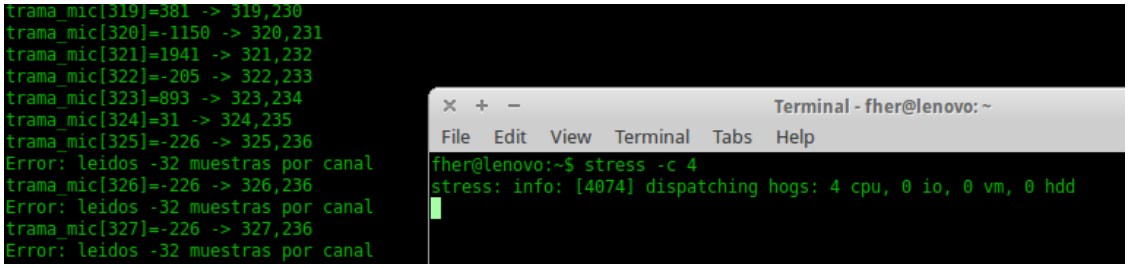

Fig.28 – Variables float, 150sec, β=2 (mejor caso)

Ahora en ambos casos tenemos una interrupción de la ejecución, aun habiendo obtenido unos buenos resultados de ejecución sin tener el procesador estresado.

Esto demuestra que en el caso de la captura de señal (ReadMic) el tiempo de procesado es crítico, ya que si el procesador no dedica la mayor parte de su potencia a ese procesado, no conseguirá tratar las 570 muestras antes de que pasen los 12,9ms que tiene de máximo antes de comenzar a capturar las 570 siguientes.

7. CONCLUSIONES

A pesar de todas las dificultades con las que nos hemos topado a lo largo del desarrollo de este Trabajo de Fin de Grado, se han logrado cumplir con la mayoría de objetivos.

Dada mi escasa experiencia con la programación en C, este Trabajo ha sido de gran utilidad para conocer más a fondo cómo funciona este lenguaje. Una de las dificultades mencionadas se debe a que, como la aplicación de acompañamiento musical desarrollada por el equipo de software de la Universidad de Oviedo se trata de un proyecto de investigación el cual está en constante desarrollo (y por ello carece aún de un manual de uso), desde un principio fue complicado llegar a entender cómo funciona realmente. Además, un continuo cambio de versiones (las cuales optimizaban los tiempos de ejecución y añadían nuevas mejoras) también hizo que fuese más complejo su estudio.

Más complicaciones fueron causadas a la hora de *flashear* el Kit de desarrollo Nvidia Jetson TK1 [17]. Este Kit de desarrollo no se ha popularizado tanto como lo han hecho Arduino o Raspberri Pi, y por ello ha sido más complicado encontrar información en la red para su correcta puesta a punto. Además tras el *flasheo*, hubo que habilitar la señal microfónica de forma manual a través de unas sentencias que se encontraron en los foros de la web de desarrolladores de Nvidia.

Otro problema lo hemos tenido con la API Alsa [14], ya que desde un principio teníamos claro que la captura se haría con un solo canal, y resultó que este conjunto de utilidades de audio es incompatible con captura en mono, viéndonos obligados a capturar en estéreo y haciendo una conversión a mono. Otra incompatibilidad desconocida de esta API es la utilización de los dispositivos de audio para capturar y reproducir de forma simultánea.

A pesar de todo ello, pienso que ha merecido la pena colaborar en el desarrollo de esta sorprendente aplicación. Me hubiese encantado seguir aportando, ya que si no hubiésemos tenido las mencionadas complicaciones seguro que hubiésemos llegado más lejos y podríamos haber perfeccionado algunos detalles.

Sería interesante seguir evolucionando la aplicación y poder dotarla de un sintetizado de la señal de la partitura que acompañase al mismo ritmo al músico (de la misma forma que lo hace la aplicación Antescofo [3]). Esto se planteó como opción de colaboración con un Trabajo de Fin de Máster de un compañero. Esto sería posible si primero se lograse implementar la captura y reproducción de audio a través de un único dispositivo sin que se tenga que cerrar captura para poder abrir reproducción para cada trama, ya que esto provocaría un severo retardo.

Otra posible mejora, la cual favorecería enormemente al Jetson TK1, es su uso con una versión del Sistema Operativo sin interfaz gráfica, sobre todo cuando el procesado se hace únicamente desde su CPU. Esto no sería necesario en caso de que todo el procesado se hiciese con la potencia de la GPU (haciendo uso además de la memoria que ésta contiene), ya que la CPU y memoria RAM mueven con soltura el Sistema Operativo.

8. ANEXOS

8.1 Flasheo del kit Jetson TK1

En primer lugar necesitamos un PC con sistema operativo Ubuntu 12.04 de 64bit, ya sea instalado directamente o ejecutado desde una máquina virtual, en cuyo caso habría que instalar los Guest Services de la máquina virtual (para VirtualBox: `sudo apt-get install virtualbox-guest-dkms`).

Los materiales necesarios para conseguir flashear el Kit de Desarrollo de NVIDIA Jetson TK1 son:

-Placa Jetson TK1 conectada a la red eléctrica.

-Cable USB macho tipo B a micro USB macho tipo B.

-Cable HDMI.

-Monitor con puerto HDMI.

-HUB USB de al menos dos puertos.

-Teclado y ratón USB PnP.

-Ordenador con Sistema Operativo Ubuntu 12.04 y puerto USB.

-Conexión a internet.

La conexión de los dispositivos debe seguir el siguiente esquema (en caso de que difiera algo, el proceso de flasheo no se llevará a cabo correctamente):

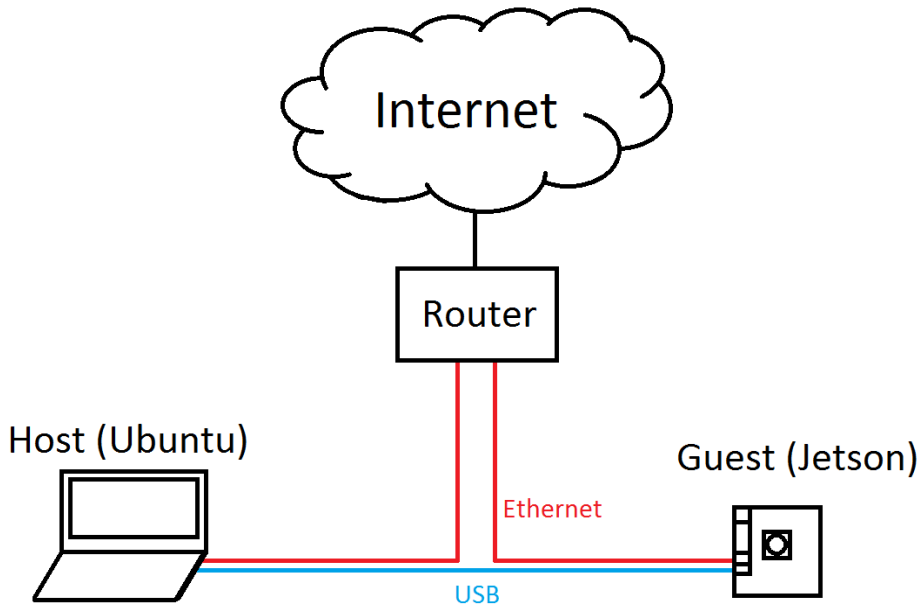

Fig.29 – Topología flasheo Jetson

Teniendo los dispositivos conectados de esta forma podemos continuar.

En el equipo Host descargamos el JetPack TK1 en su versión 1.2 de la web de desarrolladores de Nvidia (antes hay que registrarse y participar en el programa de desarrolladores de Nvidia). Lo ejecutamos (modo Completo) y vamos siguiendo las instrucciones de instalación [17]. Este proceso descargará e instalará primero en el equipo Host todos los elementos necesarios.

Una vez se ha completado la instalación en el Host, ponemos el equipo Guest en modo Recovery (manteniendo pulsado el botón Recovery mientras pulsamos y soltamos el botón de encendido). Una vez esté en modo Recovery (una pantalla totalmente en negro a la salida del HDMI de la placa Jetson), continuamos con la instalación, la cual se produce mediante USB.

Tras instalar una versión ARM de Ubuntu en la placa a través de USB, el equipo Host se conectará por SSH al equipo Guest y lo *flasheará* (haciendo *push* de todos los elementos de desarrollo de Nvidia; CUDA, OpenCV y PerfKit).

Una vez *flasheada* la placa hay que hacer ciertas correcciones a la entrada de micrófono mini-jack de 3,5mm para que capte sonido. Para ello abrimos una ventana de Terminal (todo esto ya en la placa Jetson), y ejecutamos los siguientes comandos:

```
amixer cset name="Stereo ADC MIXL ADC2 Switch" 0
amixer cset name="Stereo ADC MIXR ADC2 Switch" 0
amixer cset name="Int Mic Switch" 0
amixer cset name="ADC Capture Switch" 1
amixer cset name="RECMIXL BST1 Switch" 0
amixer cset name="RECMIXR BST1 Switch" 0
amixer cset name="RECMIXL BST2 Switch" 1
amixer cset name="RECMIXR BST2 Switch" 1
amixer cset name="Stereo ADC L1 Mux" "ADC"
amixer cset name="Stereo ADC R1 Mux" "ADC"
amixer cset name="Stereo ADC MIXL ADC1 Switch" 1
amixer cset name="Stereo ADC MIXR ADC1 Switch" 1
amixer cset name="Stereo ADC MIXL ADC2 Switch" 0
amixer cset name="Stereo ADC MIXR ADC2 Switch" 0
amixer cset name="IN1 Mode Control" "Single ended"
amixer cset name="IN2 Mode Control" "Single ended"
amixer cset name="Mic Jack Switch" 1
```

Tras estos cambios podemos probar que la entrada de micrófono capta señal yendo a la configuración de sonido de Ubuntu o con los comandos `arecord` y `aplay`.

8.2 Preparación del software

Las aplicaciones de filtrado de audio (ecualizador Matlab y en C) son totalmente portables, por lo que no es necesario detallar ningún tipo de preparación debido a que se pueden ejecutar desde cualquier directorio.

8.2.1 Librerías FFT (fftw3):

Para el caso del portátil (versión completa basada en Linux de 64bit (Host)) vamos a la web http://www.fftw.org/download.html [16] y descargamos la última versión para Linux (3.3.4). A continuación lo descomprimimos y lo instalamos con los siguientes comandos en el Terminal (estando ubicados donde hemos descomprimido los archivos):

```
./configure
make
make install
```

En el caso de la versión ARM (la instalada en el kit de desarrollo Jetson TK1), descomprimimos la versión modificada por el equipo de Software de la Universidad de Oviedo en /tmp (`Librerias-Jetson-post_22_03_2016.tgz`) y la movemos al directorio /opt con los siguientes comandos ejecutados a través del Terminal de Linux (estando ubicados donde hemos descomprimido este fichero .tgz):

```
tar –zxvf Librerias-Jetson-post_22_03_2016.tgz
sudo mv /tmp/opt/* /opt
```

8.2.2 Aplicación Preproceso_CPU

Tanto en el Jetson como en el PC descomprimimos todos los ficheros y directorios que contiene el archivo *InstVirtual_Minimal_18_06_2016.tar* a excepción de la carpeta *Preproceso_GPU*. Ahora cambiamos las propiedades de los ficheros *Build_Data.sh* y *Build_Verify.sh* para permitir que sean ejecutables y los ejecutamos, lo cual nos generará los parámetros y datos que utiliza la aplicación (veremos que se ha creado también una carpeta llamada *Datosf,* la cual es una versión con variables float de *Datos*, la cual contiene variables en double).

Ahora podemos descomprimir y reemplazar la versión para este Trabajo. En el caso del PC lo contenido en *Preproceso_CPU FINAL_Intel.zip* y en el caso del Jetson lo contenido en *Preproceso_CPU FINAL_Jetson.zip* (ambos adjuntos a la memoria).

8.2.3 Aplicaciones extra:

La primera aplicación a instalar es la API de **Alsa** [14], gracias a la cual podremos capturar señal de micrófono con código en lenguaje C. Para ello, desde una ventana del Terminal ejecutamos el comando: `sudo apt-get install libasound2-dev`.

En caso de que la distribución de Linux no incluya por defecto el compilador **GCC** es muy sencillo de instalar. Basta con que introduzcamos en el Terminal el comando: `sudo apt-get install gcc`, y una vez completada su instalación introducimos: `sudo apt-get install build-essential`.

Debido a que la versión de Ubuntu de la placa Jetson no es una versión completa, las siguientes aplicaciones solo funcionan en el PC con procesador Intel. Las instalaremos a través del Terminal:

audacity (`sudo apt-get install audacity`): para reproducir y modificar archivos de audio. Muy útil para comprobar qué hemos grabado, extraer la señal sintetizada incluida en el código del equipo de Software de la Universidad de Oviedo, y para generar nuevas versiones con distinto tempo.

stress (`sudo apt-get install stress`): para someter al procesador a un estado de estrés, el cual le aumenta considerablemente la carga de trabajo.

8.2.4 Compilador ICC:

El compilador ICC (Intel C++ Compiler) [10] forma parte de un conjunto de utilidades que proporciona Intel para la optimización de procesos de sus procesadores (y también de algunos procesadores AMD compatibles). Este paquete de mejoras, denominado Intel Parallel Studio XE 2016 no es gratuito, teniendo tres diferentes versiones con un coste que va desde los 699 (Composer Edition) a los 2949 dólares americanos (Cluster Edition). Aparte de la venta de licencias de este software, Intel también ofrece una versión de prueba de estas 3 ediciones. En mi caso seleccioné el paquete intermedio (Professional Edition), cuya licencia completa tiene un coste de 1599 dólares. En cualquiera de estas versiones, el compilador ICC está incluido.

En primer lugar accedemos a la web de software de Intel:

https://software.intel.com/en-us/intel-parallel-studio-xe/

En ella seleccionamos el sistema operativo en el que se vaya a instalar y la versión deseada. En el caso que nos concierne (Linux), el proceso de instalación es muy sencillo. Solamente hay que cambiar las propiedades del archivo .sh que se nos descargará para hacerlo ejecutable y ejecutarlo a través del Terminal de Linux como hemos hecho hasta ahora (**`./`** delante del nombre del fichero ejecutable).

Tras esto, a través del propio Terminal nos irán apareciendo los pasos de instalación, que van desde la comprobación de compatibilidad del sistema hasta la instalación de todos los componentes de este conjunto de aplicaciones, pasando por la selección de los elementos que queramos instalar y su descarga. Para la selección de todas las opciones de instalación se hará uso del teclado, marcando numéricamente la opción deseada.

El proceso de descarga a instalación puede tardar alrededor de una hora, dependiendo de la velocidad de conexión a internet que se posea (crítico a la hora de la descarga de las aplicaciones).

Una tengamos el paquete Intel Parallel Studio XE instalado, habrá que añadir el compilador ICC (`icc` a través del terminal) a los compiladores que el Terminal puede manejar (aparte del `icc` se añadirá el `icpc`, el cual sirve para compilar código en C++). Lo primero que debemos hacer es hallar la ruta completa donde se han instalado los compiladores, normalmente en */home/{user}/intel/bin* o en */opt/intel/bin.* Para ello ejecutamos el siguiente comando en el Terminal de Linux (siendo {user} nuestro nombre de usuario de Ubuntu):

```
source /home/{user}/Intel/bin/compilervars.sh intel64
```

Una vez hecho esto ya podemos invocar al compilador `icc`. En nuestro caso lo usaremos a través del archivo Makefile, el cual incluye una variable que señala con qué compilador se hará el *make*.

8.3 Manual técnico de la aplicación de acompañamiento musical

A continuación se explica la estructura principal de ficheros de la aplicación de acompañamiento musical acompañada por una estructura en árbol para comprender su distribución:

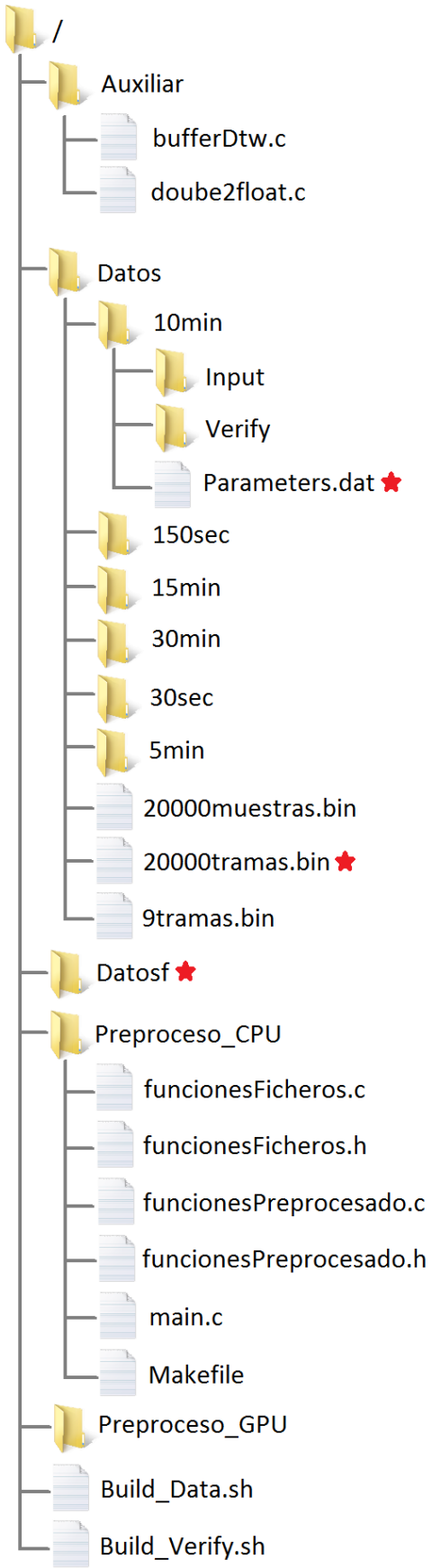

Fig.30 – Árbol de archivos de la aplicación

Una vez se ejecutan los ficheros Build_Data.sh y Build_Verify.sh se crean los ficheros y carpetas marcadas con la estrella roja.

Al hacerlo, se ejecuta de forma automática una serie de algoritmos que hacen uso de las funciones incluídas en los programas de la carpeta Auxiliar. Aparte de los archivos de parámetros, también crea una copia con variables float de la carpeta Datos con la función doube2float.c (llamada Datosf).

20000tramas.bin (salida de la función bufferDtw.c cuya entrada es 20000muestras.bin) es el archivo que contiene la señal de partitura.

La carpeta Preproceso_CPU contiene la aplicación como tal que hace el acompañamiento musical haciendo uso del microprocesador del kit de desarrollo, mientras que en Preproceso_GPU incluye una versión que trabaja con el chip gráfico que encierra el SoC TK1.

El archivo main.c contiene la función main, la cual marca la columna vertebral de la aplicación. Para compilarla se hace uso del Makefile, archivo que contiene variables editables para distintos modos de compilación.

Los archivos funcionesFicheros.c y funcionesPreprocesado.c contienen las funciones de las que hace uso la aplicación troncal main.c. Estos van acompañados de su homónimo en .h, los que incluyen los correctos modos de ejecución de las funciones dentro de la aplicación principal.

En nuestro caso hemos añadido la función ReadMic dentro de funcionesFicheros.c (y su estructura de uso en funcionesFicheros.h), y el resto sentencias necesarias en el propio main.c y Makefile.

La selección del tiempo de captura en modo ReadMic se modifica en el fichero main.c (variable `tiempoMic`).

El compilador que queremos usar se especifica en el fichero Makefile (variable CC). En éste también se elige el tipo de variables utilizado en el modo ReadFrame, e incluso la elección del modo de funcionamiento (modo ReadMic MIC=si; modo ReadFrame MIC=no). Como archivo Makefile que es, también incluye sentencias que proporcionan de forma automática (dependiendo de las variables escogidas) los *flags* con los que se va a ejecutar la compilación.

8.4 Funcionamiento DTW

El DTW (Dynamic Time Warping – Alineamiento Dinámico Temporal) [12] [13] consiste en una técnica que comprime y expande de forma no lineal una señal (entrada) para alinear los valores de ésta con otra señal (referencia).

Como resultado de esta compresión/expansión obtenemos el camino de alineamiento, el cual nos indica los puntos donde las señales convergen. Esta técnica surge a raíz de la imposibilidad de sincronizar dos señales de naturalezas distintas por completo (una pieza musical con un tempo exacto en nuestro caso).

A diferencia del alineamiento temporal (las dos señales se conocen por completo), el alineamiento **dinámico** temporal se aplica cuando una de las señales (generalmente la de entrada) aún no se conoce, por lo que esta técnica se irá aplicando a medida que se vayan recibiendo trozos de señal y teniéndose en cuenta la señal recibida hasta ahora. De esta forma, la DTW puede clasificarse en 3 modos distintos:

-Hacia atrás: se analizan las muestras que ya se han recibido. Es el método más fiable pero el que más retardo tiene.

-Hacia delante: se basa en la estimación de las siguientes notas. Apenas tiene retardo, pero la estimación hace que no sea muy preciso.

-Acompañamiento: es una técnica la cual se basa en una señal de tempo. Esta señal de tempo se irá actualizando con cada trama (por si hay que aumentar o disminuir el tempo).

Denominaremos la señal de referencia como **A**, la señal de entrada será **B** y el camino de alineamiento **C**.

$$A = \{a1, a2, a3 \dots ai \dots am\};\ B = \{b1, b2, b3 \dots bj \dots bn\};\ C = \{c1, c2, c3 \dots ck \dots cK\}; \quad (8)$$

Cada *ai* (hasta *am*) representa un punto de la señal de referencia, de forma equivalente ocurre con *bj.* En el caso de cada *ck*, representará un par de puntos que comparará de A y B:

$$ck = (ik, jk) \quad (9)$$

Y como resultado de esta comparación obtendremos una función de coste (cuyos valores mínimos representan la función de alineamiento, es decir, el camino de alineamiento).

$$d(ck) = \delta(aik, bjk) \quad (10)$$

Esta función de coste, dependerán de una variable llamada β (beta). A razón de esta variable se aplicarán distintas funciones de coste sustituyéndola en la siguiente función general:

$$D_{\beta}(x,y) = \begin{cases} \dfrac{\left(x^{\beta} + \mathrm{y}^{\beta} * \ (\beta - 1) - \beta * \mathrm{x} * y^{\beta-1}\right)}{\left(\beta * (\beta - 1)\right)} & \beta \in (0,1) \cup (1,2] \\ \mathrm{x} * \log\left(\dfrac{x}{y}\right) - \mathrm{x} + \mathrm{y} & \beta = 1 \\ \left(\dfrac{x}{y}\right) + \log\left(\dfrac{x}{y}\right) - 1 & \beta = 0 \end{cases} \quad (11)$$

Siendo β=0 la distancia Euclídea, β=1 la divergencia de Kullback-Leibler, y β=2 la de Itakura-Saito.

El algoritmo de DTW podría resumirse de la siguiente manera:

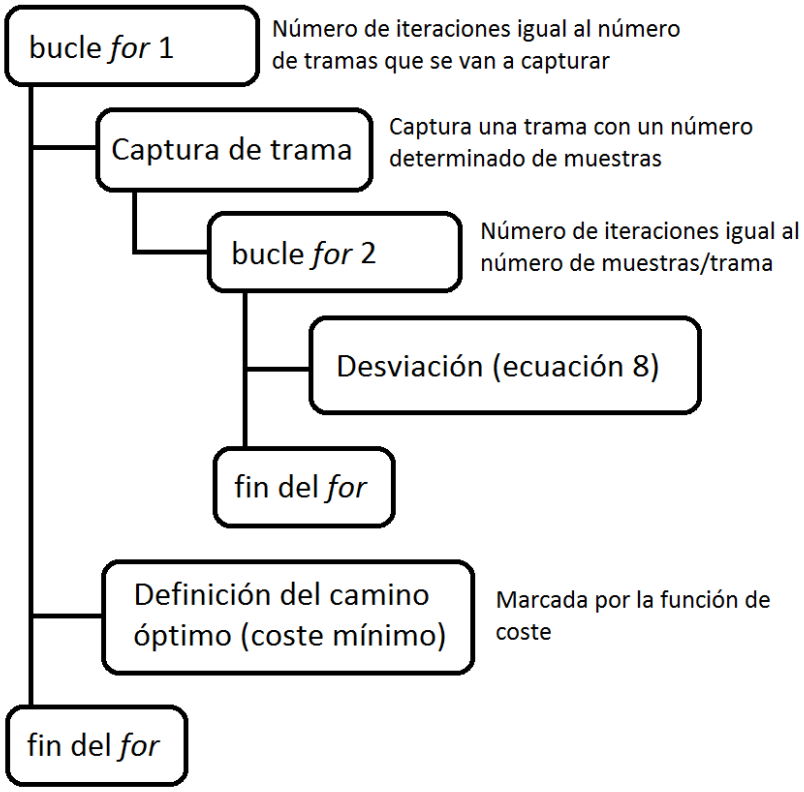

Fig.31 – Diagrama de flujo de DTW

9. REFERENCIAS BIBLIOGRÁFICAS

1. MOORE, G.E. *"Lithography and the Future of Moore's Law"*, SPIE Vol. 2437, Mayo 1995.
2. *"Tonara"*, Tonara LTD, http://tonara.com/company/
3. *"Antescofo",* IRCAM – Music Representations Team, http://repmus.ircam.fr/antescofo
4. ORIO, N., LEMOUTON, S., SCHWARZ, D. *"Score Following: State of the Art and New Developments",* Proceedings of the 2003 Conference of New Interfaces for Musical Expression, Montreal, Canada. 2003
5. *"ADX TRAX PRO 3"*, Audionamix, http://audionamix.com/technology/adx-trax-pro/
6. ZEMLIN, J. *"Jetson TK1"*, Embedded Linux Wiki. The Linux Foundation, http://elinux.org/Jetson_TK1
7. KINGHORN, D. *"NVIDIA Jetson TK1 CUDA performance"*, Puget systems, https://www.pugetsystems.com/labs/hpc/NVIDIA-Jetson-TK1-CUDA-performance-569/
8. *"GPU GFLOPS for Game console, ARM and X86 SoC"*, Taiwan, China. 2016. http://kyokojap.myweb.hinet.net/gpu_gflops/
9. HORAK, V. *"Asteroids@home",* Astronomical Institute, Charles University, Prague, https://asteroidsathome.net/boinc/cpu_list.php
10. *"Intel Developer Zone"*, Intel Corporation, https://software.intel.com/en-us/c-compilers/ipsxe
11. POULSEN, D. *"The OpenMP API Specification for parallel programming"*, OpenMP ARB, http://openmp.org
12. ALONSO, P., CORTINA, R., RODRÍGUEZ-SERRANO, F.J., VERA-CANDEAS, P., ALONSO-GONZÁLEZ, M. & RANILLA, J. *"Parallel Online Time Warping for Real-Time Audio-to-Score Alignment in Multi-core Systems",* Journal of Supercomputing. 2015
13. RODRÍGUEZ-SERRANO, F.J., CARABIAS-ORTI, J.J., VERA-CANDEAS, P. & MARTÍNEZ-MUÑOZ, D. *"Automatic Accompaniment using Spectral Decomposition and Online Dynamic Time Warping"*, ACM Transactions on Embedded Computing Systems, Vol.9, No 4, Article 39. Marzo 2010
14. KYSELA, J. *"ALSA Project",* Alsa Team, http://www.alsa-project.org/alsa-doc/alsa-lib/
15. MAZZONI, D. *"Audacity"*, Audacity Team, http://www.audacityteam.org/
16. *"Fastest Fourier Transform in the West",* FFTW. http://www.fftw.org

17. *“NVIDIA EMBEDDED COMPUTING”*, NVIDIA Corporation, https://developer.nvidia.com/embedded-computing