Trabajo Final: ecualizador y analizador de espectro optimizados en código nativo para la plataforma Andorid

Organización del Computador II, Departamento de Computación, Universidad de Buenos Aires

Guillermo Gallardo Diez y Luis Scoccola

gagdiez.c@gmail.com
luis.scoccola@gmail.com

Trabajo Final: ecualizador y analizador de espectro optimizados en código nativo para la plataforma Andorid

Organización del Computador II, Departamento de Computación, Universidad de Buenos Aires

Índice

Resumen

Se implementó un reproductor de PCM WAV para el sistema operativo ANDROID. El reproductor tiene un ecualizador gráfico de cinco bandas y un analizador de espectro que muestra la energía en función del tiempo de 30 bandas.

La interfaz del usuario fue programada en Java, mientras que la manipulación del audio se programó en C y assembler. Mediante un *profiler* se analizó la estructura dinámica del programa y se optimizaron, en assembler, las funciones más críticas.

Index Terms

FFT, ecualizador, analizador de espectro, código nativo, Android

I. Aclaraciones generales sobre algunas decisiones tomadas

El único formato de audio aceptado por el programa es PCM WAV. Este es un tipo específico de WAV. Es el más utilizado por su simpleza. Si bien es un formato no comprimido, y por ende no recomendable para almacenamiento de música en dispositivos portátiles, resulta muy útil a la hora de centrarse en otros aspectos que no tienen que ver en sí con el formato del audio utilizado, como es el caso de este proyecto.

El lenguaje de la capa superior de la aplicación es JAVA, pues así lo requiere el sistema operativo. El código para procesamiento de audio fue escrito en C y luego se realizaron optimizaciones en assembler. Es importante notar que la vasta mayoría de dispositivos que utilizan ANDROID tienen arquitectura ARM, con lo cual fue utilizado lenguaje de máquina de dicha arquitectura. Como queríamos explorar la arquitectura en detalle y realizar optimizaciones precisas, no utilizamos *inline-assembler*. Con esto perdimos generalidad, es decir, si se quieren utilizar todas las optimizaciones del código, se deberá usar una arquitectura particular (ARMv7), pero ganamos eficiencia.

II. Arquitectura

Explicamos brevemente los registros de los que se dispone a la hora de trabajar con esta arquitectura y un poco de la arquitectura en sí. Los registros de propósito general son $r0 - r12$. De estos pueden utilizarse $r0 - r3$ sin guardar los valores previos, mientras que los valores de $r4 - r12$ deben ser preservados luego del llamado a función. Estos registros son de $32bits$ y, funcionalmente, permiten hacer todo cómputo necesario. Para realizar cómputos costos, sin embargo, es conveniente usar registros e instrucciones especializadas. Por esto existe el *VFP* (*Vector Floating Point*) y la extensión *Advanced SIMD* (conocida como NEON). El primero es un coprocesador para números de punto flotante. Puede trabajar con precisión simple o doble. La segunda es una extensión que tiene instrucciones *SIMD*, no necesariamente de punto flotante, para los registros de *VFP*. Los registros *SIMD* de la arquitectura específica que utilizamos (ARMv7) son $s0 - s31$ de $32bits$, $d0 - d31$ de $64bits$ y $q0 - q15$ de $128bits$. Estos conjuntos de registros no son disjuntos, en el siguiente sentido: Los registros $s0 - s31$ son las partes bajas y altas de los registros $d0 - d15$. Por otro lado, los registros $d0 - d31$ los partes altas y bajas de los registros $q0 - q15$. Notar que **no** se dispone de registros $s32 - s64$. Para hacer referencia directa a éstos debe utilizarse $d16 - d31$ con un "subíndice", por ejemplo $d16[0]$ o $d17[1]$.

III. Herramientas

Para resolver distintos aspectos del proyecto tuvimos que utilizar varias herramientas. En esta sección detallamos las más importantes. Los paquetes principales en los cuales se encuentra la totalidad de las herramientas son el NDK (*Native Development Kit*) y el *ADT* (*Android Developer Tools*) que a su vez contiene una versión reducida del SDK (*Software Development Kit*).

El compilador utilizado no es más que una versión de GCC adaptada al sistema operativo ANDROID; este se encuentra en el paquete NDK. Aquí se encuentra tambien una versión del OBJDUMP, escencial a la hora de estudiar el código compilado, ya sea optimizador por el compidor o escrito por nosotros. Tambíen para estudiar código, pero particularmente útil para *debuguear* y analizar el comportamiento dinámico del código nativo es justamente la versión de GDB que también se encuentra en este paquete. Finalmente, no incluido en el paquete, pero listo para ser agregado al mismo, está una versión del *profiler* clásico GPROF, utilizado en la parte de *testing*.

En el paquete SDK encontramos herramientas para instalar aplicaciones en el dispositivo, conectarnos a la terminal del dispositivo y subir y bajar archivos a la memoria del dispositivo, entre otras tareas. La herramienta más importante se denomina ADB.

IV. SIGNAL FLOW

Signal flow refiere al camino que realiza una señal (usualmente de audio) desde un cierto *input* hata un cierto *output*. En este caso nuestro *input* es el archivo de audio que quiere ecualizarse y el *output* es un *buffer* de reproducción.

Con respecto al formato `PCM WAV` basta decir que (mas allá de un *header* bastante simple) el audio se representa mediante un vector de *samples* codificados como enteros, con signo, de 16 *bits*. En el caso de una canción *stereo* se usa *interleaving* de los *samples*.

Como usamos *floats* para los cálculos internos, escalamos cada sample al rango $[-1, 1]$, aprovechando la mayor precisión de dicho rango.

Introducimos, ahora, el concepto de "ventana" (*window*) de audio. Una ventana es simplemente una pequeña porción de audio (usualmente de tamaño fijo), se trata de una tira de *samples* consecutivos. En el dominio digital, las señales se procesan de a ventanas, sobre todo cuando no se usan filtros recursivos (como es el caso de este proyecto). A esto se contrapone el dominio analógico donde básicamente no existe este concepto.

Entonces vemos nuestro audio como un vector de ventanas. Para minimizar artefactos al realizar la ecualización en sí, se utilizan ventanas entrecruzadas. Esto es, en lugar de separar el archivo en prociones de audio consecutivas y disjuntas, lo separamos en porciones que comparten la mitad de sus *samples*. A cada una de estas ventanas se la multiplica por una curva particular de forma tal de que la suma de todas devuelva la señal original [2, Generalized Window Method]. En nuestro caso, esta curva es una ventana de Bartlett.

Para realizar la ecualización, se realiza la transformada discreta de Fouriere de la ventana. Una vez en el dominio de las frecuencias se le aplica un filtro a la señal. Esto simplemente se traduce en multiplicar por un número entre 0 y 1 a las bandas que se quieran atenuar. Nuestra implementación particular permite atenuar hasta $5db$ y amplificar sólo $1dB$. En general siempre es conveniente atenuar, sobre todo mientras más avanzado en la vida del audio uno se encuentre. En este caso es el final, justo el momento antes de la reproducción. Nuevamente para evitar artefactos, esta atenuación no puede ser arbitraria y es conveniente utilizar funciones suaves. Para esto elegimos un *spline* cúbico que interpola los puntos elegidos mediante la manipulación del ecualizador gráfico. La idea del *spline* es recomendada en [2, Generalized Win-

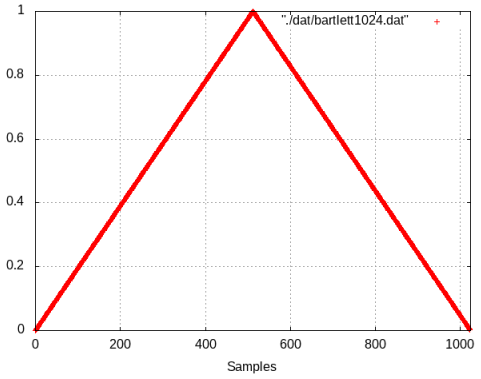


Figura 1
VENTANA TIPO BARTLETT, UTILIZANDO 1024 *samples*

dow Method]. Una vez modificada la señal en el dominio de las frecuencias con el *spline* interpolador, retornamos al dominio del tiempo mediante la transformada inversa de Fouriere.

Finalmente reconstruimos la señal (ecualizada) sumando las ventanas consecutivas. Esta señal ecualizada es reproducida y, en paralelo, analizada, para mostrar su espectro. El análisis es muy simple, se trata de, nuevamente aplicar una función conveniente a cada ventana, en nuestro caso la función de Blackman, aplicar la transformada de Fouriere, y promediar bandas consecutivas hasta obtener la cantidad de bandas deseada. Esto se debe a que la transformada nos devuelve tantas bandas como samples tenga la ventana (4096 en nuestro caso), y es usual querer visualizar menos (30 en nuestro caso).

Un detalle no menor es el uso del *dither*. En el formato `PCM WAV` las señales se encuentran codificadas mediante un vector de enteros de $16bits$, con signo, y nuestro ecualizador trabaja con *floats*. Como ahora queremos volver a codificar el audio con enteros de $16bits$, debemos utilizar un *dither* como se explica en [3, Chapter 3: ADC and DAC]. El *dither* introduce ruido blanco de muy baja amplitud para evitar que el redondeo de un mismo número se produzca de la misma manera cada vez. En particular, el *dither* implementado, fue sacado de [7] y ajustado a nuestras necesidades (se realizó una optimización y luego se implementó en *assembler*). Tiene la particularidad de usar un filtro recursivo para el error cometido en cada redondeo, lo cual produce una ecualización del ruido introducido, evitando ruido de bajas frecuencias, que tiene demasiada energía.

V. Interfaces y organización del código

Los directorios donde se encuentra la parte interesante[1] de la implementación son dos:

- `Ecualizador/src/com/ecualizador` para archivos `.java`
- `Ecualizador/jni` para archivos `.c`

A. Capas (más abstracción a menos abstracción)

A.1 Interfaz gráfica de la aplicación

Puede explorarse la tarjeta `SD` en busca del archivo `WAV` que quiera reproducirse. Si se eligiese un archivo con un formato inválido, esto se muestra en pantalla[2]. Mientras se reproduce puede filtrarse el audio mediante un ecualizador gráfico. También se muestran gráficamente las cinco bandas ecualizadas con distintos colores (analizador de espectro). Cada una de estas bandas está, a su vez, separada en seis bandas para poder apreciarse mejor la forma del espectro.

Además de modificar la ganancia de las cinco bandas mientras se reproduce, puede encenderse y apagarse el ecualizador, cambiarse el *threshold* del analizador de espectro (que modifica a partir de qué amplitud empiezan a visualizase las bandas), y cambiarse el *refresh rate* del analizador. Esto ultimo permite promediar varios espectros a lo largo del tiempo para tener una idea del espectro menos instántanea. Por una cuestión de (mucha) simpleza de código, este parámetro se modificará una vez que el audio sea parado y reproducido nuevamente.

Los archivos en los cuales se encuentra implementado lo recién explicado son:

- `Informacion.java`
- `Inicio.java`

[1] Debieron escribirse archivos tangenciales al proyecto pero necesarios para su funcionalidad, como por ejemplo *layouts* y configuraciones de la aplicación para el SO.

[2] Por una cuestión de simplicidad del código no todos los casos de error son chequeados, con lo cual, con algunos archivos particulares esto puede fallar

- `Wave.java`

A.2 Interfaz Java-C

El código en Java lee el archivo de audio, escribe sobre el buffer de reproducción y grafica en pantalla. Para esto debe manejar una serie de *threads* que son básicamente productores-consumidores. Para que estos *threads* puedan manipular el audio, se les da una interfaz a las funciones implementadas en código nativo. Esta fue denominada `LibC`, y para Java no es más que una clase usual. En `LibC.java` se encuentran declarados los métodos del lado de Java. El dual de este archivo es `interfazParaJava.neon.c`, aquí se encuentran escritas en C, las implementaciones de lo métodos declarados en `LibC.java`. El archivo `interfazParaJava.neon.c` es muy simple en esencia, pero complicado en síntaxis. A continuación explicaremos un poco la sintaxis.

Las funciones de los dos archivos arriba mencionados son las mismas. En C debe usarse, para nombrar a las funciones, una convención un tanto verborrágica, pero que hace referencia directa a la estructura del proyecto de Java. La estrucura de los nombres se encuentra detallada en [8, Chapter 2: Resolving Native Method Names]. Aparte de los nombres de las funciones, a los parámetros de las funciones se les agregan dos parámetros (que no debimos usar en nuestro caso); que son el *environment* de Java en donde se está ejecutando la función, y el objeto al que se le está aplicando el método (recordemos que para Java, `LibC` es una clase como cualquier otra). La convención se encuentra en [8, Chapter 2: Native Method Arguments].

Lo último que hay para notar aquí es la forma de pasar datos. Como la representación interna de los tipos y estructuras es distinta en los dos lenguajes debe realizarse una conversión. Para dar más flexibilidad esta conversión no es automática y, en C, recibimos estructuras de tipos particulares (por ejemplo `jbyteArray`, `jfloatArray`, `jint`, etc.). En el caso de tipos simples como `jint` la conversión se trata solo de un *casting*. Para el caso de arreglos debe llamarse a una *API*. Es así que tenemos funciones como `GetByteArrayElements` que internamente realiza un `malloc` y devuelve un puntero a un arreglo de *bytes*. Algo similar se hace a la hora de pasar un arreglo en dirección contraria, o de avisar que el arreglo ya no se necesita más. Si bien puede parecer que la conversión debería ser automática, es importante notar que dicha conversión tiene un *overhead*. Por esto resulta conveniente poder evitar conversiones si, por ejemplo, el llamado particular a la función no necesita de algunas estructuras. Otro caso interesante que tiene que ver con ésto es el momento de "devolver" un arrelgo a Java. Existen dos formas básicas de hacerlo, con *write back* o sin *write back*. Es claro que la segunda modalidad se ahorra cerca de la mitad del *overhead*. En nuestro caso utilizamos ambas modalidades en distintas situaciones. El ecualizador debe retornar la ventana ecualizada y por ende utiliza *write back*. Pero el analizador, que puede tomar varias ventanas antes de devolver un análisis (dependiendo del *refresh rate*) y para el cual, además, la entrada y la salida son distintas (arreglo de 4096 *floats* de entrada y 30 *floats* de salida), no utiliza *write back*. Esto se logra pasando `JNI_ABORT` como último parámetro de la función `ReleaseByteArrayElements`.

Finalmente explicamos la utilidad del archivo `Android.mk`. Se trata de un archivo muy semejante a un *makefile*. En él se explicitan, por ejemplo, *flags* de compilador así como directorios donde haya bibliotecas que deban ser importadas.

Archivos relevantes:

- `Programa.java`
- `LibC.java`
- `interfazParaJava.neon.c`
- `Android.mk`

A.3 Interfaz C-*assembler*

Esta parte es mucho más simple pues se trata únicamente de declarar las funciones escritas en *assembler* en algún *header*. La convención de pasaje de parámetros es semejante a la de INTEL. Se pasan los primeros cuatro parámetros en los registros de $r0$ a $r3$, y luego se *pushean* a la pila. Las variables globales se tratan como posiciones de memoria. La mayor complejidad en esta parte fue la sintaxis de GAS que está pobremente documentada. Ejemplo de uso de variables globales es la función `getIntsInt16_ditherASM`.

VI. EXPLICACIÓN DEL CÓDIGO

A. Explicación código en JAVA

Separamos por *threads* aclarando si son productores o consumidores. Las funciones explicadas a continuación se encuentran todas implementadas en el archivo `Programa.java`.

A.1 `procesadorAudio` (productor)

Lee directamente del archivo `WAV` y escribe sobre una serie de *buffers* el audio ya ecualizado. Cada *buffer* tiene espacio para exactamente una ventana de audio. Estos *buffers* permiten dar un margen para que el `procesadorAudio` se quede bloqueado por bastante tiempo. Esto es un *tradeoff* entre estabilidad (más *buffers* darán más tolerancia a períodos donde no corre el *thread*) y respuesta (demasiados *buffers* generarán una respuesta muy lenta cuando, por ejemplo, se modifique la ecualización, pues es posible que haya una gran cantidad de ventanas todavía ecualizadas con la vieja ecualización).

A.2 `reproductor` (consumidor)

Lee de los *buffers* escritos por el `procesadorAudio` y los escribe en un *buffer* interno del sistema operativo mediante el método `write()` de la clase `track`.

A.3 `analizador` (consumidor-productor)

Lee de los *buffers* ya mencionados, analiza el espectro y, cada *refresh rate* ventanas, escribe en otra serie de *buffers* de los que leerá el graficador.

A.4 `graficador` (consumidor)

De forma análoga al *reproductor*, lee de los *buffers*, esta vez escritos por el `analizador`, y muestra en pantalla el análisis.

A.5 *listener* del *track*

Lo explicado hasta recién se logra mediante semáforos, *mutex* y *read-write locks*, que, por suerte, ya están implementados en JAVA. Pero esto todavía no resuelve todo el problema, falta una sincronización esencial: entre el reproductor interno de ANDROID y el `graficador`. Esto es porque cuando el `reproductor` nuestro escribe sobre el *buffer* de reproducción interno del sistema operativo, la reproducción no es inmediata y por lo tanto la sincronización no puede ser hecha directamente con este *thread*. Lo que se utiliza en este caso es un *listener*, que no es más que un *thread* que se despierta cuando sucede un evento particular. En este caso el evento es la reproducción de *refresh rate* cantidad de ventanas de audio. En este momento el *listener* se despierta y realiza un *signal* al graficador que está esperando este momento para graficar las ventanas que están por ser reproducidas.

B. Explicación código en C

B.1 Estructura `ecualizador`

La estructura contiene varios arreglos internos para evitar realizar `mallocs` cuando se está reproduciendo. La función principal es `ecualizar`. La complejidad de la función está en realizar las sumas de las ventanas con los *offsets* correctos. Recordemos que a nivel interno, el ecualizador utiliza ventanas superpuestas en un 50 %. Cada una es ecualizada y luego debe ser sumada con la ventana siguiente. La idea general de la función es la resumida en [2, Generalized Window Method] y explicada en detalle en capítulos anteriores y posteriores. La diferencia entre *mono* y *stereo* se hace importante de aquí en adelante.

B.2 Estructura `analizador`

Al igual que `ecualizador` tiene varios arreglos internos. La función `analizar` se encarga de conseguir, mediante la transformada de Fouriere, el espectro de la ventana pasada como parámetro y reducirlo a 30 bandas. Los valores son acumulados en un vector que luego se promedia y devuelve mediante la función `dameAnalisis`.

B.3 Biblioteca `funciones`

Este archivo contiene varias funciones auxiliares interesantes. Destacaremos las que tienen una lógica no-trivial.

El *dither* es una de ellas. Se trata de la función `getIntsInt16_dither`. El tipo de *dithering* ya fue comentado en la sección de *signal flow*. La optimización realizada aquí fue generar un vector de números aleatoreos durante la inicialización del *dither* de forma tal de no llamar a la función `rand` durante la reproducción de la canción. El vector tiene la longitud de una ventana, esto no genera mayores inconvenientes ya que la ventana dura más de un décimo de segundo, y el oído humano no llega a distinguir frecuencias de menos de $40Hz$.

Aquí se encuentran las funciones para generar ventanas de diversos estilos, en particular los dos ya mencionados: Bartlett y Blackman. La implementación se trata de *samplear* el gráfico de las funciones en un arreglo de *floats*.

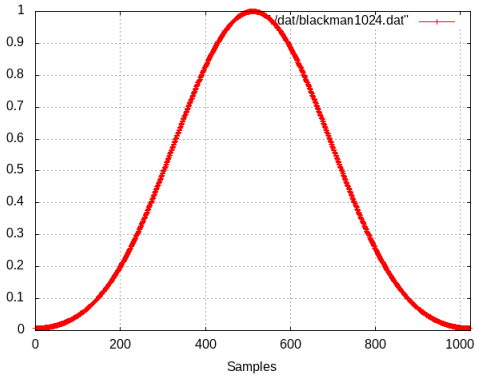
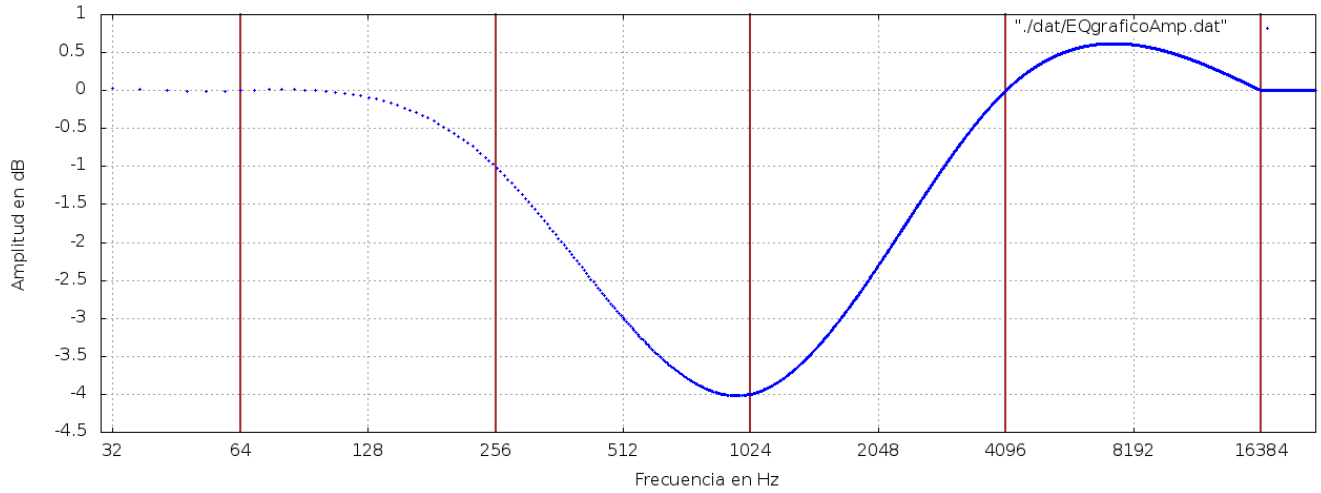


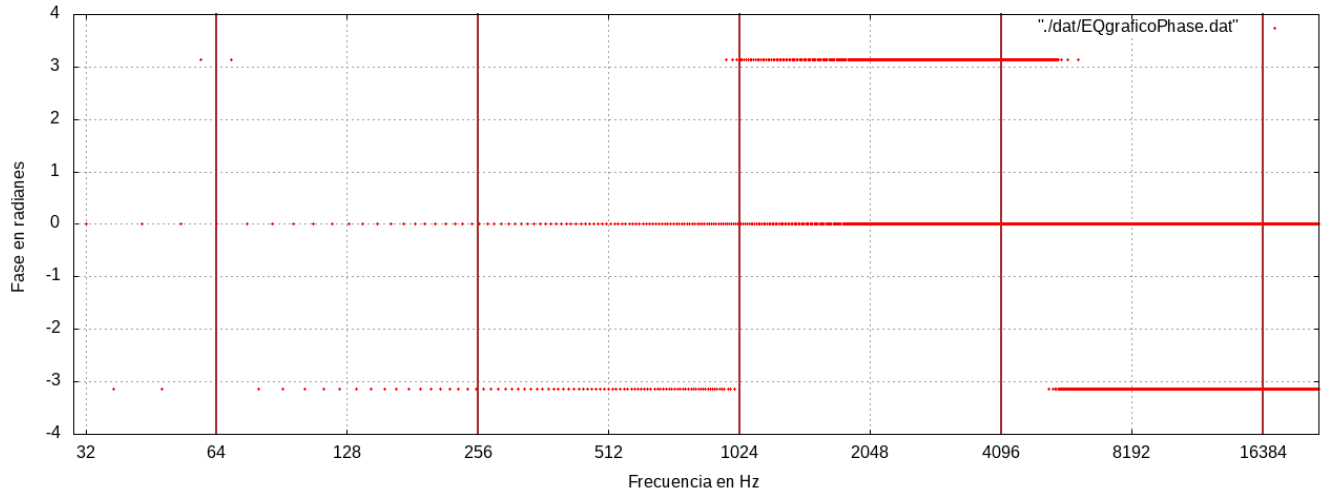
Figura 2
VENTANA TIPO BLACKMAN, UTILIZANDO 1024 *samples*

Finalmente mencionamos las funciones `splineCubico` y `polinomio`. Dados los cinco valores (en decibeles) de los parámetros elegidos para ecualizar se los interpola mediante un *spline* cúbico. Para esto se usa la función `polinomio` que genera el *spline* en el rango $[0, 5]$. Luego esto debe interpretarse como en escala logarítmica, pues así escucha el oído humano. Hay un problema con utilizar exactamente la curva generada por `polinomio`. Este es que fuera de las cinco frecuencias manipuladas, el gráfico de la función puede amplificar o atenuar mucho algunas frecuencias. Si bien estas se encuentran fuera del rango audible por el ser humano, pueden deformar e incluso sacar del rango dinámico posible con 16 *bits* al audio. Por ejemplo si las frecuencias de menos de $64Hz$ se amplifican demasiado. Con los agudos esto es peor, pues si bien, arriba de la frecuencia de Nyquist, no hay energía real, el ruido generado por el sampleo y la cuantización pueden agregar energía en dichas

frecuencias. Si estas se amplifican mucho también arruinarán el audio (como pudimos notar durante la implementación). El resultado fue que perdíamos rango dinámico al realizar la transformada inversa, obteníamos una señal de varios dBs menos. Para evitar este problema aplicamos una vuelta a ganancia unitaria paulatina a partir de la frecuencia de Nyquist y, en el caso de los bajos, antes de $64Hz$. El *spline* se genera en base a resolver un sistema líneal, el algoritmo se encuentra explicado en [6]. Un detalle interesante es que, al estar usando un filtro digital podemos sintetizar la fase que queramos. En este caso logramos un ecualizador de fase 0, como puede ser apreciado en la imagen[3].



Spline sampleado a partir de las ganancias $[0, -1, -4, 0, 0]$



Fase del ecualizador

B.4 Biblioteca `realdft`

A nivel matemático, esta es probabablemente la porción más interesante de código. Detallaremos la función `fftFunction` luego. Pero por ahora basta saber que realiza la transformada compleja, es decir, tranforma un vector de números complejos en otro de iguales características. Claramente la señal de audio es puramente real. Para obtener el espectro está la opción de agregar una parte

[3] Notar que, por una cuestión de redondeo, en algunos puntos, la fase parece ser $+\pi$ o $-\pi$ que, sabemos, es lo mismo que fase 0.

imaginaria trivial (todos ceros) para poder usar la función `fftFunction`. Esto supone una pérdida de eficiencia enorme, pues la mitad de los datos procesados son irrelevantes. Por esto seguimos la estrategia utilizada en [5]. Mediante algunas simples deducciones matemáticas, se muestra que pueden transformarse dos porciones de audio de misma longitud, utilizando como parte real una y como parte imaginaria otra. Luego se deben realizar algunas operaciones (líneales en el tamaño del arreglo) para obtener las dos transformadas. Es importante notar que la transformada rápida tiene una complejidad de $O(nlog(n))$, donde n es el tamaño del arreglo. Con lo cual, asintóticamente, este enfoque es mucho más eficiente. Para el caso de un audio *mono*, se utilizan dos arreglos consecutivos de audio para generar la parte real y la imaginaria. En el caso *stereo* se utiliza el lado izquierdo y el derecho. Las funciones que realizan estas operaciones son:

- `splitMono`
- `splitStere`
- `realFFTyaOrdenadosMono`
- `realIFFTyaOrdenadosMono`
- `realFFTyaOrdenadosStereo`
- `realIFFTyaOrdenadosStereo`

Comentamos ahora la implementación de la tranformada en sí. La idea del código fue tomada de [4] y se realizaron varias optimizaciones. Estas son: se implementó de forma iterativa en lugar de recursiva para evitar el *overhead* generado llamados a función y se redujo la escritura de datos al mínimo (organizando el orden en que se escriben los arreglos). Luego, se escribió en *assembler* el bucle central de la función.

C. Explicación código assembler

Las funciones que fueron pasadas a *assembler* son:

- `sumarSenalesASM`
- `multiplicarVentanasMonoASM`
- `multiplicarVentanasStereoASM`
- `getFlotasInt16ASM`
- `getIntsInt16_ditherASM`
- `multiplicarEspectrosASM`
- `bucleMedioFftASM`
- `splitMonoASM`
- `bucleMedioSplitStereoASM`
- `bucleEspectroMonoASM`
- `rIFFTyaosASM_1`
- `rIFFTyaosASM_2`
- `rIFFTyaomASM_1`
- `rIFFTyaomASM_2`

La optimización de las funciones que no representan una mejora significativa para el programa en su conjunto (pues se ejecutan durante una porción muy pequeña del tiempo total de ejecución) fue hecha por una cuestión de aprender la metodología del lenguaje y sus instrucciones. Además, las funciones simples permiten explicar mejor la sintaxis y la semántica de las instrucciones del lenguaje. Como se verá en la parte de testing, las funciones que no necesitaban realmente optimización, son las primeras 4 o 5.

A continuación realizamos una explicación de la implementación de cada función, detallando algunas instrucciones y metodologías.

C.1 `sumarSenalesASM`

Notamos primero el uso de la instrucción `vld` que permite cargar datos de memoria a registros. El ejemplo es `vld1.32 d0-d3, [r0]!`. Esta instrucción es muy interesante. Primero, permite cargar datos de forma *interleaved*, hecho que se comenta en la explicación de la función `multiplicarEspectrosASM`. En este caso el *interleave* es 1. Luego, permite cargar tantos elementos como quepan en un registro o una lista de hasta cuatro registros, esto puede amortizar mucho los accesos a memoria. En este caso se cargan tantos elementos de $32bits$ como quepan en los registros $d0$ a $d3$. Uno puede preguntarse para qué se explicita el tamaño de los elementos que quieran cargarse, si no se está realizando ninguna operación con ellos. Esta pregunta se responderá sola cuando veamos el uso del *interleaved load*. Préstese atención al "`!`" luego de la referencia a la posición apuntada por $r0$. Esto significa que, luego de la carga de los registros, el puntero $r0$ debe ser modificado, para apuntar a la posición siguiente a la del último elemento recién cargado, esto nos ahorra una suma luego de cada *load* y *store*.

Se utiliza la instrucción NEON aritmética `vadd`. Como se irá viendo, este *assembler* se maneja con una suerte de tipos. Un ejemplo es la línea `vadd.f32 d0, d0, d4`. Esta línea se traduce como "sumar el contenido de $d4$ (tercer operando) y el de $d0$ (segundo operando), interpretandolo como *floats* (`.f32`), y guardar el resultado en $d0$ (primer operando)". Notar que los registros $d0 - d31$ son de $64bits$, con lo cual, en ellos caben dos *floats*. Las instrucciones de este tipo son vectoriales, con lo cual, si llegaran a entrar varios elementos del tipo especificado en los registros especificados, se operará con ellos como si fueran vectores de elementos de ese tipo, realizando las operaciones índice a índice.

Otra cosa que hay para destacar es la *MACRO* utilizada (`sumate_8_flotas`). Esto es muy frecuente en *assembler* y resulta totalmente no-trivial en GAS, debido a su sintaxis complicada y a la falta de documentación. Esta *MACRO* fue hecha, no solo para que el código sea legible y mantenible, sino para explorar el comportamiento del *pipeline*. En particular, para intentar mantener el *pipeline* lleno. La idea es realizar operaciones de distintos tipos entrecruzadas. En este caso se combinan sumas con accesos a memoria, con el propósito de amortizar los accesos a memoria, aprovechando el tiempo para realizar operaciones aritméticas.

Finalmente notamos el hecho de que, por cada iteración del *loop* se suman 16 elementos de los vectores. Esto se denomina *loop unrolling* y reduce el *overhead* por saltos en el código que pueden ocasionar que se invalide el *pipeline*.

C.2 `multiplicarVentanasMonoASM`

Esta función es totalmente análoga a la explicada anteriormente, con la única diferencia de que se multiplican números.

C.3 `getFlotasInt16ASM`

En esta función destacamos la forma en que se convierten enteros a *floats*. Esta el la línea que realiza la conversión: `vcvt.f32.s32 q1, q1`. Gracias a la forma explícita en que se aclaran los tipos en el lenguaje y teniendo en cuenta que la operción es vectorial, la línea no podría ser más clara.

C.4 `multiplicarEspectrosASM`

Esta función no hace más que multiplicar dos vectores de números complejos coordenada a coordenada. Esta función sí se lleva una porción relativamente signficativa de tiempo de procesamiento.

Notamos primero el uso de cargas de memoria y a memoria de tipo *interleaved*. Esto se logra poniendo un 2, en este caso, luego del `vld` y el `vst`. Con esto se logra que dos elementos consecutivos se carguen en partes bajas de registros distintos. En este caso resulta muy útil para realizar la multiplicación de complejos, ya que, semánticamente, la distinción está en parte real e imaginaria; debemos operar con vectores de reales y vectores de imaginarios. En esta función, y en funciones posteriores, hacemos uso de índices dentro de un registro, por ejemplo $d1[0]$. Con esto se está haciendo referencia a la parte baja (en el ejemplo) o alta del mismo. Si bien $d0[0]$ es simplemente un alias de $s0$ debemos hacer uso del primero por una cuestión sintáctica del lenguaje, que no encontramos explicada en el manual.

Finalmente resaltamos el uso de las instrucciones `vmla` y `vmls`, multiplicaciones con acumulación. La primera multiplica el segundo y tecer parámetro y suma el resultado al primer parámetro. La segunda es análoga, pero lo resta. De esta manera nos ahorramos una multiplicación por cada elemento de los vectores.

C.5 `getIntsInt16_ditherASM`

La función comienza trayendo varias variables globales y guardándolas en registros. Las variables globales son interpretadas como posiciones de memoria, básicamente punteros.

Esta es la primera función de lógica no-trivial que pasamos a *assembler*. Las instrucciones usadas son básicamente las que ya explicamos, si bien vale la pena resaltar un detalle del lenguaje que resulta muy cómodo y que usamos en todas las funciones hasta ahora, sin mencionarlo: casi todas las instrucciones del lenguaje pueden ser condicionales. Es decir, puede agregárseles un sufijo de forma tal de que se ejecuten solamente si algún *flag* específico se encuentra levantado. Este es el caso del los `bne` que se encuentran al final de cada *loop* de las funciones hasta ahora explicadas, que *branchean*, saltan, únicamente si el *flag* de *Not Equal* se encuentra levantado. Un caso más interesante, donde usamos también esta *feature*, es en la instrucción `submi`. Si se mira el código en C de esta función se notará que, condicionalmente, se hace una resta de 1 como parte de un redondeo. Esta condicional resulta particularmente eficiente al usar los sufijos mencionados, en este caso, para chequear que la conversión anterior haya devuelto un número menor a cero.

C.6 `bucleMedioFftASM`

La función comienza tomando algunos parámetros que fueron pasados por pila. Luego entra directamente al *loop*. Como se trata de la función más costosa de todas se trabajó más en ella. El inconveniente principal de este bucle es que, a diferencia de otros casos, el número de ciclos a realizarse no tiene por qué ser un multiplo de 4 (sí sabemos que debe ser una potencia de 2, pero podía ser 1 o 2). Con lo cual no es inmediato realizar *loop unrolling*. Para poder lograrlo utilizamos una técnica clásica. Al comienzo del *loop* se chequea si el número de ciclos que quedan es mayor o igual a 4, en ese caso se opera con 4 elementos en paralelo. En caso contrario se chequea si la cantidad de ciclos restante es igual a 2. En tal caso se realizan 2 operaciones en paralelo. Si no se hace una sola.

Las operaciones en sí no son demasiado complejas y pueden entenderse mirando el código en C y leyendo la definición de DFT. En esta función, como en otras, puede notarse cómo se intercalan funciones aritméticas con accesos a memoria para mantener el *pipeline* lo más ocupado posible.

C.7 `splitMonoASM` y `bucleMedioSplitStereoASM`

Estas son las dos funciones de [5] que fueron optimizadas en *assembler*. Si bien en el artículo también se optimizan estas funciones en lenguaje de máquina, el lenguaje es muy distinto al que

utilizamos nosotros y, para evitar complicaciones, decidimos implementarlo de forma independiente.

En las dos funciones se pasan más de cuatro parámetros. De forma análoga a la convención INTEL los parámetros mas allá del cuarto se pasan por pila.

La lógica de las funciones es muy simple y casi todas las instrucciones utilizadas ya fueron explicadas en las funciones anteriores. Las únicas dos instrucciones nuevas son `vrev64` y `vswp`. Podría decirse que la razón de su uso es interesante, a continuación explicamos brevemente. Las funciones son una parte esencial de la transformación de dos señales reales a partir de una transformación compleja. En ambas deben realizarse operaciones entre un elementos a distancia k del inicio del arreglo y otro a distancia k del final del mismo. Como utilizamos instrucciones *SIMD*, debemos "espejar" los elementos de la segunda mitad del arreglo, una vez que los levantamos. Para esto utilizamos las dos instrucciones. Como en este caso utilizamos los registros $q0 - q8$, de $128bits$, y los elementos son de $32bits$, y, por otro lado, ninguna instrucción nos permite invertir el orden de elementos de $32bits$ en un registro de 128, recurrimos a dos inversiones. La primera invierte los elementos de la parte alta y la parte baja del registro, por ejemplo, $q7$. La segunda se usa, justamente, con las partes altas y las partes bajas, a las que hacemos referencia con los alias, en nuestro ejemplo, $d14$ y $d15$ respectivamente.

C.8 `bucleEspectroMono`, `rIFFTya*`

Ponemos todas estas funciones juntas pues no introducen metodologías o instrucciones novedosas. Destacamos, únicamente, el uso de la instrucción `vneg`, que toma un registro de 32 o $64bits$, interpreta su contenido como un número de simple o doble precisión, y le cambia el signo.

VII. ANÁLISIS DE *performance* Y *Testing*

Los *tests* fueron corridos en un *Samsung GT-N8013*. El sistema operativo es ANDROID 4,1. El procesador es de $1{,}4GHz$ y es un *Quad-Core*. La arquitectura es ARMv7. Promediamos 50 corridas en cada caso.

Los archivos de *testing* utilizados son dos audios de 10 minutos, que contienen ruido blanco. Uno es *mono* y el otro *stereo*.

A. Análisis de funciones puntuales

Usamos un *profiler* para analizar el tiempo utilizado por cada función de la clase `LibC`.

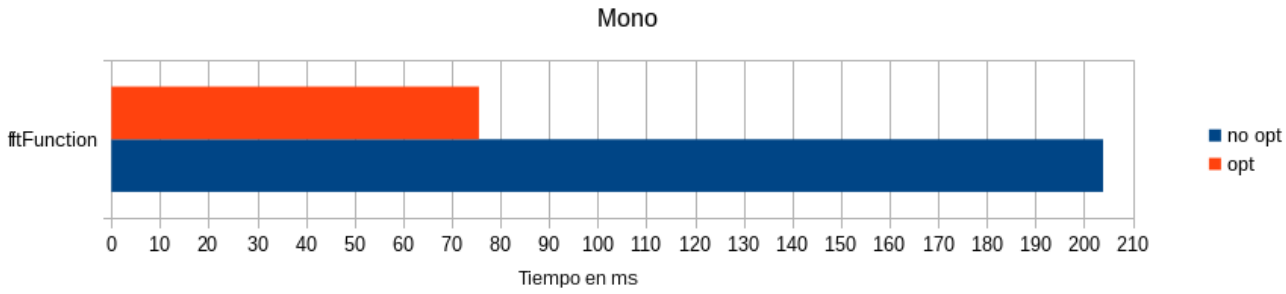
Como se especifica en [9, 16.1.1 Gprof], no es conveniente realizar este tipo de análisis utilizando optimizaciones del compilador. Entre otros motivos por cuestiones de *inlining* de funciones. Por esto comparamos el comportamiendo al utilizar la `LibC` escrita en C y al utilizar nuestras optimizaciones escritas en *assembler*.

En el apéndice de resultados numéricos puede verse que las funciones que toman mucho tiempo de ejecución fueron optimizadas. La única que faltó es `analizar`, pero por las carácteristicas de la misma, y no siendo particularmente costosa, no fue escrita en *assembler*[4].

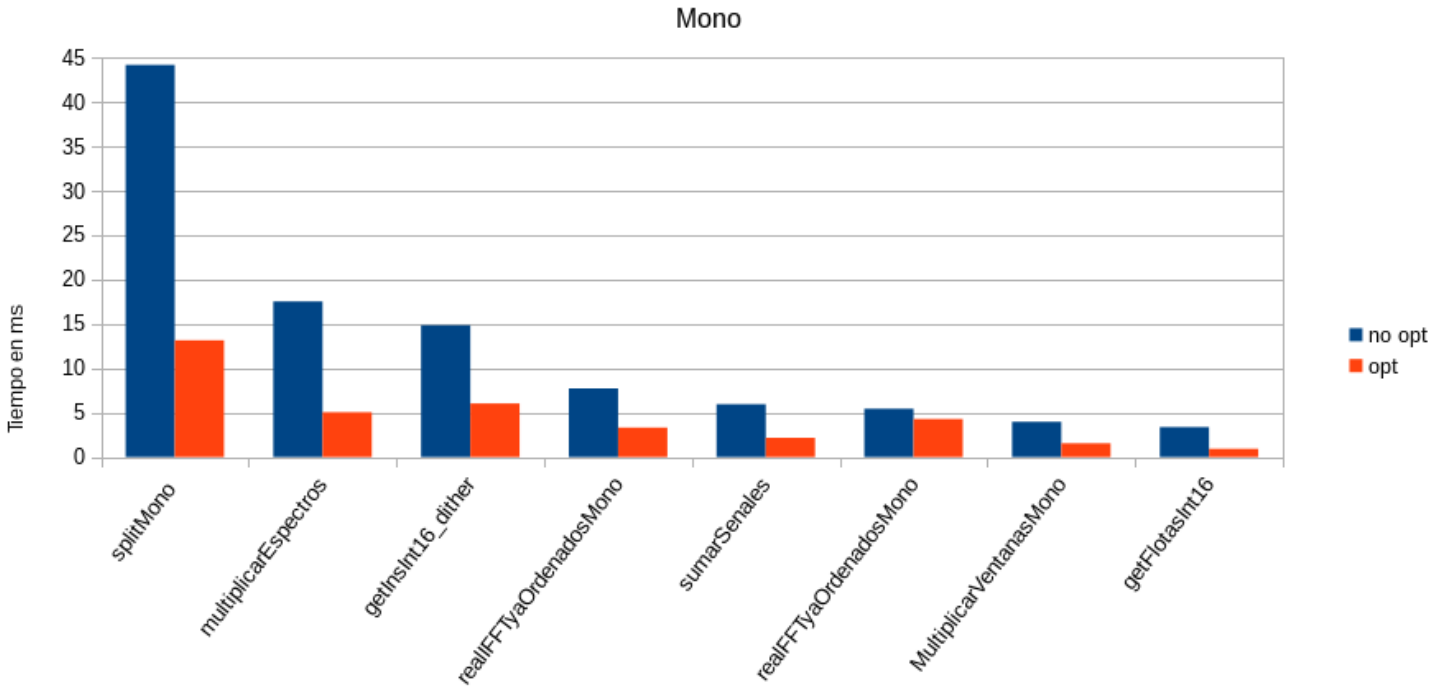
Otra cosa para notar, es que, de algunas funciones, solo se optimizaron algunos bucles, como es el caso de `splitStereo` y `fftFunction`. Para poder realizar la comparación con las funciones sin optimizar, se sumó el tiempo de cómputo de todas las porciones que conforman las funciones originales.

A continuación los resultados:

[4] El problema con la función es que realiza *loops* de distintas longitudes con una guarda complicada de chequar.



Profile de `funcionfft` corrida en canción *Mono*



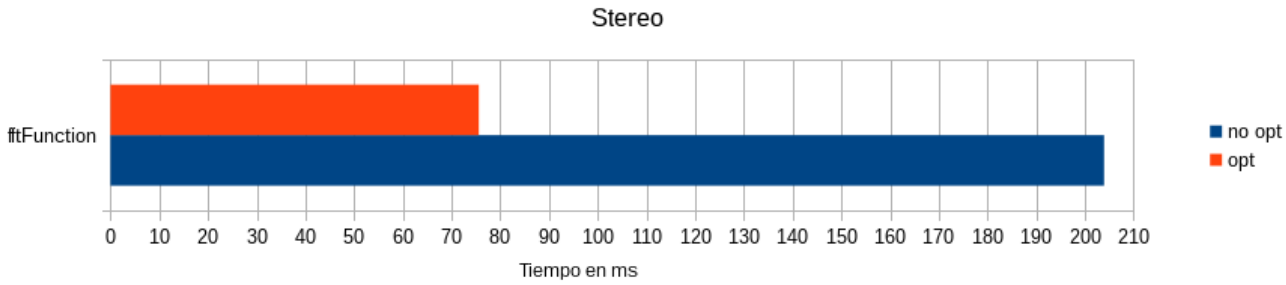
Profile del resto de las funciones corridas en canción *Mono*

	fftF	splitMono	multipEsp	getIntsInt	realIFFTy	sumarSen	realFFTya	multiVen	getFlotas
%	37	29.7	28.6	40.6	42.8	36.3	78.6	38.8	27.3

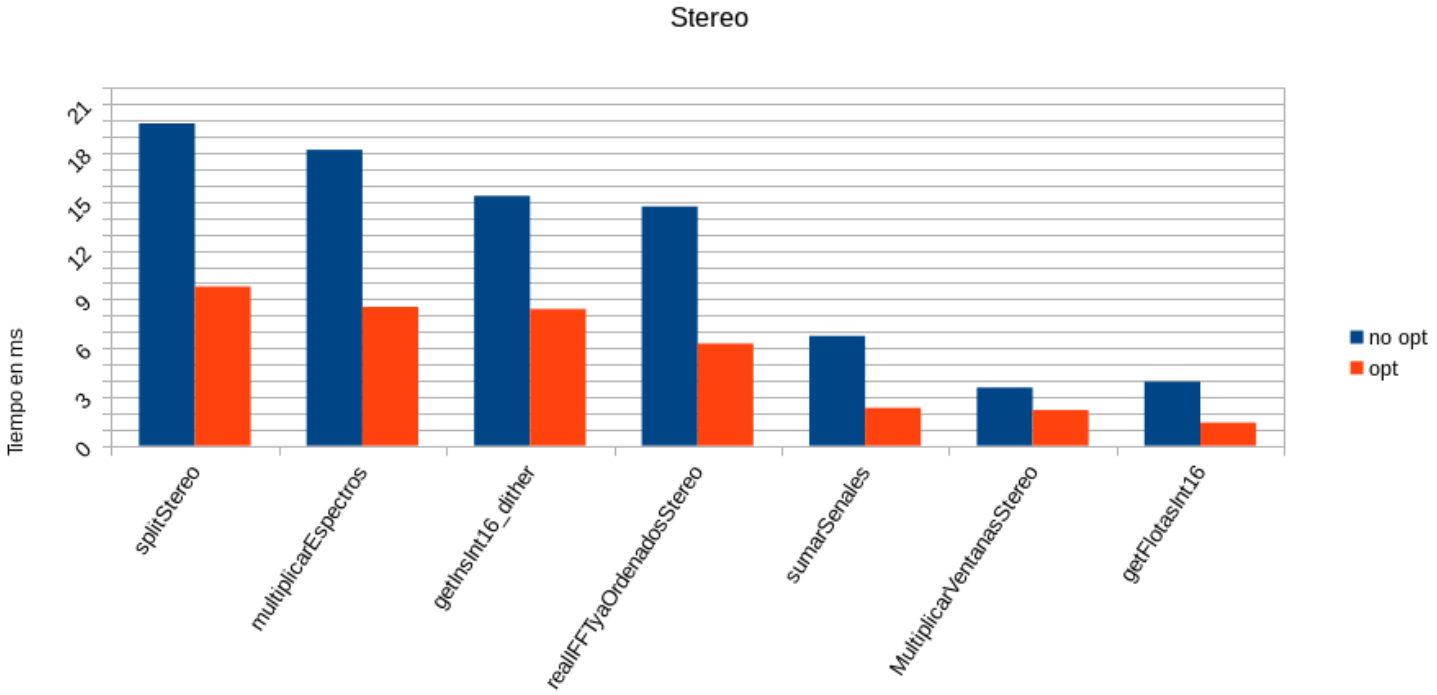
Porcentaje que toma la función optimizada con respecto a la original (*Mono*)

no opt	306.9
opt	112
%	36.5

Suma del tiempo consumido por las funciones sin optimizar y optimizadas (*Mono*)



Profile de `funcionfft` corrida en canción *Stereo*



Profile de funciones corridas en canción *Stereo*

	fftF	splitStereo	multipEsp	getIntsInt	realIFFTy	sumarSen	multiVen	getFlotas
%	37	49.4	46.9	54.7	42.7	34.4	61.3	36.1

Porcentaje que toma la función optimizada con respecto a la original (*Stereo*)

no opt	286.3
opt	114.5
%	40

Suma del tiempo consumido por las funciones sin optimizar y optimizadas (*Stereo*)

B. Análisis de performance global

El programa ejecutado para realizar este *test* se encuentra en el directorio `standAlone`. Los archivos utilizados son exactamente los mismos que los utilizados en la aplicación, menos los archivos de JAVA, y más un pequeño *main* en C.

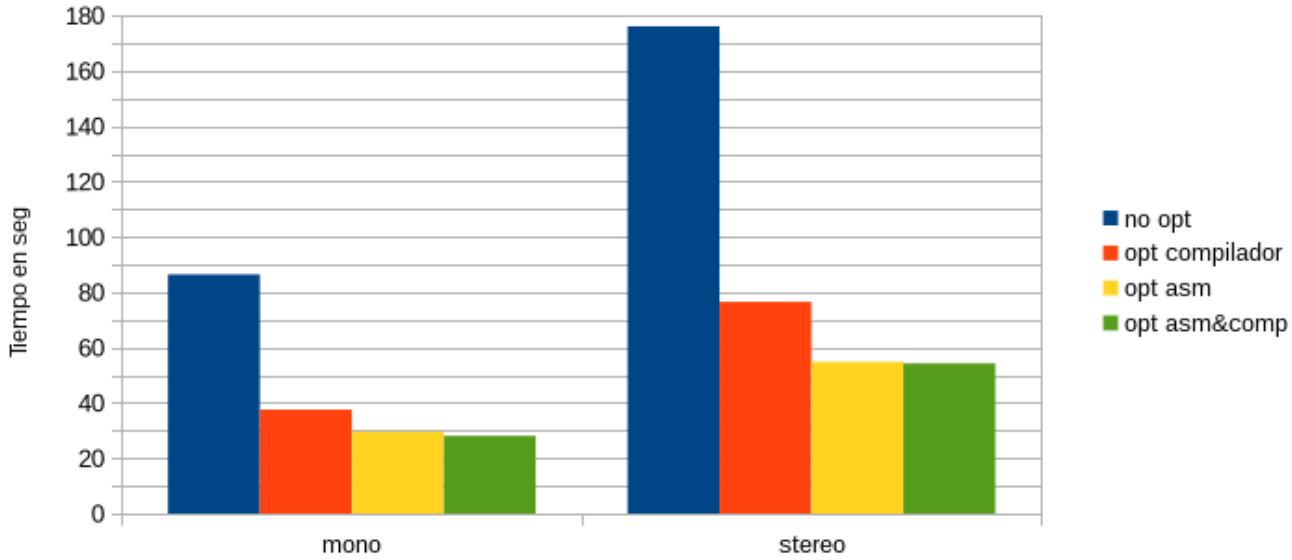
El programa simplemente ecualiza una canción entera, como si se tratara de la aplicación pero sin las interrupciones, pues no la reproduce. Tampoco realiza el análisis de espectro ya que todas las funciones optimizadas no son particulares al analizador.

Los temas ecualizados son los explicados anteriormente, 10*min mono* y 10*min stereo*.

Tanto para *mono* como para *stereo* realizamos el test de cuatro formas distintas:

- Sin optimizaciones de ningún tipo. Para esto usamos las funciones escritas en C y utilizamos el *flag* `-O0` al compilar.
- Con varias optimizaciones del compilador. Para esto utilizamos las funciones escritas en C y los *flags* `-O3` y `-ftree-vectorize`.
- Sin optimizaciones del compilador, pero utilizando nuestras optimizaciones. Es decir, usando todas las funciones que fueron optimizadas en *assembler*.
- Con optimizaciones del compilador y con nuestras optimizaciones.

Aprovechamos esta sección para comentar acerca de *bugs* encontrados gracias al uso de optimizaciones del compilador. Estos se debían a pisar registros que deben ser restituidos luego de los llamados a función por parte nuestra, en las funciones escritas en *assmebler*. Cuando el compilador realiza optimizaciones, muchas veces utiliza estos registros para guardar variables durante un llamado a función, evitando tener que almacenarlas en memoria. El problema resultaba más misterioso cuando se trataba de registros *SIMD*. Nosotros, al no restituir esas variables, las pisabamos y generabamos cualquier tipo de probelmas, que generalmente concluían en un *segmentation fault*. Encontramos estos *bugs* utilizando GDB para explorar el código cercano a las instrucciones que generaban *segmentation fault*.



COMPARACIÓN DE DIVERSAS OPTIMIZACIONES

MONO

```
        optCompilador   optASM      optASM&comp
%       43.5            34.1        32.5

STEREO
        optCompilador   optASM      optASM&comp
%       43.4            31.1        30.8
```

Porcentaje que toma el programa optimizado con respecto al sin optimizar

C. Análisis de los resultados

Las optimizaciones manuales fueron las más eficientes, si bien las optimizaciones de compilador se le aproximan bastante. La mayor diferencia se nota al ecualizar el tema completo sin reproducirlo (segundo *test*). Suponemos que esto se debe a un menor *overhead*, ya que el programa está implementado enteramente en código nativo y además no tiene concurrencia de *threads*. La menor cantidad de funciones usadas y el hecho de no realizar cambio de contexto de los *threads* aumenta los *hits* de las memorias *cache* y reduce los *pipeline*s invalidados por saltos en el código. Un caso particular donde esto se hace notorio es en el *loop unrolling*. Esta técnica reduce considerablemente los saltos en el código, pero es menos eficiente cuando se realizan muchos cambios de contexto, como es el caso del programa corriendo en su conjunto, con todos sus *threads*. Por estos motivos, el segundo *test* revela que nuestro código optimizado es 3 veces más rápido que el código compilado sin optimizaciones, mientras que las optimizaciones de compilador producen código 2,3 veces más rápido. El primer *test*, sin embargo, revela que en la práctica, nuestras optimizaciones corren 2,5 veces más rápido que las versiones optimizadas. Suponemos que esto se debe a los motivos detallados en el parrafo anterior ya que el *test* fue corrido justamente con la aplicación en su conjunto.

Por otro lado destacamos el hecho de que optimizaciones nuestras, en conjunto con optimizaciones de compilador, no genere código notoriamente más rápido que el que contiene únicamente nuestras optimizaciones, sugiere que hemos atacado todas las funciones importantes a la hora de optimizar.

Referencias

[1] *mathematics of the discrete fourier transform (DFT) with audio applications*,
Julius O. Smith III,
W3K Publishing, 2007

[2] *Spectral Audio Signal Processing*,
Julius O. Smith III,
W3K Publishing, 2011

[3] *The Scientist and Engineer's Guide to Digitas Signal Processing*,
Steven W. Smith,
California Technical Publishing

[4] Cooley-Tukey implementation in C,
http://en.literateprograms.org/Cooley-Tukey_FFT_algorithm_(C)

[5] *Implementing Fast Fouriere Transform Algorithms of Real-Valued Sequences With the TMS320 DSP Platform*,
Robert Matusiak,
Digital Signal Processing Solutions

[6] *Cubic Spline Interpolation*,
Sky McKinley and Megan Levine,
http://online.redwoods.edu/instruct/darnold/laproj/Fall98/SkyMeg/Proj.PDF

[7] Dither with noise-shaping,
http://www.musicdsp.org/archive.php?classid=5#61

[8] *Java Native Interface Specification*,
http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html

[9] *Cortex-A Series, Programmer's Guide*,
ARM, 2011

VIII. Apéndice de resultados numéricos

Incluimos, a continuación, la explicación de los parámetros principales analizados por el *profiler*.

```
 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.
```

Parámetros analizados por el *profiler*

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
63.93	203.92	203.92	32299	6.31	6.31	fftFunction
13.85	248.09	44.17	32291	1.37	1.37	splitMono
5.50	265.62	17.53	12919	1.36	1.36	multiplicarEspectros
4.65	280.46	14.84	6460	2.30	2.30	getIntsInt16_dither
3.60	291.93	11.47	6459	1.78	10.21	analizar
2.42	299.65	7.72	12916	0.60	8.28	realIFFTyaOrdenadosMono
1.86	305.59	5.94	12920	0.46	0.46	sumarSenales
1.71	311.03	5.44	19378	0.28	7.96	realFFTyaOrdenadosMono
1.24	315.00	3.97	19369	0.20	0.20	multiplicarVentanasMono
1.06	318.37	3.37	12913	0.26	0.26	getFlotasInt16
0.03	318.46	0.09	19378	0.00	7.97	realFFT

Procesando audio *Mono* sin optimizaciones

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
33.19	43.30	43.30				bucleMedioFftASM
24.75	75.59	32.29	32296	1.00	1.00	fftFunction
13.74	93.52	17.93	6454	2.78	3.78	analizar
10.09	106.68	13.16				splitMonoASM
4.62	112.71	6.03				getIntsInt16_ditherASM
3.86	117.75	5.04				multiplicarEspectrosASM
3.28	122.03	4.28				bucleEspectroMonoASM
1.66	124.19	2.16				sumarSenalesASM
1.31	125.90	1.71				rIFFTyaomASM_2
1.23	127.50	1.60				rIFFTyaomASM_1
1.18	129.04	1.54				multiplicarVentanasMonoASM
0.71	129.96	0.92				getFlotasInt16ASM

Procesando audio *Mono* con nuestras optimizaciones

%	cumulative	self	self	total

time	seconds	seconds	calls	ms/call	ms/call	name
73.86	261.08	261.08	32291	8.09	8.09	fftFunction
5.61	280.90	19.82	19374	1.02	1.02	splitStereo
5.15	299.10	18.20	25840	0.70	0.70	multiplicarEspectros
4.35	314.46	15.36	6460	2.38	2.38	getIntsInt16_dither
4.16	329.17	14.71	12920	1.14	9.22	realIFFTyaOrdenadosStereo
2.47	337.89	8.72	6460	1.35	10.94	analizar
1.91	344.64	6.75	12920	0.52	0.52	sumarSenales
1.11	348.57	3.93	12919	0.30	0.30	getFlotasInt16
1.01	352.14	3.57	19372	0.18	0.18	multiplicarVentanasStereo

Procesando audio *Stereo* sin optimizaciones

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
40.29	76.41	76.41				bucleMedioFftASM
30.92	135.04	58.63	32295	1.82	1.82	fftFunction
6.97	148.26	13.22	6456	2.05	3.87	analizar
5.16	158.05	9.79				bucleMedioSplitStereoASM
4.50	166.59	8.54				multiplicarEspectrosASM
4.43	174.99	8.40				getIntsInt16_ditherASM
2.00	178.79	3.80				rIFFTyaosASM_1
1.31	181.27	2.48				rIFFTyaosASM_2
1.22	183.59	2.32				sumarSenalesASM
1.15	185.78	2.19				multiplicarVentanasStereoASM
0.75	187.20	1.42				getFlotasInt16ASM

Procesando audio *Stereo* con nuestras optimizaciones

	mono	stereo
no opt	86.4	176.1
opt comp	37.6	76.5
opt asm	29.5	54.8
opt asm&comp	28.1	54.3

Tiempos en segundos utilizando distintas optimizaciones