Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Centro de Computación Gráfica

Procesamiento Digital de Audio en GPU con CUDA

Trabajo Especial de Grado
presentado ante la Ilustre
Universidad Central De Venezuela
por el bachiller
Alex Alberto Pérez San Elías
para optar al título de
Licenciado en Computación

Tutor
Ernesto Coto

Caracas, 28 de Octubre de 2011

Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

ACTA DEL VEREDICTO

Quienes suscriben, miembros del jurado designado por el Consejo de la Escuela de Computación para examinar el Trabajo Especial de Grado, presentado por el Bachiller Alex Alberto Pérez San Elías, C.I. 16.461.267, con el título "Procesamiento Digital de Audio en GPU con CUDA", a los fines de cumplir con el requisito legal para optar al título de Licenciado en Computación, dejan constancia de lo siguiente:

Leído el trabajo por cada uno de los miembros del jurado, se fijó el día 28 de Octubre de 2011, a las 11:00 a.m., para que su autor lo defendiera en forma pública, en el Centro de Computación Gráfica, lo cual este realizo mediante una exposición oral de su contenido, y luego respondió satisfactoriamente a las preguntas que les fueron formuladas por el jurado, todo ello conforme a lo dispuesto en la Ley de Universidades y demás normativas vigentes de la Universidad Central de Venezuela. Finalizada la defensa pública del Trabajo Especial de Grado, el jurado decidió APROBARLO.

En fe de lo cual se levanta la presente acta, en Caracas el 28 de Octubre de 2011, dejándose también constancia de que actuó como coordinador del jurado el Profesor, Ernesto Coto, tutor del trabajo.

_____

Prof. Ernesto Coto
(Tutor)

_____

Prof. Esmitt Ramírez
(Jurado Principal)

_____

Prof. Robinson Rivas
(Jurado Principal)

Resumen

Muchos investigadores se han interesado en explorar el gran poder computacional de las últimas Unidades de Procesamiento Gráfico (GPU – *Graphics Processing Units*) en aplicaciones fuera del dominio de los gráficos. Esta tendencia al desarrollo del GPU de propósito general (GPGPU – *General Purpose on Graphics Processing Units*) se ha intensificado con el lanzamiento de diversos APIs (*Application Programming Interfaces*) para GPU entre los que se destaca el lenguaje CUDA (*Compute Unified Device Architecture*) de la empresa NVIDIA. Gracias a estas herramientas de desarrollo el GPU ha sido ampliamente utilizado para resolver problemas de diversas índoles, como los de procesamiento de señales, que en muchos casos demandan una importante carga de recursos computacionales.

En la producción musical, son empleados cada vez más algoritmos de gran demanda computacional. Para satisfacer esta demanda de potencia de cálculo, los usuarios adquieren hardware de procesamiento de audio sumamente costoso. En este Trabajo Especial de Grado se muestra una alternativa accesible y portable usando el GPU para el procesamiento digital de audio.

Este trabajo presenta un método para procesar señales digitales de audio utilizando el GPU. Este enfoque explota el paralelismo de los multiprocesadores de las tarjetas de video para alcanzar un mejor rendimiento que en las implementaciones basadas en CPU.

La efectividad de este enfoque es demostrada con la implementación en tiempo real y de forma paralela en el GPU, de algunos efectos de audio clásicos que son comúnmente utilizados en un entorno de producción musical. Se analizan dichos algoritmos tanto en CPU como en la Unidad de Procesamiento Gráfico (GPU), comparando en qué casos es mejor realizar los cálculos en el GPU en lugar de en el CPU.

Palabras Clave: GPGPU, CUDA, procesamiento digital de audio, efectos de audio, procesamiento de audio en tiempo real.

Tabla de Contenido

Índice de Figuras

Introducción

Actualmente en la producción musical la mayoría del audio es manipulado de forma digital, grabado y organizado por pistas. En estas grabaciones el procesamiento digital es aplicado para mejorar el sonido final, ya sea resaltando o atenuando frecuencias, aplicando efectos especiales o eliminando ruidos, entre otros.

En los últimos años, el acelerado desarrollo de las tarjetas gráficas y sus bajos costos han abierto las puertas del poder de la computación paralela a los consumidores. Igualmente, los programadores han prestado mayor atención a la programabilidad computacional de propósito general que los procesadores gráficos ofrecen desde la reciente aparición de lenguajes de programación para GPU (*Graphics Processing Units*) como CUDA (*Compute Unified Device Architecture*) y OpenCL (*Open Computing Language*), Estas condiciones han hecho que este recurso sea utilizado de forma muy novedosa.

Trabajos recientes han explotado el paralelismo de la computación de propósito general en los procesadores gráficos, denominado GPGPU (*General Purpose computation on Graphics Processing Units*). Se han creado aplicaciones de criptografía, simulaciones físicas y procesamiento de señales, entre otros. El interés en el aprovechamiento del GPU (*Graphics Processing Unit*) ha crecido a raíz de la creación de lenguajes de alto nivel para la programación en hardware gráfico.

Un campo reciente del procesamiento en hardware gráfico de propósito general es el procesamiento digital de audio, que ha demostrado en trabajos anteriores **[1] [2] [3] [4]** ser una herramienta útil para utilizar el GPU como procesador secundario en ciertas tareas relacionadas con el procesamiento de audio y en algunos casos se ha demostrado la capacidad del GPU de superar al CPU en algunos de estos algoritmos, además de tener a disposición librerías de FFT (*Fast Fourier Transform*) implementadas para GPU que son muy utilizadas en el ámbito de procesamiento digital de señales.

Este Trabajo Especial de Grado presenta y desarrolla una novedosa forma de procesamiento digital de audio en GPU como alternativa al procesamiento en CPU, que es el comúnmente utilizado para los cálculos de los algoritmos de procesamiento de audio.

Este trabajo está dividido en cuatro capítulos principales. El primer Capítulo describe el Planteamiento del Trabajo Especial de Grado, se describe el planteamiento general del problema, su posible solución y los objetivos a realizar en este Trabajo Especial de Grado. El Capítulo 2 contiene los fundamentos necesarios para entender la teoría del procesamiento digital de audio así como los conceptos básicos de la computación paralela en GPU con CUDA. En el Capítulo 3 se encuentran los detalles de diseño e implementación tanto de los algoritmos de audio en GPU y CPU como de la implementación de la interfaz gráfica para manipular los parámetros de los efectos de audio. El Capítulo 4 contiene el análisis y los resultados de las pruebas de rendimiento de los algoritmos implementados descritos en el capítulo de diseño e implementación, así como se muestran comparativas de velocidad de cálculo de cada procesador. Finalmente, en el Capítulo 5, se presentan las conclusiones y posibles trabajos futuros de esta investigación.

Capítulo 1. Planteamiento del TEG

1.1. Planteamiento del problema

Los cálculos relacionados con el procesamiento digital de audio generalmente recaen completamente sobre el CPU, lo que a menudo implica que el rendimiento se degrade al utilizar numerosos efectos de audio al mismo tiempo. Para expandir el potencial computacional, permitiendo utilizar más filtros y efectos sobre más pistas de audio de forma concurrente, algunas empresas han creado procesadores dedicados al procesamiento digital de audio, pero son sumamente costosos para el usuario no profesional.

Con el poder de cómputo que actualmente tienen los procesadores gráficos, su bajo costo y el alto nivel de programabilidad que se ha logrado en los últimos años se puede crear una solución al problema de la sobrecarga al CPU implementando en el GPU los algoritmos de efectos de audio más utilizados, logrando mejoras en rendimiento, velocidad de cálculo y una posible alternativa a los procesadores de audio profesionales dedicados.

1.2. Solución propuesta

La solución que se propone estará basada en un *plugin* VST (*Virtual Studio Technology*) que contendrá un conjunto de efectos de audio procesados en el GPU y contará con una interfaz gráfica de usuario con controles básicos para la manipulación de los parámetros de cada efecto de audio.

1.3. Objetivo general

Adaptar una serie de algoritmos de procesamiento digital de audio en GPU utilizando CUDA para reducir la carga al CPU cuando se utilizan numerosos plugins de audio en las aplicaciones de edición de audio.

1.4. Objetivos específicos

- Implementar una serie de algoritmos de audio en el GPU con CUDA.
- Crear un *plugin* VST compatible con la mayoría de las aplicaciones DAW que existen para el sistema operativo *Windows*.
- Hacer procesamiento en tiempo real del audio.
- Crear una interfaz gráfica de usuario para la manipulación de los efectos de audio.
- Hacer pruebas de rendimiento para hacer comparativas GPU vs. CPU.

1.5. Alcance y limitaciones de este trabajo

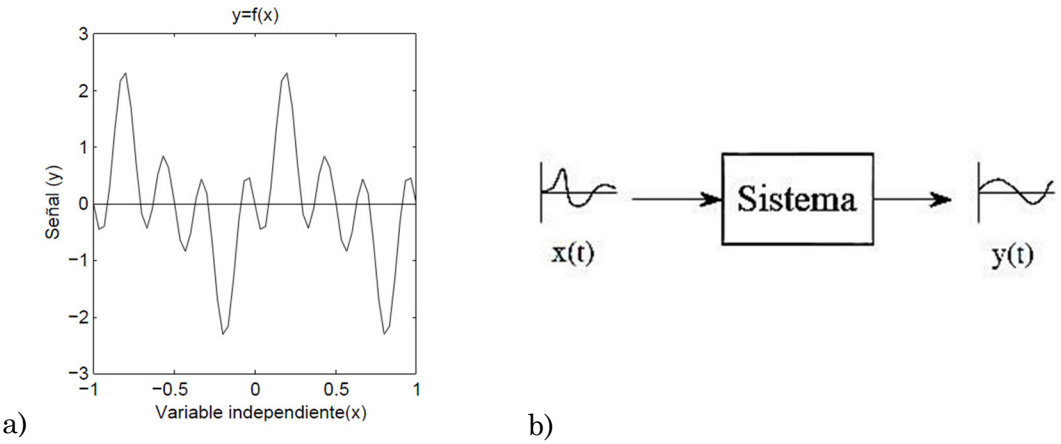
- Este trabajo está enfocado en el desarrollo de un *plugin* de audio VST, aplicando la teoría básica de procesamiento digital de señales en el contexto de programación de audio, aprovechando la capacidad de cálculo del GPU para realizar las operaciones involucradas en cada algoritmo de los efectos de audio implementados.
- Las implementaciones para GPU de los algoritmos serán desarrolladas con el lenguaje de programación CUDA, y por lo tanto solo pueden ser ejecutadas en tarjetas de video NVIDIA.
- Las tarjetas de video NVIDIA deben de tener al menos *Compute Capability 1.1*, ya que se utiliza mapeo de memoria, característica que no se encuentra en las tarjetas de video NVIDIA de menor capacidad.
- Durante las pruebas del prototipo a desarrollar se realizarán todas las pruebas con una frecuencia de muestreo de 44.100 muestras por segundo y un tamaño de bloque de audio de 512 muestras.

Capítulo 2. Marco Teórico

En este capítulo se presentan los conceptos básicos de la teoría de señales y sistemas, los cuales son fundamentales para entender y estudiar como una señal de audio puede ser modificada de distintas maneras. También se muestran los conceptos básicos de la computación paralela en GPU con el API de programación CUDA.

2.1. Señales

Una señal fundamentalmente es la representación de la variación de alguna magnitud física que se utiliza para transmitir algún tipo de información (ver **Figura 2.1.a**). Todos los tipos de señales, sean éstas naturales o artificiales, tienen características comunes que hacen posible su estudio en forma independiente de su naturaleza. Por lo general, las señales son procesadas o modificadas por sistemas (ver **Figura 2.1.b**). En el caso computacional, el computador genera o modifica una señal acústica digitalizada, por lo tanto se comporta como un sistema.



a) b)

Figura 2.1: a) Gráfica de una señal arbitraria. b) Representación de un sistema básico **[5]**.

Una señal, en forma simplificada, se puede entender como cualquier mecanismo que es utilizado para transmitir información. Algunos ejemplos de señales son: ondas electromagnéticas enviadas por un radar, ondas de sonido viajando por el aire o las ondas de la actividad del cerebro captadas por un electrocardiograma. Desde un punto de vista matemático, una señal es una variable de una o más dimensiones que toma valores de acuerdo a otra variable, como por ejemplo el tiempo en el caso del sonido, o el espacio en el caso de imágenes.

Matemáticamente, si la dependencia es temporal o espacial carece de importancia, ya que en ambos casos las señales se tratan de igual manera y sólo importa la función que modela su comportamiento **[5].**

2.2. Señales de audio

El audio es un tipo de señal producida por una perturbación física que se propaga a través de un medio (aire, agua, etc.) y que puede variar a lo largo del tiempo.

Existen distintos tipos de ondas que difieren en su naturaleza, por ejemplo ondas de radio, de luz, de agua, electromagnéticas, de sonido o rayos X. También difieren en su forma de propagación, la que puede ser longitudinal o transversal. En una onda longitudinal, el desplazamiento de las partículas es paralelo a la dirección de propagación de la onda. En una onda transversal, el desplazamiento de las partículas es perpendicular a la dirección de propagación de la onda. Las ondas de sonido son longitudinales. Cuando esta onda alcanza alguna superficie (como el tímpano del oído humano o la membrana de un micrófono), produce una vibración en dicha superficie por simpatía o resonancia. De esta forma, la energía acústica es transferida desde la fuente al receptor manteniendo las características de la vibración original.

El sonido puede viajar a través de objetos sólidos, líquidos o gases. Su velocidad es proporcional a la densidad del medio por el cual viaja. A temperatura ambiente, la velocidad del sonido en el aire es de 343 m/s y en el agua, es de 1493 m/s. La velocidad del sonido es independiente de su intensidad y su intensidad decae con la distancia en forma inversamente proporcional.

2.2.1. Sinusoides

Una señal de audio puede ser descrita a través de las señales de tipo sinusoidal. La sinusoide es una de las señales más simples e importantes que existen. Las dos funciones matemáticas sinusoidales básicas son las funciones seno y coseno, las cuales están derivadas de las funciones trigonométricas del mismo nombre.

Las señales de audio pueden describirse de acuerdo a algunos pocos parámetros (ver **Figura 2.2**). Los tres parámetros más comunes son: la amplitud, que es la altura de la señal; la frecuencia, que es el número de oscilaciones por unidad de tiempo; y la fase de la onda, que es el desplazamiento con respecto al eje del tiempo. Existen algunos otros parámetros tales como período, espectro o intensidad que pueden derivarse de los parámetros mencionados anteriormente.

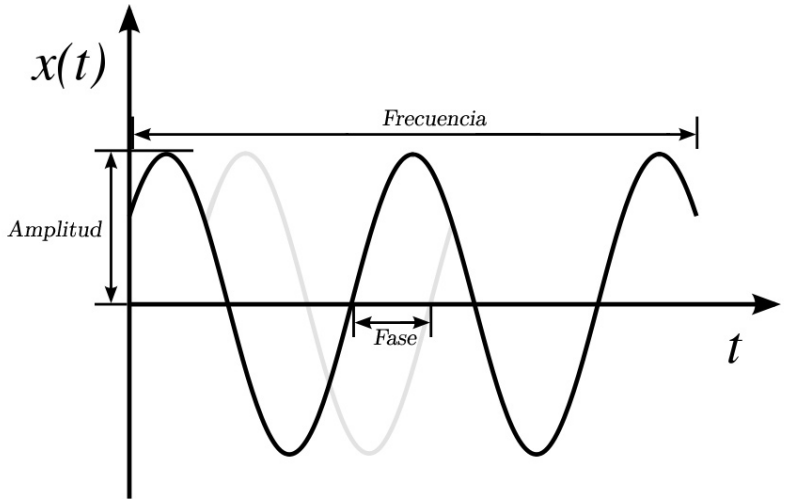


Figura 2.2: Representación de una señal de audio y sus parámetros básicos (amplitud, fase y frecuencia).

Matemáticamente una función de este tipo se puede escribir como:

$$y(t) = \sum_{i=1}^{N} A_i(t) \sin[2\pi f_i(t)t + \theta_i(t)] \qquad (\boldsymbol{Ec.\ 2.1})$$

Donde $A_i(t)$, $f_i(t)$, y $\theta_i(t)$ son el conjunto de amplitudes, frecuencias y fases de las sinusoides, con respecto a un instante de tiempo t.

Las señales de voz o música, pueden ser descritas de forma bastante precisa como la suma de una gran cantidad de sinusoides de diferentes amplitudes.

Otra razón de la importancia de las sinusoides es que constituyen funciones básicas de los sistemas lineales. Esto significa que cualquier sistema lineal puede ser estudiado a través de su respuesta a funciones sinusoidales.

En el campo del audio, las sinusoides son importantes para el análisis de filtros tales como reverberadores, ecualizadores y otros tipos de efectos.

Desde un punto de vista matemático, las sinusoides constituyen bloques fundamentales que al ser combinados de cierta forma permiten la creación o síntesis de cualquier tipo de señal, por muy compleja que sea.

2.2.2. Exponenciales

Existe otra función matemática muy importante, conocida como exponencial. La forma canónica de una función exponencial es la siguiente:

$$f(t) = Ae^{-t/\tau} \quad t \geq 0 \qquad (\boldsymbol{Ec.\ 2.2})$$

A corresponde a la amplitud máxima de la exponencial y τ se conoce como la constante de tiempo de la exponencial. La constante de tiempo es el tiempo que demora la exponencial en decaer $1/e$, es decir:

$$\frac{f(\tau)}{f(0)} = \frac{1}{e} \qquad (\boldsymbol{Ec.\ 2.3})$$

La **Figura 2.3** muestra el gráfico de una función exponencial para $A = 1\ y\ \tau = 1$. En la **Ecuación 2.4** e es el número de Euler, el cual tiene el valor irracional aproximado de 2.71828 y constituye la base de los logaritmos naturales. Este número es uno de los más importantes de la matemática y puede calcularse como:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n \approx 2.71828 \qquad (\boldsymbol{Ec.\ 2.4})$$

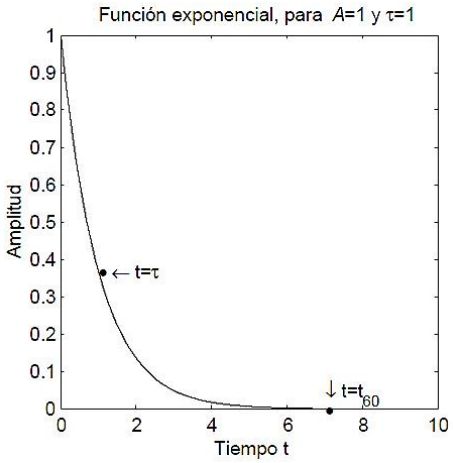


Figura 2.3: Función exponencial cuando $\boldsymbol{A = 1}$ y $\boldsymbol{\tau = 1}$ **[5]**.

El decaimiento exponencial ocurre naturalmente cuando una cantidad está decayendo a una tasa proporcional a lo que aún queda por decaer. En la naturaleza, todos los resonadores lineales, tales como las cuerdas de los instrumentos musicales y los instrumentos de vientos exhiben un decaimiento exponencial en su respuesta a una excitación momentánea. La energía reverberante en una sala de conciertos decae exponencialmente una vez finalizada la emisión de sonido. Esencialmente, todas las oscilaciones libres (sin una fuente mantenida en el tiempo) caen exponencialmente siempre que sean lineales e invariantes en el tiempo. Ejemplos de este tipo de oscilaciones incluyen la vibración de un diapasón, cuerdas pulsadas y barras de marimba o xilófonos.

El crecimiento exponencial ocurre cuando una cantidad crece a una tasa proporcional al incremento actual. El crecimiento exponencial es inestable dado que nada puede crecer para siempre sin llegar a un cierto límite.

Es necesario notar que en la **Ecuación 2.2** una constante de tiempo positiva implica a un decaimiento exponencial mientras que una constante de tiempo negativa corresponde a un crecimiento exponencial.

En sistemas de audio, un decaimiento de $1/e$ se considera muy pequeño para aplicaciones prácticas, como por ejemplo, para el diseño acústico de salas de concierto. Por lo general, se utiliza una cantidad tal que asegure que la señal ha caído 60 decibelios (dB). Esta cantidad, denotada por t_{60}, se encuentra resolviendo la ecuación:

$$\frac{f(t_{60})}{f(0)} = 10^{-60/20} = 0.001 \qquad (\boldsymbol{Ec.\ 2.5})$$

Usando la definición de exponencial de la **Ecuación 2.2**, se tiene entonces que:

$$t_{60} = -ln(0.001)\tau \approx 6.91\tau \qquad (\boldsymbol{Ec.\ 2.6})$$

Esta ecuación nos dice que la constante t_{60} es aproximadamente 7 veces la constante de tiempo τ. Esto puede verificarse en la **Figura 2.3**, donde se aprecia la ubicación de dicha cantidad en el eje del tiempo.

2.2.3. Relación entre sinusoides y exponenciales

Existen dos relaciones muy importantes para el análisis de señales y para el Teorema de Fourier, llamadas ecuaciones de Euler. Estas son:

$$e^{ix} = cos(x) + i\, sin(x) \qquad (\boldsymbol{Ec.\ 2.7})$$

$$e^{-ix} = cos(x) - i\, sin(x) \qquad (\boldsymbol{Ec.\ 2.8})$$

Esta ecuación nos dice que las exponenciales y las sinusoides están íntimamente relacionadas. En la **Ecuación 2.7**, el número i representa al número imaginario y que está definido como:

$$i = \sqrt{-1} \qquad (\boldsymbol{Ec.\ 2.9})$$

El número imaginario i tiene una importancia fundamental en el análisis de frecuencia de una señal, ya que tal y como se verá más adelante, las sinusoides pueden representarse y manejarse en forma más compacta si se utilizan números complejos. Los números complejos están constituidos por un par ordenado de números, uno real y otro imaginario, y usualmente se grafican en lo que se denomina el plano complejo, mostrado en la **Figura 2.4**, donde el eje de las ordenadas representa los números reales y el eje de las abscisas los números imaginarios. En este plano un número complejo es un vector que se puede representar de dos formas: cartesiana y polar. En la forma cartesiana, un número complejo Z se representa como la suma de su parte real con su parte imaginaria o bien $Z = x + iy$. Pero el mismo número se puede representar mediante la longitud del vector y su ángulo con respecto al eje de las ordenadas, lo que se denomina forma polar.

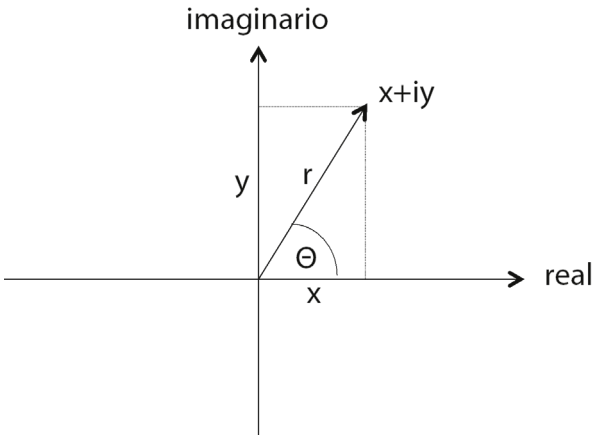


Figura 2.4: El plano complejo para la forma cartesiana.

En este caso se tiene $Z = r\angle\theta$, donde r es el módulo o magnitud del número complejo y que también suele representarse como $|Z|$.

Ambas representaciones están relacionadas por las siguientes ecuaciones:

$$r = \sqrt{x^2 + y^2} \qquad (\boldsymbol{Ec.\ 2.10})$$

$$\theta = \arctan\left(\frac{y}{x}\right) \qquad (\boldsymbol{Ec.\ 2.11})$$

Las **Ecuaciones 2.7** y **2.8** pueden utilizarse para demostrar que:

$$cos(x) = \frac{e^{ix} + e^{-ix}}{2} \qquad (\boldsymbol{Ec.\ 2.12})$$

$$sin(x) = \frac{e^{ix} - e^{-ix}}{2i} \qquad (\boldsymbol{Ec.\ 2.13})$$

Multiplicando la **Ecuación 2.7** por una amplitud $A \geq 0$ y utilizando $x = wt + \phi$ se tiene:

$$Ae^{i(wt+\phi)} = Acos(wt + \phi) + iB\ sin(wt + \phi) \qquad (\boldsymbol{Ec.\ 2.14})$$

Esta ecuación describe una sinusoide compleja. Por lo tanto, una sinusoide compleja contiene una parte real coseno y una parte imaginaria seno.

De acuerdo a las **Ecuaciones 2.12** y **2.13** y dado que e^{iwt} corresponde a una sinusoide compleja (**Ecuación 2.14**), se tiene entonces que toda sinusoide real está compuesta por una contribución equitativa de frecuencias positivas y negativas. Dicho de otra forma, una sinusoide real consiste de una suma de dos sinusoides complejas, una de frecuencia positiva y la otra de frecuencia negativa.

Esto significa que el espectro de frecuencias de una sinusoide o de una función periódica compuesta por sinusoides es simétrico respecto del origen y contiene tanto frecuencias negativas como positivas.

En la próxima subsección se describe como una señal se define digitalmente y como puede ser procesada para mejorar o modificar las señales de audio. Lo que se entiende como señal digital comparada con una señal analógica se explica también en la siguiente subsección.

2.3. Señales analógicas y digitales

Las señales analógicas como naturalmente ocurren en las señales de audio, son señales que varían en tiempo continuo. Definiremos tal señal como $x(t)$. Ellas están definidas para cada valor de tiempo en un intervalo continuo. Las señales en tiempo discreto están solamente definidas en ciertos valores de tiempo. Cuando se interpretan señales en la computadora, se necesita que estén en forma digital.

La señal analógica es convertida a digital a través de un muestreo. Para digitalizar una señal analógica es necesario discretizarla. Es decir, tomar muestras a intervalos regulares de la señal analógica original. Este proceso se conoce como digitalización o muestreo. La calidad de la señal discreta depende de la frecuencia a la cual se muestrea la señal original. Esto se ilustra en la **Figura 2.5.a.** donde cada flecha representa una medida de la muestra en un tiempo dado. La señal en consecuencia está en tiempo discreto.

Una señal digital puede ser representada solamente en un conjunto limitado de valores, por lo tanto las muestras tomadas tienen que ser cuantizadas al valor más cercano que la pueda representar como se ve en la **Figura 2.5.b.** La frecuencia de muestreo se refiere al número de muestras en una señal digital por segundo y la resolución se refiere al número de niveles de cuantización. Esto indica que una señal digital no solamente tiene un tiempo discreto sino un valor de muestra discreto también.

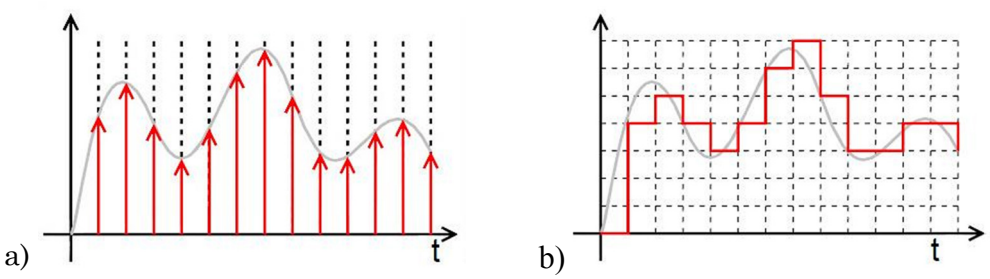


Figura 2.5: a) Toma de muestras de una señal analógica. b) Su representación de forma digital a través de la cuantización **[3]**.

Usualmente el muestreo ocurre en intervalos de tiempo fijos. Definiremos a la señal digital como $x(n)$ donde n es el número de índice, empezando desde $n = 0$. Teniendo la frecuencia adecuada de muestreo la señal analógica podrá ser perfectamente representada en forma digital. Esta frecuencia adecuada de muestreo es llamada la frecuencia de Nyquist y es dada por el teorema de muestreo. La frecuencia de muestreo es dada como el número de muestras por unidad de tiempo.

2.3.1. Teorema de muestreo de Nyquist

Si la frecuencia más alta contenida en una señal analógica $x_a(t)$ es $f_{max} = B$ y la señal es muestreada a la frecuencia $F_s > 2\, f_{max} \equiv 2B$, entonces $x_a(t)$ podrá ser recuperada exactamente a partir de los valores de las muestras.

Esto significa que la señal debe ser muestreada con una frecuencia de muestreo de al menos el doble de la frecuencia más alta contenida en la señal. Esta frecuencia de muestreo es lo que se conoce como la Frecuencia de Muestreo de Nyquist.

Para manipular el sonido, la señal puede ser transformada del dominio del tiempo al dominio de la frecuencia y viceversa. Esto es de suma utilidad para poder manejar de forma más accesible parámetros de las señales tales como la frecuencia o la fase. En la siguiente subsección se explicarán los conceptos básicos relacionados a dicha transformación, denominada Transformada de Fourier.

2.4. Series de Fourier

En el año 1811, el matemático Jean Baptiste Fourier demostró que cualquier señal periódica razonable puede expresarse como la suma de una o más sinusoides de distinta frecuencia, fase y amplitud. Se entiende por una señal razonable, una que posee valores máximos menores que infinito y un número finito de saltos en un período.

Estas sinusoides están armónicamente relacionadas entre sí, es decir, sus frecuencias son múltiplos enteros de una frecuencia fundamental. Esta suma ponderada de señales sinusoidales se conoce como Serie de Fourier.

La Serie de Fourier de una señal periódica $f(x)$ de período T_0 puede escribirse como:

$$f(x) = a_0 + \sum_{k=1}^{\infty} (a_k \cos(kw_0 x) + b_k \sin(kw_0 x)) \qquad (\boldsymbol{Ec.\ 2.15})$$

Donde w_0 es la frecuencia fundamental ($w_0 = 2\pi\ /\ T_0$) y los coeficientes a_0, a_k y b_k constituyen un conjunto de coeficientes relacionados unívocamente con la función $f(x)$. Esta forma de escribir la Serie de Fourier se conoce como la forma trigonométrica.

Utilizando las ecuaciones de Euler (**Ecuaciones 2.7** y **2.8**), se puede reescribir la Serie de Fourier como:

$$f(x) = \sum_{k=-\infty}^{\infty} C_k\, e^{ikw_0 x} \qquad (\boldsymbol{Ec.\ 2.16})$$

Esta forma de escribir la ecuación se conoce como forma compleja.

La **Ecuación 2.15** nos dice que para cualquier señal periódica $f(x)$ pueden encontrarse coeficientes $a_0, a_1, b_1, a_2, b_2 \ldots$ tales que multiplicados por sinusoides de frecuencias $w_0, 2w_0, 3w_0 \ldots$ den como resultado la función $f(x)$ cuando estas sinusoides se suman.

Entonces, toda función periódica de frecuencia w, cualquiera sea su naturaleza, está compuesta por la suma de varias sinusoides de frecuencias mayores o iguales a w, cada una de las cuales tiene distinta amplitud, frecuencia y fase. Las frecuencias involucradas están armónicamente relacionadas entre sí.

Lo anterior implica dos cosas:

1. Si se toman varias sinusoides de distintas frecuencias, fases y amplitudes y se suman, se obtendrá una señal periódica.

2. Dada una señal periódica $f(x)$ cualquiera, ésta siempre podrá descomponerse en sus componentes de frecuencia, mediante la determinación de las sinusoides que la conforman. El gráfico de las frecuencias de estas sinusoides con respecto a la amplitud de cada una de ellas, se conoce como espectro de frecuencias.

En la **Figura 2.6** se muestran varias sinusoides de frecuencias relacionadas y su suma. Allí puede observarse que al sumar estas sinusoides se obtiene una nueva forma de onda de frecuencia igual a la frecuencia de la onda fundamental. Esto nos permite comprobar de manera empírica el descubrimiento de Fourier.

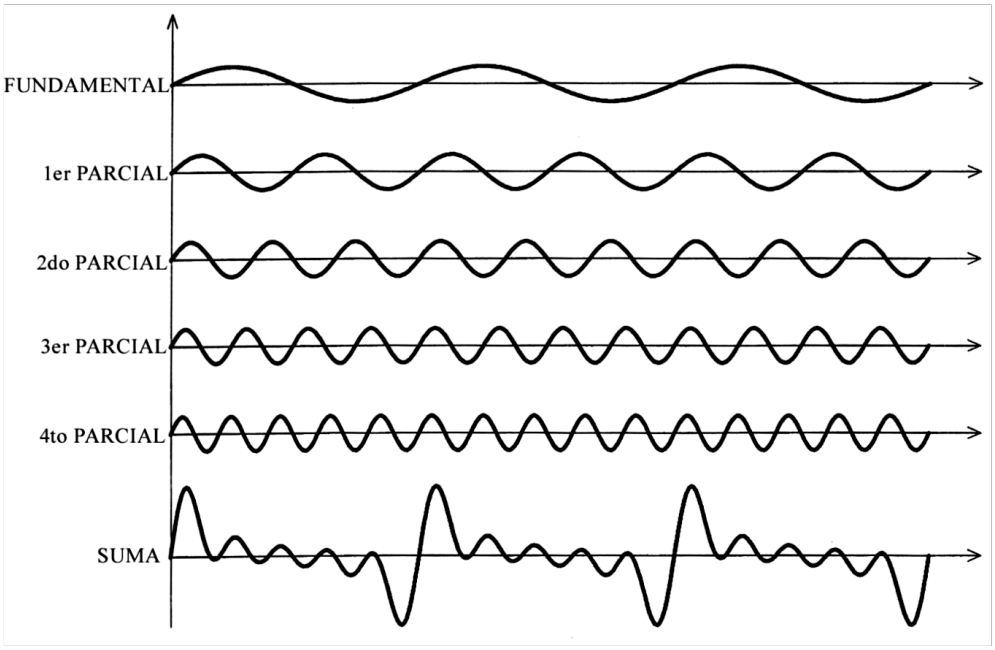


Figura 2.6: Suma de señales armónicas **[5]**.

2.4.1. Transformada de Fourier

Una forma de extender las series de Fourier a funciones no periódicas, es truncar las señales en un cierto punto y suponer que la zona truncada se repite hasta el infinito desde ahí hacia adelante.

Otra forma es suponer que éstas poseen un período infinito y expandir un tanto las ecuaciones para poder trabajar con este tipo de señales. Esto da pie a la Transformada de Fourier. La Transformada de Fourier es una generalización de las Series de Fourier.

En rigor, esta transformada se aplica a funciones continuas y aperiódicas, pero también es posible aplicarla a funciones discretas mediante la utilización de funciones impulso. La Transformada de Fourier es una herramienta clave para traducir señales desde los dominios del tiempo a la frecuencia y viceversa.

La Transformada de Fourier es una operación que transforma una función descomponiéndola en varias sinusoides de diferentes frecuencias y amplitudes. La función transformada es la representación en el dominio de la frecuencia de la función original.

La Transformada de Fourier se define como:

$$F(w) = \int_{-\infty}^{\infty} f(x) e^{-iwx}\, dx \qquad (\boldsymbol{Ec.\ 2.17})$$

Donde w denota la frecuencia y F a la transformada. Esta ecuación corresponde a integrar la función $f(x)$ multiplicada por una sinusoide compleja en todo el intervalo de la variable x (desde $-\infty$ hasta ∞).

La transformada inversa, la que permite encontrar $f(x)$ cuando se conoce $F(w)$, está dada por la ecuación:

$$f(x) = \int_{-\infty}^{\infty} F(w) e^{iwx}\, dx \qquad (\boldsymbol{Ec.\ 2.18})$$

Utilizando estas relaciones, siempre es posible encontrar una correspondencia entre una función y su espectro o viceversa, es decir:

$$f(x) \leftrightarrow F(w) \qquad (\boldsymbol{Ec.\ 2.19})$$

La Transformada de Fourier está definida para todas las funciones continuas, pero en nuestro caso solo será desarrollada para la forma discreta denominada también Transformada Discreta de Fourier.

Esta transformada nos permite conocer el espectro de frecuencias de cualquier señal $f(x)$, sea ésta periódica o aperiódica y cualquiera sea su naturaleza. Es decir, dada una función $f(x)$ cualquiera, mediante esta transformada podemos saber que frecuencias están presentes en ella.

2.4.2. Transformada de Fourier discreta

La Transformada de Fourier, tal como se presentó en la sección anterior, nos permite encontrar transformadas para señales continuas. Sin embargo, esto no es aplicable directamente a señales discretas o digitales, que son las que nos interesan tratar. La Transformada de Fourier Discreta, o bien DFT (*Discrete Fourier Transform*), se emplea para encontrar el contenido de frecuencia de señales que son discretas. Esto implica que en el dominio de la frecuencia estas señales también serán periódicas y discretas.

La forma compleja de la Transformada de Fourier Discreta se expresa de la siguiente forma:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-\frac{j2\pi nk}{N}} \qquad k = 0, 1, \dots, N-1 \quad (\boldsymbol{Ec.\ 2.20})$$

Donde n denota el número de muestras de la señal en el dominio del tiempo $x(n)$, $X(k)$ el espectro de frecuencias de $x(n)$ y $e^{-\frac{j2\pi nk}{N}}$ es la raíz primitiva n-ésima de la unidad en el plano complejo (factor de fase de rotación).

Cuando se lleva a cabo la Transformada Discreta de Fourier de una señal, se obtiene una secuencia de números complejos de longitud $N/2 + 1$ (ver **Figura 2.7**) que consta de dos señales a la salida, las cuales contienen información de las magnitudes de los componentes cosenoidales y senoidales, es decir, la parte real e imaginaria.

La Transformada Inversa de Fourier Discreta se calcula mediante la ecuación:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{\frac{j2\pi nk}{N}} \qquad (\boldsymbol{Ec.\ 2.21})$$

Al procedimiento en el cual se transforma una señal que se encuentra en el dominio del tiempo al dominio de la frecuencia se le llama análisis o *forward DFT*. Por el contrario, si se tiene una señal en el dominio de la frecuencia y se lleva al dominio del tiempo, se le llama síntesis o *inverse DFT*.

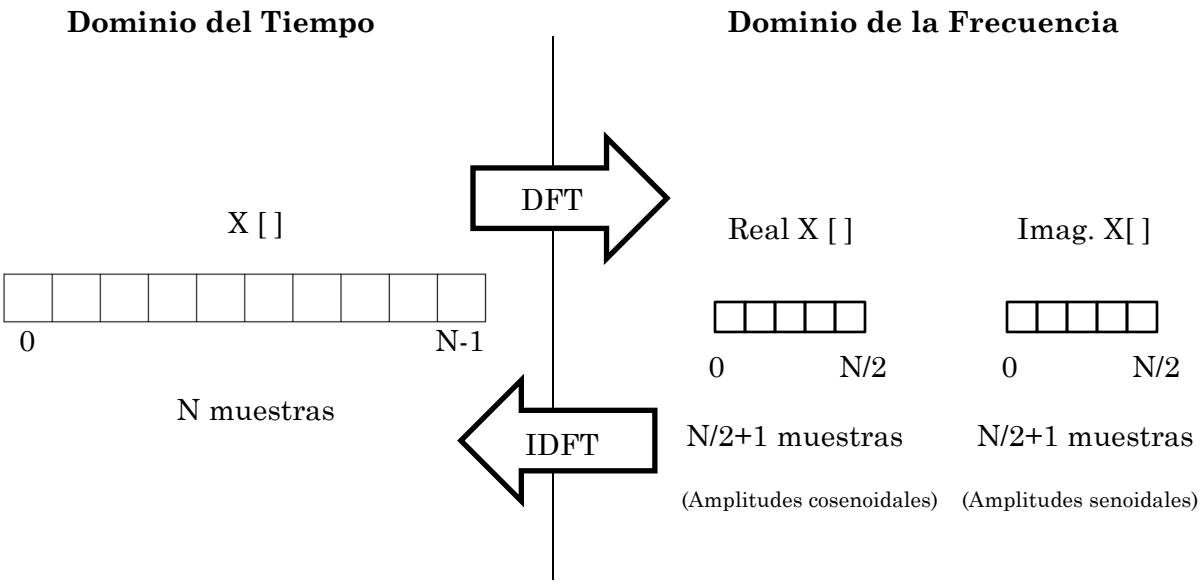


Figura 2.7: Proceso de cambio de dominio del tiempo al dominio de la frecuencia con la Transformada Discreta de Fourier **[6]**.

Cuando se lleva a cabo una transformada inversa IDFT, es necesario que las muestras en la entrada se encuentren en el orden correcto de tal forma que las frecuencias negativas sean incluidas de manera apropiada, ya que el resultado sería una señal diferente a la que se quiere reconstruir. Para comprobar que la simetría es la correcta después de hacer la transformación, se verifica que las muestras de la parte imaginaria sean equivalentes a cero.

El desarrollo de la DFT históricamente se dio en forma paralela al de la transformada de Fourier continua. A pesar de su existencia, la DFT prácticamente no se utiliza dado que el cálculo de la transformada discreta es un proceso complejo y lento computacionalmente. Lo que se utiliza en la mayoría de los casos para calcular espectros de señales discretas se llama Transformada Rápida de Fourier, o bien FFT (*Fast Fourier Transform*), que es un algoritmo desarrollado para obtener la DFT de una forma más rápida y eficiente computacionalmente. El tiempo de procesamiento de la FFT es considerablemente más rápido que calcular la DFT directamente.

2.4.3. Transformada rápida de Fourier

La FFT (*Fast Fourier Transform*) es un algoritmo creado por J.W. Cooley y J.W. Tukey **[7]** para llevar a cabo de manera eficaz la Transformada Discreta de Fourier. En el año 1969, se llevó a cabo un análisis sísmico de 2048 puntos diferentes, el cual tomaba aproximadamente 13 ½ horas en obtener resultados. Usando la FFT, este mismo cálculo tomó 2.4 segundos en la misma máquina.

Constituye uno de los mayores desarrollos en la tecnología en tratamiento de señales. La evolución de la computación, particularmente la del computador personal, ha hecho de la FFT una herramienta de análisis manejable y potente.

La rutina de la FFT está basada en la DFT compleja, ya que se utiliza una serie de números complejos de tamaño N en las señales de entrada al sistema. Sin embargo, a diferencia de la real se obtendrán dos señales de tamaño N en la salida, cuyo espectro real estará definido desde 0 hasta $N/2$, es decir, los componentes de frecuencia positiva. De esta manera, al incluir las frecuencias negativas, se tratará de un dominio de la frecuencia periódico.

Otro punto a considerar es que el algoritmo de la FFT implementa una DFT real utilizando una DFT compleja, por lo que las muestras correspondientes a la parte imaginaria de la entrada, deberán ser iguales a cero.

Este algoritmo tiene una complejidad en tiempo de $O(n\,log(n))$, lo que indica que las convoluciones pueden ser calculadas más rápidamente en el dominio de la frecuencia haciendo uso de la FFT en vez de hacer la convolución en el dominio del tiempo.

2.4.3.1. Algoritmo de FFT

Para llevar a cabo la Transformada Rápida de Fourier se requieren cuatro etapas:

Primero, se divide una señal de N muestras en el dominio del tiempo en N señales cada una conformada por un solo punto. En la **Figura 2.8**, se muestra la descomposición en el tiempo de una señal de 16 muestras:

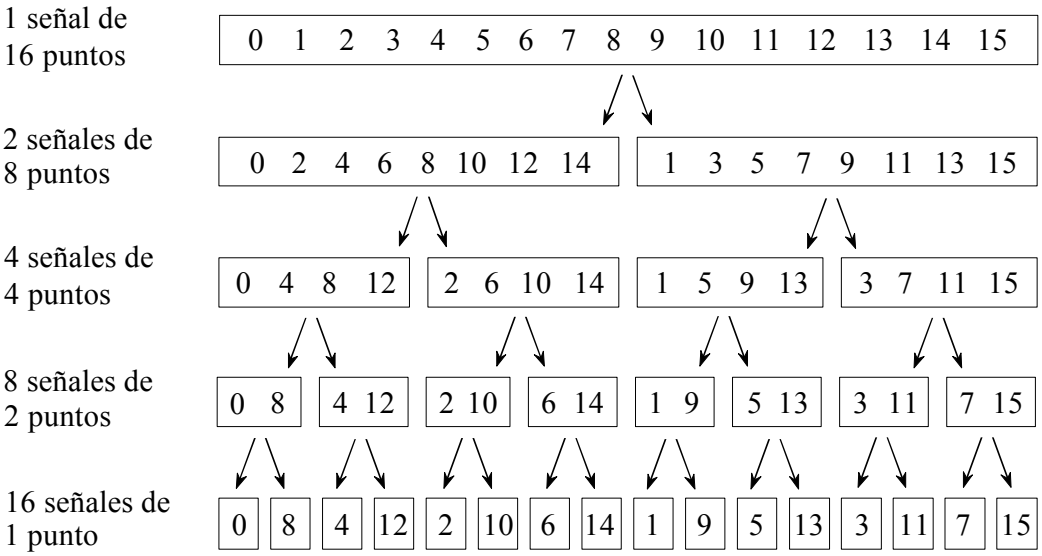


Figura 2.8: Descomposición de las muestras de una señal de 16 puntos **[6]**.

Al segundo paso del algoritmo se le conoce como Algoritmo de Inversión de Bits, ya que las N muestras son reordenadas contando en binario con los bits rotados de izquierda a derecha. Por ejemplo, la muestra 3 que en binario es "0011" será intercambiada por la muestra 12 "1100" y así sucesivamente. Para este paso se requieren $log_2 N$ etapas.

El tercer paso se trata de obtener el espectro de frecuencia correspondiente a cada una de las muestras, el cual es equivalente a su valor actual, ya que debido al paso anterior, se encuentran en el dominio de la frecuencia.

Por último, los espectros de frecuencia en la FFT se combinan en pares, duplicándolos y sumándolos de tal manera que ambos espectros coincidan.

Esto se logra agregando ceros en el dominio del tiempo a ambas señales. En una señal los puntos impares son ceros mientras que en la otra señal los puntos pares serán ceros, es decir, una de las señales será desplazada a la derecha por una muestra. Esto es equivalente a multiplicar su espectro por una señal sinusoidal, como se muestra en la **Figura 2.9**.

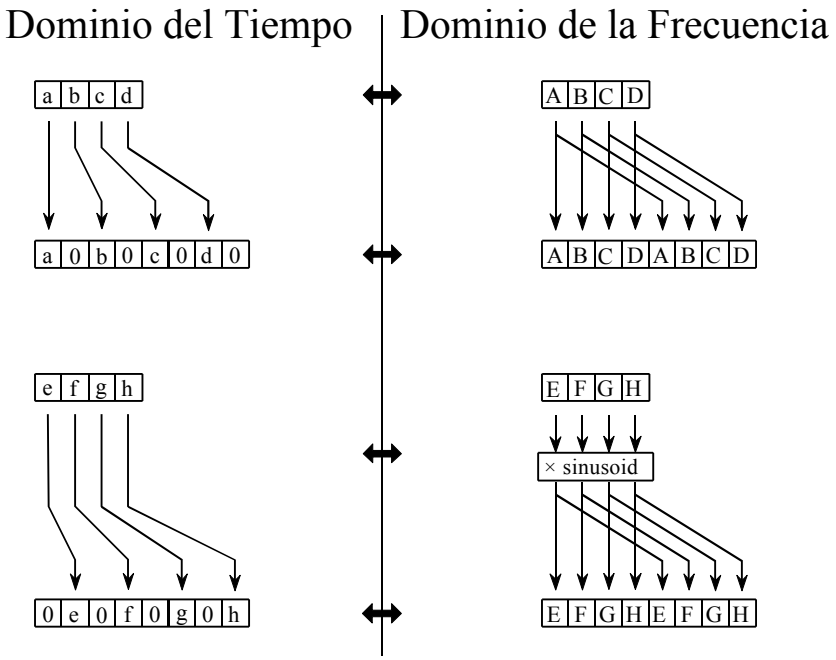


Figura 2.9: Síntesis de espectros en frecuencia **[6]**.

De esta manera, al sumar ambos espectros en el orden mostrado, se obtendrá el espectro final a la salida de la FFT (ver **Figura 2.10**).

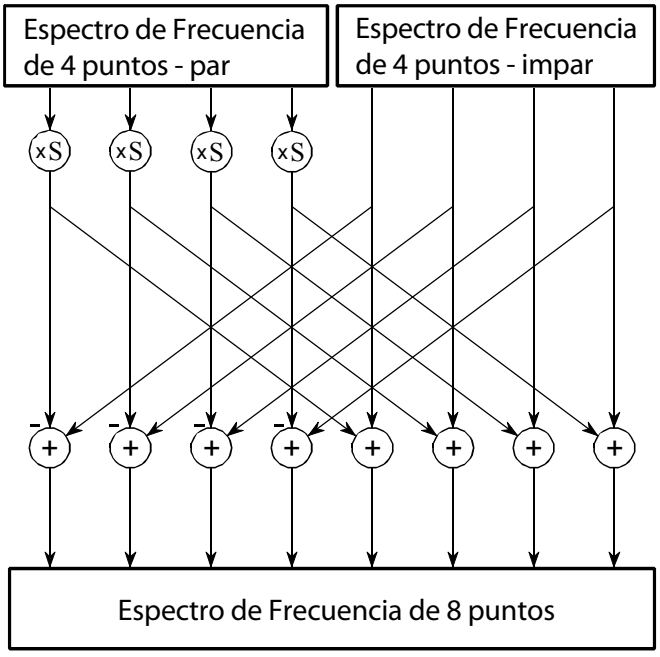


Figura 2.10: Método para obtener el espectro final **[6]**.

2.5. Respuesta al impulso y respuesta de frecuencia

Para estudiar el comportamiento de un sistema lineal e invariante se utilizan normalmente dos tipos de entradas, un impulso unitario o una señal constante (ver **Figura 2.11**).

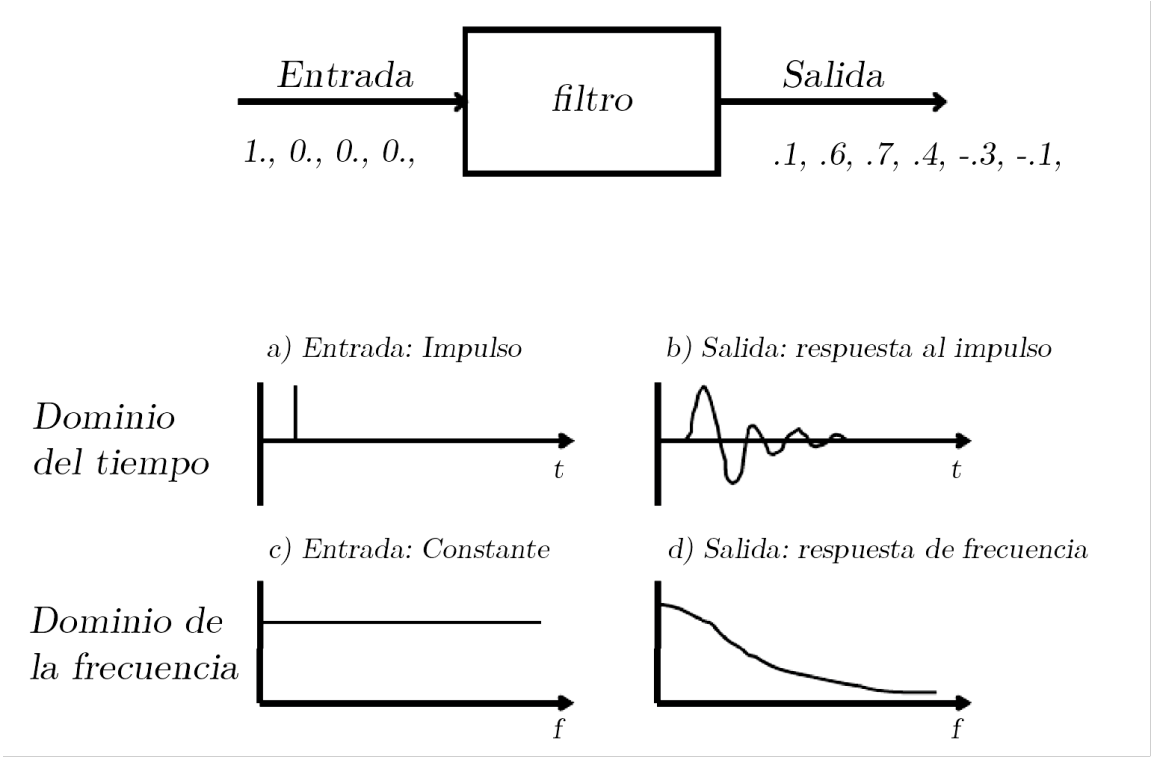


Figura 2.11: Respuesta al impulso **[5]**.

La respuesta al impulso corresponde a la respuesta de un sistema ante un impulso cuando el sistema se encuentra en estado de reposo. Un impulso es una función matemática abstracta que tiene una amplitud infinita y una duración casi nula.

Esta señal es ideal para estudiar sistemas, ya que permite estimar la respuesta de un sistema cualquiera a señales con un contenido de frecuencias previamente determinado.

Otra respuesta muy utilizada para diseñar sistemas es la respuesta de frecuencia, que se define como la respuesta del sistema en el dominio de la frecuencia. Esta respuesta puede calcularse como la Transformada de Fourier de la respuesta al impulso, o bien puede medirse o estimarse directamente, si se utilizan como entrada señales de tipo sinusoidal.

Dado que cualquier señal puede descomponerse en muchas sinusoides individuales, de acuerdo a la serie o transformada de Fourier, la respuesta total del sistema puede calcularse mediante el principio de superposición, es decir, si un sistema posee varias entradas, pueden analizarse por separado las respuestas del sistema a cada una de las entradas y luego calcular la salida como la suma de las respuestas individuales. En el ámbito de los sistemas digitales, dado que un impulso es una abstracción matemática imposible de representar en un computador, un impulso se implementa como una secuencia de números con valor cero salvo una sola muestra que toma el valor uno.

Un tren de impulsos, en cambio, es una secuencia de muestras con valor unitario. La **Figura 2.11** muestra en forma gráfica la relación entre la respuesta al impulso y la respuesta de frecuencia de un sistema digital o filtro. El estudio de la respuesta de frecuencia de un sistema, es clave en la teoría y diseño de los filtros digitales.

Si se conoce la respuesta al impulso, basta realizar una operación matemática denominada convolución entre una señal de entrada cualquiera y la respuesta al impulso para obtener la respuesta del sistema a esa entrada.

2.6. Convolución

La convolución es una operación matemática que se utiliza para combinar dos señales y formar una tercera señal. Es la técnica más simple e importante en el procesamiento de señales. Usando la estrategia de descomposición de un impulso los sistemas pueden ser descritos por una señal llamada respuesta al impulso (ver glosario). La descomposición en impulsos es importante ya que permite examinar las muestras de una señal.

La convolución es importante porque contiene la relación de tres señales de interés: la señal de entrada, la de salida y la respuesta al impulso **[6]**.

Mediante la convolución es posible calcular la respuesta de un sistema a una entrada arbitraria.

Cuando una señal de entrada $x(n)$ es introducida en un sistema lineal con una respuesta al impulso $h(n)$, el resultado de la convolución de ambas será la señal $y(n)$, cuyo número total de muestras será la suma de la entrada con la respuesta al impulso menos uno (ver **Figura 2.12**).

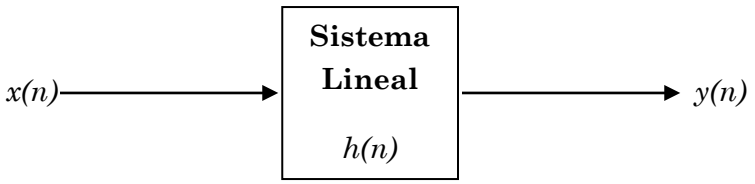


Figura 2.12: Sistema Lineal con una respuesta al impulso $\boldsymbol{h(n)}$.

El operador de convolución se expresa como se muestra en la siguiente ecuación:

$$x(n) \otimes h(n) = y(n) \qquad (\boldsymbol{Ec.\ 2.22})$$

La convolución también puede ser expresada como la suma de los productos escalares de las señales del sistema, es decir, la de entrada y la respuesta al impulso:

$$\sum_{k=0}^{n} x(k)h(n-k) \qquad (\boldsymbol{Ec.\ 2.23})$$

Hay que considerar que los límites de la ecuación anterior comienzan en $k = 0$, ya que los sistemas en muchos casos no responden a valores de entrada más allá del tiempo presente, ya que $h\,(n-k) = 0$ cuando $n - k < 0$.

Cuando se trata de hacer procesamiento digital de una señal no tiene sentido hablar de convoluciones aplicando estrictamente la definición ya que sólo disponemos de valores en instantes discretos de tiempo. Es necesario para esto realizar una aproximación numérica.

La convolución puede ser calculada en el dominio de la frecuencia (utilizando la FFT), ya que una de sus propiedades es la de ser equivalente a la multiplicación compleja en el dominio de la frecuencia. Una vez ubicados en este dominio se multiplica la entrada $x(n)$ por la respuesta al impulso $b(k)$ y finalmente se realiza la transformada inversa (IFFT) para obtener el resultado del producto hecho en el dominio de la frecuencia.

2.6.1. Teorema de Convolución

Sea $\boldsymbol{\mathcal{F}}$ el operador Transformada de Fourier. Entonces, para la función $x(n)$, $\boldsymbol{\mathcal{F}}(x(n))$ es la Transformada de Fourier de $x(n)$ y $\boldsymbol{\mathcal{F}^{-1}}(x(n))$ la Transformada Inversa de Fourier de $x(n)$.

El Teorema de Convolución establece si $x(n)$ y $h(k)$ son secuencias infinitas:

$$\boldsymbol{\mathcal{F}}(x(n) \otimes h(k)) = \boldsymbol{\mathcal{F}}(x(n)) \cdot \boldsymbol{\mathcal{F}}(h(k)) \qquad (\boldsymbol{Ec.\ 2.24})$$

Aplicando la Transformada Inversa de Fourier, la convolución de $x(n)$ y $h(k)$, $x(n) \otimes h(k)$ puede ser calculada como:

$$x(n) \otimes h(k) = \boldsymbol{\mathcal{F}}^{-1}(\boldsymbol{\mathcal{F}}(x(n)) \cdot \boldsymbol{\mathcal{F}}(h(k))) \qquad (\boldsymbol{Ec.\ 2.25})$$

Los conceptos básicos de señales y ciertas operaciones que se pueden aplicar en ellas a través del procesamiento digital de señales han sido introducidos, en la siguiente subsección se explicará cómo manipular una señal de audio a través de filtros. Esto puede ser de utilidad para reducir ruido o aplicar ecualización.

2.7. Filtros digitales

Un filtro es básicamente una caja con una entrada y una salida (ver **Figura 2.13**). Si la salida es diferente a la entrada, significa que la señal original ha sido filtrada. Cualquier medio por el cual una señal de audio pasa, cualquiera sea su forma, puede describirse como un filtro.

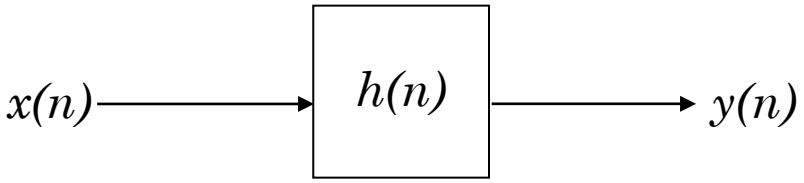


Figura 2.13: Un filtro como una caja negra **[5]**.

Los filtros pueden ser analógicos, como el caso de un filtro solar de un telescopio, o digitales, como un eliminador de ruido implementado en el computador. Un filtro digital es un sistema de tiempo discreto que deja pasar ciertos componentes de frecuencia de una secuencia de entrada sin distorsión y bloquea o atenúa otros. Se trata simplemente de un filtro que opera sobre señales digitales, tales como las que operan en un computador. Un filtro digital es un cálculo que se realiza al recibir una secuencia de datos (la señal de entrada) y produce una nueva (la señal de salida).

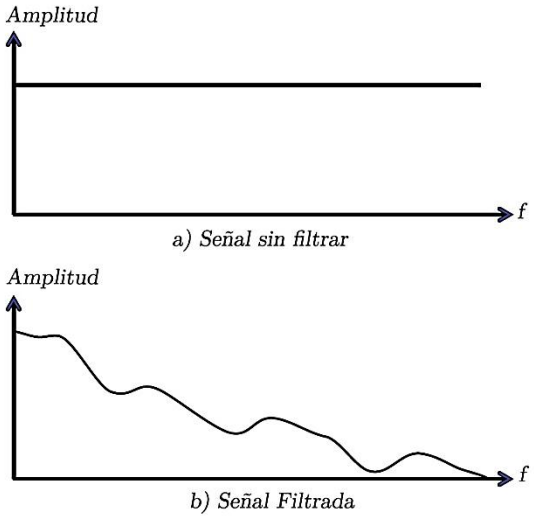


Figura 2.14: Cambio que se produce en una señal que ha sido pasada a través de un filtro **[5]**.

La **Figura 2.14** muestra el cambio que produce un filtro en el dominio de la frecuencia. En el primer caso el espectro de frecuencias se mantiene igual, lo que indica que el filtro no realiza cambio alguno en la señal de entrada. El segundo caso claramente muestra una modificación en el contenido de frecuencias de la señal original.

Los filtros analógicos pueden ser usados para cualquier tarea, sin embargo, los filtros digitales pueden alcanzar resultados muy superiores. Un filtro digital puede hacer todo lo que un filtro analógico es capaz de realizar.

2.7.1. Ecuación de diferencias

En forma general, cualquier filtro digital puede ser descrito por la siguiente ecuación de diferencias **[5]**:

$$y(n) = \sum_{i=0}^{M} b_i x(n-i) - \sum_{k=1}^{N} a_k y(n-k) \qquad (\boldsymbol{Ec.\ 2.26})$$

De esta ecuación se puede concluir que lo único que se necesita para implementar un filtro digital son sumas (o restas), multiplicaciones y retrasos, ya que nos dice que la salida del filtro depende de versiones presentes y pasadas de la entrada menos versiones pasadas de la salida.

El índice n es un número entero que representa unidades discretas y los números a_k y b_i se denominan los coeficientes del filtro. Los coeficientes a_k multiplican a valores pasados de la salida y los coeficientes b_i a valores pasados de la entrada.

Si todos los coeficientes a_k son cero, entonces el filtro sólo depende de valores anteriores y actuales de la entrada y se trata de un filtro no recursivo o FIR *(Finite Impulse Response*). Si los valores a_k no son todos nulos, entonces se trata de un filtro recursivo o IIR (*Infinite Impulse Response*).

2.7.2. Filtros FIR

En un filtro con Respuesta Finita al Impulso o FIR, los coeficientes a_k son cero, lo que significa que la respuesta del filtro depende solamente de la entrada y no de valores pasados de la salida. Este tipo de filtros tiene una respuesta finita ya que no exhiben recursión.

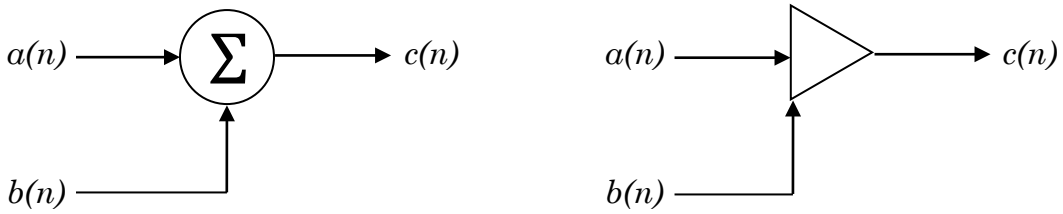
$$y(n) = \sum_{i=0}^{M} b_i x(n-i) \qquad (\boldsymbol{Ec.\ 2.27})$$

Los filtros pueden ser graficados como un diagrama de bloques, en este tipo de diagramas se pueden definir los elementos que se requieren para implementar el filtro. En la **Figura 2.15** se muestran las operaciones básicas que se pueden llevar a cabo para construir un filtro.

- **Adiciones y/o multiplicaciones:** Se utilizan para obtener el resultado de una operación (suma o multiplicación) con dos elementos.

- **Retraso unitario:** Se utiliza para almacenar un valor anterior al actual para su uso posterior.

Sumador: $c(n) = a(n) + b(n)$ **Multiplicador:** $c(n) = a(n) * b(n)$



Retraso Unitario: $b(n) = a(n-1)$

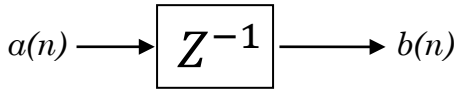


Figura 2.15: Operaciones básicas con señales digitales **[8]**.

La **Figura 2.16** muestra la implementación de un filtro FIR. Una de las propiedades más interesantes de este tipo de filtros es la simetría de sus coeficientes y el hecho que pueden ser diseñados de tal forma de exhibir una respuesta de fase lineal.

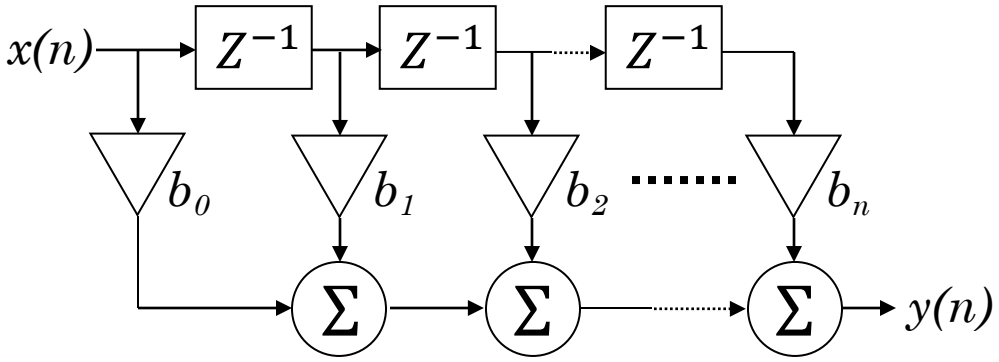


Figura 2.16: Esquema de implementación de un filtro FIR **[5]**.

Los filtros FIR tienen las siguientes propiedades:

- Respuesta de fase lineal (si se diseñan adecuadamente)
- Estabilidad con coeficientes cuantizados
- En general, requieren de un mayor orden que los filtros IIR

El diseño de filtros FIR se basa usualmente en una aproximación directa de la magnitud de la respuesta deseada a la cual se le agrega la condición de una respuesta de fase lineal.

2.7.2.1. Diseño de filtros FIR

Un filtro FIR puede ser diseñado utilizando distintas técnicas tanto en el dominio del tiempo como de la frecuencia. Existe un método óptimo para producir filtros FIR llamado Algoritmo de Parks McClellan **[9]** que produce coeficientes óptimos y fase lineal. Este método es muy popular dado su facilidad de uso, flexibilidad y excelentes resultados.

Las técnicas de aplicación de ventanas (*windowing*), básicamente toman la respuesta al impulso de un filtro ideal con duración infinita y aplican una ventana para truncar y limitar los coeficientes a un número específico, para luego desplazar la secuencia de manera de garantizar causalidad. Esta técnica es simple y produce resultados razonables, pero con distorsión respecto a la respuesta ideal.

Otras técnicas se basan en muestreo en la frecuencia. La idea es muestrear la respuesta de amplitud deseada a intervalos regulares y luego utilizar la DFT inversa para generar una respuesta al impulso de duración finita. El número de muestras determina el nivel de precisión del filtro a la respuesta ideal. Esta técnica es bastante sofisticada y requiere una tasa de muestreo alta.

Por último, hay una técnica ad-hoc denominada posicionamiento de ceros (ver glosario) **[10]**. La idea es colocar ceros en el plano Z de forma estratégica con el fin de aproximarse a la respuesta deseada y realizar el cálculo de los coeficientes en forma gráfica. Mientras más cercano esté el cero al círculo unitario, mayor efecto tendrá en la respuesta de frecuencia del filtro para ese rango de frecuencias. Esta técnica es rápida y no muy precisa, pero puede funcionar para aplicaciones simples.

2.8. Algoritmos de procesamiento digital de audio

Existen distintos tipos de efectos de audio, clasificados comúnmente de la siguiente forma **[11]**:

- **Control de Dinámicas y Ganancia:** Este tipo de efectos modifican o alteran de algún modo la dinámica (amplitud) del sonido. Entre ellos se encuentra el *Compressor, Distortion y Over Drive.*

- **Efectos de Modulación:** Se encargan de modular la señal de entrada ya que se altera la señal de audio de forma no lineal en dos dimensiones, tiempo y frecuencia. Varias señales interactúan unas con otras para producir nuevas señales con frecuencias que no estaban presentes en las señales originales. Entre los más comunes se encuentran el *Vibrato, Chorus, Flanger, Phaser, Ring Modulator, Tremolo y Auto Panner.*

- **Filtros:** Son los efectos que se utilizan para modificar el espectro de la frecuencia del sonido. Los filtros más comunes son el filtro paso bajo, filtro paso alto y filtro paso banda. Generalmente estos filtros forman parte de un Ecualizador, el cual se puede parametrizar de una forma más específica.

- **Efectos Basados en el Tiempo:** Se utilizan para crear una sensación de eco o simular de forma artificial un espacio acústico. El *Delay* y el *Reverb* son los más conocidos.

A continuación se presenta una descripción breve de los efectos de audio implementados en este trabajo.

2.8.1. Over Drive

Este efecto consiste en saturar la señal de audio logrando que esta se distorsione ligeramente. Es muy utilizado en la guitarra eléctrica.

El *Over Drive* ofrece al instrumento un sonido cálido con mayor duración de las notas musicales (lo que se conoce como *sustain*). Así mismo adiciona armónicos (ver glosario) lo cual refuerza las frecuencias altas o agudos.

Se puede ajustar el grado de saturación de la señal, desde un sonido limpio a otro moderadamente saturado.

2.8.2. Distortion

Este efecto consiste en saturar fuertemente la señal de audio logrando que esta se distorsione. Al igual que el *Over Drive*, es muy utilizado en la guitarra eléctrica.

La distorsión le ofrece al instrumento un sonido agresivo y potente así como mayor duración a las notas musicales ya que adiciona armónicos a la señal lo cual refuerza las frecuencias altas o agudos.

Se puede ajustar el grado de distorsión de la señal, desde un sonido suave a otro fuertemente saturado.

2.8.3. Ecualizador

Un ecualizador modifica el contenido en frecuencias de la señal de entrada. Consiste básicamente en una colección de filtros, cada uno centrado en una banda de frecuencias específicas que alteran la señal recibida.

Los filtros nos permiten modificar la señal que reciben, dejando pasar a través de ellos una parte de la señal diferente a la original, todo dependiendo de su funcionamiento. La parte que dejarán pasar estará limitada entre la frecuencia de corte inferior y la frecuencia de corte superior. Ambas se corresponden con las frecuencias mínimas y máximas que pueden pasar a través del filtro.

Los filtros más comunes que tiene un ecualizador son (ver **Figura 2.17**):

- **Filtro pasa-bajo:** Permite el paso de las frecuencias menores a cierta frecuencia ω_c denominada frecuencia superior de corte y bloquea las frecuencias superiores a este valor.

- **Filtro pasa-altos:** Permite el paso de las frecuencias mayores a cierta frecuencia ω_c denominada frecuencia inferior de corte y bloquea las frecuencias inferiores a este valor.

- **Filtro pasa-banda:** Permite el paso de las frecuencias comprendidas entre dos frecuencias ω_1 y ω_2 ($\omega_1 < \omega_2$), denominadas frecuencia inferior de corte y frecuencia superior de corte, bloqueando las frecuencias restantes.

Filtro rechaza-banda: Bloquea el paso de las frecuencias comprendidas entre dos frecuencias ω_1 y ω_2 ($\omega_1 < \omega_2$), denominadas frecuencia superior de corte y frecuencia inferior de corte, dejando pasar las frecuencias restantes.

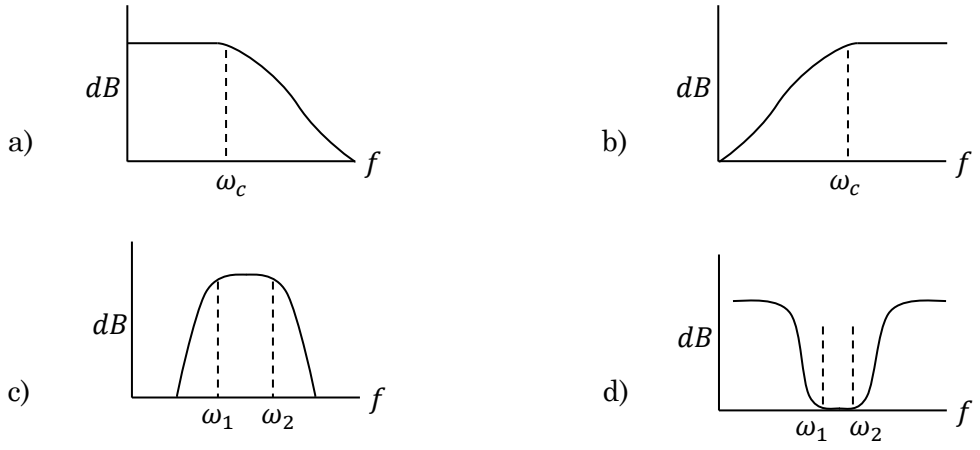


Figura 2.17: Respuesta en frecuencia de los filtros utilizados en un ecualizador, a) Filtro paso-bajo, b) Filtro paso-alto, c) Filtro pasa-banda, d) Filtro rechaza-banda.

2.8.4. Vibrato

Cuando un carro pasa a una velocidad sumamente rápida, se puede apreciar una desviación o desafinación del sonido (lo que se conoce como *Efecto Doppler*). Esta variación de tono en el sonido es dada por la variación de distancia que existe entre la fuente que emite el sonido y el receptor, esto es equivalente a variar el tiempo de retardo del efecto *delay*. Al variar periódicamente el tiempo de retardo se produce una variación periódica del tono, esto es lo que se logra con el efecto de *vibrato*.

2.8.5. Chorus

El efecto *chorus* o de coro se obtiene al mezclar una señal de entrada con vibrato junto con la señal sin procesar. Para producir el efecto, ya sea natural o artificialmente, deben mezclarse dos o más sonidos individuales con aproximadamente el mismo timbre y alturas levemente disímiles, de manera que sean percibidos como un solo sonido proveniente de una misma fuente.

El efecto de coro se enfatiza cuando los sonidos se producen en tiempos y posiciones levemente distintas. Esto es lo que típicamente sucede en un coro de voces, y es lo que origina el nombre del efecto.

2.8.6. Ring Modulator

Se refiere a la modulación de "anillo" porque la implementación en circuitos analógicos de este efecto consiste en posicionar una colección de diodos en forma de anillo. La modulación de anillo está estrechamente relacionada con la modulación de amplitud y la mezcla de frecuencias.

Las frecuencias que se generan generalmente son inarmónicas, lo que hace que se creen sonidos bastante disonantes. Este efecto está diseñado para generar sonidos con características metálicas o sonidos similares al sonido de una campana.

2.8.7. Tremolo

El efecto de *tremolo* consiste en una modulación de amplitud de la señal de entrada (volumen) que varía de forma periódica durante el tiempo. La señal portadora será la señal de audio original, mientras que la señal moduladora será una señal periódica de baja frecuencia generada por un LFO (*Low Frequency Oscilator*).

2.8.8. Auto Panner

El efecto *auto panner* modifica la posición del sonido en el espacio estereofónico. Esta posición estereofónica puede ir desde el extremo izquierdo (valores negativos) al extremo derecho (valores positivos) de los altavoces.

Este efecto tiene el mismo principio del *tremolo*, la diferencia está en que se caracteriza por ser un efecto de tipo estéreo. La señal de audio se envía a los altavoces de manera alternada.

2.8.9. Delay

Un efecto de *delay* o retraso consiste en la multiplicación y retraso de una señal de entrada, que se mezcla con la señal original, obteniéndose un efecto de eco artificial. En producción musical es utilizado de forma muy creativa. Es utilizado especialmente en grabaciones de instrumentos musicales donde se puede crear una sensación de engrandecimiento del instrumento con *delays* de corta duración y efectos rítmicos con *delays* de larga duración.

Cuando se reproduce la copia de la señal retrasada (*Delay Line*) con la señal original, el usuario puede manipular el volumen de dicha copia para generar un efecto de eco alejándose o un sonido perpetuo de la copia. También es posible definir el tiempo del retraso (desde 1 ms. hasta 1 seg. comúnmente).

2.9. Computación paralela en GPU

2.9.1. GPGPU

GPGPU son la siglas de *General Purpose computation on Graphics Processing Units* o Computación de Propósito General para Unidades de Procesamiento Gráfico, también denominado *GPU Computing*. Los GPU son procesadores con una gran cantidad de núcleos de procesamiento capaces de tener un alto rendimiento y velocidad en la computación de datos. La evolución a través del tiempo de estos procesadores, en particular de los procesadores NVIDIA, se ilustra en la **Figura 2.18**. Su nombre se deriva de que originalmente se encontraban localizados en la tarjeta gráfica y se utilizaban solamente para procesamiento gráfico.

Actualmente los GPU son procesadores paralelos de propósito general con soporte para una programación accesible con lenguajes de programación estándar como C. Los desarrolladores que han traducido sus aplicaciones para ser procesadas por el GPU han tenido muchas veces rendimientos superiores incluso comparables con versiones optimizadas para CPU. Actualmente existe el sitio web GPGPU.org, que se encarga de catalogar el uso histórico del GPU para la computación de propósito general además de proveer recursos para los desarrolladores de GPGPU.

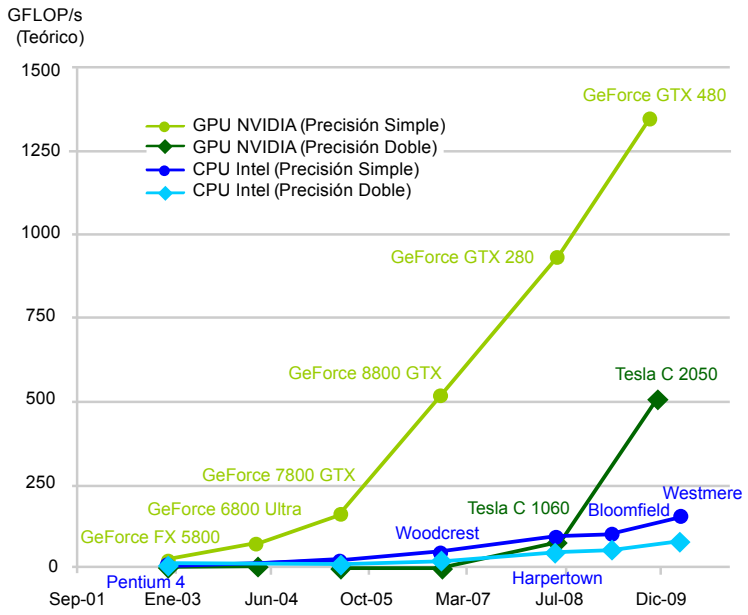


Figura 2.18: Evolución de los GPU NVIDIA con respecto a los CPU Intel **[12]**.

El término GPGPU fue creado por Mark Harris en el año 2002 **[13]**, cuando empezó a notar una tendencia inicial de utilizar el GPU para aplicaciones no gráficas. GPGPU.org, por su lado, ha pasado de ser un sitio visitado por unos pocos a ser un sitio sumamente popular para los desarrolladores e investigadores de esta área.

Uno de los primeros ejemplos de programación genérica en GPU fue "El Juego de la Vida" **[14]**. Con este ejemplo, Greg James, pretendía ilustrar las capacidades que tiene la GPU para realizar cálculos dentro y también fuera del ámbito de la síntesis de gráficos pura. Este ejemplo estaba orientado a la creación de texturas dinámicas (procedurales) en tiempo real. Pronto le siguieron otros pequeños *shaders* (ver glosario) que ampliaban esas posibilidades hacia métodos numéricos aritméticamente intensivos en los que la dependencia entre datos era baja, como la resolución de ecuaciones en derivadas parciales para la simulación de fenómenos físicos.

El campo de la GPGPU tuvo una calurosa aceptación por parte de la comunidad científica, ya que abría las puertas a sistemas muy baratos, disponibles en cualquier computador personal, con un potencial de cálculo muy alto y un crecimiento exponencial. Algo que no tenía rival frente a las otras arquitecturas de aquella época.

Sin embargo, su programación ha sido durante mucho tiempo una tarea especialmente ardua y compleja. Esto debido a que las tarjetas gráficas y su *pipeline* gráfico estaban únicamente pensados para realizar cálculos para la síntesis de gráficos geométricos, lo que obligó a sus programadores a adaptar sus algoritmos a la secuencia segmentada del *pipeline*. Por lo tanto, para lidiar con la tarjeta gráfica era necesario conocer su arquitectura a muy bajo nivel si se deseaba aprovechar sus características potenciales.

Así fue creciendo el movimiento GPGPU hasta lo que hoy en día es, un campo de desarrollo e investigación muy importante. Inclusive los planes de negocio de compañías como NVIDIA, AMD/ATI, IBM e Intel han cambiado debido al potencial que tienen las arquitecturas que se utilizan en procesadores gráficos para realizar grandes cantidades de cálculos en punto flotante, lo que nos dirige cada vez más rápido a los llamados procesadores heterogéneos.

Debido a las diferencias fundamentales entre las arquitecturas de la GPU y la CPU, no cualquier problema se puede beneficiar de una implementación en la GPU. En concreto, el acceso a memoria plantea las mayores dificultades. Las CPU están diseñadas para el acceso aleatorio a memoria. Esto favorece la creación de estructuras de datos complejas, con punteros a posiciones arbitrarias en memoria. En cambio, en una GPU, el acceso a memoria está mucho más restringido.

Por ejemplo, en un procesador de vértices, se favorece el modelo *scatter*, en el que el programa lee en una posición predeterminada de la memoria, pero escribe en una o varias posiciones arbitrarias. En cambio, un procesador de píxeles, o fragmentos, favorece el modelo *gather*, pudiendo el programa leer de varias posiciones arbitrarias, pero escribir en sólo una posición predeterminada.

La tarea del diseñador de algoritmos GPGPU consiste principalmente en adaptar los accesos a memoria y las estructuras de datos a las características de la GPU.

Pese a que cualquier algoritmo que es implementable en una CPU lo es también en una GPU, esas implementaciones no serán igual de eficientes en las dos arquitecturas. En concreto, los algoritmos con un alto grado de paralelismo, sin necesidad de estructuras de datos complejas, y con una alta intensidad aritmética, son los que mayores beneficios obtienen de su implementación en la GPU.

Tradicionalmente, el desarrollo de software GPGPU se hace en ensamblador, o bien en alguno de los lenguajes específicos para aplicaciones gráficas usando la GPU, como GLSL (*OpenGL Shading Language*), Cg (*C for Graphics*) o HLSL (*High Level Shading Language*). Sin embargo, recientemente han surgido herramientas para facilitar el desarrollo de aplicaciones GPGPU, al abstraer muchos de los detalles relacionados con los gráficos y presentar una interfaz de más alto nivel. Son ejemplos de esto el lenguaje de programación CUDA (*Compute Unified Device Architecture*) creado por la empresa NVIDIA, o el lenguaje OpenCL (*Open Computing Language*) creado por Khronos Group para la programación en GPU y CPU de forma paralelizada y con estándar abierto y libre, el cual AMD ha decidido apoyar en lugar de su antigua API *Close to Metal*.

Los GPUs están diseñados específicamente para procesar gráficos y son sumamente restrictivos en términos de operacionalidad y programabilidad. Por su naturaleza, los GPUs son solamente efectivos abordando problemas que puedan ser resueltos usando *stream processing* y el hardware solo puede ser usado de ciertas maneras.

Un *stream* es simplemente un conjunto de registros que requieren de una computación similar. Los *streams* proporcionan paralelismo de datos. Los *kernels* son funciones que son aplicadas a cada elemento del *stream*. En los GPUs, los vértices y fragmentos son los elementos en un *stream* y los *vextex* y *pixel shaders* son los *kernels* que se ejecutan sobre cada uno de los elementos.

Los GPUs solo pueden procesar de forma independiente vértices y fragmentos, pero pueden procesar muchos de ellos en paralelo. Esto es específicamente efectivo cuando el programador necesita procesar varios vértices o fragmentos con una operación especifica. En este sentido, los GPUs son *stream processors*, procesadores que pueden operar en paralelo ejecutando un *kernel* en varios registros en un sentido a la vez.

Ya que los GPUs procesan elementos de forma independiente no hay forma de tener datos en forma estática o compartida. Para cada elemento solo se puede hacer lectura de la entrada, llevar a cabo operaciones en estos elementos y escribir los resultados en la salida. Está permitido tener entradas y salidas múltiples, pero nunca una parte de la memoria puede ser leída y además escrita.

La intensidad aritmética es definida por el número de operaciones a llevar a cabo por palabra o bloque de memoria transferida. Es importante para las aplicaciones de GPGPU que exista una alta intensidad aritmética, de lo contrario la latencia del acceso a memoria limitará la velocidad de computación de los datos.

Las aplicaciones GPGPU ideales son las que tienen gran cantidad de datos, alto paralelismo y mínima dependencia entre los elementos a procesar.

En la siguiente parte de este capítulo se presentará una visión general del lenguaje de programación CUDA para entender como es su arquitectura y modelo de programación.

2.9.2. CUDA

En Noviembre de 2006 NVIDIA lanzo CUDA, un SDK (*Software Development Kit*) y API (*Application Programming Interface*) que les permite a los programadores usar el lenguaje de programación C para implementar algoritmos para su ejecución en las tarjetas de video NVIDIA de la generación GeForce 8 en adelante.

El GPU está basado en una arquitectura sumamente paralelizable ya que contiene cientos de núcleos que pueden ejecutar simultáneamente numerosos hilos de forma concurrente. NVIDIA busca explotar el paralelismo de la GPU mediante un modelo y ambiente de programación paralelo que aproveche al máximo el poder de cómputo de la unidad de procesamiento gráfico para programas de propósito general. Por ello es creada la arquitectura CUDA que permite a los programadores centrarse en el desarrollo de algoritmos paralelos y no en la traducción de los algoritmos al API de hardware gráfico.

CUDA es básicamente una extensión del lenguaje de programación C, por lo que la implementación de los algoritmos paralelos en CUDA es bastante similar a la que se acostumbra hacer en lenguaje C. Además de tener una curva rápida de aprendizaje en cuanto a la sintaxis y la semántica del lenguaje, no requiere de previo conocimiento del API gráfico o de programación en GPU.

2.9.2.1. Modelo de programación

CUDA permite la programación heterogénea, esto significa que las aplicaciones pueden usar tanto el CPU como el GPU para los algoritmos implementados (ver **Figura 2.19**).

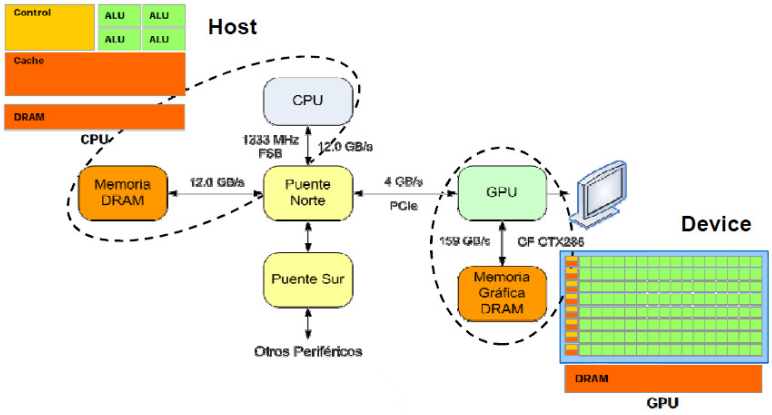


Figura 2.19: Modelo de programación heterogénea con CUDA **[15]**.

Tanto el CPU como el GPU tienen sus propios espacios de memoria los cuales pueden ser copiados de uno a otro o mapeados para tener un acceso más eficiente. El CPU es denominado (*host*), y es el que se encarga de la administración de las tareas y funciones que se van a ejecutar en el GPU (*device*), que es el que se encarga de hacer los cálculos que les fueron asignados desde el *host* a través de los *kernels* que son las funciones a ser ejecutadas en el *device*. En otras palabras, un *kernel* es una función que es llamada por un *host* y es ejecutada por un *device* (ver **Figura 2.20**). Sólo un *kernel* puede ser ejecutado en un *device* a la vez y cuando un *kernel* es ejecutado, son ejecutados varios hilos (*CUDA Threads*) de forma concurrente. Hay algunas distinciones que deben hacerse entre los hilos de CUDA y los hilos de CPU **[12]**.

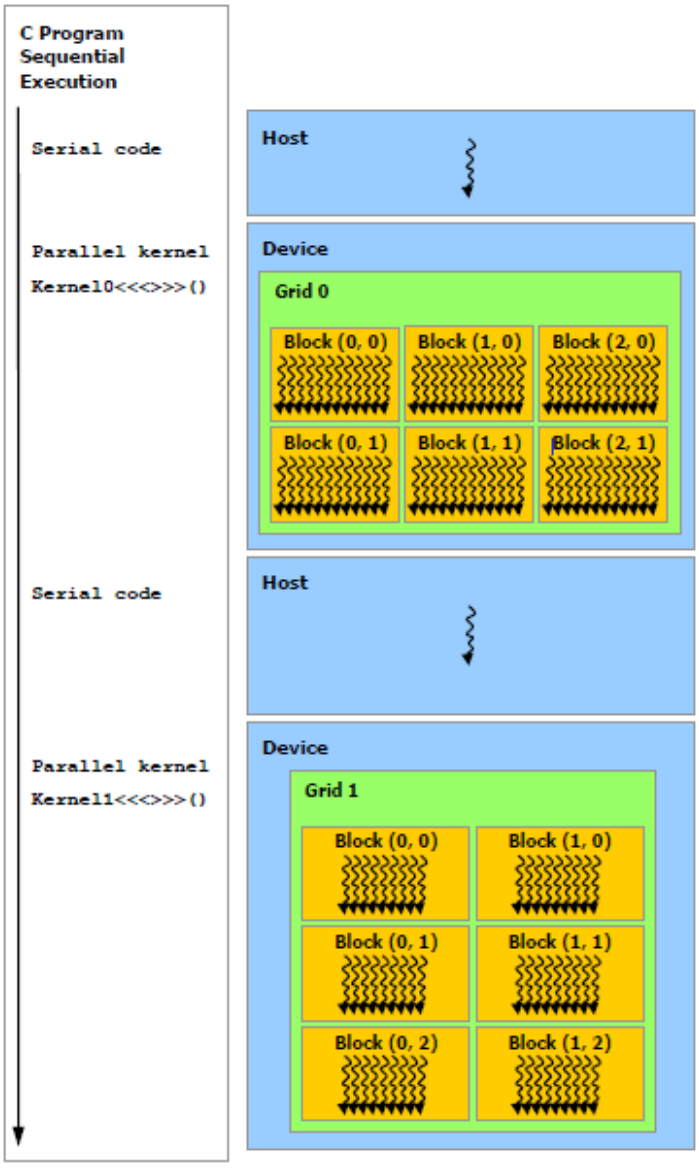


Figura 2.20: Flujo de la ejecución de un programa hecho en CUDA, el código en serial se ejecuta en el *host* mientras que el código en paralelo se ejecuta en el *device* **[12]**.

Los hilos de CUDA son sumamente ligeros en términos de creación y cambio de contexto de un hilo a otro. Miles de hilos de CUDA pueden ser creados en solo algunos ciclos de reloj y como resultado no hay sobrecarga que tenga que ser amortizada durante la ejecución del *kernel*. Dado que en CUDA el intercambio de hilos es de bajo costo, las penalizaciones de tiempo asociadas a un intercambio de hilo cuando se encuentra en el estado suspendido o bloqueado son mínimas.

La agrupación de hilos se puede hacer en *grids* y *blocks*, con el objetivo de compartir datos y sincronizar la ejecución de dichos hilos. Los *blocks* son bloques de hilos que pueden ser de una, dos o tres dimensiones y los *grids* son agrupaciones de bloques de hilos que pueden ser de una o dos dimensiones como se puede apreciar en la **Figura 2.21**.

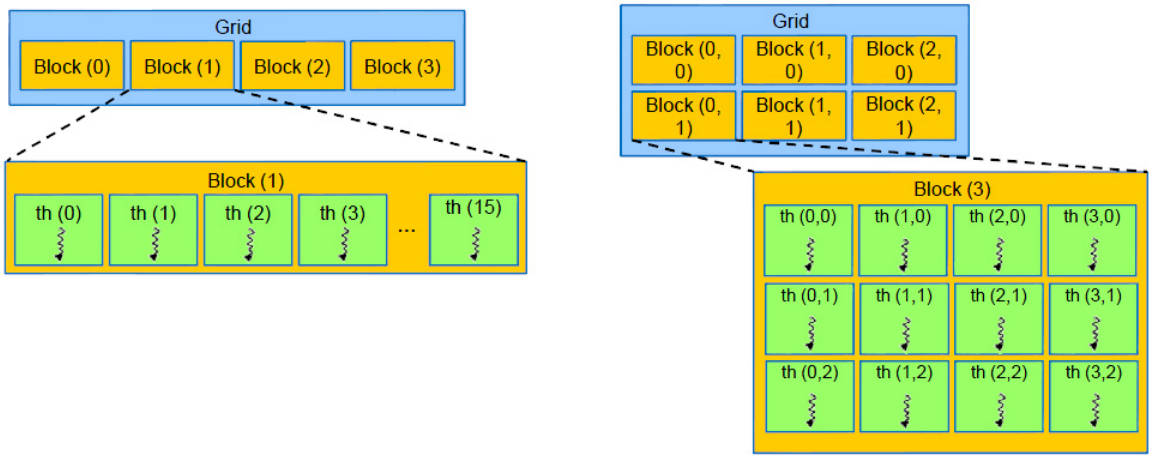


Figura 2.21: Tipos de agrupación de hilos con CUDA. A la izquierda tenemos un *grid* de una dimensión con cuatro bloques que contienen quince hilos cada uno. A la derecha tenemos un grid de dos dimensiones con seis bloques que contienen doce hilos cada uno **[12]**.

2.9.2.2. Compilación del código CUDA

CUDA compila una aplicación C/C++ de CUDA enviándola al compilador NVCC (NVIDIA C *Compiler*) que provee las herramientas necesarias para la compilación en GPU. En el NVCC se generan dos códigos objetos, uno para el CPU (*CPU Code*) y otro para el GPU. El código que va hacia el GPU es pasado previamente por el lenguaje intermedio PTX (*Parallel Thread Execution*) independiente de la plataforma, el cual transforma el código al momento de la ejecución a un binario que puede ser ejecutado por la tarjeta gráfica (ver **Figura 2.22**).

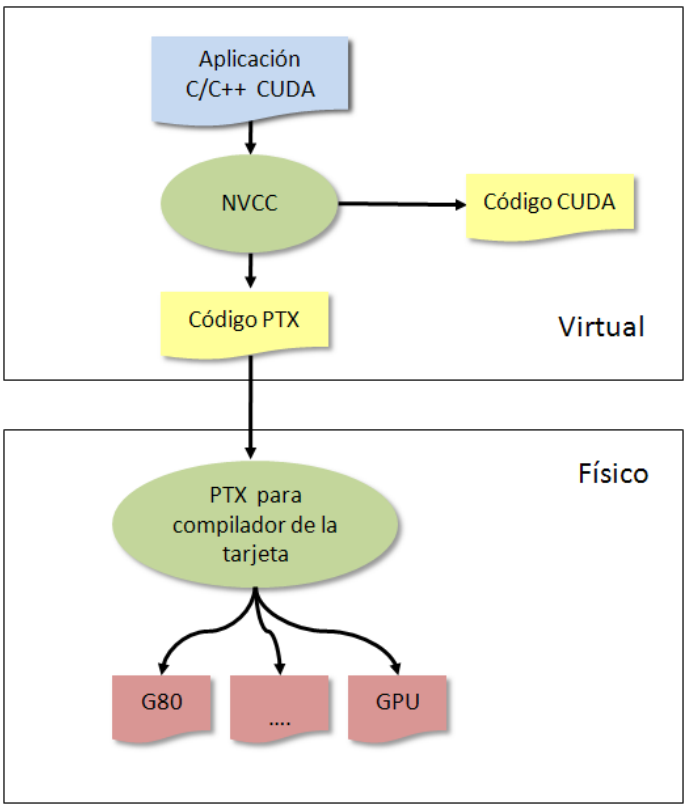


Figura 2.22: Pasos de compilación del código CUDA.

2.9.2.3. Modelo de memoria

El CPU y el GPU tienen espacios de memoria separados, el código del *host* (CPU) maneja la memoria del *device* (GPU). Entre las operaciones básicas de manejo de memoria se encuentran la asignación y liberación de memoria del GPU, la copia de datos desde el *host* al *device* y viceversa y la copia de memoria de un *device* a otro *device*. Estas operaciones aplican para la memoria global del (de los) *device (s).*

Cada multiprocesador posee un pequeño espacio de memoria (*Registers*) así como memoria compartida (*Shared Memory*) para habilitar la comunicación y colaboración entre los hilos. Además de los espacios de memoria de cada multiprocesador dentro del *device*, éste también posee un espacio de memoria (DRAM) el cual se encuentra dividido en memoria local y global.

El *host* y el *device* se comunican a través de la memoria global, la cual puede ser accedida por el *host* para lectura y escritura, tal como se muestra en la **Figura 2.23**.

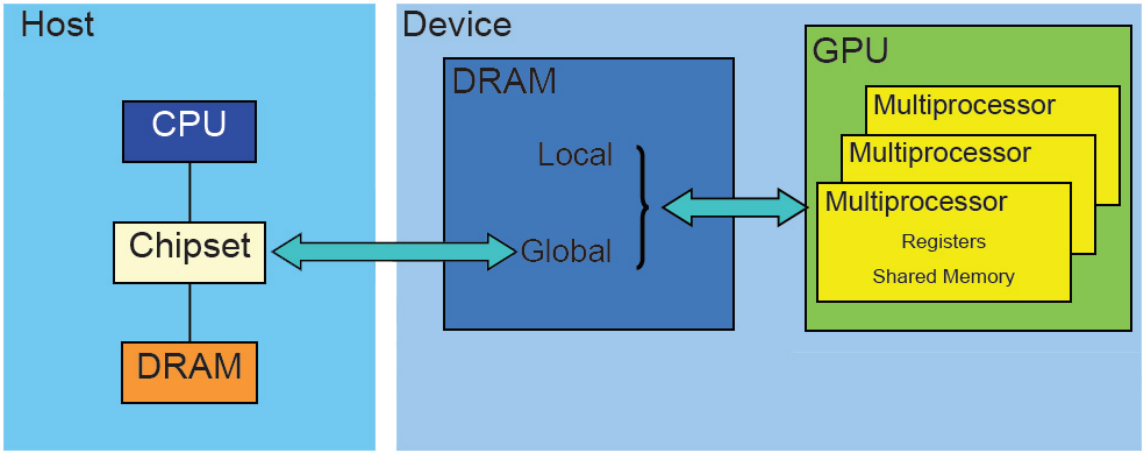


Figura 2.23: Manejo de memoria entre el CPU (host) y el GPU (device).

Los hilos en CUDA pueden acceder a los espacios de memoria múltiple durante su ejecución, como se ilustra en la **Figura 2.24.** Cada hilo tiene una memoria privada local. Cada bloque de hilos tiene una memoria compartida que es visible entre todos los hilos de dicho bloque. Todos los hilos tienen acceso a la misma memoria global.

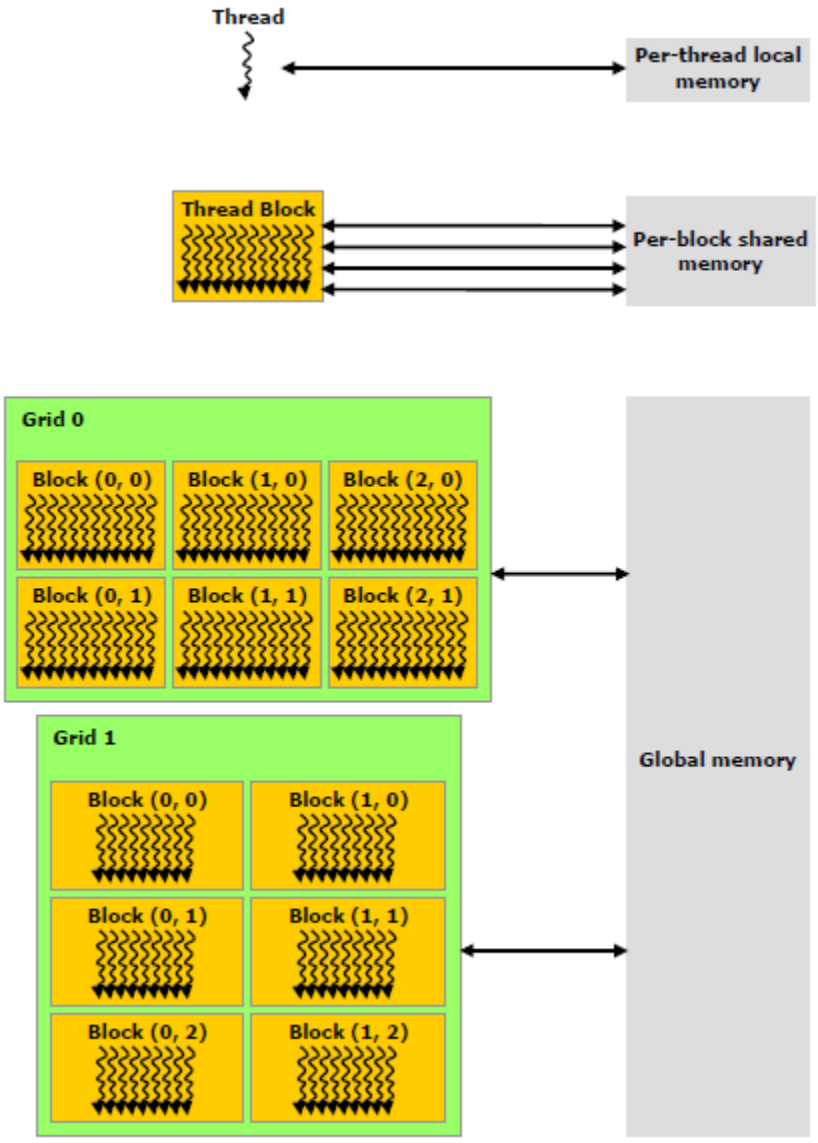


Figura 2.24: Tipos de acceso a memoria por parte de los hilos de CUDA **[12]**.

2.9.2.4. CUDA Framework

CUDA está compuesto en su totalidad por los siguientes componentes:

Driver NVIDIA CUDA: Habilita el acceso al hardware y así poder ejecutar los cálculos en el GPU con CUDA.

CUDA Toolkit: Contiene un conjunto de herramientas necesarias para la creación de programas con CUDA. Entre las más importantes se encuentran:

- Compilador NVCC para C/C++
- Librería CUBLAS (CUDA *Basic Linear Algebra Subprograms*)
- Librería CUFF (CUDA *Fast Fourier Transform*)
- Librería para operaciones con Matrices Dispersas
- Librería para generación de números aleatorios
- Herramientas adicionales y documentación

CUDA SDK: El Kit de desarrollo de CUDA que contiene ejemplos para su utilización y documentación adicional.

En la próxima sección se hablara un poco acerca de la librería CUFFT de CUDA, la cual es de suma importancia para el procesamiento digital de señales.

2.9.2.5. CUFFT

CUFFT (CUDA *Fast Fourier Transform*) es una librería que forma parte de las herramientas de desarrollo de CUDA para calcular de forma eficiente la Transformada Discreta de Fourier (ver Capitulo 2, sección 2.4.2). La implementación del algoritmo de FFT está basado en el paradigma del algoritmo "Divide y Vencerás" y es uno de los algoritmos más importantes y ampliamente usados de los algoritmos numéricos, más que todo en el área de física y procesamiento digital de señales. La librería CUFTT proporciona una interfaz simple para calcular la FFT en forma paralela en los GPUs NVIDIA, además de permitir a los usuarios tener el poder de procesar números en punto flotante y aprovechar el paralelismo del GPU sin tener que crear una implementación personalizada del algoritmo FFT **[16]**.

Las librerías de FFT varían en términos de tamaños de la transformación y del tipo de datos. Por ejemplo, algunas librerías solo implementan FFTs Radix-2, restringiendo la transformación a tamaños de solo potencias de dos, mientras que otras implementaciones soportan implementaciones de transformaciones de tamaño arbitrario.

La librería CUFFT soporta las siguientes características:

- Transformaciones 1D, 2D y 3D de datos complejos y reales.
- Ejecución por lotes para hacer múltiples transformaciones de cualquier dimensión en paralelo.
- Transformaciones en 2D y 3D de tamaños en el rango de 2 a 16384 en cualquier dimensión.
- Transformaciones 1D de hasta 8 millones de elementos.
- Transformaciones de números en punto flotante de doble precisión para hardware compatible (GPUs GT200 y superiores).
- Soporte para ejecución en *streaming*, habilitando cálculos simultáneos a la vez que se hacen movimientos de datos.

Capítulo 3. Diseño e Implementación

3.1. Detalles de implementación

Para la implementación de la aplicación se decidió utilizar el lenguaje de programación C++ sobre el entorno de desarrollo Microsoft Visual Studio 2010, la arquitectura CUDA es usada para realizar las operaciones de los algoritmos que se ejecutan de forma paralela en el GPU y el Kit de Desarrollo Steinberg VST SDK 3.5 para la creación del *plugin* de audio.

Para la interfaz del programa se empleó VSTGUI, la cual es una biblioteca multiplataforma para desarrollar interfaces gráficas de *plugins* de audio. Se empleó el software DAW Cakewalk SONAR 8.5 compatible con el *plugin* VST para hacer las pruebas necesarias de su funcionamiento.

En cuanto a la metodología de programación, se utilizó la programación Orientada a Objetos para la creación de la interfaz de usuario. Para la implementación de los algoritmos de audio en GPU se utilizó la programación paralela.

Una de las limitaciones que podemos encontrar al utilizar CUDA es la utilización de un hardware específico, y es que para poder utilizar esta herramienta es necesario poseer una tarjeta gráfica NVIDIA compatible con CUDA. Dado que la implementación requiere algunas operaciones especiales en CUDA (mapeo de memoria), es imprescindible que posea *Compute Capability* versión 1.1.

En las siguientes secciones se explica de forma más específica la implementación de cada una de las partes que componen el *plugin* VST.

3.2. Implementación general del plugin VST

Virtual Studio Technology (Tecnología de Estudio Virtual) o VST es una interfaz estándar desarrollada por la empresa alemana Steinberg para conectar sintetizadores de audio y *plugins* de efectos a editores de audio y sistemas de grabación **[17]**.

Para poder programar *plugins* para las aplicaciones de audio, la empresa Steinberg creó un kit de desarrollo para *plugins* VST llamado VST SDK. Es un paquete para programadores, con documentación, ejemplos y un conjunto amplio de librerías para desarrollar *plugins* VST. De forma más específica el VST SDK es un conjunto de clases hechas en el lenguaje de programación C++. Este SDK puede ser descargado desde la web de Steinberg **[18]**.

Un *plugin* VST es un proceso de audio que debe ser ejecutado mediante una aplicación que soporte esta tecnología. A esta aplicación se le llama VST *Host*, que maneja el flujo de audio y hace uso del proceso del *plugin* VST como un complemento, en este caso se utiliza el software DAW Cakewalk SONAR 8.5 (ver **Figura 3.1**).

De forma general, el *plugin* VST puede tomar un flujo de datos de audio, aplicarle un procesamiento y enviar el resultado a la aplicación *host*. Los *plugins* VST normalmente hacen su procesamiento usando el CPU de la computadora, sin embargo en este caso se hace uso del GPU para realizar dicho procesamiento.

Desde el punto de vista de la aplicación *host* el *plugin* VST es una caja negra con una cantidad arbitraria de entradas y salidas, ya sean MIDI (ver glosario) o de audio y sus parámetros asociados. La aplicación *host* no necesita saber de qué forma el *plugin* procesa los datos para saber si puede usar el *plugin* o no.

Figura 3.1: Software Cakewalk SONAR 8.5 (*host*) con un *plugin* VST activo.

El proceso que hace el *plugin* tiene una cantidad determinada de parámetros que pueden ser mostrados a través de una interfaz gráfica contenida en una ventana de la aplicación *host*. Estos parámetros se pueden manipular de forma manual o automática si la aplicación *host* lo permite.

La tecnología VST es compatible con los sistemas operativos *Windows* y *Mac OS*. En este caso (*Windows*) el *plugin* VST es un archivo DLL (*Dynamic Link Library*) capaz de ejecutarse en la mayoría de las aplicaciones DAW. Como estos archivos binarios son dependientes de la plataforma donde se ejecutan, un *plugin* VST compilado para *Mac OS* no funcionará en *Windows* y recíprocamente.

El sistema de procesamiento de audio utilizado en la mayoría de las aplicaciones DAW es de procesamiento por bloques. La forma de procesamiento de un *plugin* VST es un ejemplo de ello. El flujo de audio es segmentado en una serie de bloques donde la aplicación *host* le provee al *plugin* VST los bloques de forma secuencial. En el programa *host* se determina de qué tamaño van a ser los bloques de audio a procesar.

Cada muestra de audio es acumulada hasta obtener la cantidad definida por bloque de procesamiento que generalmente va del rango de 64 muestras (baja latencia) a 8192 muestras (alta latencia), comúnmente siendo potencia de 2, para luego ser pasada a una función predefinida en el *plugin* VST que se encarga de hacer los cálculos correspondientes.

Para que este sistema de procesamiento de audio en tiempo real funcione correctamente el cálculo de los datos tiene que ser rápido. Por ejemplo, un bloque de datos de 512 muestras con una frecuencia de muestreo de 44.100 muestras por segundo tiene que ser procesado a lo sumo en 11,6 milisegundos. Este estimado de tiempo se calcula del siguiente modo:

$$\frac{1}{fs} \cdot BlkSz \cdot 1.000\, ms. \qquad (\boldsymbol{Ec.3.1})$$

donde fs es la frecuencia de muestreo y $BlkSz$ el número de muestras del bloque de datos de audio a procesar.

Si el procesamiento requiere más tiempo que este el sistema se cargaría a su máxima capacidad (exceso de uso de CPU) y ocurrirían fallas de procesamiento o suspensión del sistema de reproducción de audio.

Los datos de audio están dados como un arreglo de números de punto flotante, con magnitudes en el rango de -1.0 a +1.0. Cada número representa la amplitud de una muestra de señal de audio.

En la próxima sección se explica con detalle cómo está conformado el *plugin* VST a nivel de código, su estructura de clases y métodos asociados.

3.3. Estructura de clases del plugin VST

Para desarrollar el *plugin*, este debe ser implementado como un conjunto de clases. Ya que el VST SDK provee un conjunto de clases predefinidas no es necesario construir todas las estructuras desde el principio. La estructura interna del *plugin* VST está dividida en varias secciones como se muestra a continuación.

Clase *GPUAFXProcessor*: Hereda de la clase *AudioEffect* definida en el VST SDK para crear un nuevo efecto de audio (ver **Figura 3.2** y **Figura 3.8**). Es donde se inicializan y ajustan los parámetros principales del *plugin* VST, como la inicialización del búfer de audio, el tamaño de bloque de audio, la tasa de muestreo, el método en donde se realiza el procesamiento del audio, etc. Ademas de esto, instancia un objeto de la clase CUDA que es un *thread* de *Windows* se encarga del pase de parámetros y la comunicación con el GPU a través de CUDA.

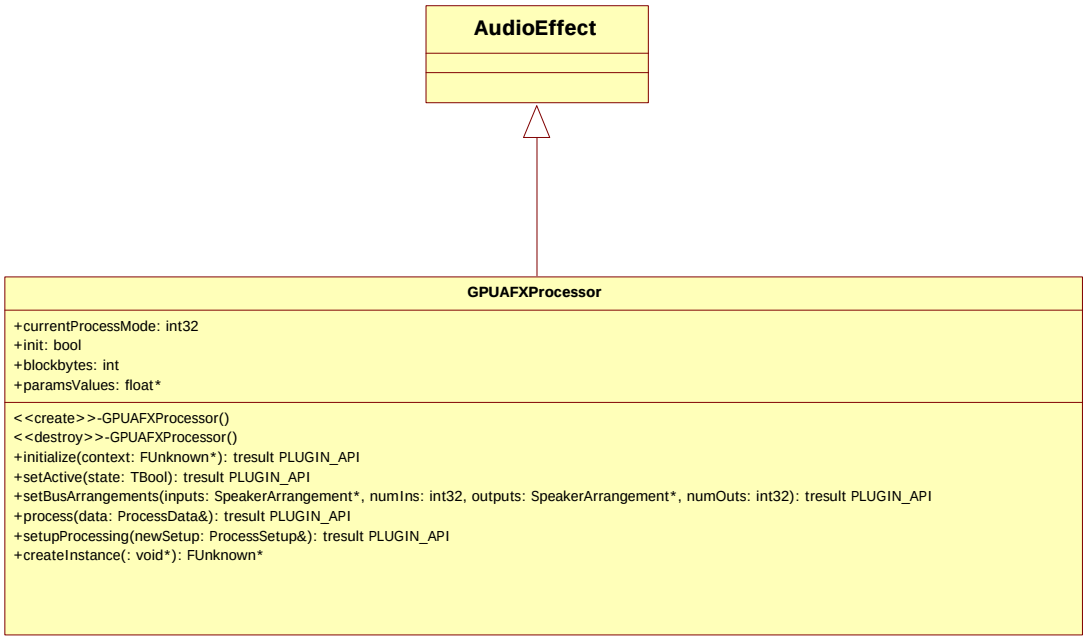


Figura 3.2: Diagrama de clases de la clase *GPUAFXProcessor*.

Esta clase tiene constructor y destructor; el método *createInstance()* se encarga de instanciar el efecto de audio y el método *initialize()* inicializa las variables que se vayan a utilizar en la instancia del efecto de audio. El método *setBusArrangements()* se encarga de definir cuántos canales de audio van a ser procesados (mono o estéreo). El método *setupProcessing()* inicializa los búferes de audio, tamaño de bloque y frecuencia de muestreo según como se hayan definido en la aplicación *host*.

Finalmente, el método *process()* se encarga de ejecutar todos los cálculos que sean definidos para transformar los búferes de audio y eviarlos de vuelta a la aplicación *host*.

Clase *Thread*: Se utiliza para crear un *thread* de *Windows* para procesar el audio con CUDA (ver **Figura 3.3** y **Figura 3.8**). Cada vez que el Sistema Operativo solicita el CPU para cederselo a un hilo de otro proceso, los registros y la pila (es decir, el contexto del hilo) contienen unos valores determinados. El Sistema Operativo guarda todos esos datos en cada cambio, de modo que al volver a conmutar el hilo inicial, pueda restaurar el contexto en el que quedo anteriormente. Si el procesamiento no se hace de este modo, los contextos se destruirían al terminarse cada llamada a la función *audioProcessing()* de CUDA, ya que el tiempo de vida de una función es solo durante su llamada y no a lo largo de toda la ejecución del proceso. En cambio con la creación del *thread* mantenemos todas las asignaciones de memoria y estados de CUDA.

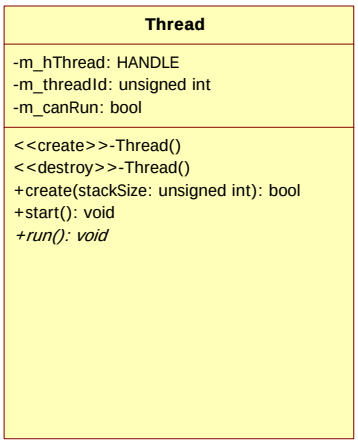


Figura 3.3: Diagrama de clases de la clase *Thread*.

Esta clase tiene su constructor y destructor, el método *create()* se encarga de instanciar un *thread* de *Windows,* el método *run()* ejecuta las operaciones definidas en el *thread* (llamadas a funciones, etc.). Finalmente el método *start()* inicia el *thread* una vez creado.

Clase CUDA: Hereda de la clase *Thread* y es instanciada por la clase *GPUAFXProcessor*. Es donde se mandan a ejecutar todos los cálculos de los algoritmos de audio (ver **Figura 3.4** y **Figura 3.8**). Para ello se han creado *kernels* para cada efecto de audio. En esta clase también se hace el manejo del orden de los efectos y la sincronización con el GPU.

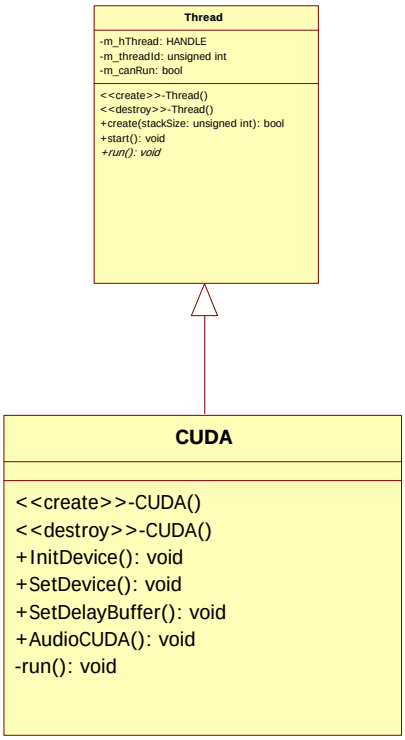


Figura 3.4: Diagrama de clases de la clase CUDA.

Esta clase tiene constructor y destructor; el método *initDevice()* se encarga de inicializar los parámetros del *device* de CUDA, como por ejemplo el identificador del *device*, las banderas de estados como fijar el mapeo de memoria, etc. El método *setDelayBufer()* se encarga de inicializar en el *device* el búfer de memoria para el efecto *delay*; *setDevice()* determina la cantidad de bloques e hilos a utilizar por los *kernels* de CUDA; además de esto, inicializa todos los búferes de audio a utilizar por el GPU. El método *run()* heredado de la clase *Thread* contiene la llamada al método *audioProcessing()* que es el que se encarga de ejecutar los *kernels* de los efectos de audio, así como administrar los pases de parámetros a los *kernels* y la activación o desactivación de los efectos.

Clase Knob*Parameter*: Hereda de la clase *Parameter* del VST SDK y es instanciada pro la clase *GPUAFXBaseController* (ver **Figura 3.5** y **Figura 3.8**). En esta clase se definen los parámetros (perillas y switches) de cada efecto para poder manipularlos de forma independiente.

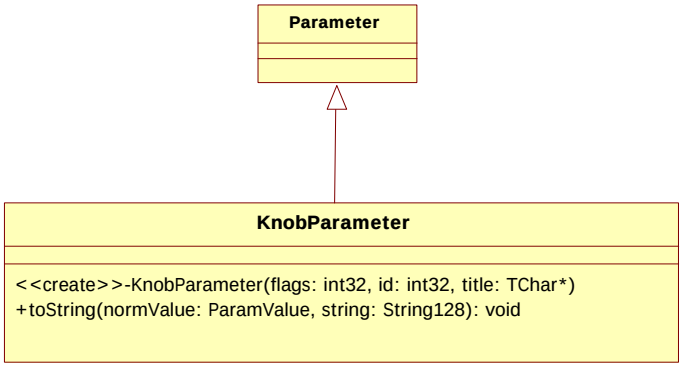


Figura 3.5: Diagrama de clases de la clase *KnobParameter*.

Esta clase tiene un constructor para definir sus parámetros por defecto (identificador, nombre, valor por defecto). El método *toString()* se encarga de enviar una cadena de texto que identifica a cada parámetro con su rango de valores específico.

Clase *GPUAFXBaseController*: Hereda de la clase *EditController* y *VST3EditDelegate* que forman parte del VST SDK. En esta clase se asocian los identificadores de cada parámetro con su valor respectivo instanciando a la clase *KnobParameter* por la cantidad de parámetros preestablecidos (ver **Figura 3.6** y **Figura 3.8**). También gestiona la forma en que se van a desplegar los elementos de la interfaz gráfica creada.

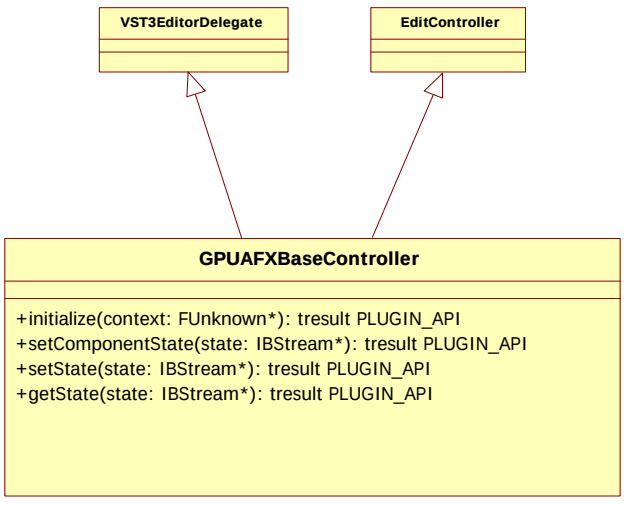


Figura 3.6: Diagrama de clases de la clase *GPUAFXBaseController*.

El método *initialize()* se encarga de instanciar los parámetros que se crean de la clase *KnobParameter* y guardar dichas instancias en un arreglo de parámetros para ser posteriormente utilizados por la clase *GPUAFXProcessor* (ver **Figura 3.8**). El método *setComponentState()* se encarga de guardar el estado de los parámetros instanciados en un *preset* definido por el usuario desde la aplicación *host*. El método *setState()* guarda los valores asociados a los parámetros instanciados y *getState()* carga dichos valores.

Clase *GPUAFXController*: Hereda de la clase *GPUAFXBaseController*. Esta clase (ver **Figura 3.7** y **Figura 3.8**) se encarga de crear las vistas de la interfaz gráfica de usuario así como crear subcontroladores que se pueden asignar a los parámetros para un comportamiento personalizado.

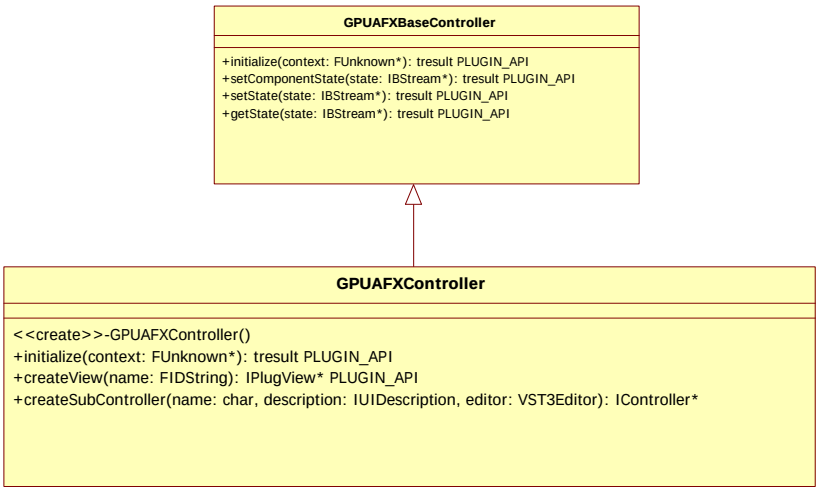


Figura 3.7: Diagrama de clases de la clase *GPUAFXController*.

El método *initialize()* se encarga de inicializar los parámetros que pueden ser usados en el método *createView()* o *createSubController()* (ver **Figura 3.8**). El método *createView()* define la vista general de la interfaz gráfica de usuario, en este método se hace la carga del archivo XML que contiene la estructura y parámetros de la vista. El método *createSubController()* sirve para dar un comportamiento personalizado (*tooltips*, cambios de texturas, color, tipo de fuente de texto, etc.) a los parámetros que se le asignen el uso de esta función.

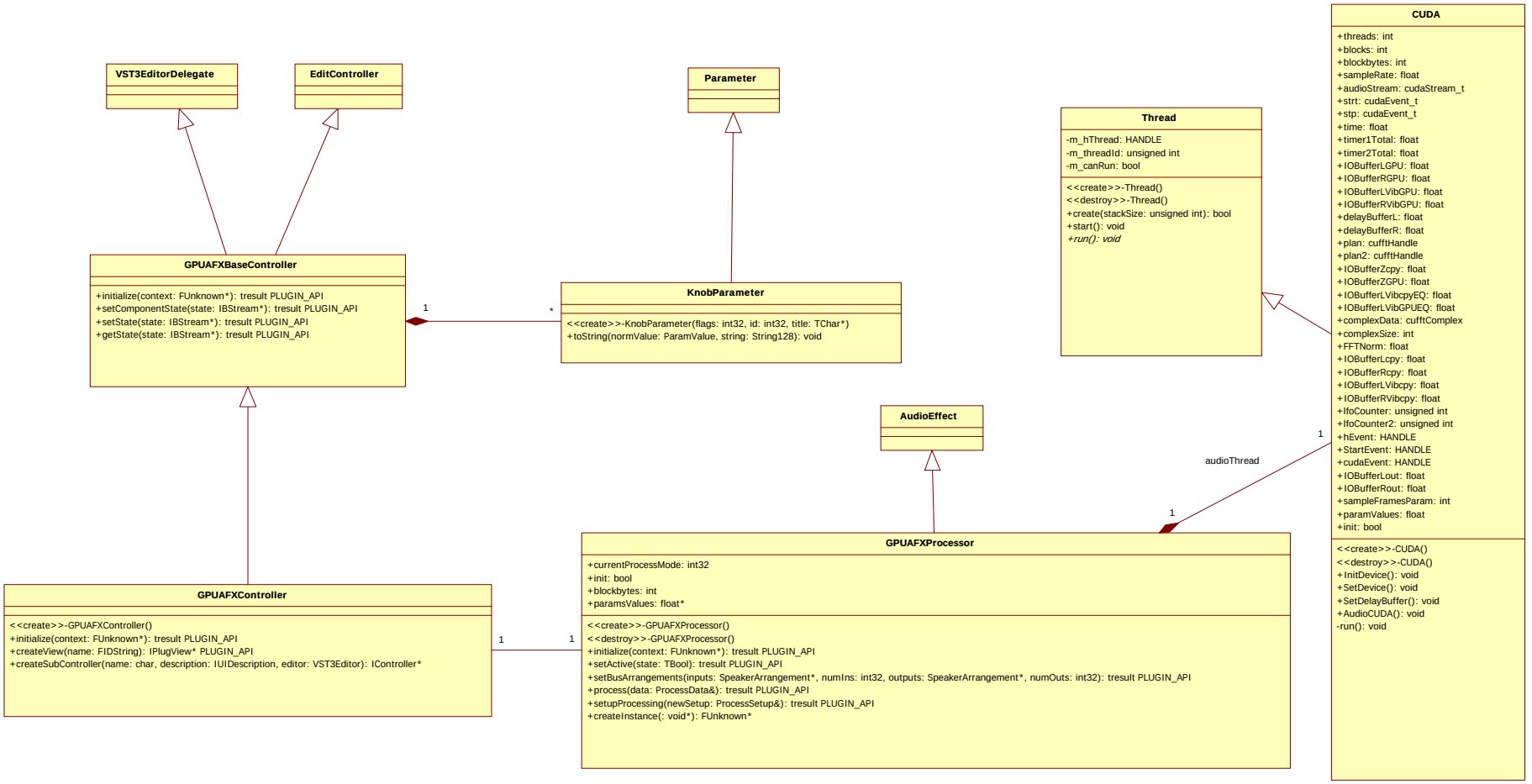


Figura 3.8: Diagrama de clases del proyecto GPUAFX.

En la siguiente sección se explica cómo se realizó la implementación de la interfaz gráfica de usuario con la librería VSTGUI.

3.3. Implementación del GUI

VSTGUI es una librería multi-plataforma para C++ diseñada principalmente para crear interfaces gráficas de usuario para *plugins* de audio. Esta herramienta gráfica proporciona manejo de ventanas y gestión de eventos de usuario para poder interactuar con la interfaz creada.

Funciona en los sistemas operativos *Windows* y *Mac OSX*. Entre los objetos más comunes de esta librería están los contenedores de vista, perillas, sliders y switches. Estos crean la sensación de estar interactuando con interfaces de audio reales logrando que el trabajo sea más natural para el usuario.

Las interfaces se pueden construir con implementaciones específicas de los objetos de la librería en C++ o describiendo la estructura de la interfaz de audio en archivos XML.

Para la implementación del GUI, se creó un contenedor de efectos ordenados, con una barra de desplazamiento vertical para poder desplazarse entre los efectos. La disposición de los objetos dentro del GUI está definida en un archivo XML. En este archivo se declaran los objetos con su etiqueta correspondiente y de forma jerárquica. Adicionalmente el VSTGUI tiene la opción de habilitar un editor visual para poder realizar cambios de forma más sencilla.

Para la creación de las interfaces, se utilizaron las siguientes clases de la librería:

***CAnimKnob*:** Es la clase con la cual se implementan las perillas de cada efecto, con las cuales se pueden ajustar los parámetros de dichos efectos y así percibir los cambios auditivamente. Su función es intercambiar una serie de mapas de bits en secuencia para crear un efecto de perilla animada al momento de posicionar el mouse encima, también envía a la clase *GPUAFXProcessor* el valor del parámetro actual el cual esta normalizado entre 0 y 1.

***COnOffButton*:** Es la clase con la cual se implementa el encendido o apagado de cada efecto. Su función es intercambiar dos mapas de bits según el estado que tenga al hacer click, así como su valor que puede ser de 0 o 1.

***CViewContainer*:** Es la clase que contiene los objetos con los cuales el usuario interactúa (perillas, switches, sliders, etc.). Esta clase puede tener un mapa de bits de fondo.

***CScrollView*:** Esta clase hereda de la clase *CViewContainer* y su función es ser un contenedor pero además tener un slider para poder navegar en dicho contenedor. Es sumamente útil cuando la cantidad de objetos es considerable y se necesita navegar a través de ellos.

Para cada efecto se implementó el uso de *CAnimKnob* para el manejo de las perillas y un *COnOffButton* para activar o desactivar el efecto. Estos objetos están contenidos en un objeto *CScrollView* que hereda de la clase *CViewContainer* (ver **Figura 3.9**).

Figura 3.9: Contenedor de efectos del *plugin* GPUAFX y sus clases de GUI asociadas.

En la siguiente sección se muestran los detalles de implementación de cada uno de los efectos propuestos para la solución, los parámetros de cada efecto y su algoritmo en pseudocódigo.

3.4. Implementación de los efectos de audio

Las implementaciones en CPU se hicieron de forma secuencial mientras que en GPU se hicieron de forma paralela. En algunos casos, para adaptar algunos algoritmos de forma paralela se tuvo que hacer ajustes en la forma en que se procesan, incluso haciendo uso de más de un *kernel* de CUDA para llevar a cabo la implementación.

3.4.1. Over Drive

El algoritmo consiste en hacer pasar la señal por una función de transferencia no lineal (ver **Figura 3.10**), lo cual cambia la forma de la señal, es decir, distorsionará la señal de audio. La nueva forma de la señal dependerá de la forma de la función de transferencia:

$$f(x) = \begin{cases} -\sqrt{-x}, & x \leq 0 \\ \sqrt{x}, & x > 0 \end{cases} \qquad (\boldsymbol{Ec.3.2})$$

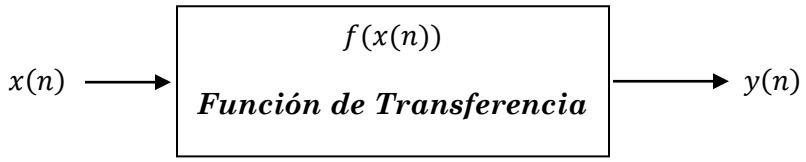


Figura 3.10: Diagrama de bloques del efecto *Over Drive*.

Los parámetros de este efecto son (ver **Figura 3.11**):

- **Ganancia (*Gain*):** Es la cantidad de saturación que se le aplica a la señal de entrada.
- **Nivel de salida (*Level*):** Se define como la cantidad de volumen que tendrá el efecto.

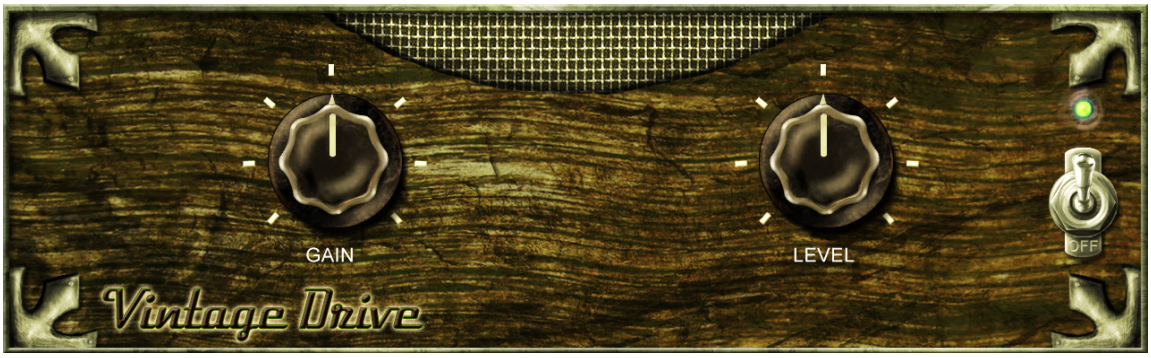


Figura 3.11: Interfaz gráfica del efecto *Over Drive*.

El **Algoritmo 1** corresponde a la versión secuencial para CPU del efecto *Over Drive*. Para la implementación en CUDA se elimina el ciclo **para**, ya que todo el código de la función tiene que ser ejecutado de forma paralela por cada hilo asignado al *kernel* de CUDA de este efecto.

Algoritmo 1 Algoritmo del efecto *Over Drive*

```
 1: función OverDrive(array in, array out, int N, float gain, float lvl)
 2:
 3:   float TFResult
 4:
 5:   para int i ← 0 hasta i < N hacer
 6:
 7:     si in[i] > 0 entonces
 8:       TFResult ← sqrtf(in[i])
 9:     si no
10:       TFResult ← -sqrtf(-in[i])
11:     fin si
12:
13:     out[i] ← (lvl · 2) · (gain · (TFResult - in[i]) + in[i])
14:
15:   fin para
16: fin función
```

En el **Algoritmo 1**, línea 3, se encuentra la variable **TFResult**, la cual va a contener el resultado del cálculo de la función de transferencia (ver **Figura 3.12**) aplicada a la señal de entrada. Luego en las líneas 7-11 para cada elemento del arreglo de entrada **in** se calcula la relación que tiene su valor con respecto a la función de transferencia $\sqrt{x}$ siempre y cuando su valor sea mayor a 0, de lo contrario se calcula con la función de transferencia $-\sqrt{-x}$. Una vez obtenido el resultado, en la línea 13 se procede a determinar la proporción de mezcla entre la señal original y la señal procesada por la función de transferencia con la variable **gain**. Con la variable **lvl** se define la cantidad de volumen del efecto.

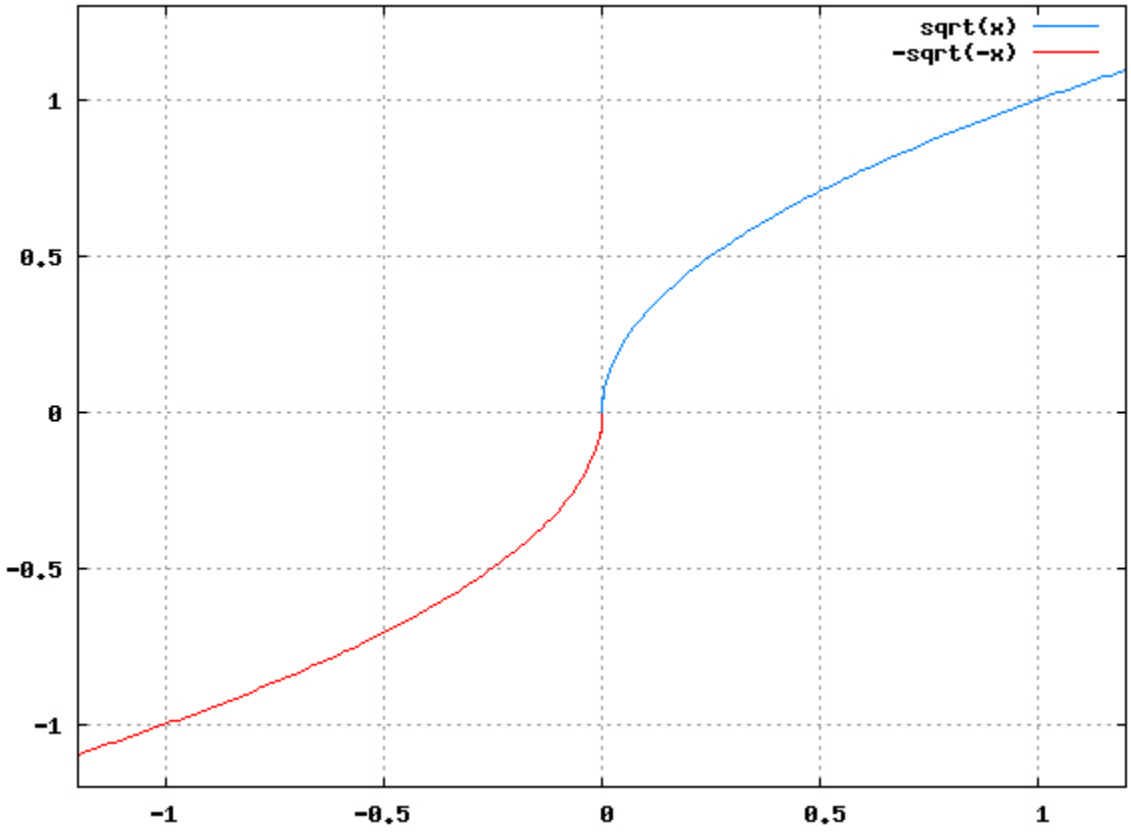


Figura 3.12: Función de transferencia del efecto *Over Drive*.

3.4.2. Distortion

Al igual que el *Over Drive*, el algoritmo consiste en hacer pasar la señal por una función de transferencia no lineal (ver **Figura 3.13**), pero en este caso la forma de la señal cambiará de un modo más brusco. La nueva forma de la señal dependerá de la forma de la función de transferencia tangente hiperbólica ($\tanh(x)$).

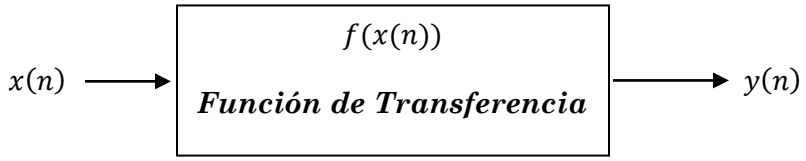


Figura 3.13: Diagrama de bloques del efecto *Distortion*.

Los parámetros generales de este efecto son (ver **Figura 3.14**):

- **Ganancia (*Gain*):** Es la cantidad de saturación que se le aplica a la señal de entrada.
- **Nivel de salida (*Level*):** Se define como la cantidad de volumen que tendrá el efecto.

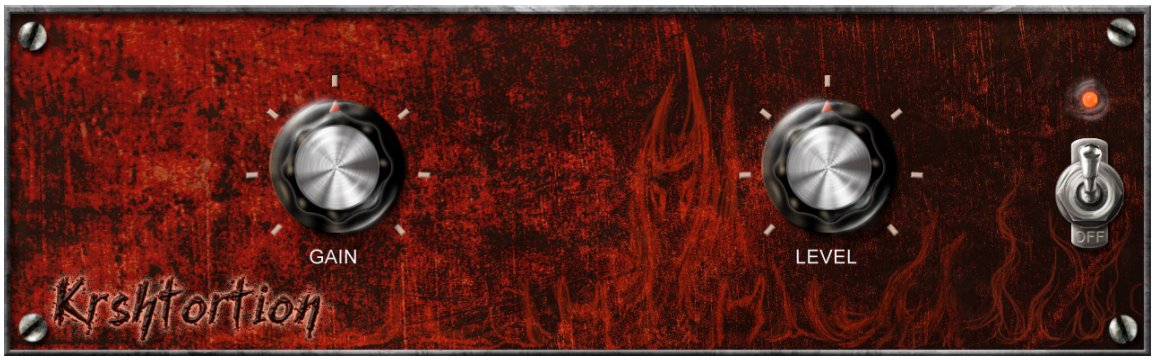


Figura 3.14: Interfaz gráfica del efecto *Distortion*.

El **Algoritmo 2** corresponde a la versión secuencial para CPU del efecto *Distortion*. Para la implementación en CUDA se elimina el ciclo **para**, ya que todo el código de la función tiene que ser ejecutado de forma paralela por cada hilo asignado al *kernel* de CUDA de este efecto.

Algoritmo 2 Algoritmo del efecto *Distortion*

```
 1: función Distortion(array in, array out, int N, float gain, float lvl)
 2:
 3:   float TFResult
 4:
 5:   para int i ← 0 hasta i < N hacer
 6:
 7:     TFResult ← 0.8 · tanh((gain · 1023 + 1) · in[i])
 8:
 9:     out[i] ← (lvl · 2) · (gain · (TFResult - in[i]) + in[i])
10:
11:   fin para
12: fin función
```

En el **Algoritmo 2**, línea 3, se encuentra la variable **TFResult**, la cual va a contener el resultado del cálculo de la función de transferencia aplicada a la señal de entrada. Luego para cada elemento del arreglo de entrada **in** se calcula la relación que tiene su valor con respecto a la función de transferencia *tanh(x)*. La variable **gain** define la forma de la función de transferencia, a medida que incrementa su valor, la pendiente de la recta tangente a la función en el punto cero tiende hacia el infinito (ver **Figura 3.15**). Una vez obtenido el resultado, en la línea 9 se procede a determinar la proporción de mezcla entre la señal original y la señal procesada por la función de transferencia con la variable **gain**. Con la variable **lvl** se define la cantidad de volumen del efecto.

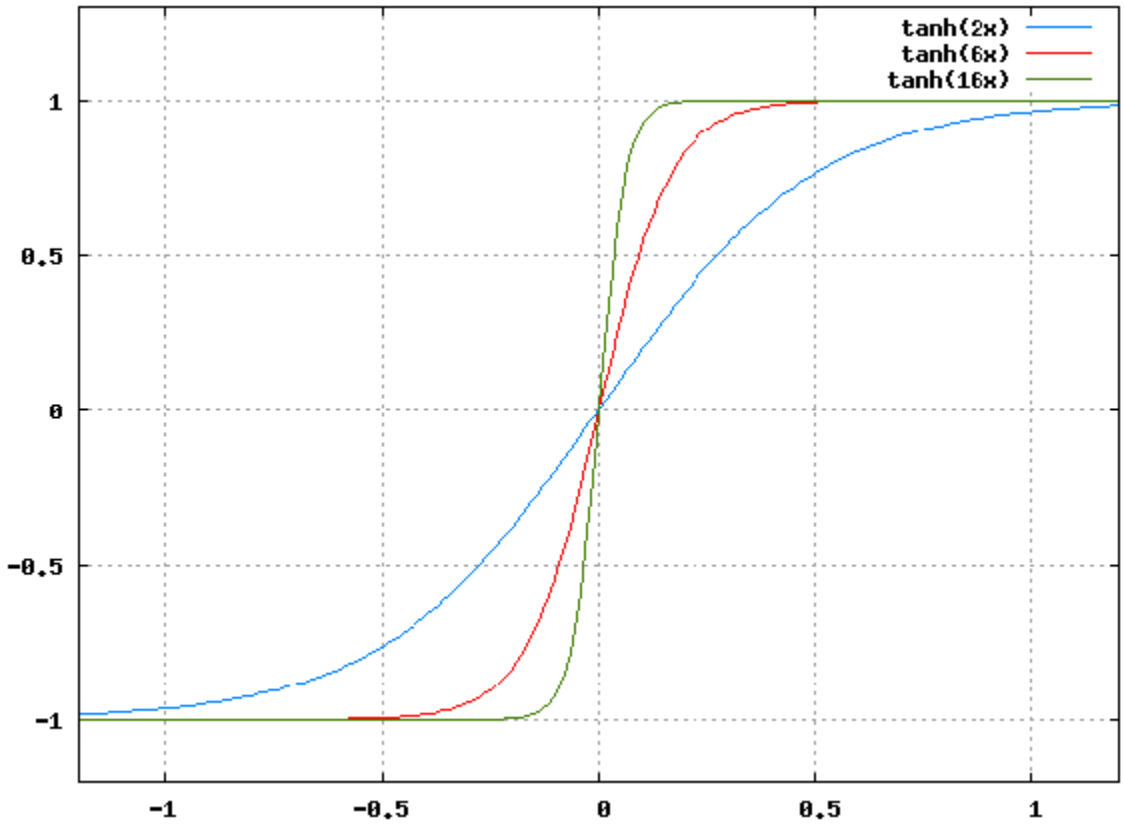


Figura 3.15: Función de transferencia del efecto *Distortion*.

3.4.3. Ecualizador

El algoritmo del ecualizador consiste en tomar la señal y aplicarle la Transformada de Fourier, una vez en el dominio de la frecuencia se puede multiplicar por un conjunto de filtros con sus niveles de ganancia respectivos y así determinar la nueva forma de la señal. Luego de esto se procede a hacer la Transformada Inversa de Fourier de la señal modificada para regresarla al dominio del tiempo (ver **Figura 3.16**).

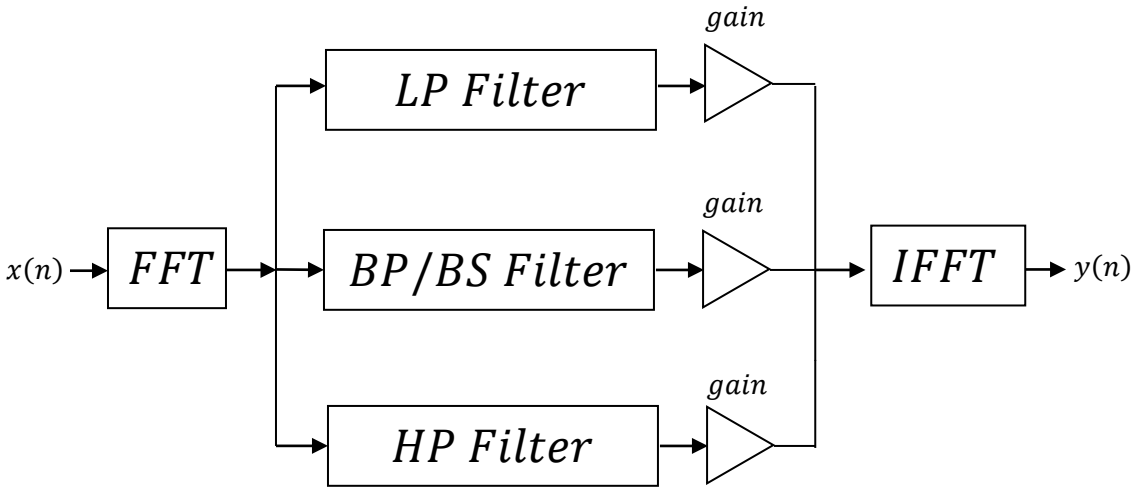


Figura 3.16: Diagrama de bloques del Ecualizador.

Los parámetros que se pueden modificar en un ecualizador son (ver **Figura 3.17**):

- **Frecuencias Bajas (*Low*):** Es el parámetro que modifica el rango de ganancia de las frecuencias bajas.

- **Frecuencias Medias (*Mid*):** Es el parámetro que modifica el rango de ganancia de las frecuencias medias.

- **Frecuencias Altas (*High*):** Es el parámetro que modifica el rango de ganancia de las frecuencias altas.

Figura 3.17: Interfaz gráfica del efecto *Ecualizador*.

El **Algoritmo 3** corresponde a la versión secuencial para CPU del efecto *Ecualizador*. Para la implementación en CUDA se elimina el ciclo **para**, ya que todo el código de la función tiene que ser ejecutado de forma paralela por cada hilo asignado al *kernel* de CUDA de este efecto.

Algoritmo 3 Algoritmo del efecto *Ecualizador*

```
1: función EQ(array2D inComplex, array2D outComplex, int N, float scale,
2: float low, float mid, float high)
3:
4:   float mag, magin, phi, p, q
5:   float dB2lin ← 0,1155
6:
7:   low ← exp(((low · 48) – 24) · dB2lin)
8:   mid ← exp(((mid · 48) – 24) · dB2lin)
9:   high ← exp(((high · 48) – 24) · dB2lin)
10:
11:  para int i ← 0 hasta i < N hacer
12:    magin ← sqrt(inComplex.x · inComplex.x + inComplex.y · inComplex.y)
13:
14:    si i < (N/128) entonces
15:      mag ← low · magin
16:    si i >= (N/128) Y i < (N/16) entonces
17:      q ← (mid-low) / (N/16 – N/128)
18:      p ← low + (i – N/128) · q
19:      mag ← p · magin
20:    si i >= (N/16) Y i < (N/4) entonces
21:      q ← (high-mid) / (N/4 – N/16)
22:      p ← mid + (i – N/16) · q
```

```
23:          mag ← p · magin
24:       si no
25:          mag ← high · magin
26:       fin si
27:
28:       phi ← atan2(inComplex.y, inComplex.x)
29:
30:       outComplex.x ← scale · (mag · cos(phi))
31:       outComplex.y ← scale · (mag · sin(phi))
32:
33:    fin para
34: fin función
```

Algoritmo 4 Ejecución del algoritmo del efecto *Ecualizador*

```
1: FFT(in, inComplex, N)
2: EQ(inComplex, outComplex, N, scale, low, mid, high)
3: IFFT(outComplex, out, N)
```

En el Algoritmo 3, líneas 7-9, los parámetros **low**, **mid** y **high** que están en el rango de -24 a +24 dB son transformados de escala en decibeles a escala lineal con la fórmula $e^{x \cdot 0{,}1155}$ **[3]**. Luego de esto se procede a manipular cada muestra de la entrada. En la línea 12 se calcula la magnitud de cada muestra con la fórmula $\sqrt{x^2 + y^2}$. Seguidamente en las líneas 14-26 se calcula el valor de magnitud de cada banda de frecuencia con una interpolación lineal entre los puntos centrales definidos para cada banda, estos se multiplican por la magnitud de la entrada para obtener la nueva respuesta en frecuencia de cada muestra. Posteriormente en la línea 28 se determina la fase de cada elemento con la fórmula $atan2(y, x)$, que es el cálculo del arcotangente en el plano complejo. Luego en la línea 30 y 31 se convierten los datos de forma polar a forma rectangular normalizados con el parámetro **scale** que es la unidad dividida entre el número de elementos a procesar **N**.

El **Algoritmo 4** comienza convirtiendo la señal de entrada **in** del dominio del tiempo al dominio de la frecuencia con la Transformada Rápida de Fourier (FFT). Esto da como resultado un arreglo de números complejos **inComplex** el cual se utiliza para realizar los cálculos correspondientes de ecualización. Se llama a la función **EQ** descrita en el **Algoritmo 3** y finalmente se hace la Transformada Inversa de Fourier para regresar los datos resultantes al dominio del tiempo y envíalos al búfer de salida. Para realizar la FFT en CPU se usó la librería FFTW **[19]** y para GPU se utilizó la librería CUFFT **[16]** que forma parte de CUDA.

3.4.4. Vibrato

Para implementar este efecto se necesita controlar la cantidad de retardo de línea (generalmente entre 5 ms. y 10 ms. de *delay*) mediante un LFO (siglas *de Low Frequency Oscilator*), del cual se tiene control de su amplitud y frecuencia. La amplitud del *Vibrato* determinará cuál es el retardo inicial sobre el cual oscilará la señal sinusoidal (ver **Figura 3.18**).

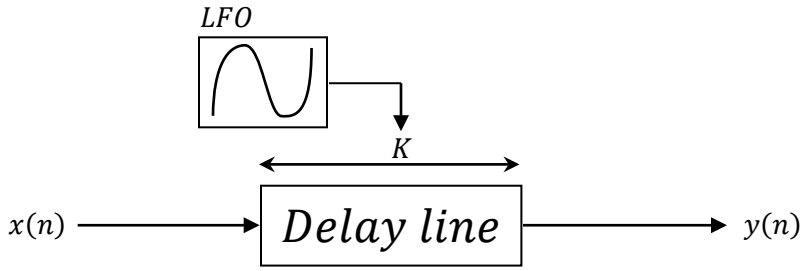


Figura 3.18: Diagrama de bloques del efecto *Vibrato*.

Los parámetros del efecto *Vibrato* son (ver **Figura 3.19**):

- **Frecuencia (*Rate*):** Es la velocidad con la cual se hace la variación de tono del *Vibrato*. Generalmente la frecuencia del LFO oscila entre 0 Hz. y 10 Hz.

- **Profundidad (*Depth*):** Se define como la intensidad de *Vibrato* que se le aplica a la señal.

- **Proporción de Mezcla (*Mix*):** Es la cantidad de efecto *Vibrato* que se le aplica a la señal original.

Figura 3.19: Interfaz gráfica del efecto *Vibrato*.

El **Algoritmo 5** corresponde a la versión secuencial para CPU del efecto V*ibrato*. Para la implementación en CUDA se elimina el ciclo **para**, ya que todo el código de la función tiene que ser ejecutado de forma paralela por cada hilo asignado al *kernel* de CUDA de este efecto.

Algoritmo 5 Algoritmo del efecto *Vibrato*

```
 1: función Vibrato(array in, array out, array oldIn, int N, int counter,
 2: int sampleRate, float rate, float depth, float mix)
 3:
 4:   float TWOPI ← 6,283185307
 5:   float modifier, index, frac, next
 6:   int indexInt
 7:
 8:   rate ← rate · 10
 9:
10:   float t ← (TWOPI · rate) / sampleRate
11:
12:   para int i ← 0 hasta i < N hacer
13:
14:     modifier ← depth · sin(t · (N · counter + i))
15:     index ← i + (modifier · N/2 + N/2)
16:     indexInt ← index
17:     frac ← index – indexInt
18:     next ← oldIn[i+1]
19:
20:     out[i] ← (1 - mix) · in[i] + mix · (oldIn[i] + frac · (next – oldIn[i]))
21:
22:   fin para
23: fin función
24:
25: counter ← counter + 1
```

En el **Algoritmo 5**, línea 8, se encuentra la variable **rate** que es multiplicada por 10, ya que las oscilaciones del modificador de este efecto es de máximo 10 Hz. La variable **t** es la proporción de cambio de fase de la onda senoidal que se va a generar para manipular el efecto. En la línea 14, en la variable **modifier**, se calcula el valor de la onda seno que se va a utilizar para recorrer el índice del arreglo **oldIn**, el cual es un arreglo de tamaño $2 \cdot N$ que contiene el bloque de audio anterior y actual. Como el índice de la onda seno tiene que estar dentro del rango del tamaño del arreglo **oldIn** en la línea 15 se calcula en la variable **index** un valor que esté dentro de este rango.

En la línea 16 se calcula la parte entera del índice y en la siguiente línea el resto que queda de la variable **index**. Se calcula la siguiente posición en el índice del arreglo **oldIn** que se utilizará para calcular una interpolación lineal entre la posición actual del índice y la siguiente posición. En la línea 20 se calcula la proporción de mezcla de la señal original con respecto a la señal modificada por el efecto. Finalmente al terminar la función en la línea 25 se suma 1 a la variable **counter**, con el propósito de poder seguir la forma de onda del modificador para el siguiente bloque de audio a procesar.

3.4.5. Chorus

Este efecto se consigue mezclando la señal original con una copia ligeramente retardada y modulada de la misma (entre 10 ms. y 30 ms. de *delay*) donde el tiempo del retardo varía de forma constante a través de un LFO (ver **Figura 3.20**). Este retraso es muy corto así como su modulación. Ambos crean el efecto de coro que logra a su vez un efecto de engrandecimiento en el sonido.

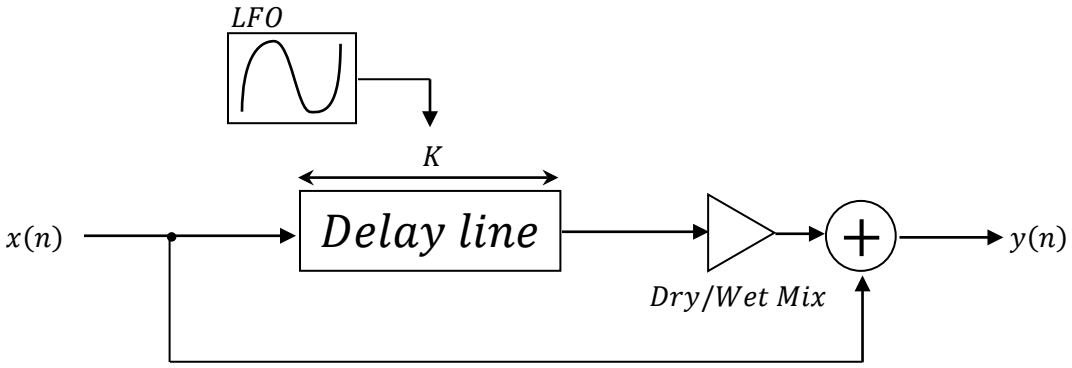


Figura 3.20: Diagrama de bloques del efecto *Chorus*.

Los parámetros del efecto *Chorus* son (ver **Figura 3.21**):

- **Frecuencia (*Rate*):** Es la velocidad con la cual se hace la variación de tono del *Chorus*. Generalmente la frecuencia del LFO oscila entre 0 Hz. y 2 Hz.

- **Profundidad (*Depth*):** Este parámetro modifica el tiempo total de retraso que varía durante el tiempo.

- **Proporción de Mezcla (*Mix*):** Es la cantidad de efecto *Chorus* que se le aplica a la señal original.

Figura 3.21: Interfaz gráfica del efecto *Chorus*.

El **Algoritmo 6** corresponde a la versión secuencial para CPU del efecto *Chorus*, que es muy similar al **Algoritmo 5** del efecto *Vibrato*, ya que es un efecto derivado de este. La diferencia con respecto al efecto *Vibrato* es el rango de las oscilaciones por segundo que tiene el efecto *Chorus* y la suma de la señal original con la señal procesada en la mezcla final. Para la implementación en CUDA se elimina el ciclo **para**, ya que toda la función tiene que ser ejecutada de forma paralela por cada hilo asignado al *kernel* de CUDA de este efecto.

Algoritmo 6 Algoritmo del efecto *Chorus*

```
1: función Chorus(array in, array out, array oldIn, int N, int counter,
2: int sampleRate, float rate, float depth, float mix)
3:
4:   float TWOPI ← 6,283185307
5:   float modifier, index, frac, next
6:   int indexInt
7:
8:   rate ← rate · 2
9:
10:  float t ← (TWOPI · rate) / sampleRate
11:
12:  para int i ← 0 hasta i < N hacer
13:
14:    modifier ← depth · sin(t · (N · counter + i))
15:    index ← i + (modifier · N/2 + N/2)
16:    indexInt ← index
17:    frac ← index – indexInt
18:    next ← oldIn[i+1]
```

```
19:
20:       out[i] ← (1 - mix) · in[i] + mix · (in[i] + oldIn[i] + frac · (next – oldIn[i]))
21:
22:    fin para
23: fin función
24:
25: counter ← counter + 1
```

En el **Algoritmo 6**, línea 8, se encuentra la variable **rate** que es multiplicada por 2, ya que el número de oscilaciones del modificador de este efecto es de máximo 2 Hz. La variable **t** indica la proporción de cambio de fase de la onda senoidal que se va a generar para manipular el efecto. En la línea 14 en la variable **modifier** se calcula el valor de la onda seno que se va a utilizar para recorrer el índice del arreglo **oldIn**, el cual es un arreglo de tamaño $2 \cdot N$ que contiene el bloque de audio anterior y el bloque de audio actual. Como el índice de la onda seno tiene que estar dentro del rango del tamaño del arreglo **oldIn**, en la línea 15 se calcula en la variable **index** un valor que este dentro de este rango. En la línea 16 se calcula la parte entera del índice, y en la siguiente línea el resto que queda de la variable **index**. Se calcula la siguiente posición en el índice del arreglo **oldIn** que se utilizará para calcular una interpolación lineal entre la posición actual del índice y la siguiente posición. En la línea 20 se calcula la proporción de mezcla de la señal original con respecto a la suma de la señal original y la señal modificada por el efecto. Finalmente al terminar la función en la línea 25 se suma 1 a la variable **counter**, con el propósito de poder seguir la forma de onda del modificador para el siguiente bloque de audio a procesar.

3.4.6. Ring Modulator

El efecto de *Ring Modulator* consiste en multiplicar en el dominio del tiempo la señal de entrada con una señal portadora como por ejemplo una señal sinusoidal (ver **Figura 3.22**). El principio de esto radica en que la multiplicación de cualquier señal por otra señal genera una suma y diferencia de frecuencias que modifican la señal final.

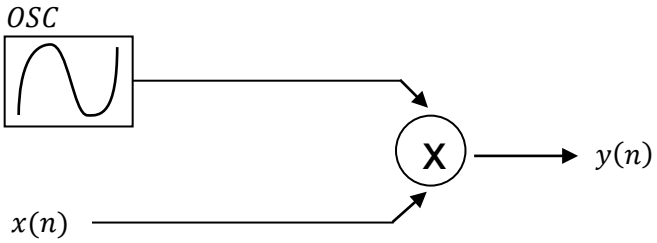


Figura 3.22: Diagrama de bloques del efecto *Ring Modulator*.

Los parámetros generales de este efecto son (ver **Figura 3.23**):

- **Frecuencia (*Rate*):** Es la velocidad con la cual se repetirán los ciclos de la señal sinusoidal portadora (de 20 Hz a 4 KHz).

- **Proporción de Mezcla (*Mix*):** Se define como la intensidad del efecto que se le aplica a la señal original.

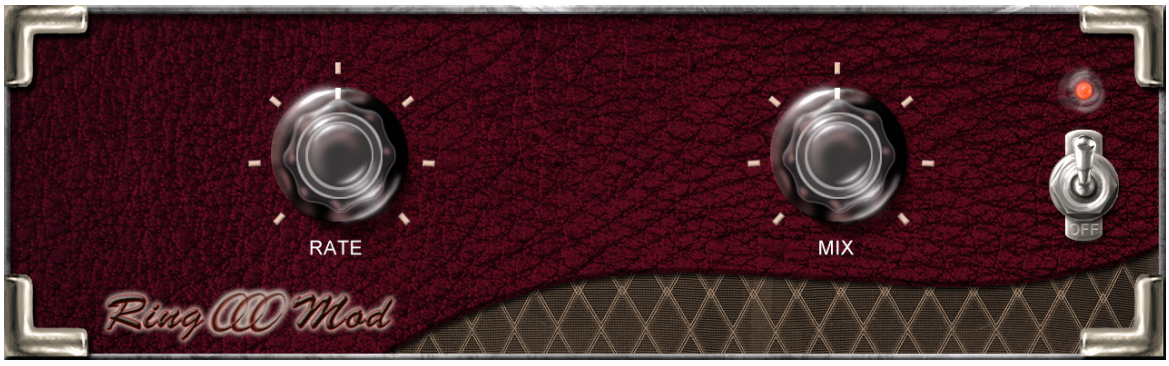


Figura 3.23: Interfaz gráfica del efecto *Ring Modulator*.

El **Algoritmo 7** corresponde a la versión secuencial para CPU del efecto *Ring Modulator*. Para la implementación en CUDA se elimina el ciclo **para**, ya que todo el código de la función tiene que ser ejecutado de forma paralela por cada hilo asignado al *kernel* de CUDA de este efecto.

Algoritmo 7 Algoritmo del efecto *Ring Modulator*

```
 1: función RingMod(array in, array out, int N, int counter, int sampleRate,
 2: float rate, float mix)
 3:
 4:   float TWOPI ← 6,283185307
 5:   float modifier
 6:
 7:   rate ← (rate · 3980) + 20
 8:
 9:   float t ← (TWOPI · rate) / sampleRate
10:
11:   para int i ← 0 hasta i < N hacer
12:
13:     modifier ← sin(t · (N · counter + i))
14:
15:     out[i] ← (1 - mix) · in[i] + mix · (in[i] · modifier)
16:
17:   fin para
18: fin función
19:
20: counter ← counter + 1
```

En el **Algoritmo 7**, línea 7, se encuentra la variable **rate** que está definida para el rango entre 20 Hz y 4000 Hz. La variable **t** indica la proporción de cambio de fase de la onda senoidal que se va a generar para manipular el efecto. En la línea 13 se encuentra la variable **modifier** en la cual se calcula el valor de la onda seno que se utiliza para multiplicar la muestra actual de la señal original. En la línea 15 se calcula la proporción de mezcla de la señal original con respecto a la señal modificada por el efecto. Finalmente al terminar la función en la línea 20 se suma 1 a la variable **counter**, con el propósito de poder seguir la forma de onda del modificador para el siguiente bloque de audio a procesar.

3.4.7. Tremolo

Este efecto consiste en enviar la señal a los altavoces a una frecuencia determinada. El algoritmo utilizado consiste en implementar en el programa una sinusoide de amplitud unitaria y de una determinada frecuencia. Luego se detecta si el valor del seno es positivo o negativo. Cuando es positivo se enviará al parlante el audio multiplicado por los valores positivos de la sinusoide y cuando es negativo simplemente se inhibirá el envío de señal (ver **Figura 3.24**). La cantidad de efecto aplicado a la señal es controlado por la Profundidad y la velocidad de las oscilaciones (Frecuencia) será determinada por los ciclos del LFO.

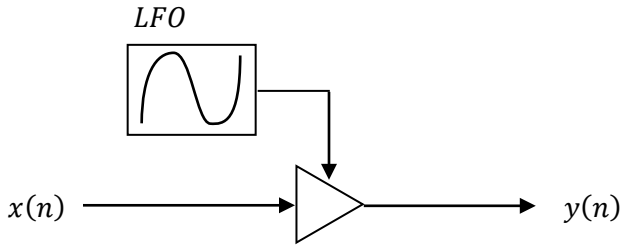


Figura 3.24: Diagrama de bloques del efecto *Tremolo*.

Los parámetros de este efecto son (ver **Figura 3.25**):

- **Frecuencia (*Rate*):** Es la velocidad con la cual se hace la variación de volumen en los altavoces (generalmente entre 0,1 Hz y 10 Hz.).

- **Profundidad (*Depth*):** Se define como la intensidad de cambio de volumen del efecto *Tremolo*.

- **Proporción de Mezcla (*Mix*):** Es la cantidad de efecto *Tremolo* que se mezcla con la señal original.

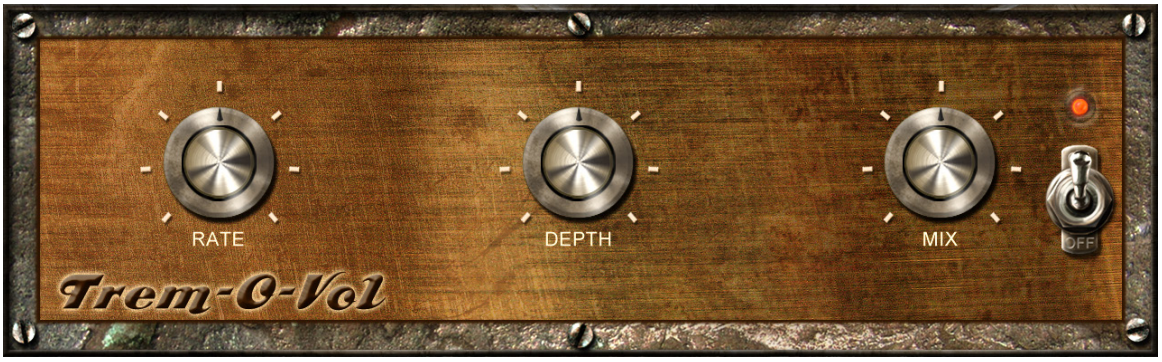


Figura 3.25: Interfaz gráfica del efecto *Tremolo*.

El **Algoritmo 8** corresponde a la versión secuencial para CPU del efecto *Tremolo*. Para la implementación en CUDA se elimina el ciclo **para**, ya que todo el código de la función tiene que ser ejecutado de forma paralela por cada hilo asignado al *kernel* de CUDA de este efecto.

Algoritmo 8 Algoritmo del efecto *Tremolo*

```
1: función Tremolo(array in, array out, int N, int counter,int sampleRate,
2: float rate, float depth, float mix)
3:
4:    float TWOPI ← 6,283185307
5:    float modifier
6:
7:    rate ← (rate · 9,9) + 0,1
8:    depth ← (depth · 9) + 1
9:
10:   float t ← rate / sampleRate
11:
12:   para int i ← 0 hasta i < N hacer
13:
14:      modifier ← 0,5 · tanh(depth · sin(TWOPI · t · (N · counter + i))) + 0,5
15:
16:      out[i] ← (1 - mix) · in[i] + mix · (in[i] · modifier)
17:
18:   fin para
19: fin función
20:
21: counter ← counter + 1
```

En el **Algoritmo 8**, línea 7, se encuentra la variable **rate** que está definida para el rango entre 0,1 Hz y 10 Hz. En la línea 8 en la variable **depth** se define la curvatura de la señal senoidal en un rango de 1 a 10, mientras mayor sea el valor, la onda senoidal tendrá una forma más cuadrada (ver **Figura 3.26**). La variable **t** indica la proporción de cambio de fase de la onda senoidal que se va a generar para manipular el efecto. En la línea 14 en la variable **modifier** se calcula el valor de la onda seno que se utiliza para multiplicar con la muestra actual de la señal original. En la línea 16 se calcula la proporción de mezcla de la señal original con respecto a la señal modificada por el efecto. Finalmente al terminar la función en la línea 21 se suma 1 a la variable **counter**, con el propósito de poder seguir la forma de onda del modificador para el siguiente bloque de audio a procesar.

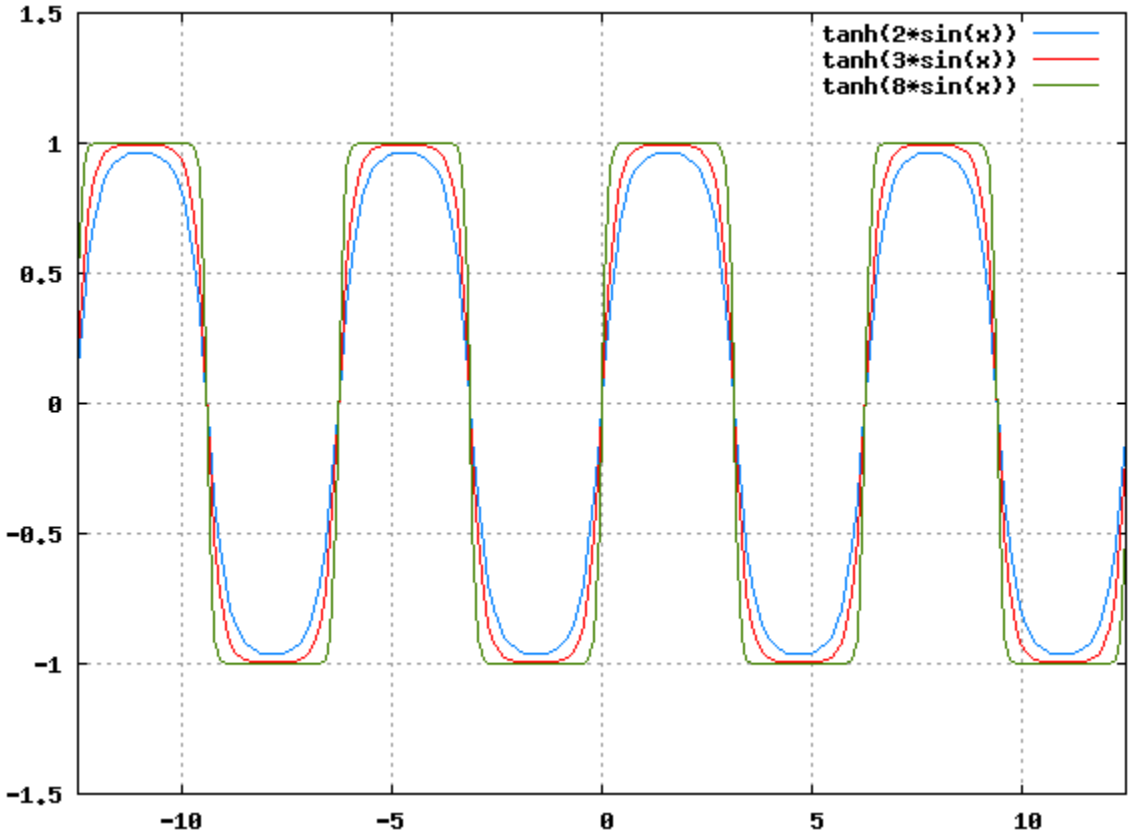


Figura 3.26: Una onda sinusoidal modificada por la función *tanh(x)*. A medida que el multiplicador dentro de la función sea mayor, la onda sinusoidal tendrá una tendencia a ser cada vez más cuadrada.

3.4.8. Auto Panner

El algoritmo consiste en implementar una sinusoide de amplitud unitaria y frecuencia determinada, generalmente un LFO (ver **Figura 3.27**). Si se detecta que el valor actual del seno es positivo entonces se envía al parlante derecho la señal de audio multiplicada por el valor positivo del seno, en caso contrario, si se detecta que el valor del seno es negativo se enviará al parlante izquierdo la señal de audio multiplicada por el valor actual del seno. En resumen, mientras un parlante emite señal el otro no emite señal alguna, luego se invierten los estados. Esto se repite en un bucle infinito hasta que el efecto sea desactivado por el usuario.

La amplitud del LFO permite determinar las posiciones mínimas y máximas de la oscilación. La frecuencia del LFO determinará el período de oscilación.

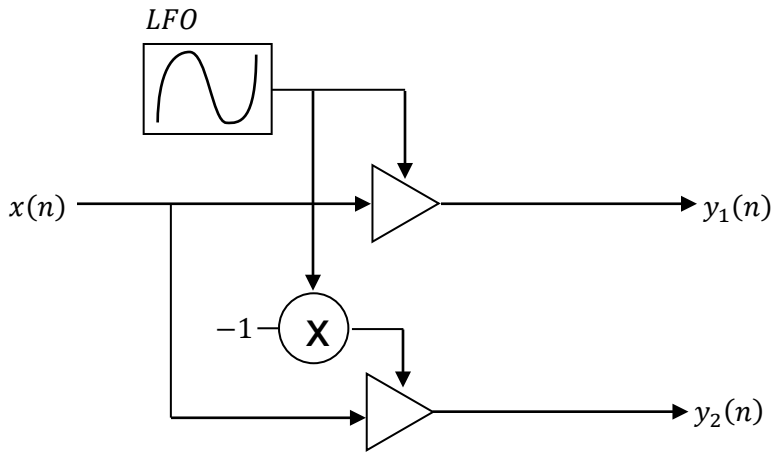


Figura 3.27: Diagrama de bloques del efecto *Auto Panner*.

Los parámetros de este efecto son (ver **Figura 3.28**):

- **Frecuencia (*Rate*):** Es la velocidad con la cual se hace la variación de volumen de izquierda a derecha entre los altavoces (generalmente entre 0,1 Hz. y 5 Hz.).

- **Profundidad (*Depth*):** Se define como la intensidad de cambio de volumen del efecto *Auto Panner*.

- **Proporción de Mezcla (*Mix*):** Es la cantidad de efecto que se mezcla con la señal original.

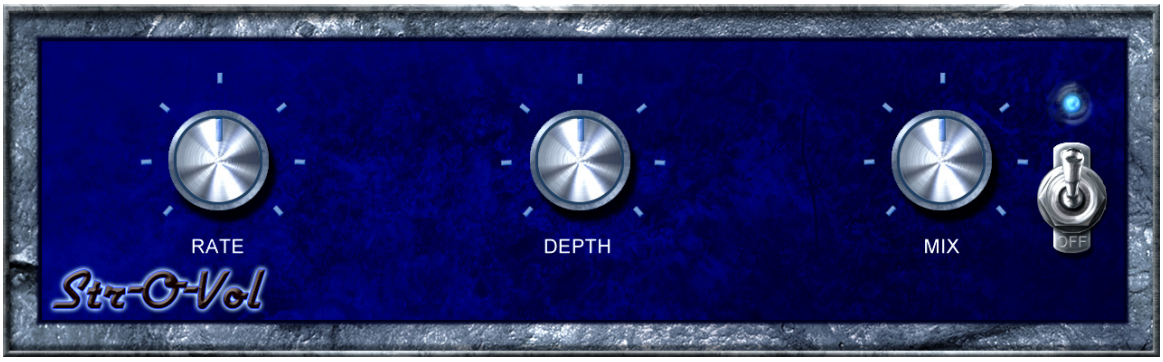


Figura 3.28: Interfaz gráfica del efecto *Auto Panner*.

El **Algoritmo 9** corresponde a la versión secuencial para CPU del efecto *Auto Panner*. Este efecto es muy similar al **Algoritmo 8** del efecto *Tremolo* ya que es un efecto derivado de este. La diferencia con respecto al efecto *Tremolo* es el rango de las oscilaciones por segundo que tiene el efecto *Auto Panner* y que se utilizan 2 canales de audio ya que es un efecto estéreo. Para la implementación en CUDA se elimina el ciclo **para**, ya que todo el código de la función tiene que ser ejecutado de forma paralela por cada hilo asignado al *kernel* de CUDA de este efecto.

Algoritmo 9 Algoritmo del efecto *Auto Panner*

```
1: función AutoPannner(array inL, array inR, array outL, array outR, int N,
2: int counter, int sampleRate, float rate, float depth, float mix)
3:
4:   float TWOPI ← 6,283185307
5:   float modifier
6:
7:   rate ← (rate · 4,9) + 0,1
8:   depth ← (depth · 9) + 1
9:
10:  float t ← rate / sampleRate
11:
12:  para int i ← 0 hasta i < N hacer
13:
14:    modifier ← 0,5 · tanh(depth · sin(TWOPI · t · (N · counter + i)))
15:
16:    outL[i] ← (1 - mix) · inL[i] + mix · (inL[i] · (modifier + 0,5))
17:    outR[i] ← (1 - mix) · inR[i] + mix · (inR[i] · (-modifier + 0,5))
```

```
18:
19:    fin para
20: fin función
21:
22: counter ← counter + 1
```

En el **Algoritmo 9**, línea 7, se encuentra la variable **rate** que está definida para el rango entre 0,1 Hz y 5 Hz. En la línea 8 en la variable **depth** se define la curvatura de la señal senoidal en un rango de 1 a 10; mientras mayor sea el valor, la onda senoidal tendrá una forma más cuadrada (ver **Figura 3.26**). La variable **t** indica la proporción de cambio de fase de la onda senoidal que se va a generar para manipular el efecto. En la línea 14, en la variable **modifier**, se calcula el valor de la onda seno que se utiliza para multiplicar la muestra actual de la señal original. En las líneas 16 y 17 se calcula la proporción de mezcla de la señal original con respecto a la señal modificada por el efecto; en la salida de audio izquierda **outL** se multiplica la señal por la variable **modifier** y en la salida de audio derecha se multiplica la señal por **-modifier** para tener la respuesta contraria a la salida izquierda y generar el efecto buscado. Finalmente al terminar la función en la línea 22 se suma 1 a la variable **counter**, con el propósito de poder seguir la forma de onda del modificador para el siguiente bloque de audio a procesar.

3.4.9. Delay

La estructura general para implementar el efecto *Delay* es descrita por la ecuación:

$$y(n) = x(n) + (y(n - K) * feedback) * dry/wet \qquad (\boldsymbol{Ec.\ 3.3})$$

donde K es el tamaño del retraso (medido en muestras), $feedback$ es un factor de amplificación aplicado a la señal retrasada y dry/wet es un factor de mezcla de la señal original con la señal retrasada. En la **Figura 3.29** se puede observar la implementación del efecto en forma de diagrama de bloques.

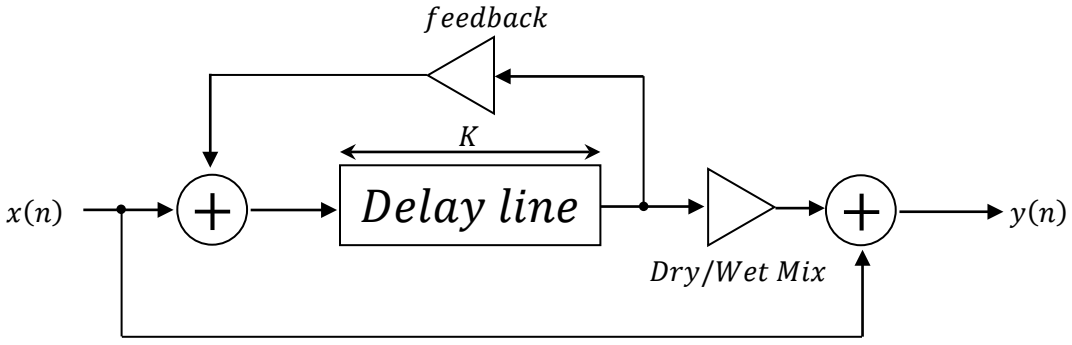


Figura 3.29: Diagrama de bloques del efecto *Delay*.

Los parámetros de un *delay* son (ver **Figura 3.30**):

- **Tiempo de retraso (*Time*):** Corresponde al tiempo que tarda en producirse un eco (K) y es ajustado en milisegundos.

- **Retroalimentación (*Feedback*):** Define la cantidad de veces que se repite la señal sonora.

- **Proporción de mezcla (*Dry/Wet*):** Es la cantidad de sonido retrasado que se mezcla con la señal original.

Figura 3.30: Interfaz gráfica del efecto *Delay*.

El **Algoritmo 10** corresponde a la versión secuencial para CPU del efecto *Delay* Para la implementación en CUDA se elimina el ciclo **para**, ya que todo el código de la función tiene que ser ejecutado de forma paralela por cada hilo asignado al *kernel* de CUDA de este efecto.

Algoritmo 10 Algoritmo del efecto *Delay*

```
1: función Delay(array in, array out, array delayBuffer, int N, int counter,
2: int sampleRate, float time, float feedback, float drywet)
3:
4:    para int i ← 0 hasta i < N hacer
5:
6:      cursor ← N · counter + i
7:
8:      delayBuffer[cursor] ← (delayBufferL[cursor] · feedback) + in[i]
9:
10:     in[i] ← (1 - drywet) · in[i] + (drywet · delayBuffer[cursor])
11:
12:   fin para
13: fin función
14:
15: counter ← counter + 1
16:
17: si counter · N > time · sampleRate entonces
18:    counter ← 0
19: fin si
```

En el **Algoritmo 10**, línea 6, se encuentra la variable **cursor** la cual va a contener el valor del índice con el cual se va a recorrer el arreglo **delayBuffer**, el cual es el arreglo donde se van a almacenar las muestras anteriores del búfer de entrada. En la línea 8 a cada posición del arreglo **delayBuffer** se le asigna un valor de la muestra actual más la muestra que tenga almacenada previamente por un factor entre 0 y 1 asignado a la variable **feedback** que hace decaer en volumen a las muestras anteriores almacenadas en el arreglo **delayBuffer** tras sucesivas iteraciones. En la línea 10 se calcula la proporción de mezcla de la señal original con respecto a la señal modificada por el efecto. Finalmente al terminar la función, en la línea 15, se suma 1 a la variable **counter**, con el propósito de poder tener un acumulador para la variable **cursor** del efecto *Delay*. En las líneas 17-19 se verifica si la variable **counter** multiplicada por **N** es mayor que el tiempo en número de muestras, de ser así, se reinicia el contador a cero.

En la siguiente sección se describe todo el proceso de las pruebas de rendimiento para diversos CPUs y GPUs, de las distintas plataformas de *hardware* seleccionadas.

Capítulo 4. Pruebas y Resultados

Una vez finalizada la etapa de diseño y desarrollo se puso a prueba el sistema para evaluar el rendimiento y eficiencia y así determinar si se alcanzan los objetivos planteados.

En este capítulo se presentarán las pruebas realizadas para el procesamiento digital de audio de los efectos anteriormente mencionados tanto en su implementación en GPU como en CPU.

4.1. Descripción del ambiente de pruebas

Para las pruebas se requirió de *hardware* en particular para poder ejecutar los algoritmos. A continuación se muestran las arquitecturas utilizadas:

Plataforma de Hardware 1 (Laptop ASUS M50VM-B2):

- Computador basado en procesador Intel Core 2 Duo T9400 2,53 GHz.
- 4 GB de memoria RAM DDR2 800 MHz.
- Tarjeta de video NVIDIA GeForce 9600M GS con 1024 MB de VRAM DDR2.

Plataforma de Hardware 2 (Laptop MacBookPro MB470xx/A):

- Computador basado en procesador Intel Core 2 Duo P8600 2,4 GHz.
- 4 GB de memoria RAM DDR3 1066 MHz.
- Tarjeta de video NVIDIA GeForce 9600M GT con 512 MB de VRAM GDDR3.

Plataforma de Hardware 3 (Desktop Personalizada):

- Computador basado en procesador Intel Core i7 920 2,66 GHz.
- 6 GB de memoria RAM DDR3 1066 MHz.
- Tarjeta de video NVIDIA GeForce 9800M GT con 1024 MB de VRAM GDDR3.

Plataforma de Hardware 4 (Desktop Personalizada):

- Computador basado en procesador Intel Core i3 540 3,06 GHz.
- 4 GB de memoria RAM DDR3 1066 MHz.
- Tarjeta de video NVIDIA GeForce GTX 470 con 1280 MB de VRAM GDDR5.

El sistema fue evaluado en la plataforma del Sistema Operativo *Microsoft Windows 7*. Se empleó el software DAW Cakewalk Sonar 8.5 que es compatible con el *plugin* VST para hacer las pruebas necesarias de su funcionamiento.

Para todas las pruebas se utilizó la tarjeta de sonido integrada de cada máquina con un software de emulación ASIO (especificación Audio Stream Input/Output) que habilita la grabación y reproducción de audio a baja latencia.

Para todas las pruebas se utilizó un tamaño de bloque de audio de 512 muestras, que es un arreglo de números en punto flotante de 32 bits a una tasa de muestreo de 44.100 muestras por segundo. Usando la **Ecuación 3.1** tenemos que:

$$\frac{1}{44100} \cdot 512 \cdot 1000 = 11{,}6\ ms.$$

Este tamaño de bloque es lo suficientemente aceptable para no percibir la latencia que genera el procesamiento de audio por bloques, así como la tasa de muestreo que es la utilizada de forma estándar para el audio contenido en un CD.

4.2. Medición de Tiempo

Para medir el tiempo en las pruebas de CPU se utilizó el contador de tiempo de alta resolución *QueryPerformanceCounter*, ya que el orden de las magnitudes a evaluar son en milisegundos y la función *clock()* o *GetTickCount()* de *Windows* no ofrecen la resolución adecuada para estos casos.

QueryPerformanceCounter se utiliza para medir el tiempo transcurrido de un fragmento de código tomando un tiempo inicial y final para luego calcular la resta entre estos dos tiempos y obtener el tiempo transcurrido del algoritmo. En la **Figura 4.1** se muestra la implementación del cálculo del tiempo en CPU.

```
__int64 ctr1, ctr2, freq;       // Variables para medir el tiempo
float resultado;                // Variable para guardar resultado final

QueryPerformanceCounter((LARGE_INTEGER *)&ctr1);    // Tiempo Inicial

// Sección de código a evaluar

QueryPerformanceCounter((LARGE_INTEGER *)&ctr2);    // Tiempo Final
QueryPerformanceFrequency((LARGE_INTEGER *)&freq); // Tics por segundo del CPU

resultado = ((ctr2-ctr1)/(float)freq * 1000.0f);   // Calculo final del tiempo

printf("Tiempo Total: %f (ms)", resultado);         // Despliegue por pantalla
                                                    // del resultado final en ms.
```

Figura 4.1: Fragmento de código que mide el tiempo de un algoritmo en CPU.

Por otro lado para la medición de tiempo en GPU usamos las funciones propias de CUDA:

Se utilizó *cudaEventRecord()* que graba un evento en CUDA. Dado que la operación es asíncrona la función *cudaEventSynchronize()* es usada para determinar cuándo un evento ha sido realmente grabado. En la **Figura 4.2** se muestra la implementación del cálculo del tiempo en GPU.

```
cudaEvent_t start, stop;        // Variables para medir el tiempo
float resultado;                // Variable para guardar resultado final

cudaEventCreate(&start);        // Creación de evento start para medir tiempo
cudaEventCreate(&stop);         // Creación de evento stop para medir tiempo

cudaEventRecord(start, 0);            // Tiempo Inicial

// Sección de código a evaluar

cudaEventRecord(stop, 0);       // Tiempo Final
cudaEventSynchronize(stop);     // Parada de la toma de muestra del tiempo

cudaEventElapsedTime(&resultado, start, stop);     // Calculo final del tiempo

printf("Tiempo Total: %f (ms)", resultado);        // Despliegue por pantalla
                                                   // del resultado final en ms.
```

Figura 4.2: Fragmento de código que mide el tiempo de un algoritmo en GPU.

Se hizo la toma de tiempo de los algoritmos en CPU a las funciones que calculan las operaciones aritméticas del procesamiento digital del audio. En GPU se toma el tiempo en dos partes; un cálculo del tiempo tomado en procesar el o los *kernels* de CUDA y un tiempo para la sincronización con el GPU.

4.3. Resultados

Para las pruebas de rendimiento los algoritmos se ejecutaron durante 20 segundos aproximadamente. Como se estima que cada algoritmo se ejecuta unas 100 veces por segundo, se estima que para cada prueba cada algoritmo se ejecutó unas 2.000 veces.

Las primeras cuatro columnas de las gráficas representan los tiempos de ejecución de cada CPU, las otras cuatro columnas representan el tiempo de ejecución y sincronización de cada GPU de las plataformas de hardware anteriormente definidas.

4.3.1. Over Drive

La gráfica de la **Figura 4.3** muestra los tiempos de ejecución de la implementación del efecto de audio *Over Drive*. En la implementación para CPU se puede apreciar que el procesador Intel Core i7 2,66 GHz es aproximadamente 13 veces más rápido el GPU GeForce 9600M GS. Los tiempos de sincronización penalizan duramente el rápido tiempo de cálculo ofrecido por el GPU. La implementación de CPU ejecutada en el procesador Intel Core i3 3,06 GHz es 2,9 veces más rápida que la tarjeta de video GeForce GTX 470. Se puede notar un retardo de tiempo del GPU GeForce 9600M GS debido a que la sincronización es mucho más lenta que la del resto de los GPUs.

Para este efecto, el procesador Intel Core i7 2,66 GHz es más rápido que el resto de los GPUs y CPUs.

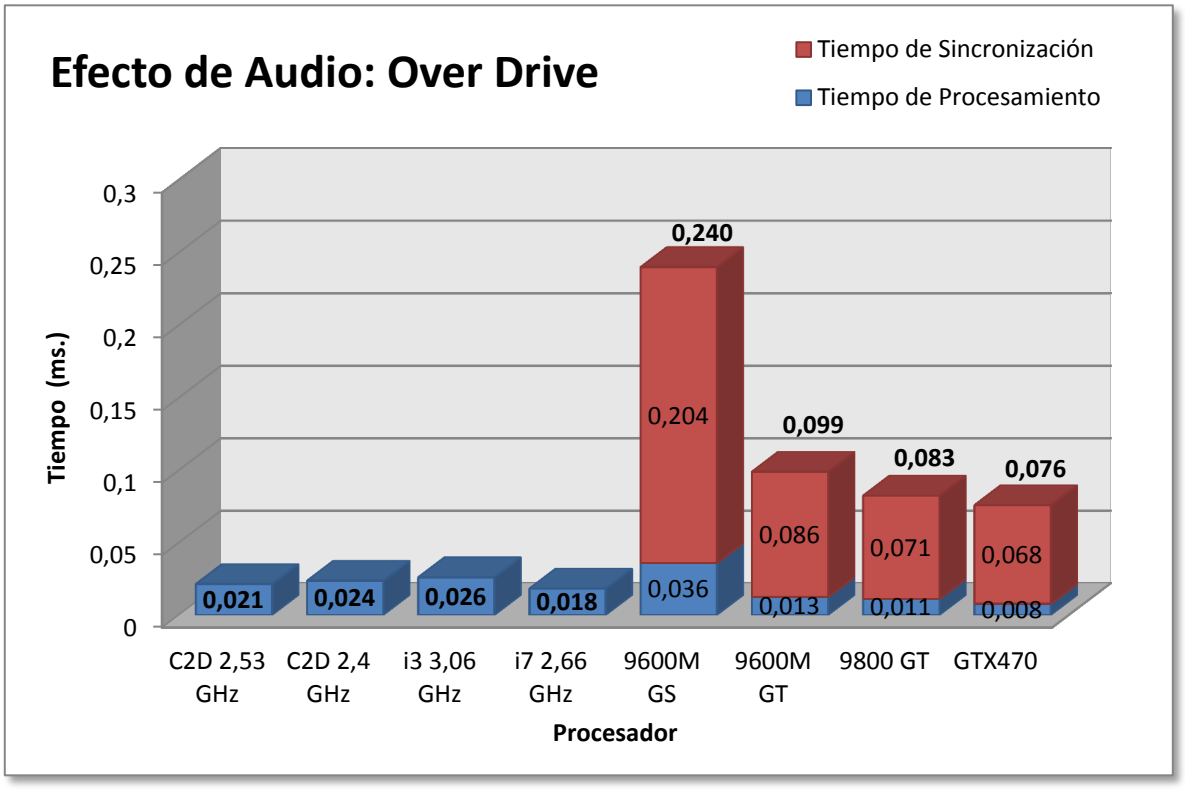


Figura 4.3: Comparación de rendimiento en tiempo para las implementaciones en CPU y GPU del efecto *Over Drive*.

4.3.2. Distortion

La gráfica de la **Figura 4.4** muestra los tiempos de ejecución de la implementación del efecto de audio *Distortion* en varias arquitecturas. En la implementación para CPU se puede apreciar que el procesador Intel Core i7 2,66 GHz es aproximadamente 1,4 veces más rápido que el CPU Intel Core i3 3,06 GHz y 2,6 veces más rápido el GPU GeForce 9600M GS. Sin embargo la implementación de GPU ejecutada en la tarjeta de video GeForce GTX 470 es más rápida que el resto de los CPUs y GPUs. Esto es debido a que la velocidad de cálculo de la función de transferencia $tanh(x)$ es realizada de forma más eficiente en GPU que en CPU, incluso tomando en cuenta el tiempo de sincronización con el GPU. Se puede notar una gran diferencia de tiempo entre el resto de los GPUs y el GPU GeForce 9600M GS debido a que su tiempo de sincronización es mucho más lento.

Para este efecto, los GPUs GeForce GTX 470, 9800GT son más rápidos que todos los CPUs evaluados.

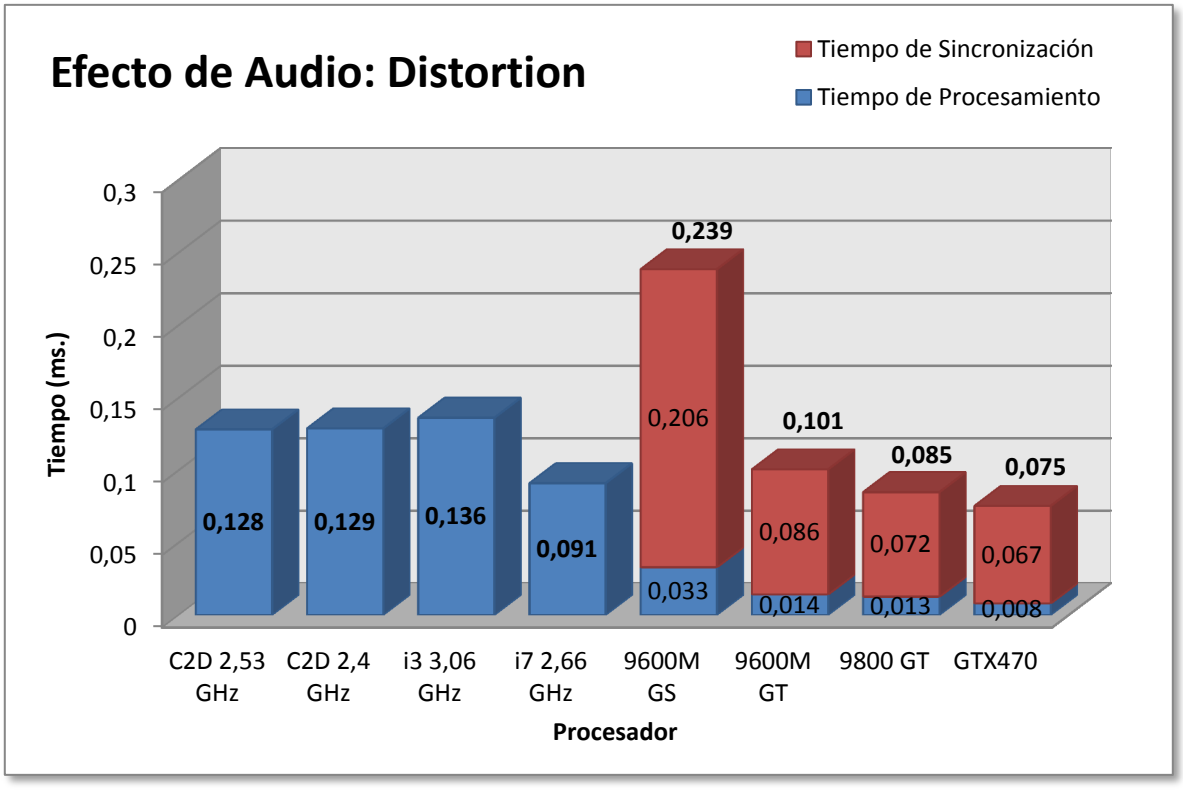


Figura 4.4: Comparación de rendimiento en tiempo para las implementaciones en CPU y GPU del efecto *Distortion*.

4.3.3. Ecualizador

La gráfica de la **Figura 4.5** muestra los tiempos de ejecución de la implementación del efecto *Ecualizador* en varias arquitecturas. En la implementación para CPU se puede apreciar que el procesador Intel Core i7 2,66 GHz es aproximadamente 1,4 veces más rápido que el CPU Intel Core 2 Duo 2,4 GHz y 1,9 veces más rápido que el GPU GeForce 9600M GS. Sin embargo la tarjeta de video GeForce GTX 470 es 2,5 veces más rápida que el CPU Intel Core i7 2,66 GHz y 3,6 veces mas rápida que el procesador Intel Core 2 Duo 2,4 GHz. Cabe acotar que la tarjeta de video GeForce GTX 470 es más rápida que el resto de los CPUs y GPUs, esto es debido a que la velocidad de cálculo de la Transformada de Fourier es mucho más eficiente en GPU que en CPU. Se puede notar una demora de tiempo del GPU GeForce 9600M GS con respecto a los otros GPUs debido a que su tiempo de sincronización es mucho más lento.

Para este efecto, la tarjeta de video GeForce GTX 470 es más rápida que el resto de los GPUs y CPUs.

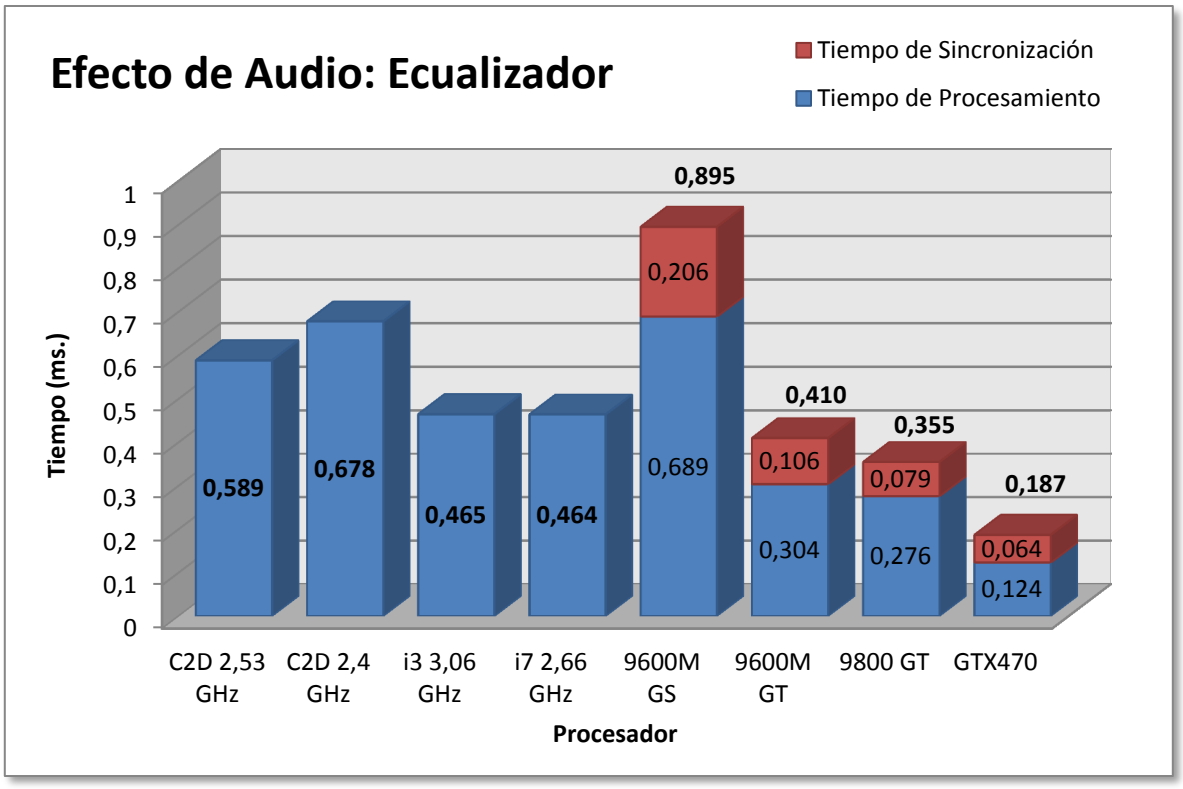


Figura 4.5: Comparación de rendimiento en tiempo para las implementaciones en CPU y GPU del *Ecualizador*.

4.3.4. Vibrato

La gráfica de la **Figura 4.6** muestra los tiempos de ejecución de la implementación del efecto de audio *Vibrato* en varias arquitecturas. En la implementación para CPU se puede apreciar que el procesador Intel Core i7 2,66 GHz es aproximadamente 1,4 veces más rápido que el CPU Intel Core i3 3,06 GHz y 7,1 veces más rápido que el GPU GeForce 9600M GS. Los tiempos de sincronización penalizan duramente el rápido tiempo de cálculo ofrecido por el GPU. Sin embargo la implementación de CPU ejecutada en el procesador Intel Core i3 3,06 GHz es solo 1,4 veces más rápida que la tarjeta de video GeForce GTX 470. Se puede notar un retardo de tiempo del GPU GeForce 9600M GS debido a que su sincronización es mucho más lenta que la del resto de los GPUs.

Para este efecto, los CPUs son más rápidos que todos los GPUs, sin embargo, el procesador Intel Core i3 3,06 GHz es solamente un poco más rápido que la tarjeta de video GeForce GTX 470.

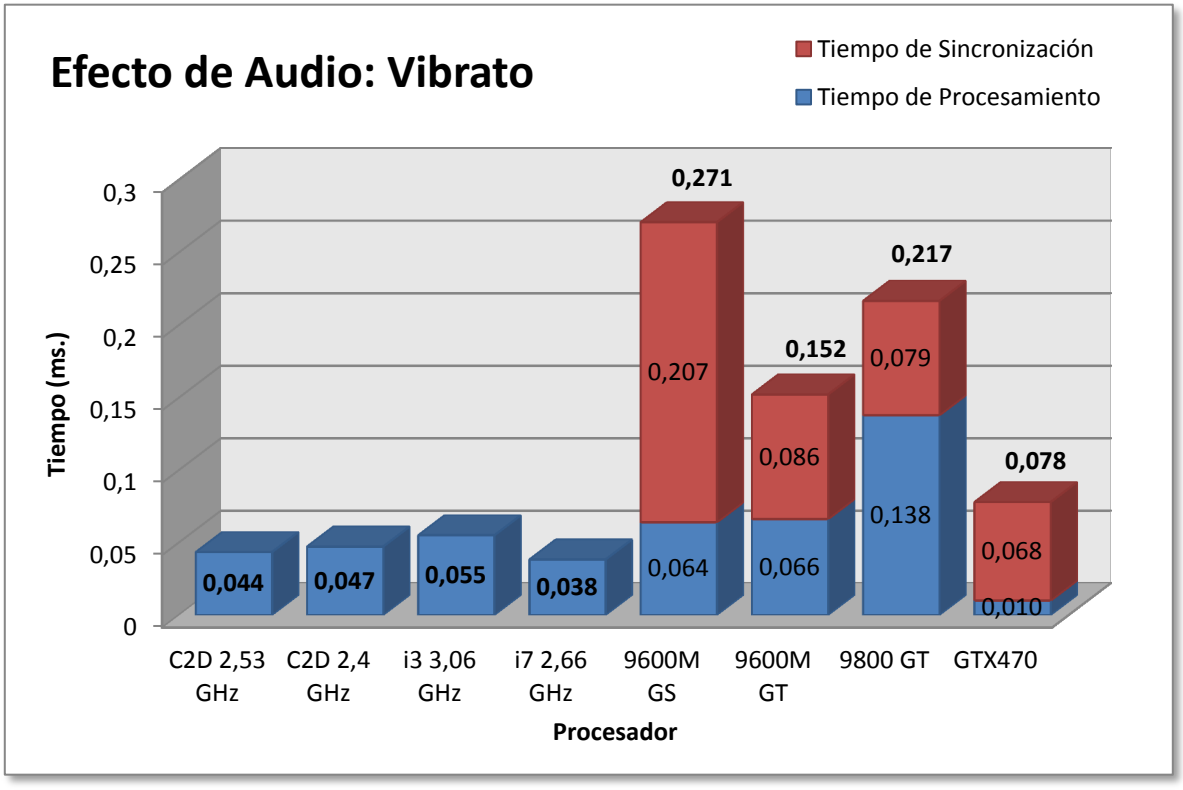


Figura 4.6: Comparación de rendimiento en tiempo para las implementaciones en CPU y GPU del efecto *Vibrato*.

4.3.5. Chorus

La gráfica de la **Figura 4.7** muestra los tiempos de ejecución de la implementación del efecto de audio *Chorus* en varias arquitecturas. Los resultados son muy similares al del efecto *Vibrato*, ya que el *Chorus* es un derivado de este efecto. En la implementación para CPU se puede apreciar que el procesador Intel Core i7 2,66 GHz es aproximadamente 1,4 veces más rápido que el CPU Intel Core i3 3,06 GHz y 6,7 veces más rápido que el GPU GeForce 9600M GS. Los tiempos de sincronización penalizan duramente el rápido tiempo de cálculo ofrecido por el GPU. Sin embargo la implementación de CPU ejecutada en el procesador Intel Core i3 3,06 GHz es solo 1,3 veces más rápida que la tarjeta de video GeForce GTX 470. Se puede notar un retardo de tiempo del GPU GeForce 9600M GS debido a que su sincronización es mucho más lenta que la del resto de los GPUs.

Para este efecto, los CPUs son más rápidos que todos los GPUs, sin embargo, el procesador Intel Core i3 3,06 GHz es solamente un poco más rápido que la tarjeta de video GeForce GTX 470.

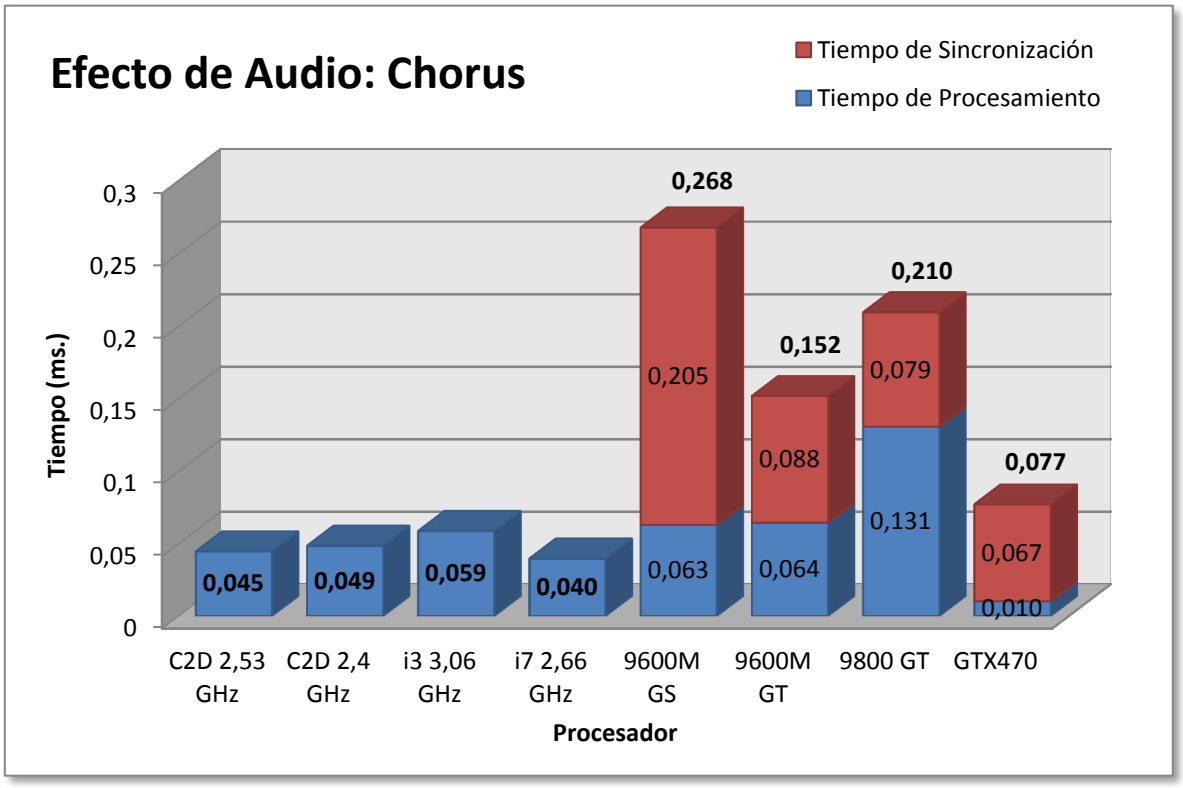


Figura 4.7: Comparación de rendimiento en tiempo para las implementaciones en CPU y GPU del efecto *Chorus*.

4.3.6. Ring Modulator

La gráfica de la **Figura 4.8** muestra los tiempos de ejecución de la implementación del efecto de audio *Ring Modulator* en varias arquitecturas. En la implementación para CPU se puede apreciar que el procesador Intel Core i7 2,66 GHz es aproximadamente 1,4 veces más rápido que el CPU Intel Core i3 3,06 GHz y 5 veces más rápido que el GPU GeForce 9600M GS. Los tiempos de sincronización penalizan duramente el rápido tiempo de cálculo ofrecido por el GPU. Sin embargo el procesador Intel Core i3 3,06 GHz es apenas 1,1 veces mas rápido que la tarjeta de video GeForce GTX 470. Se puede notar un retardo de tiempo del GPU GeForce 9600M GS debido a que su sincronización es mucho más lenta que la del resto de los GPUs.

Para este efecto, los CPUs son más rápidos que todos los GPUs, sin embargo, el procesador Intel Core i3 3,06 GHz es solamente un poco más rápido que la tarjeta de video GeForce GTX 470.

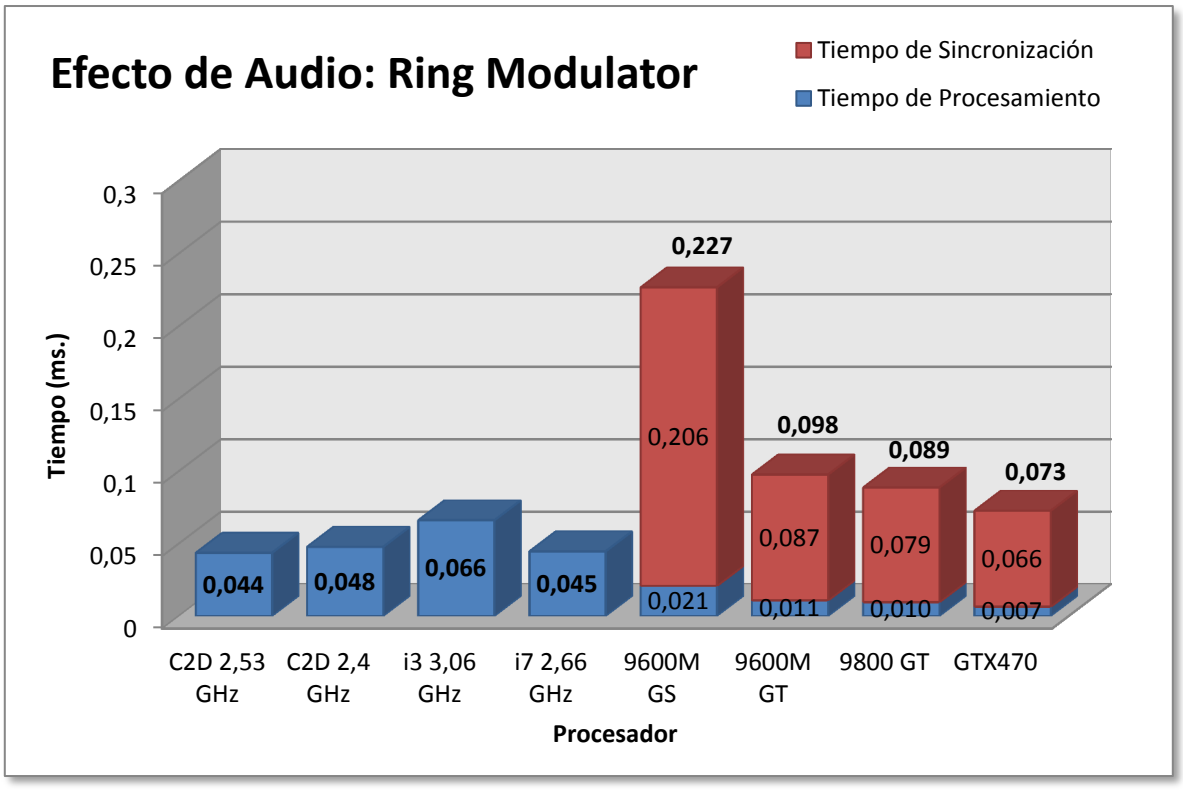


Figura 4.8: Comparación de rendimiento en tiempo para las implementaciones en CPU y GPU del efecto *Ring Modulator*.

4.3.7. Tremolo

La gráfica de la **Figura 4.9** muestra los tiempos de ejecución de la implementación del efecto de audio *Tremolo* en varias arquitecturas. En la implementación para CPU se puede apreciar que el procesador Intel Core i7 2,66 GHz es aproximadamente 1,4 veces más rápido que el CPU Intel Core i3 3,06 GHz y 3 veces más rápido que el GPU GeForce 9600M GS. Los tiempos de sincronización penalizan duramente el rápido tiempo de cálculo ofrecido por el GPU. Sin embargo la implementación de GPU ejecutada en el resto de las tarjetas de video es ligeramente más rápida que la ejecutada en el resto de los CPUs. Se puede notar un retardo de tiempo del GPU GeForce 9600M GS debido a que su sincronización es mucho más lenta que la del resto de los GPUs.

Para este efecto, los GPUs y CPUs tienen un tiempo de procesamiento muy similar entre ellos, excepto el GPU de la tarjeta de video GeForce 9600M GS.

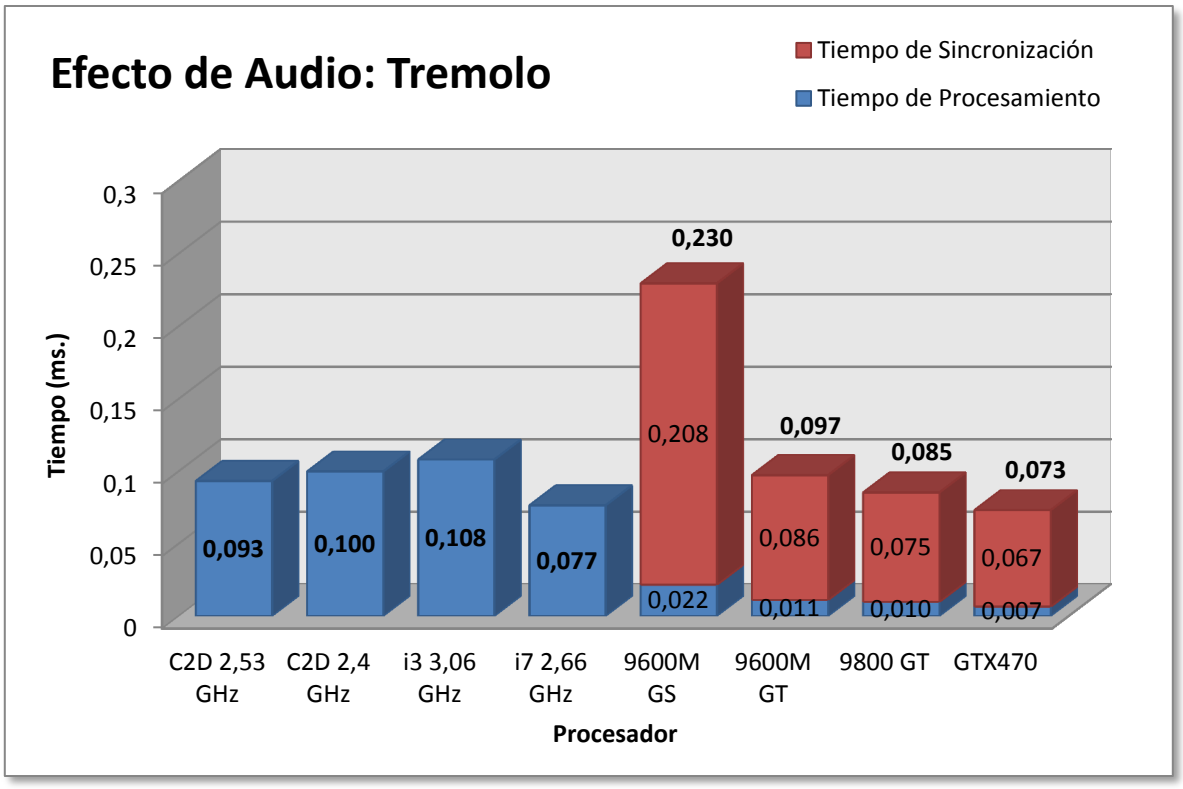


Figura 4.9: Comparación de rendimiento en tiempo para las implementaciones en CPU y GPU del efecto *Tremolo*.

4.3.8. Auto Panner

La gráfica de la **Figura 4.10** muestra los tiempos de ejecución de la implementación del efecto de audio *Auto Panner* en varias arquitecturas. Los resultados son muy similares al del efecto *Tremolo*, ya que el *Auto Panner* es un derivado de este efecto. En la implementación para CPU se puede apreciar que el procesador Intel Core i7 2,66 GHz es aproximadamente 1,4 veces más rápido que el CPU Intel Core i3 3,06 GHz y 3 veces más rápido que el GPU GeForce 9600M GS. Los tiempos de sincronización penalizan duramente el rápido tiempo de cálculo ofrecido por el GPU. Sin embargo la implementación de GPU ejecutada en el resto de las tarjetas de video es ligeramente más rápida que la ejecutada en el resto de los CPUs. Se puede notar un retardo de tiempo del GPU GeForce 9600M GS debido a que su sincronización es mucho más lenta que la del resto de los GPUs.

Para este efecto, los GPUs y CPUs tienen un tiempo de procesamiento muy similar entre ellos, excepto el GPU de la tarjeta de video GeForce 9600M GS.

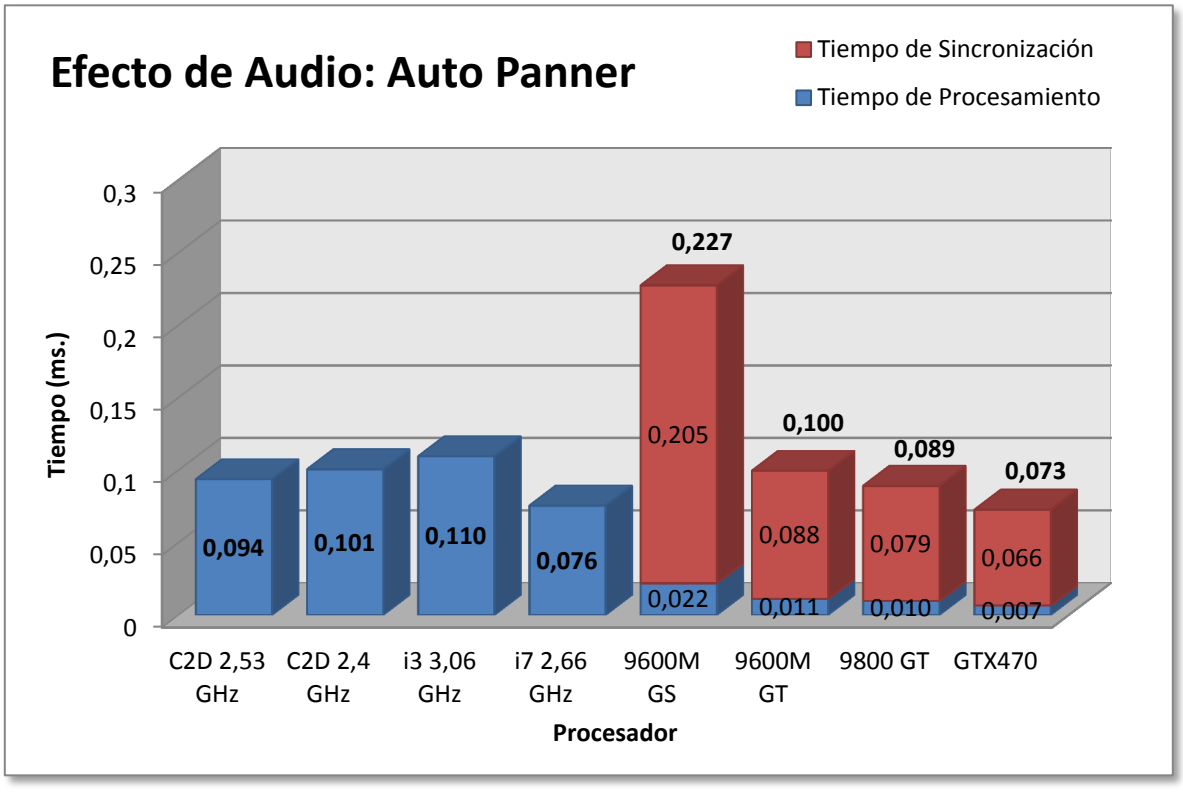


Figura 4.10: Comparación de rendimiento en tiempo para las implementaciones en CPU y GPU del efecto *Auto Panner*.

4.3.9. Delay

La gráfica de la **Figura 4.11** muestra los tiempos de ejecución de la implementación del efecto de audio *Delay* en varias arquitecturas. En la implementación para CPU se puede apreciar que el procesador Intel Core i7 2,66 GHz es 1,6 veces más rápido que el CPU Intel Core 2 Duo 2,4 GHz y 46 veces más rápido el GPU GeForce 9600M GS. La diferencia entre los resultados de las ejecuciones en CPU y en GPU son arrasantes. No hay lugar a duda de que la implementación para GPU es la menos indicada para conseguir un buen rendimiento en este efecto. Se puede notar un retardo de tiempo del GPU GeForce 9600M GS debido a que su sincronización es mucho más lenta que la del resto de los GPUs.

Para este efecto, todos los CPUs son más rápidos que los GPUs aquí evaluados.

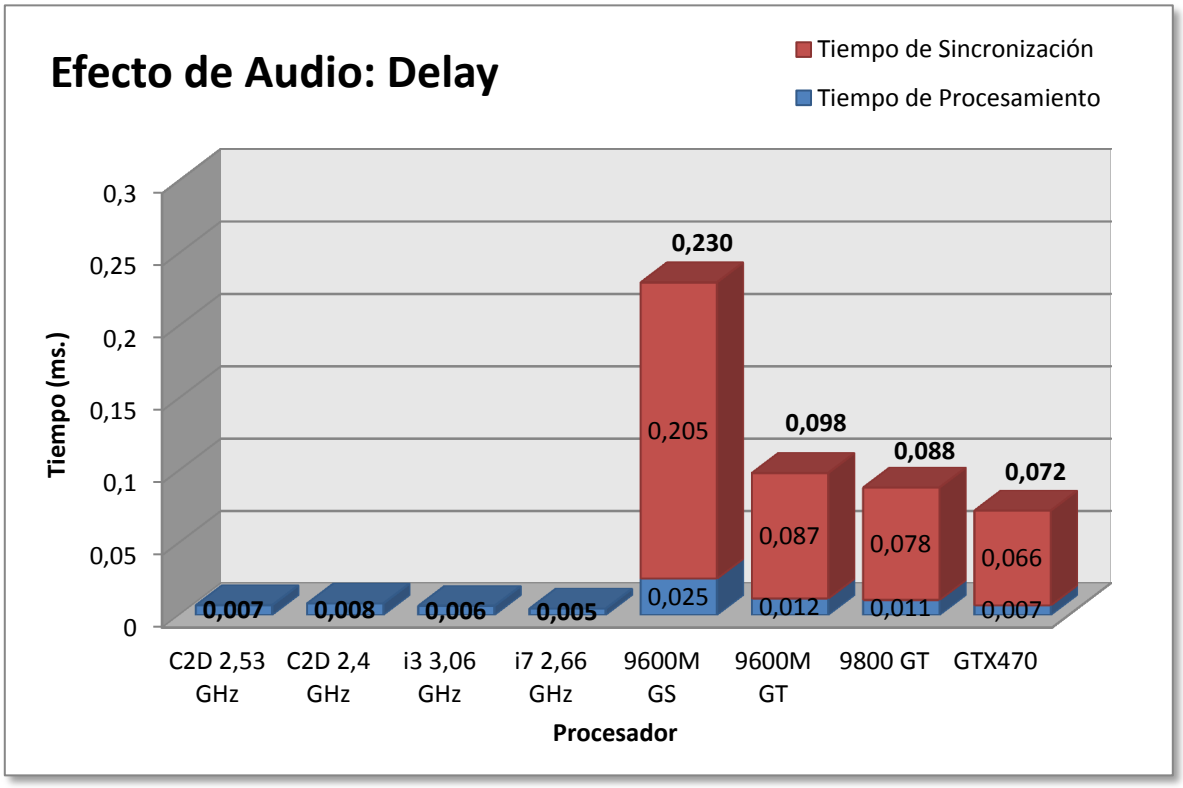


Figura 4.11: Comparación de rendimiento en tiempo para las implementaciones en CPU y GPU del efecto *Delay*.

4.3.10. Tiempo de ejecución promedio de un solo efecto

La gráfica de la **Figura 4.12** muestra el tiempo promedio de ejecución de un solo efecto en varias arquitecturas. Se puede notar que el procesador Intel Core i7 2,66 GHz es mas rápido que el resto de los CPUs. Sin embargo, el GPU GeForce GTX 470 es ligeramente más rápido que el CPU Intel Core i7 2,66 GHz. Esto es debido a que este GPU tiene una gran ventaja en el efecto *Ecualizador* realizando el cálculo de la Transformada Rápida de Fourier. También se puede observar que los tiempos entre el procesador Core 2 Duo 2,4 Ghz y el GPU GeForce 9800M GT son muy similares. El GPU GeForce 9600M GS es el más lento con respecto al resto de los GPUs y CPUs.

En promedio, la tarjeta de video GeForce GTX 470 es más rápida que el resto de los GPUs y CPUs.

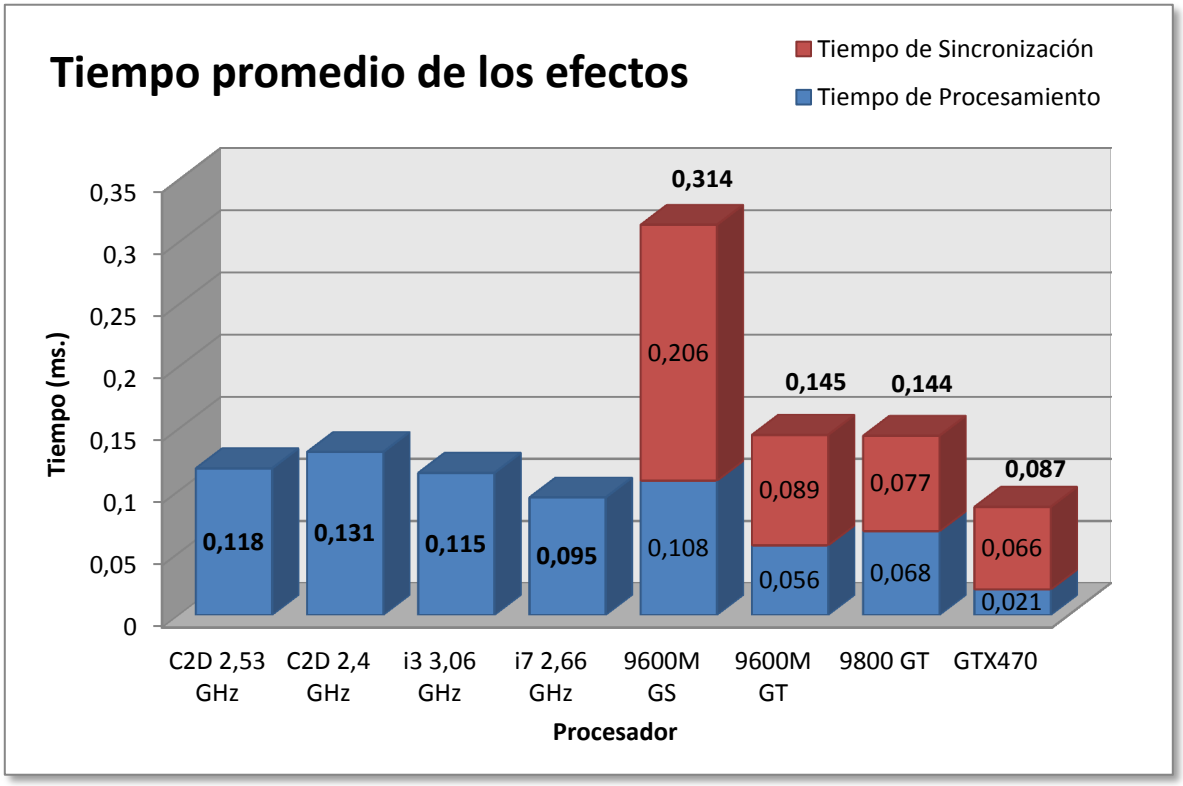


Figura 4.12: Comparación del tiempo de ejecución promedio de los efectos en CPU y GPU.

4.3.11. Nueve efectos de audio simultáneos

La gráfica de la **Figura 4.13** muestra los tiempos de ejecución de 9 efectos activados de forma simultánea en varias arquitecturas. Los efectos activados son todos los aquí presentados. Se puede notar que se puede aprovechar el potencial del GPU al realizar bastantes cálculos en él. En la implementación para GPU se puede apreciar que la tarjeta de video GeForce GTX 470 es 3 veces más rápida que el CPU Intel Core i7 2.66 GHz y 4,3 veces más rápido que el procesador Intel Core 2 Duo 2,4 GHz. En este caso el tiempo de sincronización con el GPU se equilibra con el rápido tiempo de cálculo que ofrecen los GPUs.

El procesador Intel Core i7 2.66 GHz es 1,4 veces más rápido que el GPU GeForce 9600M GS, sin embargo el resto de las ejecuciones hechas en los GPUs son más rápidas que las ejecuciones hechas en CPU.

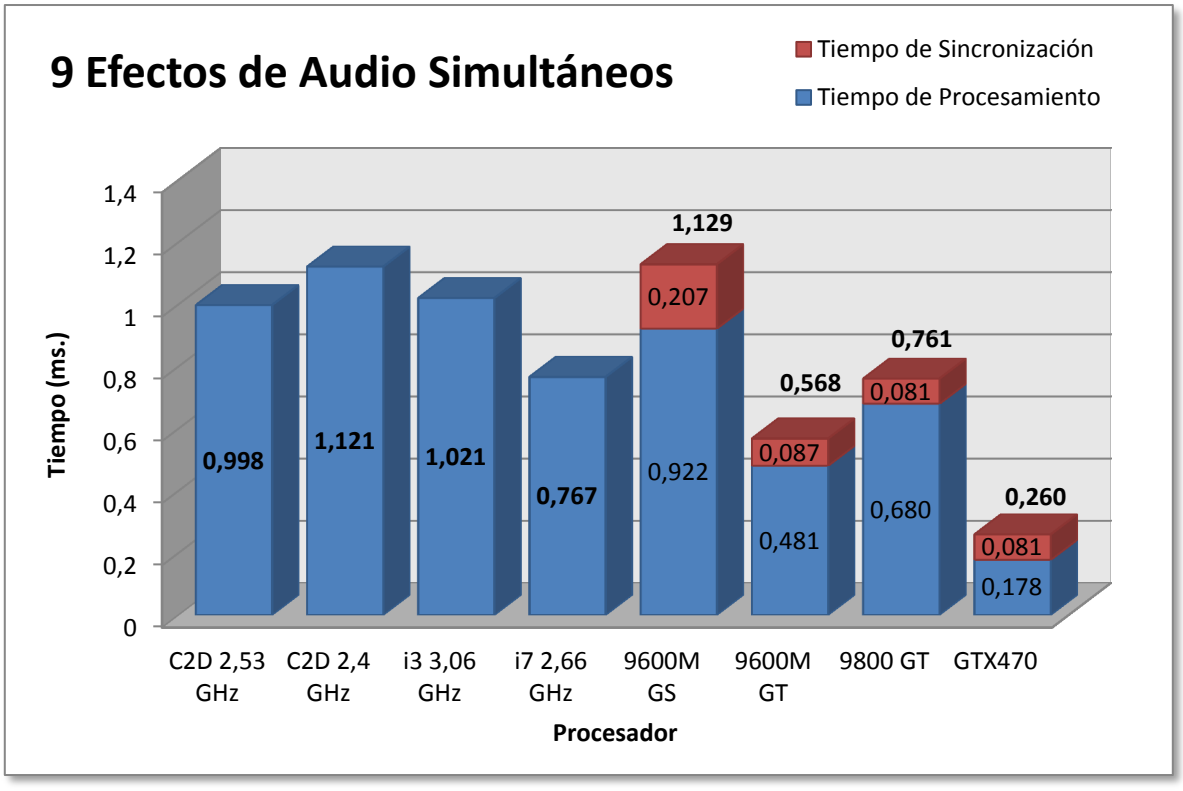


Figura 4.13: Comparación de rendimiento en tiempo de 9 efectos implementados ejecutándose de forma simultanea (CPU y GPU).

Capítulo 5. Conclusiones y Trabajos Futuros

Con la propuesta de este proyecto, se logró entrar en un área no explorada a lo largo de la carrera como lo es el audio digital, puesto que el pensum actual comprende algunos apectos de señales de manera generalizada o bien enfocada al procesamiento digital de imágenes.

En este trabajo se ha presentado la implementación de un sistema de procesamiento digital de audio en tiempo real en GPU logrando los objetivos planteados, demostrando que es factible procesar dichos efectos de audio en GPU y de forma eficiente.

Se observó un incremento en la velocidad de procesamiento de los algoritmos de audio en el GPU con respecto al CPU cuando son activados varios efectos.

Se notó que al activar solo un efecto no se aprovecha la capacidad del GPU ya que se está subutilizando y la sincronización penaliza los tiempos esperados. En el caso del efecto *Delay* la diferencia de tiempos es muy marcada.

Se apreció una particularidad en el efecto *Distortion* en cuanto al tiempo de procesamiento, ya que el cálculo de la función de transferencia para este efecto ($tanh(x)$) es más costoso en CPU que en la implementación para GPU.

El efecto *Ecualizador* implementado para GPU conlleva una gran ventaja para las tarjetas de video más recientes, ya que el cálculo de la Transformada Rápida de Fourier se realiza de forma muy eficiente con respecto a la implementación en CPU.

Se demostró que el poder de cálculo de esta última generación de GPUs puede ofrecer un aumento de velocidad significativo en los cálculos, sin embargo, se espera que en las próximas generaciones de GPUs existan mejoras en cuanto a la latencia que existe en la comunicación y transferencia de datos entre el CPU y el GPU.

Con este proyecto se logra un beneficio económico para los usuarios no profesionales al tener la posibilidad de utilizar un procesador digital de audio con una tarjeta de video NVIDIA de precio accesible.

Como trabajo futuro se puede plantear la utilización del lenguaje OpenCL como alternativa para obtener mayor portabilidad entre las tarjetas de video, ya que este trabajo tiene como limitación el uso de la arquitectura CUDA que es solo compatible con las tarjetas de video NVIDIA. Al implementar el código en OpenCL se lograría conseguir un rango mayor de compatibilidad con las tarjetas de video que soporten este lenguaje.

Igualmente en el futuro se pueden hacer pruebas en interfaces de audio que puedan manejar una mayor frecuencia de muestreo, ya que con bloques de datos de mayor tamaño se podría explotar aún más la capacidad de los GPU para procesar datos en paralelo. Una dura prueba sería el procesamiento de bloques de audio de 11.6 ms. a una tasa de muestreo de 172.400 muestras por segundo, ya que esto implicaría un procesamiento de 4 veces más datos que las pruebas hechas en este trabajo.

También, se puede intentar hacer pruebas en pistas de audio con canales Surround 7.1, ya que explotaría la capacidad de paralelismo del GPU no solo procesando 2 canales en estéreo sino hasta 8 canales de forma simultánea.

De ser factible, también se podría implementar otros efectos como el *reverb* de convolución, *compressor* (ver glosario) o algún efecto que tenga que ser procesado en el dominio de la frecuencia (e.g., reducción de ruido o *pitch shifting*).

Finalmente se recomienda implementar mejoras en la GUI, para poder obtener más flexibilidad al manipular los efectos, como por ejemplo cambiar el orden en que se procesan los efectos, agregar o quitar efectos, etc.

Referencias Bibliográficas

[1] WHALEN, S. *Audio and the Graphics Processing Unit.* [En línea] Disponible en: <http://www.extalin.com/gpudsp/gpuaudio.pdf>.

[2] JEDRZEJEWSKI, M.; MARASEK, K. "Computation of Room Acoustics Using Programmable Video Hardware." 2004. Computer Vision and Graphics International Conference, ICCVG 2004, Warsaw, Polonia, Septiembre de 2004.

[3] FABRITIUS, F. *Audio Processing Algorithms on the GPU.* Technical University of Denmark. 2009. Tesis de Maestría.

[4] GALLO, E.; TSINGOS, N. "Efficient 3D Audio Processing with the GPU." 2004. Proceedings of the ACM Workshop on General Purpose Computing on Graphics Processors. pág. C-42.

[5] CADIZ, R. *Introducción a la Música Computacional.* Centro de Investigación en Tecnologías de Audio, Instituto de Música, Pontificia Universidad Católica de Chile, 2008.

[6] SMITH, S. W. *The Scientist and Engineer's Guide to Digital Signal Processing.* 2da. Edición. San Diego, California. California Technical Publishing, 1999.

[7] COOLEY, J.W.; TUKEY, J.W. "An Algorithm for the Machine Calculation of Complex Fourier Series." *Mathematics of Computation.* 1965, vol. 19, págs. 297-301.

[8] BORES SIGNAL PROCESSING. *Introduction to DSP - filtering: digital filter equation.* [En línea] Disponible en: <http://bores.com/courses/intro/filters/4_eq.htm>.

[9] Parks-McClellan filter design algorithm. [En línea] Disponible en: <http://en.wikipedia.org/wiki/Parks-McClellan_filter_design_algorithm>.

[10] CONNEXIONS. *Diseño de Filtros usando la Gráfica de Polos y Ceros de la Transformada-Z.* [En línea] Disponible en: <http://cnx.org/content/m12967/latest/>.

[11] ZÖLZER, U. *DAFX - Digital Audio Effects.* Wiley, 2002.

[12] CUDA Programming Guide. *Versión 3.1.* NVIDIA Corporation, 2010.

[13] GPGPU. [En línea] [Citado el: 14 de Noviembre de 2010.] Disponible en: <http://www.gpgpu.org>.

[14] JAMES, G., "Operations for Hardware-Accellerated Procedural Texture Animation." Ed. M. Deloura. *Game Programming Gems 2.* Charles River Media, 2001, págs. 497-509.

[15] MARTINEZ, M. *Computación en GPU con NVIDIA CUDA.* Tutorial dentro del II Workshop en Aplicaciones de Nuevas Arquitecturas de Consumo y Altas Prestaciones (ANACAP 2009), España, Noviembre de 2009.

[16] *CUDA CUFFT Library. Versión 3.1.* NVIDIA Corporation, 2010.

[17] Virtual Studio Technology. [En línea] Disponible en: <http://en.wikipedia.org/wiki/Virtual_Studio_Technology>.

[18] STEINBERG. *3rd Party Developer Area.* [En línea] Disponible en: <http://www.steinberg.net/en/company/developer.html>.

[19] FFTW. [En línea] [Citado el: 5 de Octubre de 2011.] Disponible en: <http://www.fftw.org>.

[20] GÓMEZ, E. *Introducción al Filtrado Digital.* Departamento de Sonología, Escuela Superior de Música de Catalunya. Barcelona, España, 2009.

[21] GÓMEZ, E. *Efectos Digitales Básicos.* Departamento de Sonología, Escuela Superior de Música de Catalunya. Barcelona, España, 2009.

[22] MICEA, M.; STRATULAR M.; ARDELEAN, D.; AIOANEI, D. *Implementing Professional Audio Effects with DSPs.* Software and Computer Engineering Department, Politehnica University of Timisoara, Rumania, 2001.

Glosario

Altura tonal: En psicoacústica, la altura es un parámetro utilizado para determinar la percepción del tono (frecuencia) de un sonido. El que un sonido sea agudo o grave depende de su frecuencia.

Armónicos: Son los componentes de frecuencia de una señal periódica. Consiste siempre en múltiplos enteros de la frecuencia fundamental. La frecuencia fundamental es el primer armónico.

ASIO (*Audio Stream Input/Output*): Es un protocolo de comunicación de hardware para audio digital de la empresa Steinberg, que provee una baja latencia y una interfaz de alta fidelidad entre la aplicación de audio, el hardware y la tarjeta de sonido.

AU: Siglas de *Audio Units*. Es un formato de *plugin* desarrollado por Apple Computer para manipular audio en tiempo real, usado por el software DAW LogicPro.

Cero: En el contexto matemático de plano complejo, un cero de una función $f(z)$ es un punto $z = a$ tal que $f(z)$ se hace cero en dicho punto a.

Compressor: Es un efecto de audio que tiene un sistema automático de control de ganancia que constantemente monitorea la señal y ajusta la ganancia para maximizar la razón señal a ruido sin producir distorsión. Es importante no confundir a este tipo de compresores de amplitud de audio con la compresión de información digital o perceptual.

DAW: Son las siglas de *Digital Audio Workstation*. Es un sistema electrónico dedicado a la grabación y edición de audio digital por medio de un software de edición de audio.

Decibelio (*dB*): El decibelio es la unidad de medida utilizada para el nivel de potencia de la señal y el nivel de intensidad del ruido. El decibelio, es una unidad logarítmica.

Device: En el contexto de CUDA, es la unidad de cálculo (GPU) en la cual se ejecutan los *kernels*.

DSP: Son las siglas de *Digital Signal Processing*. Es el análisis y manipulación de los datos en forma digital. En este trabajo se refiere al procesamiento digital de los datos de audio.

Frecuencia de Muestreo: Es el número de muestras por unidad de tiempo que se toman de una señal continua para producir una señal discreta, durante el proceso necesario para convertirla de analógica a digital.

Función Delta: Es un impulso normalizado. La función discreta delta es una señal compuesta por ceros, excepto la muestra cero que tiene valor unitario.

Host: En el contexto de CUDA, es el elemento que interactúa con el *device* para ejecutar y administrar las tareas y funciones asignadas a procesar en el GPU.

Impulso: Es una señal compuesta por una colección de muestras con valor cero, excepto para algunos pulsos muy breves.

Impulso Unitario: Otro nombre para la Función Delta.

Kernel: Es una función declarada en el código del programa a ser ejecutado en el GPU.

Kernel de Convolución: Es la respuesta al impulso de un filtro implementado por convolución. También es conocido como Kernel del Filtro.

LFO (*Low Frequency Oscilator*): (Traducido como "Oscilador de Baja Frecuencia"), se refiere a una señal de audio normalmente por debajo de 20 Hz que crea un ritmo palpitante en vez de un tono audible.

MIDI: Siglas de (Musical Instrument Digital Interface). Es un protocolo de comunicación serial que permite a los computadores, sintetizadores, secuenciadores, controladores y otros dispositivos musicales electrónicos comunicarse y compartir información.

Octava: Es un factor de dos en la frecuencia de una señal.

OSC (Oscilador): Es un circuito que produce una señal electrónica repetitiva, a menudo una onda sinusoidal o una onda cuadrada.

Polo: En el contexto matemático de plano complejo, un polo de una función $f(z)$ es un punto $z = a$ tal que $f(z)$ tiende a infinito a medida que z tiende a a.

Respuesta al Impulso: Es la salida de un sistema cuando la entrada es un impulso normalizado (función delta).

Reverberación (*reverb*): Es un fenómeno derivado de la reflexión de sonido consistente en una ligera permanencia del sonido una vez que se ha extinguido el original, debido a las ondas reflejadas.

RTAS: Siglas de *Real-Time Audio Suite*, es un formato de *plugin* desarrollado por la empresa Avid Technology para su gama de programas DAW ProTools.

SDK: Son las siglas de *Software Development Kit*. Es un paquete que generalmente le permite a los desarrolladores crear software para alguna plataforma en particular.

Sesgo: Es el nivel de error sistemático en un proceso que causa que todos los valores medidos se desvíen a valores mayores o menores a los valores esperados.

Shader: Es un conjunto de instrucciones que son usadas generalmente para calcular efectos de *rendering* en hardware gráfico con un gran nivel de flexibilidad en cuanto a programación.