Desarrollo de una aplicación de audio multicanal utilizando el paralelismo de las GPU

Autor: José Antonio Belloch Rodríguez

Directores: Antonio Manuel Vidal Maciá
Francisco José Martínez Zaldívar

Abstract

Massive convolution is the basic operation in multichannel acoustic signal processing. This field has experienced a major development in recent years. One reason for this has been the increase in the number of sound sources used in playback applications available to users. Another reason is the growing need to incorporate new effects and to improve the hearing experience [1]. Massive convolution requires high computing capacity. GPUs offer the possibility of parallelizing these operations. This allows us to obtain the processing result in much less time and to free up CPU resources. One important aspect lies in the possibility of overlapping the transfer of data from CPU to GPU and vice versa with the computation, in order to carry out real-time applications. Thus, a synthesis of 3D sound scenes could be achieved with only a peer-to-peer music streaming environment using a simple GPU in your computer, while the CPU in the computer is being used for other tasks. Nowadays, these effects are obtained in theaters or funfairs at a very high cost, requiring a large quantity of resources. Thus, our work focuses on two main points: to describe an efficient massive convolution implementation and to incorporate this task to real-time multichannel-sound applications.

Author: Belloch Rodríguez, José Antonio, email: jobelrod@iteam.upv.es
Directors: Vidal Maciá, Antonio Manuel, email: avidal@dsic.upv.es
Martínez Zaldívar, Francisco José, email: fjmartin@dcom.upv.es

Agradecimientos

Este tesis de máster ha sido financiada por los siguiente proyectos: PROMETEO/2009/013 (Generalitat Valenciana), TEC2009-13741 (Ministerio de Ciencia e Innovación, España), TIN2008-06570-C04-02 y PAID-05-09.

Índice general

1. Introducción **1**
1.1. Motivación . . . 1
1.2. Raíces del problema . . . 2
1.3. Objetivos . . . 3
1.4. Organización de esta tesis de máster . . . 4

2. Programación en GPU **5**
2.1. Modelo de programación . . . 5
2.2. Organización de los threads de ejecución . . . 7
2.3. Arquitectura de la GPU . . . 9
2.3.1. Localización de memorias . . . 12
2.3.2. Runtime de la GPU . . . 13
2.3.3. Productos GPU de NVIDIA . . . 14
2.3.4. Plataformas utilizadas . . . 14

3. La Convolución **16**
3.1. Definición . . . 16
3.2. Convolución en señales largas . . . 17
3.3. Convolución en tiempo real . . . 18
3.3.1. Overlap-save . . . 18
3.4. Conclusiones . . . 20

4. Convolución multicanal sobre GPU **22**
4.1. Aproximación de convolución a la GPU . . . 22
4.2. Convolución sobre GPU . . . 23
4.3. Extrapolación a múltiples canales . . . 26
4.4. Configuración *Pipeline* del Algoritmo de convolución . . . 27

5. Implementación de Aplicaciones Multicanal sobre GPU **30**
5.1. Aproximación al desarrollo de un *kernel* para la convolución . . . 30
5.1.1. Señales de audio . . . 31
5.2. Convolución Masiva . . . 32
5.3. Aplicación Multicanal . . . 34

5.4. Análisis de Prestaciones . . . 36
5.4.1. Análisis temporal GPU-CPU . . . 36
5.4.2. Audio multicanal en tiempo real . . . 38
5.4.3. Conclusiones . . . 39

6. Implementación de un Cancelador de Crosstalk **41**
6.1. Introducción . . . 41
6.2. Planteamiento del problema . . . 41
6.3. Herramientas lógicas y físicas utilizadas . . . 44
6.3.1. Medida de las respuestas al impulso . . . 44
6.3.2. Preparación del sistema . . . 47
6.4. Demostración y Pruebas . . . 48
6.4.1. Convolución multicanal . . . 49
6.4.2. Cancelador de Crosstalk . . . 49

7. Conclusiones **51**
7.1. Líneas Futuras . . . 51
7.2. Aportaciones . . . 51
7.2.1. Revistas Internacionales . . . 52
7.2.2. Revistas Nacionales . . . 52
7.2.3. Congresos Internacionales . . . 52

Bibliografía **54**

Anexos **55**

Capítulo 1

Introducción

La Computación de Altas Prestaciones está expandiendo su ámbito de aplicación a numerosos problemas científicos y de ingeniería, entre éstos a problemas de Procesado de Señal para desarrollar aplicaciones de usuario en el prometedor mercado del procesado, la transmisión y la reproducción de contenidos multimedia. La utilización cada vez más generalizada de procesadores gráficos (GPU) en aplicaciones de propósito general, supone a la vez un importante reto y una gran oportunidad: la potencia de cálculo de estas nuevas arquitecturas permite resolver complejos problemas que requieren computación intensiva en computadores personales, si se desarrollan los algoritmos de computación de altas prestaciones apropiados, dando lugar a la disponibilidad, al alcance del usuario no experto, de aplicaciones que hasta hace poco tiempo eran impensables en el mercado de consumo.

1.1. Motivación

El campo del procesado de señales de audio multicanal (*Multichannel acoustic signal processing*) ha experimentado un gran desarrollo en los últimos años debido al aumento del número de fuentes sonoras utilizadas en las aplicaciones de reproducción disponibles para los usuarios, y en la necesidad creciente de incorporar nuevos efectos y potencialidades a la experiencia de la audición. [1][2][3].

La incorporación de dichos efectos ha estado siempre supeditado a la gran carga computacional que supone el procesado de señal multicanal, lo que hacia que únicamente pudieran ser obtenidos estos efectos en grandes espectáculos como teatros, parques de atracciones, centros comerciales, y siempre a través del uso de recursos potentes produciendo un gran gasto de energia.

A su vez, el creciente mercado de la distribución de contenidos multimedia con nuevos efectos sonoros para su utilización en el hogar, crea la necesidad de disponer de forma generalizada de herramientas de procesado de señal sonora multicanal que permitan extraer todas las potencialidades que sea posible incorporar en dichos contenidos multimedia. Es aquí donde la GPU juega un papel fundamental pues permitiría que dichos

contenidos puedan ser producidos en el comedor de una casa, obteniendo los mismos efectos que se conseguian en los grandes espectaculos, sin necesidad de utilizar los recursos computacionales de un PC, y ahorrando energía al mismo tiempo (figura 1.1).

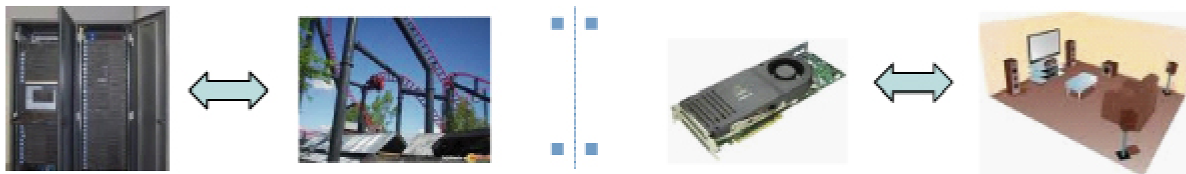


Figura 1.1: Efectos que necesitan gran capacidad computacional, se pueden conseguir usando una GPU.

1.2. Raíces del problema

El problema básico de la reproducción sonora multicanal podría describirse a partir del siguiente esquema (figura 1.2), donde un número de señales a reproducir se procesan a través de un sistema de procesado de señal multicanal que tiene en cuenta ciertas características o parámetros del auditorio y de la escena sonora a reproducir. Un gran número de aplicaciones de sonido espacial pueden derivar del sistema mostrado en dicha figura, tanto aquellas aplicaciones que realizan la reproducción mediante altavoces como mediante auriculares.

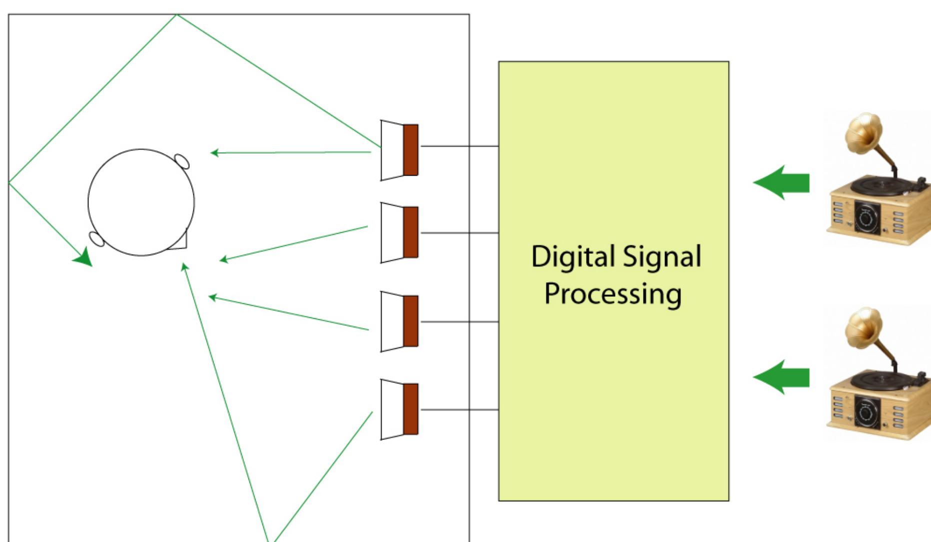


Figura 1.2: Ejemplo de reproducción sonora multicanal con dos señales de entrada y cuatro altavoces

La convolución masiva y las distintas combinaciones entre diferentes canales son las operaciones básicas del procesado de audio multicanal. En la figura 1.2, se aprecia un

sistema formado por 2 fuentes y 4 altavoces. Las dos fuentes proporcionan muestras de audio que son procesadas digitalmente para ser posteriormente reproducidas en los distintos altavoces. Este procesado permite obtener efectos de cancelación, sonido en tres dimensiones, etc.

La aparición de la GPU y su uso en el campo del procesado de señal ha logrado la liberación de recursos de CPU que permiten que ésta sea utilizada para otras tareas, explotando al mismo tiempo los recursos de la GPU y permitiendo abundante procesado simultáneo multicanal, que tanto cuesta realizar en una CPU. Sin embargo, el hecho de tener que transferir las muestras de señal desde la CPU a la GPU y viceversa, impedía desarrollar hasta ahora aplicaciones en tiempo real. En el desarrollo de este trabajo veremos como utilizando algoritmos con una estructura *pipeline* es posible minimizar el problema del trasvase de datos en ambos sentidos.

1.3. Objetivos

Esta tesis de máster se centra en solventar los problemas computacionales que surgen en el procesado de señales de audio cuando en éste intervienen varios canales que deben ser procesados en tiempo real y de forma concurrente.

La figura 1.2 muestra claramente el funcionamiento del procesado de audio multicanal. El bloque *Digital Signal Processing* se encarga de realizar multiples convoluciones, es decir, FFTs [4], multiplicaciones, sumas ... que deben ser ejecutadas sobre todos los canales entrantes. A su vez, en sistemas en tiempo real, tiene vital importancia la rapidez con la cual se ejecuten dichas operaciones, pues es deseable que acústicamente el flujo de la señal auditiva sea continuo.

Es por eso que el paralelismo que ofrecen, tanto las operaciones a realizar, como el hecho de que éstas se realicen de igual modo en diferentes canales, juega un papel fundamental en el procesado de audio. Por eso, explotando al máximo el paralelismo en canales y operaciones, es posible desarrollar nuevas aplicaciones que requieren de máxima rapidez, que hasta ahora no habían sido posibles.

El uso de las GPU como hardware de propósito general y su aprovechamiento como estructura SIMD [5] (*Single Instruction Multiple Data*) encaja perfectamente a las necesidades del procesado de audio multicanal.

Esta es la razón que nos lleva a implementar un algoritmo de convolución masiva sobre una GPU, que permita a su vez, poder realizar combinaciones multicanal de cara a conseguir los máximos efectos sonoros posibles.

La caracteristica principal de esta implementación es la flexibilidad en el número de fuentes sonoras y de altavoces (figura 1.2), así como que permita la posibilidad de poder ejecutarse diferentes aplicaciones de audio espacial.

En este trabajo abordaremos por tanto los siguientes objetivos:

- Implementación de la convolución masiva.

- Desarrollo de una aplicación multicanal que se ejecute sobre la GPU.
- Implementación de una aplicación de audio multicanal sobre una GPU dentro de un ordenador personal.
- Análisis y optimización de las prestaciones que se puede alcanzar con dicha GPU.

1.4. Organización de esta tesis de máster

Una vez conocidos los objetivos, podemos presentar la estructura que tendrá esta tesis. En el capítulo 2, describimos la arquitectura de la GPU y la organización de sus memorias. Los capítulos 3 y 4 están dedicados a describir el algoritmo de la convolución masiva y analizar las diversas formas de plasmar dicho algoritmo en la GPU. Diferentes implementaciones de la convolución masiva con sus resultados se examina en el capítulo 5. El capítulo 6 presenta como se ha implementado un cancelador crosstalk multicanal acústico, donde se abordan las diferentes librerías de audio utilizadas, así como los entornos para la programación de dichas librerías, y el material utilizado para su puesta en marcha. Finalmente, los capítulos 6 y 7 detallan las conclusiones alcanzadas, así como las distintas publicaciones que ha generado este trabajo, sin olvidarnos de las referencias bibliográficas.

Capítulo 2

Programación en GPU

Actualmente, la computación paralela se ha incorporado a la informática de consumo. Las empresas que diseñan y venden computadores la han incorporado en todas las gamas de sus productos, a veces con el mismo hardware en productos de línea alta y línea económica. Gran parte de este éxito ha sido debido a la industria de los videojuegos, donde la demanda de los usuarios para obtener mejores gráficos y prestaciones ha provocado una gran inversión en las Unidades de Proceso Gráfico, las GPU. El potencial de cálculo de dichas unidades no ha pasado desapercibido en los principales centros de investigación, y se están utilizando dichas unidades de proceso para cálculos de propósito general. De hecho, es tal el auge, que las nuevas GPU que salen al mercado ya no tienen salida gráfica y están orientadas principalmente para cálculos de altas prestaciones. Su éxito se basa en la utilización de la réplica de operaciones y de datos obteniéndose un grado alto de paralelismo, aumentando más velocidad en el proceso.

2.1. Modelo de programación

La tecnología NVIDIA CUDA™ (*Compute Unified Device Architecture*) [6] es un entorno basado en el lenguaje C que permite a los programadores escribir software para resolver problemas computacionales complejos en menos tiempo aprovechando la gran cantidad de hilos de ejecución de las GPU, a los que tambien llamaremos $threads$.

La programación en CUDA implica escribir código en dos partes diferentes (figura 2.1):

- Un HOST donde reside una o más CPU.
- Un DEVICE donde reside una o más GPU.

HOST y DEVICE se comunican a través del BUS PCI express, siendo éste un cuello de botella habitual de las aplicaciones desarrolladas en GPU, pues en muchas de ellas, se invierte más tiempo en trasladar datos entre HOST y DEVICE, que la propia ejecución sobre la GPU [7].

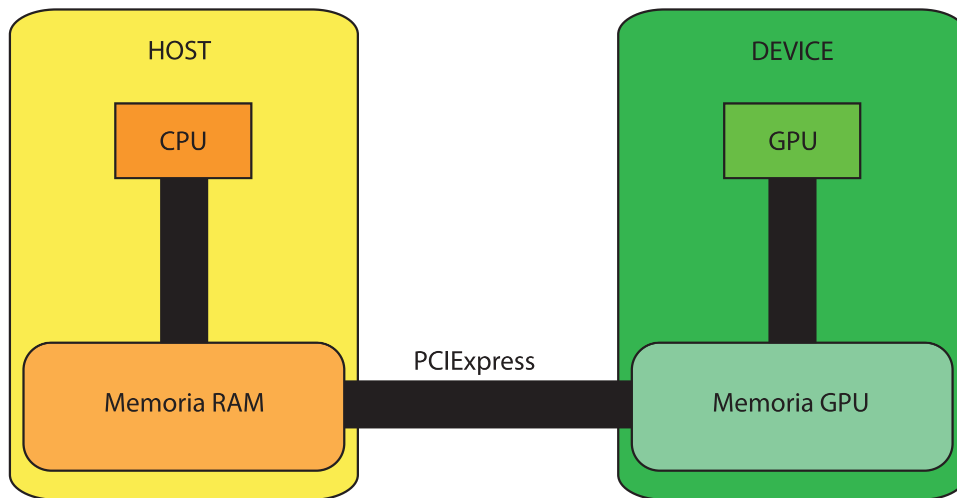


Figura 2.1: HOST y DEVICE en el modelo de programación de la GPU

La estructura lógica de un programa en CUDA consistirá en:

- Cargar los datos en la memoria RAM de la CPU.
- Trasladar los datos desde la memoria de la CPU a la memoria en la GPU.
- Ejecutar las operaciones en la GPU, con los datos trasladados.
- Obtener unos resultados, y trasladar esos resultados de vuelta a la memoria de la CPU

La zona de código donde la GPU realiza sus operaciones, recibe el nombre de $kernel$, el cual, tiene que ser lanzado desde la CPU, es decir, el HOST es el que manda ejecutar el $kernel$ en el DEVICE (figura 2.2).

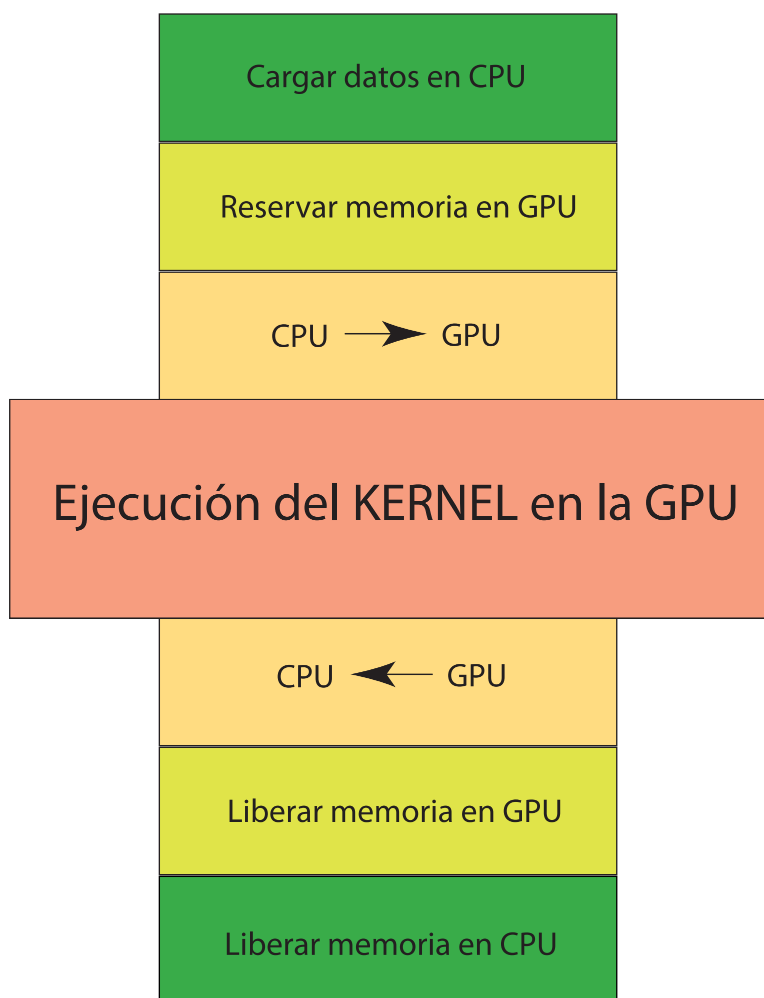


Figura 2.2: Modelo de programación en CUDA

El código paralelo o el código del *kernel* es ejecutado en el *Device* por muchos *threads*. El número de *threads* a utilizar dependerá de la cantidad de datos, de la forma que tengan éstos y de la operación que vaya a ser ejecutada. El código que se escribe en el *kernel* se hace para un solo *thread*, y todos los *threads* habilitados ejecutarán ese mismo código.

2.2. Organización de los threads de ejecución

Según el tipo de datos que tengamos, la operación que se ejecute sobre éstos y los resultados que se esperen, modificaremos la configuración de los *threads* dentro de la GPU. Los *threads* se agrupan en Bloques cuya dimensión será **BlockDim.x** (número de *threads* que contiene cada bloque). Los *threads* dentro de un bloque pueden comunicarse utilizando una memoria especial llamada memoria compartida de la que hablaremos más adelante. Existe un máximo de 512 *threads* por Bloque (según GPU). Así pues, llamaremos Grid al conjunto de bloques de *threads* lanzados a ejecución, y su organización dependerá también de los tipos de datos que se estén utilizando. Cada *thread* tendrá un

único Identificador **threadIdx.x** , dentro de cada bloque, que a su vez tendrá también un identificador **blockIdx.x** dentro de cada Grid. A cada *thread* se accederá de manera única a través de un índice único **Idx**. Así, si queremos inicializar todas las componentes de un vector de 15 componentes a 7, por ejemplo, podríamos definir una disposición de 3 bloques de *threads* con 5 *threads* cada uno, de forma que la **threadIdx.x**, **bloackIdx.x** y **BlockDim.x** quedarán asi (figura 2.3):

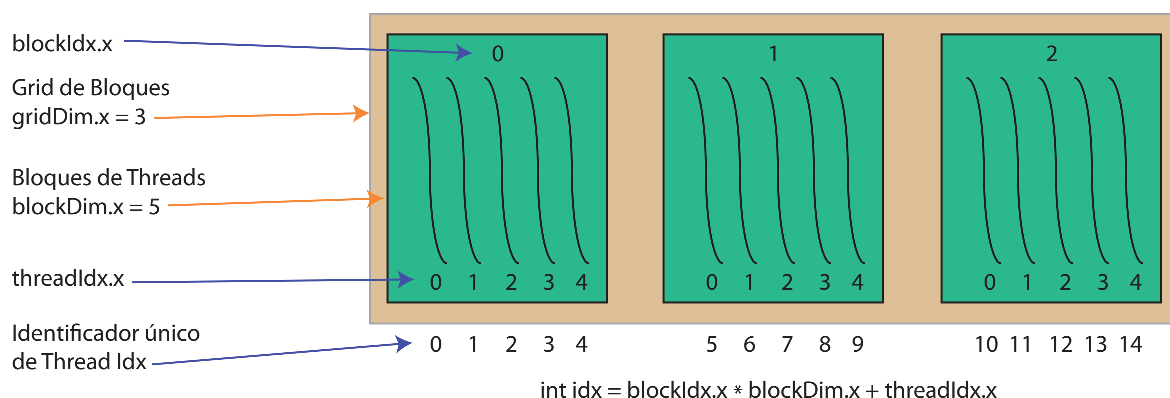


Figura 2.3: Disposición de los threads

De esta manera, si hicieramos un *kernel*, en el cual, cada *thread* se encargara de inicializar una componente de un vector con el valor 7. El *kernel* quedaría de la siguiente manera:

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

Es importante resaltar que las funciones que son llamadas desde el HOST y que son ejecutadas en el DEVICE son precedidas por la palabra `__global__`

Como se ha podido observar, tanto el nombre de los identificadores como de la dimensión del nímero de bloques por Grid, iban acompañadas al final por un **.x**. (**blockIdx.x**, **threadIdx.x**, **blockDim.x**). Esto nos indica, que estas variables tienen más de una dimensión (figura 2.4).

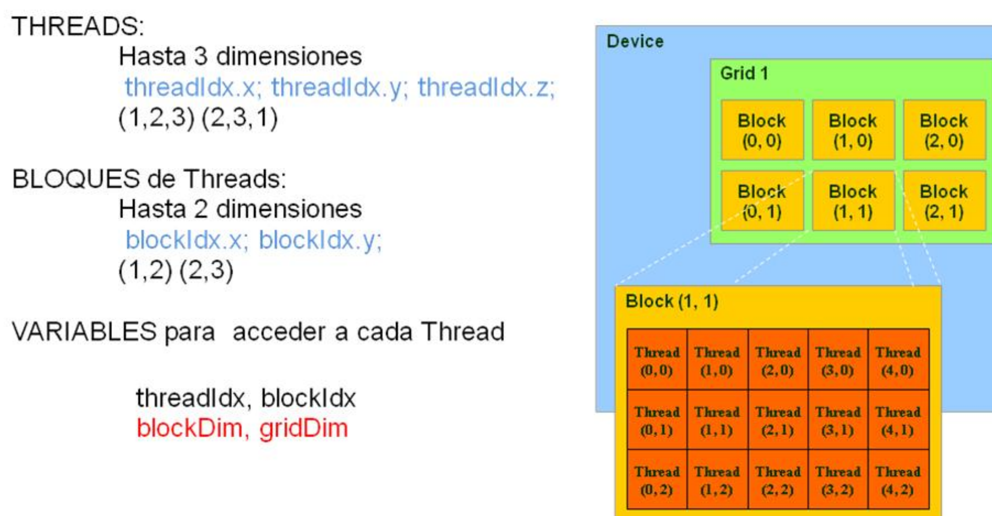


Figura 2.4: Disposición de threads dentro del bloque y del grid

Por tanto, los *threads* se ordenan en bloques cuya dimension puede ser unidimensional, bidimensional o tridimensional, y luego los bloques se ordenan dentro de un grid que puede ser unidimensional o bidimensional.

Toda esta organización en diferentes niveles se produce para guiar al programador de cara a obtener las maximas prestaciones de la GPU [8]. Antes de lanzar una ejecución en CUDA, el programador debe indicar las diferentes dimensiones mediante las variables de tipo `dim3` **gridDim** y **blockDim**:

- Numero de *threads* de ejecución por bloque.
- Disposición de los *threads* dentro del bloque.
- Numero de bloques de ejecución por Grid.
- Disposición de los bloques dentro del grid.

2.3. Arquitectura de la GPU

Las primera arquitectura de las GPGPU (*General-Purpose Computing on Graphics Hardware*) [9], también llamada arquitectura TESLA, y que caracteriza a la mayoría de las tarjetas de los modelos Tesla y Geforce de NVIDIA [10] están basadas en la unidad denominada Multiprocesador, SM (*Streaming Multiprocessor*) según las siglas utilizadas por NVIDIA [8]. Este multiprocesador está formado por ocho procesadores escalares de simple precisión, denominados tambien CUDA core, uno de doble precisión, 16 KB de memoria compartida y un banco de registros (16384 registros). Dependiendo de la tarjeta, unas tendrán mayor o menor número de SMs. En el caso de una TESLA C1060, tenemos 30 SM, y por tanto 240 CUDA cores. Todos los SMs tienen acceso a la memoria de la tarjeta, que llamaremos en adelante, memoria global. (figura 2.5).

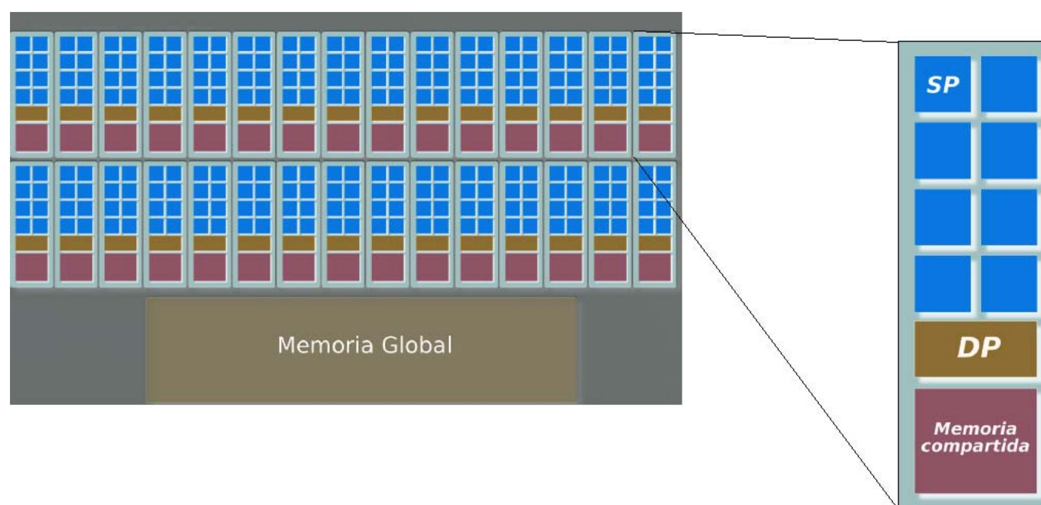


Figura 2.5: Arquitectura de la GPU formada por SMs y una memoria global

Las primeras versiones de esta arquitectura que tenían una capacidad de computación (*Compute Capability*) entre 1.0 y 1.1 [11] requerían amplios conocimientos a la hora de gestionar el acceso de los $threads$ a la memoria global. Existía la posiblidad de realizar 16 accesos a memoria en una sola transacción, pero se requería que los $threads$ consecutivos (con identificadores de $threads$ consecutivos) accedieran a posiciones de memoria consecutivas. Con la aparición de la versión 1.2, ésta última restricción ya permitía que $threads$ consecutivos pudieran acceder a posiciones de memoria no consecutivas, pero con ciertas restricciones [12].

Por otra parte, con la versión 1.2, se dio un gran salto cualitativo y es que si antes, como se ve en la figura 2.2, había que trasladar todos los datos a la GPU para comenzar a ejecutar el $kernel$, con ésta, se permitía poder solapar el trasvase de datos entre el HOST y el DEVICE con la ejecución del kernel, siempre y cuando el tipo de datos y operación lo permitieran. Es decir, se permitía particionar los datos en bloques, e ir enviando bloque a bloque los datos a la GPU (figura 2.6). En el momento en que llegara un bloque a la GPU, ésta ya podía empezar su ejecución dentro del $kernel$, mientras el segundo bloque de datos esta todavía siendo enviado a través del bus PCI-Express desde el HOST al DEVICE [12].

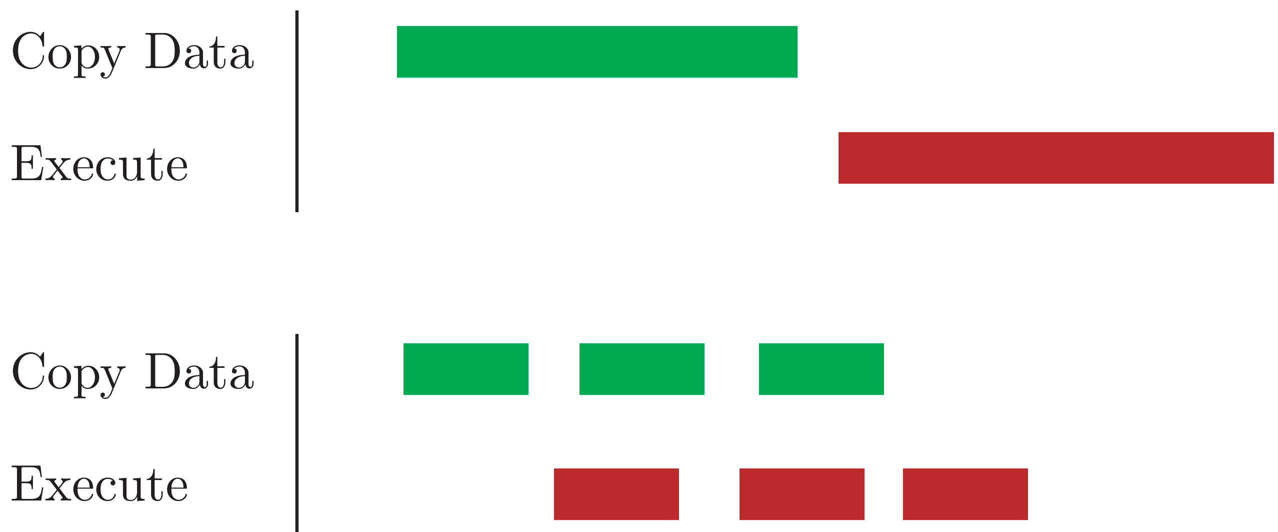


Figura 2.6: Particionado de datos en bloques y solapamiento de envio de datos y ejecución del *kernel*

La versión 1.3 posibilitó que las operaciones, pudieran ejecutarse con datos en doble precisión.

A finales del año 2009 NVIDIA anunció la aparición de una nueva arquitectura utilizada por las GPU llamada FERMI [13] y que ya incorporan las nuevas tarjetas gráficas que salen al mercado, la última de ellas la GTX-580. La arquitectura Fermi engloba a las tarjetas con capacidad computacional 2.0, un gran cambio después de la versión 1.3.

Esta nueva arquitectura destaca porque reduce el número de SM por tarjeta a 16, pero en cambio, aumenta el número de procesadores a 32, permitiendo realizar operaciones en doble precisión 8 veces más rápido que en las arquitecturas anteriores (figura 2.7). Por otra parte, presentan una memoria RAM de 64 KB que puede ser redimensionada para ser utilizada como una memoria compartida por los bloques (caso de arquitectura TESLA), como memoria Cache L1, o como ambas a la vez, pero en menor cantidad.

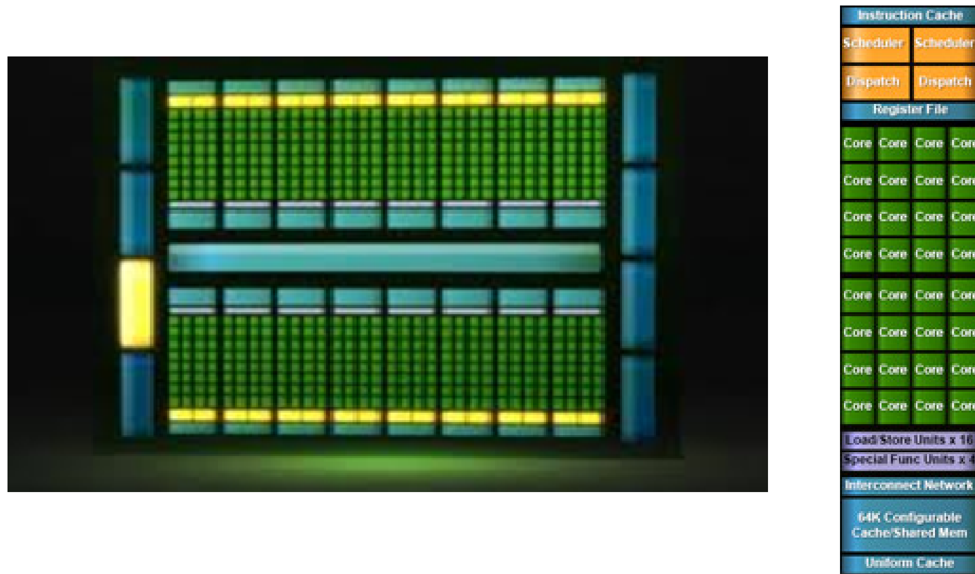


Figura 2.7: Arquitectura Fermi

2.3.1. Localización de memorias

La diferencia entre memoria global y compartida es que esta última es compartida por los *threads* de dentro de un mismo bloque, y será utilizada por ellos para intercambiar datos, o bien, como caché para reutilizar datos. El acceso a la memoria compartida es 10 veces más rápido que a la memoria global. Por contra, si tenemos que intercambiar datos entre *threads* de distintos bloques, no nos quedará más remedio que pasar por la memoria global (figura 2.8).

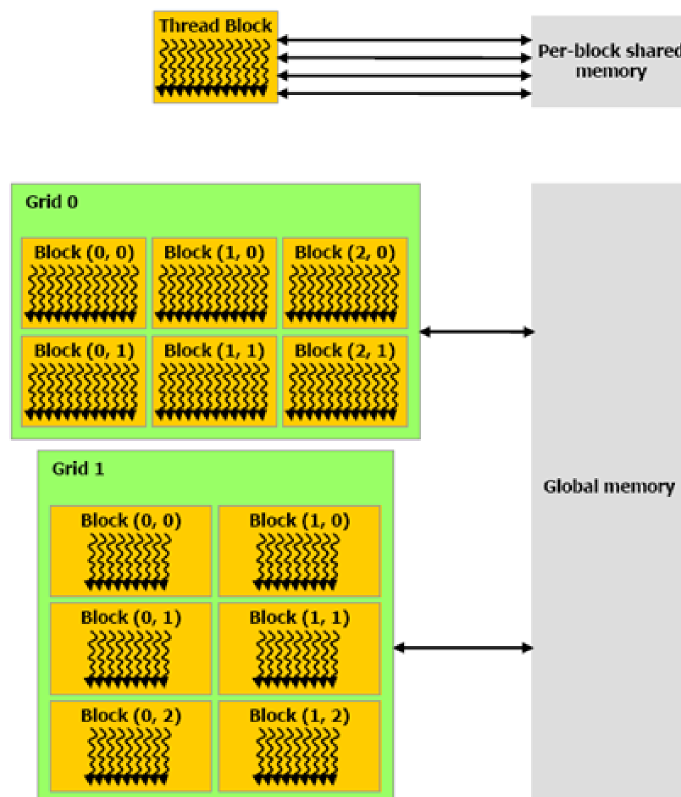


Figura 2.8: Arquitectura de la GPU formada por SMs y una memoria global

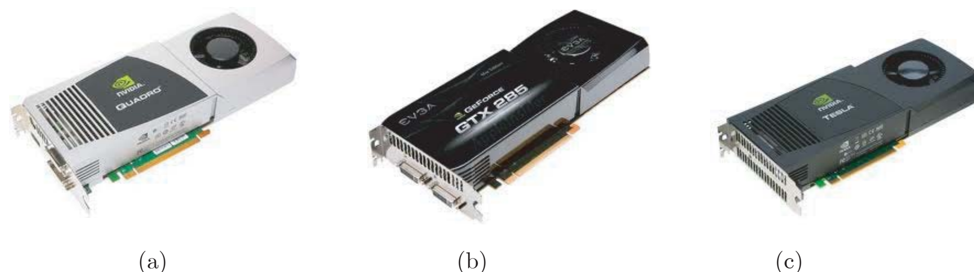
2.3.2. Runtime de la GPU

Cuando se lanza una ejecución en CUDA. Los bloques son distribuidos entre todos los SMs que tiene la tarjeta gráfica. Una vez repartidos, el SM hace grupos de 32 *threads*, a los que llama *warp*, independientemente del número de bloques que le hayan sido asignados y los ejecuta concurrentemente utilizando sus 8 procesadores. Cuantos más bloques haya, más *threads* y por tanto más *warps*. Por eso, es importante hacer grupos de 32 *threads* tal y como indica [8].

2.3.3. Productos GPU de NVIDIA

Atendiendo a su uso, existen tres tipos de tarjetas NVIDIA (figura 2.9):

- **GeForce**: A pesar de ser la más económica, presenta un gran potencial computacional. De hecho, el modelo GTX-295 doblaba el número de SM de la arquitectura TESLA, es decir, 60 SM por tarjeta.
- **Quadro**: Usan la misma arquitectura que la GeForce, pero está más orienta al uso de gráficos para videojuegos.
- **Tesla**: Esta tarjeta está más orientada a computación de altas prestaciones. De hecho, algunas no tienen salida de video.



(a) (b) (c)

Figura 2.9: (a) Quadro FX 5800, (b) GeForce GTX 285 y (c) Tesla C1060

De la misma manera, podemos encontrar tarjetas equivalentes que usan la misma GPU en cuanto a su arquitectura y su número de procesadores en los tres modelos, como por ejemplo las tarjetas, TESLA C1060, GeForce GTX-285 o la Quadro FX 5800.

Las tarjetas gráficas que permiten el uso de la tecnología CUDA están descritas en [10]. Por otra parte, en el Apendice A del *NVIDIA CUDA C Programming Guide* [11] puede consultarse el tipo de arquitectura y el número de SM que posee cada una de ellas.

2.3.4. Plataformas utilizadas

A lo largo de la tesis, se han utilizado diversos tipos de tarjetas. Todas ellas poseen una arquitectura Tesla (Ver sección 2.3).

Las primeras pruebas se realizaron en una GTX-285 con 1GB de Memoria, situada en un multiprocesador intel Core i7. A nivel software se utilizó una distribución Linux Ubuntu 9.04 de 64 bits. Estas pruebas iniciales giraban en torno a buscar como obtener el ancho de banda al acceder a cada una de las memorias más usadas de la tarjeta: la memoria global y la memoria compartida.

Una primera versión del algoritmo de la convolución fue probado sobre una TESLA C1060 que se diferenciaba de la anterior únicamente en la cantidad de memoria que tenía, que en este caso es de 4GB, y por tanto, podía almacenar más datos en la memoria global.

Finalmente, la aplicación de audio multicanal, Cancelador Crosstalk, es implementada en tiempo real sobre una tarjeta GeForce GTS 360M, que se encuentra dentro de un computador personal. Esta tarjeta se caracteriza porque está diseñada para un dispositivo móvil y como se puede leer en [11], está compuesta por 12 SM, y tiene 96 CUDA cores.

Capítulo 3

La Convolución

En este capítulo definiremos la operación de convolución formalmente con una única señal, abordando dos escenarios, uno en el caso en que la duración de la señal sea conocida y otro dentro de una aplicación en tiempo real donde las muestras de señal van llegando a una frecuencia dada.

3.1. Definición

La convolución es una operación que permite, dado un sistema caracterizado por una respuesta al impulso $h(t)$, predecir cuál va a ser la señal que encontramos a la salida $y(t)$. Si consideramos sólo sistemas lineales e invariantes con el tiempo, para una determinada entrada $x(t)$, la salida $y(t)$ se obtiene de forma teórica como [4]:

$$y(t) = \int_{-\infty}^{+\infty} x(\tau)h(t-\tau)\mathrm{d}\tau \tag{3.1}$$

En el dominio discreto, las integrales se convierten en sumatorios:

$$y[n] = \sum_{k=-\infty}^{+\infty} x[k]h[n-k] \tag{3.2}$$

Viendo esta operación, observamos que la convolución consiste en realizar sumas y multiplicaciones entre las distintas muestras de la señales $x[n]$ y $h[n]$.

De forma que cuando se convolucione una señal $x[n]$ de duración N y una respuesta al impulso $h[n]$ de duración M, obtendremos como resultado la señal $y[n]$ de duración $N+M-1$.

En ciertas aplicaciones es deseable considerar la convolución de dos secuencias periódicas $x_1[n]$ y $x_2[n]$, con **periodo común** N. Si en la ecuación (3.2), hacemos $k = rN + m$ y transformamos la suma en k en una doble suma en r y m, se tiene:

$$y[n] = \sum_{k=-\infty}^{+\infty} x_1[k]x_2[n-k] = \sum_{r=-\infty}^{+\infty} \sum_{m=0}^{N-1} x_1[rN+m]x_2[n-rN-m] \tag{3.3}$$

Como las dos secuencias del lado derecho son periódicas de periodo N, tenemos que:

$$y[n] = \sum_{r=-\infty}^{+\infty} \sum_{m=0}^{N-1} x_1[m]x_2[n-m] \tag{3.4}$$

Dado un valor fijo de n, la suma interna es constante, por lo que la suma infinita no converge. Para solucionar este problema, definimos una forma diferente de convolución de señales periódicas, que se denomina **convolución periódica o circular**:

$$y[n] = \sum_{k=0}^{N-1} x_1[k]x_2[n-k] \tag{3.5}$$

Nótese que la suma del lado derecho sólo tiene N sumandos. Esta operación la expresamos así:

$$y[n] = x_1[n] \otimes x_2[n]; \tag{3.6}$$

Como la ecuación (3.2) representa la salida de un sistema lineal, es usual denominarla **convolución lineal**, para distinguirla de la **convolución circular**.

Hay que recalcar que la convolución periódica se define sólo para secuencias del mismo período. En el caso de que no coincidan, habrá que extender al período con menor duración hasta N muestras para que ambos periodos duren igual. Al ser una convolución de señales periódicas, sólo estaremos interesados en los valores de n en el intervalo $0 \leq n \leq N-1$

3.2. Convolución en señales largas

Si aprovechamos las propiedades de la **Tranformada discreta de Fourier**, también llamada DFT (*Discrete Fourier Transform*), la convolución se puede convertir en una simple multiplicación de las dos señales (muestra a muestra) en el dominio de la frecuencia [14]:

$$y[n] = x_1[n] * x_2[n] \tag{3.7}$$

$$\Downarrow$$

$$X[k] = DFT(x[n]) \tag{3.8}$$

$$H[k] = DFT(h[n]) \tag{3.9}$$

$$\Downarrow$$

$$Y[k] = X[k]H[k] \tag{3.10}$$

Sin embargo, el producto $X[K]H[k]$ de las dos $DFTs$ corresponde a la **convolución periódica o circular** de $x[n]$ y $h[n]$.

$$IDFT(Y[k]) = x[n] \otimes h[n] \quad (3.11)$$

La pregunta que surge es si se puede utilizar la DFT para realizar una **convolución lineal**.

Tal y como se muestra en [4], si las señales iniciales $x[n]$ y $h[n]$ son expandidas ambas añadiendo ceros hasta una longitud K tal que $K \geq M + N - 1$. Entonces, la convolución circular, coincidirá con la convoluución lineal.

$$IDFT(Y[k]) = x[n] \otimes h[n] = x[n] * h[n] = y[n] \quad (3.12)$$

De esta manera se puede lograr la convolución lineal de $x[n]$ y $h[n]$.

3.3. Convolución en tiempo real

Nótese como en los apartados anteriores se han considerado únicamente señales con una longitud finita y conocida de antemano. Sin embargo, en la mayoría de las aplicaciones, se necesita un método para lograr convolucionar señales que puedan funcionar en tiempo real, proporcionando la salida correspondiente a medida que se muestrea una señal de entrada. Es más, aunque se conociera toda la señal de entrada y se quisiera hallar la correspondiente salida, la longitud de las señales involucradas podrían agotar los recursos disponibles del sistema. De aquí nace la necesidad de utilizar diferentes métodos de convolución [14] que nos permitan trabajar con señales de larga duración como es el caso del *Overlap-save*.

3.3.1. Overlap-save

Tomamos como referencia la señal de antes $x[n]$ en un sistema que funciona en tiempo real, en el cual nos van llegando muestras a una cierta frecuencia pero que no sabemos cuando va a terminar, y una respuesta al impulso $h[n]$ de duración M.

Con las muestras de entrada, se forman bloques de L muestras. A cada bloque lo denotaremos como $x_k[n]$. El valor de L es arbitrario aunque normalmente se suele escoger un valor potencia de 2 más grande que la longitud de $h[n]$

Estos bloques se caracterizan porque solapan $M - 1$ muestras del bloque anterior. Mención especial tiene el primer bloque, el $x_0[n]$, el cual, se solapará con $M - 1$ ceros añadidos al principio. (Figura 3.1):

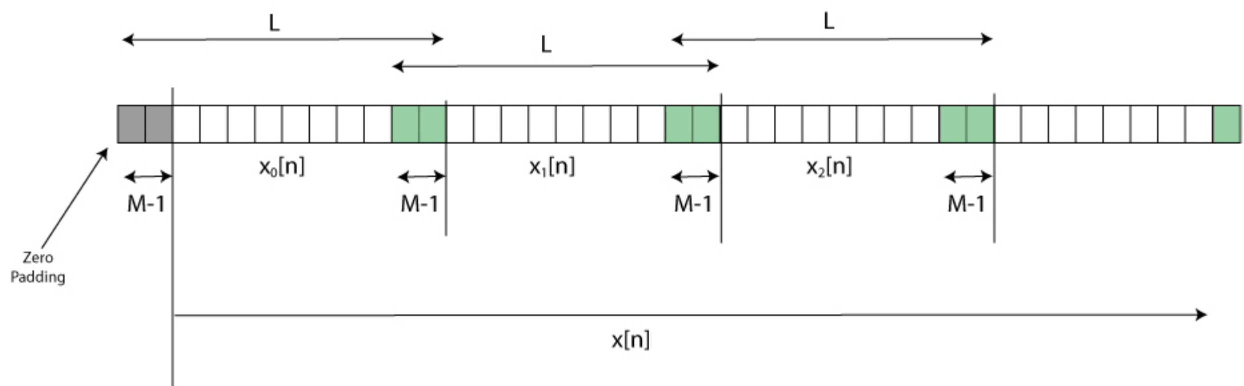


Figura 3.1: División en bloques de L muestras

Añadimos ceros también a la señal $h[n]$ hasta que su longitud sea L, de forma que las señales $x_k[n]$ y $h[n]$ tendrán la misma longitud, y es entonces cuando ya estamos en condiciones de aplicar el algoritmo Overlap-Save:

1. Cogemos cada uno de los segmentos $x_k[n]$ y realizamos la convolución circular con $h[n]$, de forma que $y_k[n] = x_k[n] \otimes h[n]$. Para obtenerla, tal y como hemos escrito anteriormente, solo hay que aplicar la DFT y $IDFT$ a los bloques $x_k[n]$ y a $h[n]$ expandida hasta longitud L (Figura 3.2).

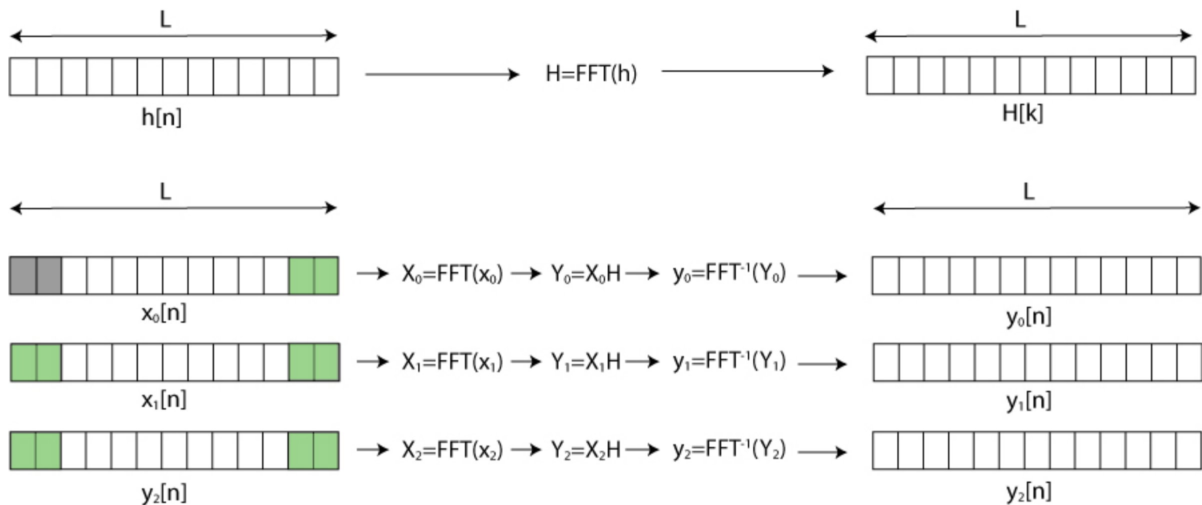


Figura 3.2: División en bloques de L muestras

2. De los bloques de señal obtenidos $y_k[n]$, despreciamos las primeras $M-1$ muestras, y vamos agrupando los bloques de manera que ya tenemos la señal de salida del sistema (Figura 3.3).

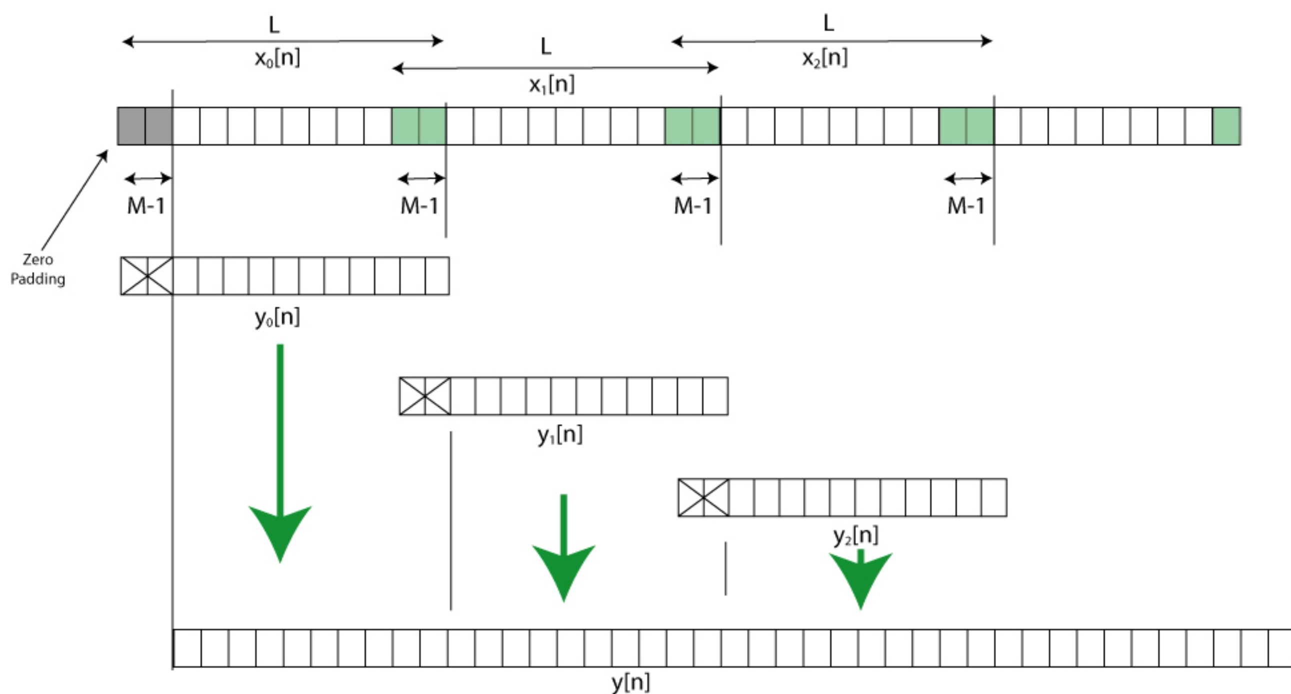


Figura 3.3: División en bloques de L muestras

3.4. Conclusiones

La implementación del *Overlap-save* vista en la sección anterior presenta un alto grado de paralelismo. Se puede observar que sobre cada uno de los fragmentos en los que queda dividida la señal $x[n]$ se realizan las mismas operaciones (figura 3.2).

El comportamiento de la GPU a la hora de ejecutar un programa y explotar el paralelismo de datos, invita a desarrollar una aplicación de convolución masiva donde se puedan dar multiples señales y multiples filtros (diferentes $h_i[n]$ [14]). De esta manera, no sólo se explota el paralelismo hallado en los diferentes pasos a la hora de obtener la convolución, sino también, el hecho de que ésta pueda ser obtenida para múltiples señales en paralelo.

En el campo del audio multicanal, una aplicación de convolución masiva sería muy interesante por el gran ahorro computacional que supondría la realización en paralelo de las diferentes operaciones de filtrado, de cara a implementar aplicaciones de sonido multicanal (figura 3.4) . Estas aplicaciones requieren de un alto número de fuentes sonoras y de altavoces.

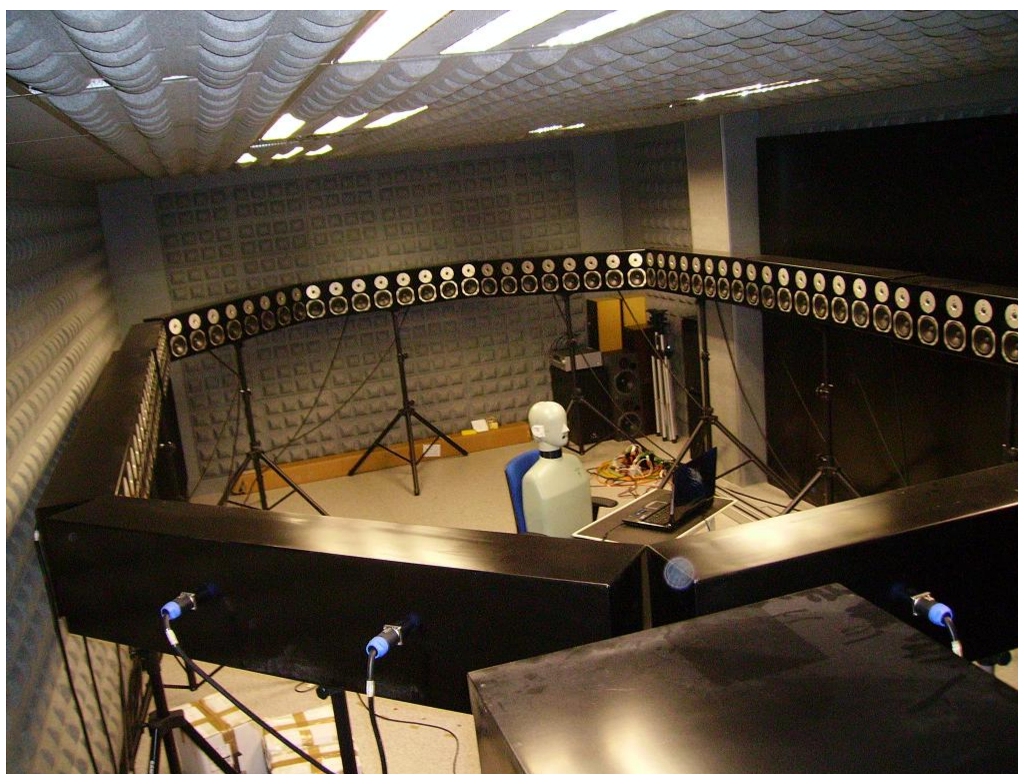


Figura 3.4: Sistema de sonido espacial con 96 altavoces

Sin embargo, el problema real en el campo del audio multicanal no gira en torno a la cantidad de operaciones que se deben realizar sino a la rapidez con que deben ser realizadas. En aplicaciones en tiempo real con calidad CD, la frecuencia de muestreo es de f_s=44.1kHz, es decir, una muestra de señal de audio es leída cada 22μs. Este valor es importante, pues dependiendo del tamaño del buffer de muestras que tengamos y la gestión que realicemos sobre él, una aplicación multicanal tendrá éxito o no. Una aplicación multicanal en tiempo real permite no sólo la convolución paralela de diferentes canales (*Convolución Masiva*), sino poder combinarlos entre ellos (*Sonido Espacial, Ecualizador de sonido 3D, Cancelador Crosstalk*).

Un factor muy importante en aplicaciones en tiempo real, es la continuidad entre la lectura de las muestras en las fuentes sonoras y la salida por los altavoces. El uso de las GPU para aplicaciones de estilo ha resultado siempre dificultoso. En los capítulos posteriores tratamos de implementar una aplicación que pueda funcionar en tiempo real. Para ello tratamos de sacar el máximo rendimiento a la capacidad de computación 1.2 (*Compute Capability*, ver sección 2.3) que ofrecen las tarjetas utilizadas en esta tesis, las cuales, que permiten solapar ejecución en GPU con transferencia de datos CPU $\leftrightarrow$ GPU.

Capítulo 4

Convolución multicanal sobre GPU

En este capítulo plasmamos el algoritmo de la convolución visto en el capítulo anterior sobre una GPU. Posteriormente extrapolamos el funcionamiento de la convolución de un canal a multiples canales. Finalmente, veremos como la convolución de multiples canales nos permite llevar a cabo aplicaciones de audio multicanal.

4.1. Aproximación de convolución a la GPU

NVIDIA, dentro de su página dedicada a los desarrolladores [15], ofrece un SDK (*Software Development Kit*), en el cual, se dan varios ejemplos de aplicaciones. Una de ellas gira en torno a la convolución.

Esta aplicación realiza únicamente la convolución de una señal finita, utilizando el teorema de la convolución [4] y siguiendo el siguiente esquema (figura 4.1):

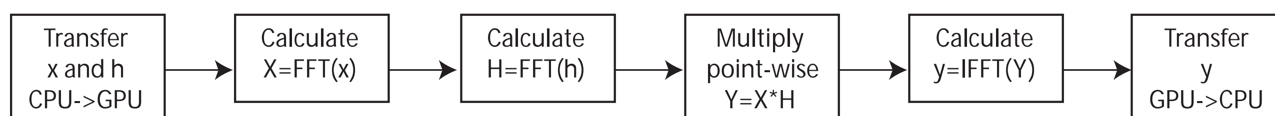


Figura 4.1: Esquema de convolución que sigue el ejemplo

NVIDIA utiliza para hacer la DFT su propia libreria CUFFT [16], que está optimizada para ejecutar tanto transformaciones de Fourier directas como inversas, así como multiples transformaciones en paralelo.

Sin embargo, este esquema no permite realizar convoluciones en tiempo real, ni permite explotar el paralelismo de multiples canales. Además en el caso de que la señal fuera muy larga, se invertiría mucho tiempo en el trasvase de datos CPU $\Leftrightarrow$ GPU. Por supuesto, se necesita otro modelo de convolución a seguir, amparado en los conocimientos de señal explicados en la seccion 3.

4.2. Convolución sobre GPU

La implementación del *Overlap-save* vista en el capítulo anterior presenta alto grado de paralelismo. Se puede observar que sobre cada uno de los fragmentos en los que queda dividida la señal $x[n]$ se realizan las mismas operaciones (figura 3.2).

Si adoptamos una configuración en forma de matriz (figura 4.2), sacaremos máximo rendimiento de la librería CUFFT [16], pues ésta permite dada una matriz, poder realizar tantas FFT paralelas como filas tiene esa matriz.

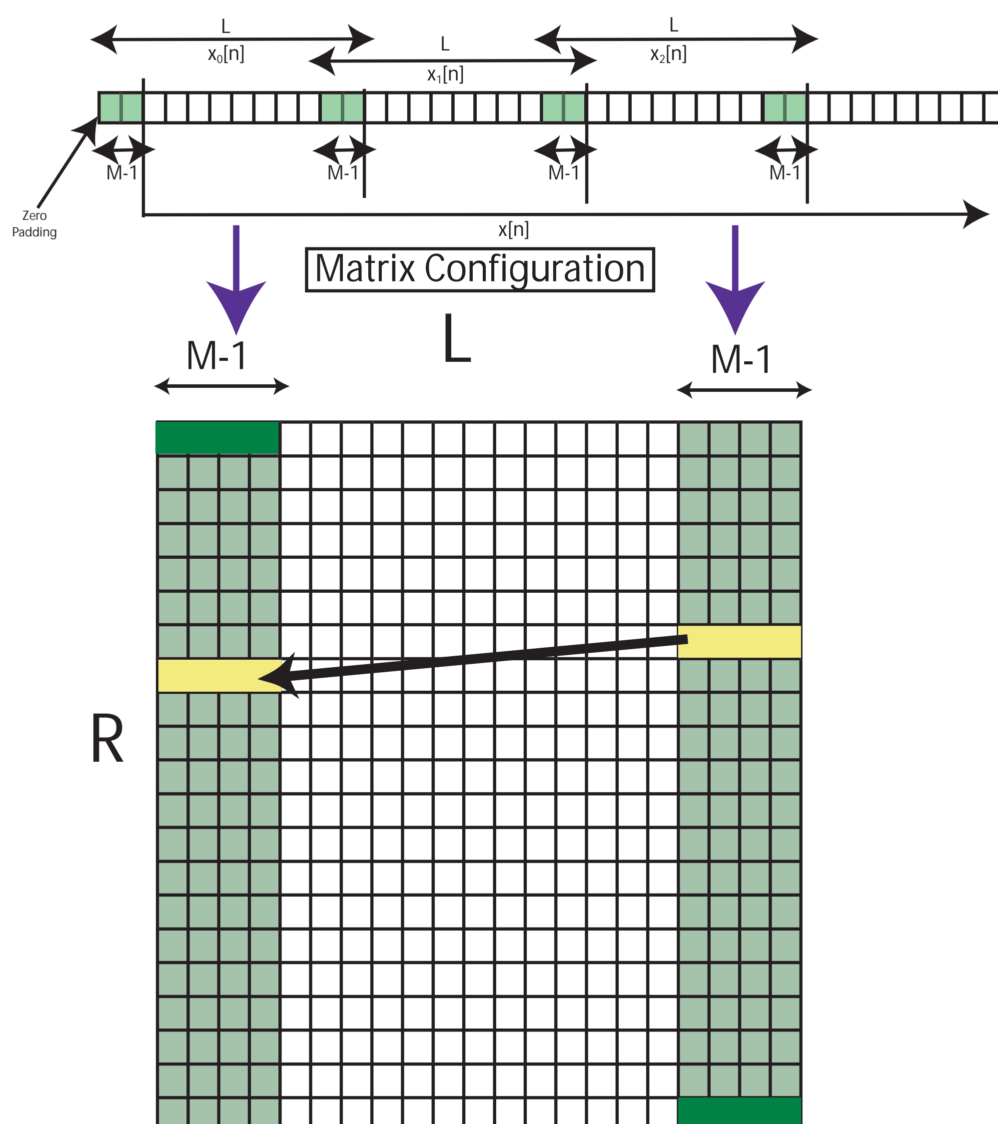


Figura 4.2: Esquema de convolución que sigue el ejemplo

Para conseguir esta disposición matricial, haremos que cada fragmento $x_k[n]$ ocupe una fila de la matriz. Así, hasta un número de filas R definido por el programador. A la hora de organizar las muestras procedentes de las fuente sonora, es necesario duplicar las $M-1$ últimas muestras de una fila (M es la longitud de la $h[n]$ vista en la sección 3.3.1) y

colocarlas en la siguiente fila (figura 4.2).

Una vez tenemos la matriz rellenada, se envía dicha matriz a la GPU y se procede a ejecutar el algoritmo *Overlap-save*:

1. Sobre la matriz recibida en GPU, utilizamos la librería CUFFT para ejecutar R FFTs, tantas como filas tiene la matriz. El resultado será una matriz pero ahora las muestras pertenecerán al dominio frecuencial (figura 4.3).

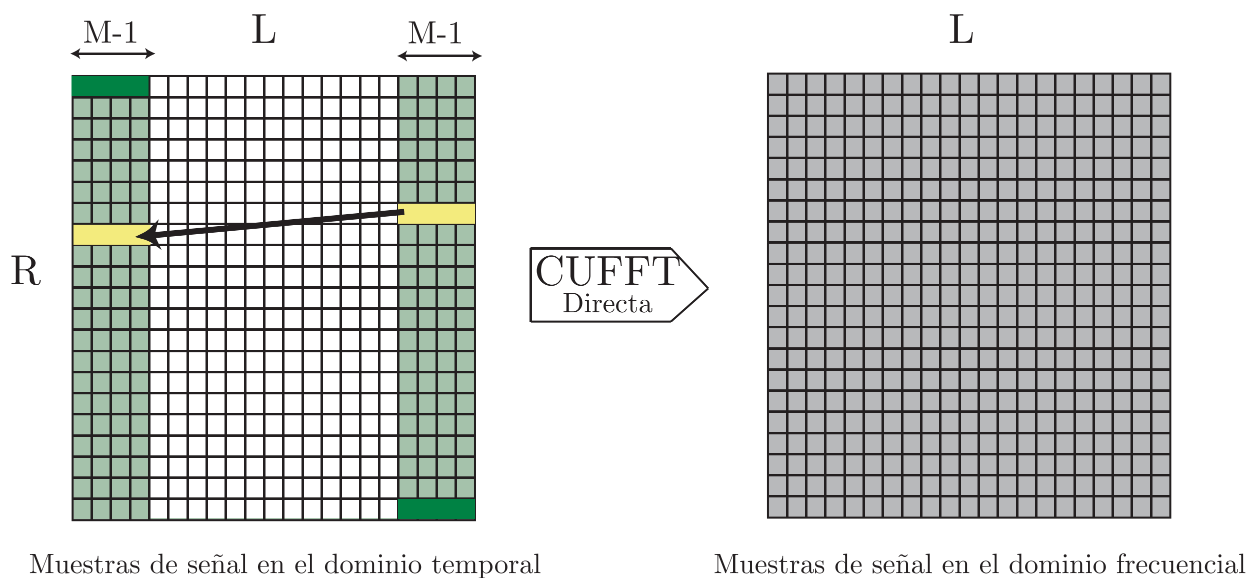


Figura 4.3: Ejecución de tantas FFTs como filas tiene la matriz en paralelo usando CUFFT

2. Previamente al envío de la matriz con las muestras, se habrá mandado a la GPU el $\mathbf{h}[n]$ expandido hasta la longitud L (Ver sección 3.3.1). Obtenemos su FFT y lo multiplicamos punto a punto por cada una de las filas de la matriz (figura 4.4).

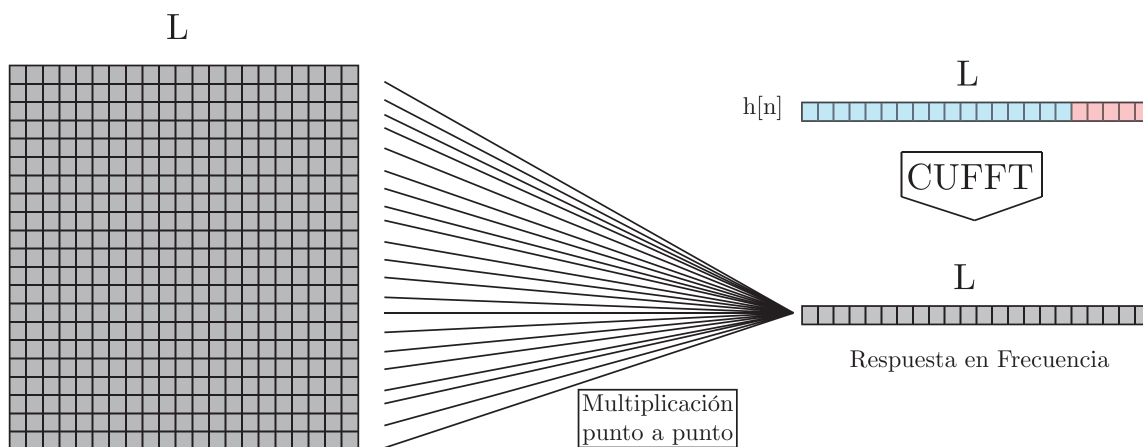


Figura 4.4: FFT de $\mathbf{h}[n]$ y Multiplicación punto a punto con cada fila de la matriz

3. Una vez tenemos las muestras ya filtras en el dominio de la frecuencia, debemos realizar la transformada inversa de Fourier, la IFFT [4]. Para ello, usaremos la CUFFT inversa y ya obtenemos las muestras señal en el dominio temporal (figura 4.5).

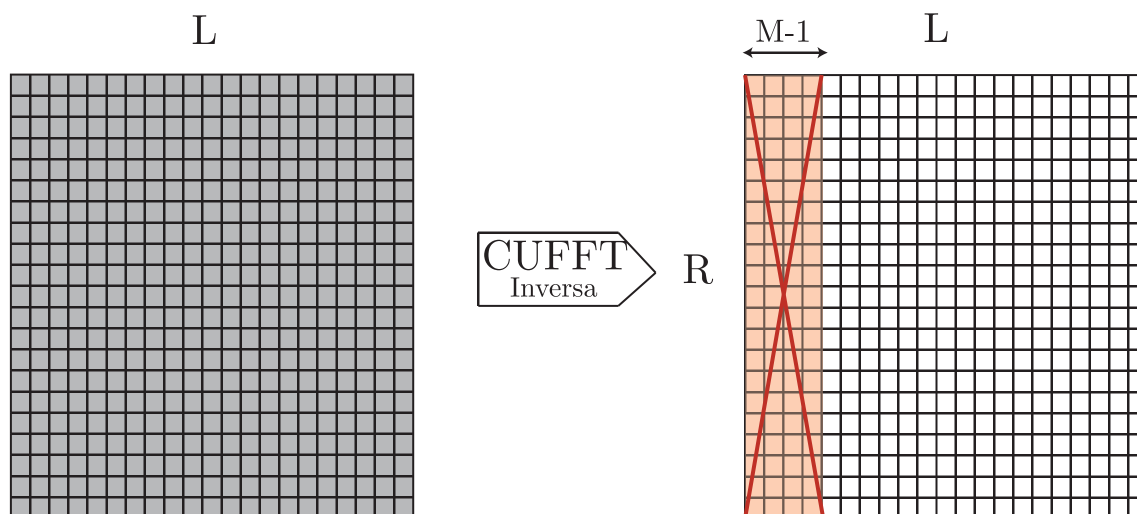


Figura 4.5: Calculo de la IFFT de cada una de las filas y posterior procesado para obtener las muestras temporales.

4. Con las muestras ya filtradas en dominio temporal, la matriz de datos puede ser devuelta a la CPU. Una vez allí, habrá que descartar las primeras $M-1$ pues éstas

han sido ya procesadas en el bloque anterior (sec 3.3.1).

4.3. Extrapolación a múltiples canales

Considerando que un canal de audio está compuesto por una señal, o una fuente sonora, podemos decir que en la sección anterior acabamos de ver como implementar la convolución de un canal. El paso para poder realizar dicha convolución con múltiples canales es inmediato. Este paso está basado en la **división de recursos**:

En el caso de ejecutar la convolución de dos canales, la matriz con las muestras de señal quedaría dividida en dos partes (figura 4.6):

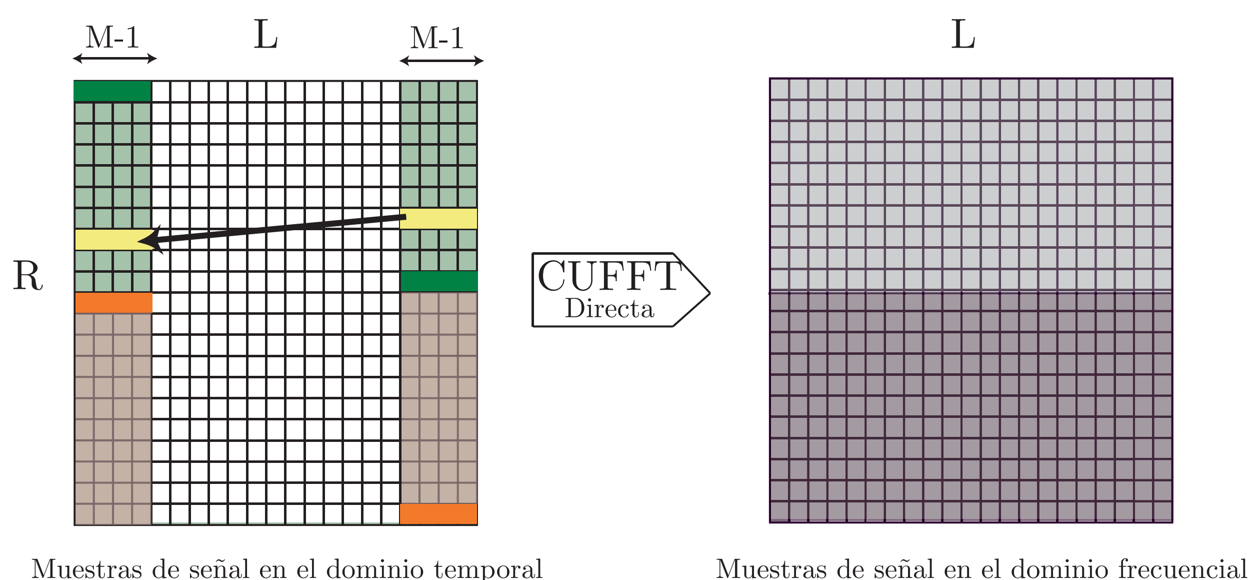


Figura 4.6: Matriz de señal dividida para cada uno de los canales

Ya con las muestras en dominio de la frecuencia, podemos bien, utilizar la misma $\mathbf{h}[n]$ para ambos canales, realizando el mismo filtrado para ambas señales (misma situación que en (figura 4.4) pero ahora las muestras de señal pertenecerían a dos canales) o bien, podemos utilizar filtros diferentes para cada canal, es decir, $\mathbf{h}_1[n]$ para un canal y $\mathbf{h}_2[n]$ para el otro canal. Todos los cálculos se siguen ejecutando en la GPU de forma paralela (figura 4.7).

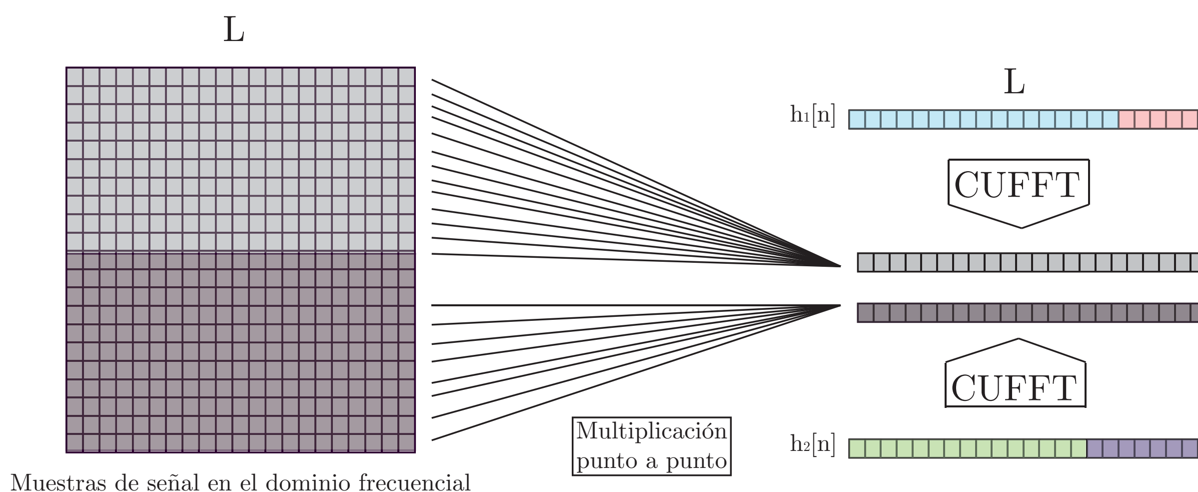


Figura 4.7: Un canal convoluciona con $h_1[n]$ y otro con $h_2[n]$ en paralelo

Esta división de recursos se puede extrapolar a más canales siempre y cuando existan recursos suficientes.

4.4. Configuración $Pipeline$ del Algoritmo de convolución

A partir de la versión 3.1 del Toolkit NVIDIA CUDA, se incorporó a la librería CUFFT, la propiedad *Concurrent Copy and Execution*, que poseían las tarjetas que tenían una capacidad de computación 1.2 (Ver sección 2.3). Esto permitía solapar la ejecución de la librería CUFFT con la transferencia de datos entre CPU y GPU. La consecuencia principal de esta medida, no sólo se refiere a la posibilidad de que el algoritmo de la convolución vaya más rápido sino, a la posibilidad real de realizar una aplicación de audio en tiempo real cuyo procesado sea llevado a cabo por una GPU.

De esta forma podríamos definir el algoritmo en las siguientes fases:

1. En una primera fase, se almacenan las muestras procedentes de las fuente sonoras en una matriz buffer, a la que denominaremos Matriz-Buffer A.

2. Una vez que se ha llenado la matriz con muestras de señal, ésta se envía de forma asíncrona a la GPU, de forma que el control vuelve inmediatamente a la CPU, que continúa rellenando otro Buffer, al que denominaremos Matriz-Buffer B. Como el flujo de muestras es continuo, cuando se envíe una matriz a la GPU, es necesario almacenar en un Buffer auxiliar las muestras de solapamiento (las denominadas $M_i - 1$ muestras) de cada uno de los canales, para que puedan ser utilizadas en la siguiente Matriz-Buffer a ser rellenada (figura 4.8).

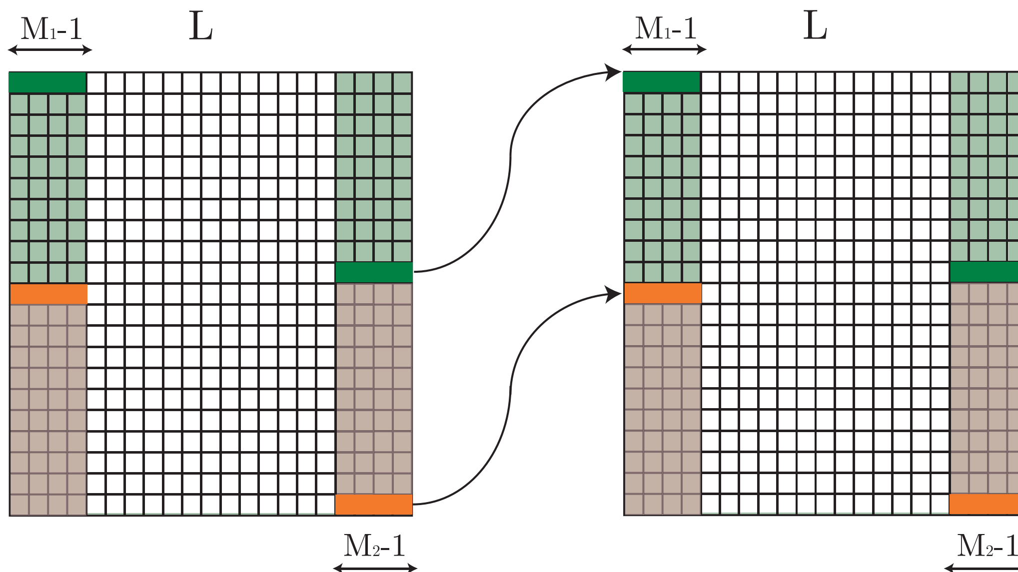


Figura 4.8: Un canal convoluciona con $h_1[n]$ y otro con $h_2[n]$ en paralelo

3. Con el paso anterior acabado, sobre los datos de la Matriz-Buffer A comenzarán a ejecutarse las operaciones descritas en la sección 4.2 . Al utilizar una GPU con capacidad de computación 1.2, podremos al mismo tiempo transferir de forma asíncrona la Matriz-Buffer B a la propia GPU, y por tanto, continuar rellenando otro Buffer, que llamaremos Matriz-Buffer C (figura 4.9).

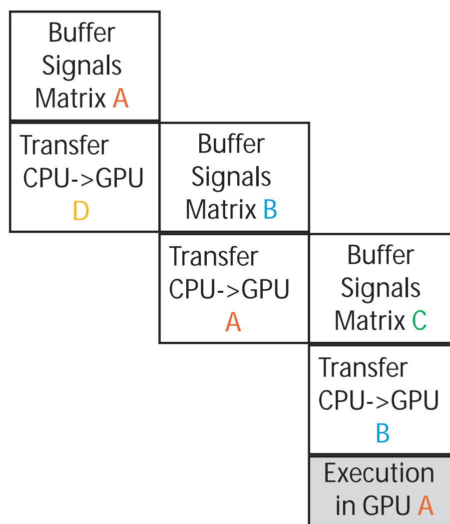


Figura 4.9: Se ejecuta el kernel en Matriz-Buffer A, se transfiere a la GPU la Matriz-Buffer B y se rellena con nuevas muestras Matriz-Buffer C

4. Posteriormente, se procede a: transferir la Matriz-Buffer A de vuelta a la CPU, ya con

los datos de la señal filtrados, a ejecutar las operaciones del kernel sobre la Matriz-Buffer B, a transferir asíncronamente la Matriz-Buffer C a la GPU, y por último, a continuar rellenando otro buffer, Matriz-Buffer D.

5. Con la Matriz-Buffer A ya devuelta en la CPU, se obtienen de ella las señales ya filtrados, y posteriormentes comienza a ser de nuevo rellenada por nuevas muestras comenzando el ciclo otra vez, obteniéndose un algoritmo con estructura *pipeline* (figura 4.10).

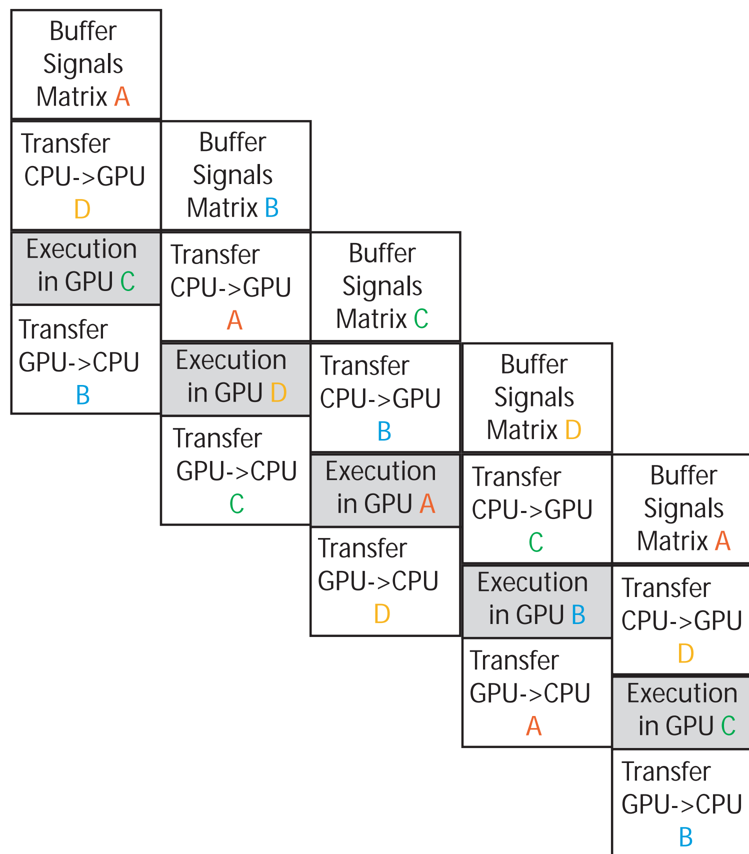


Figura 4.10: Estructura *pipeline* del algoritmo de la convolución utilizando 4 Matriz-Bufferes

En capítulos posteriores se analizará las prestaciones de este algoritmo.

Capítulo 5

Implementación de Aplicaciones Multicanal sobre GPU

En los capítulos anteriores hemos analizado las prestaciones que ofrece una GPU y hemos comprobado como este hardware puede ser utilizado para ejecutar sobre él aplicaciones de audio multicanal. En este capítulo vamos a tratar de analizar las prestaciones en procesado de audio que se alcanzan cuando aplicaciones de audio multicanal se ejecutan sobre GPU. Estas aplicaciones siguen el esquema $pipeline$ mostrado en el capítulo anterior, centrándonos en este capítulo en las diferentes implementaciones de $kernel$, pues su correcta configuración marcará la eficiencia del algoritmo.

5.1. Aproximación al desarrollo de un $kernel$ para la convolución

Las primeras pruebas se llevaron a cabo en una GPU Tesla C1060, para poder comprobar la capacidad de mejora que tenía el algoritmo *Overlap-save* con respecto al algoritmo que presentaba NVIDIA en su SDK [15].

El $kernel$ implementado era el más básico, pues no hacia uso de la memoria compartida. Se optó por una configuración en bloques de 16x16 $threads$, y se analizaron diferentes configuraciones de matriz, variando el número de filas y de columnas, resultando la óptima para R=32 y L=512.

Para evitar problemas de acceso consecutivo en la memoria global, se duplicó R veces el filtro en la CPU (número de filas que posee el Matriz-Buffer de señal). Posteriormente se trasladaban a la GPU ambas matrices. El $kernel$ implementa la FFT de cada una de las filas de ambas matrices, y posteriormente las multiplica punto a punto. Por último, se ejecuta la IFFT sobre la matriz resultante. Las operaciones de la FFT e IFFT se llevan a cabo usando la librería CUFFT.

El problema principal de este $kernel$ inicial era la latencia, pues debía trasladar dos matrices a la GPU al principio del algoritmo. A pesar de la sencillez del algoritmo, esta implementación lograba reducir en la mitad de tiempo el algoritmo presentado por NVI-

DIA en su SDK *Software Development Kit*. Ver Tabla 5.1. Un estudio más exhaustivo puede verse en [17].

Tipo de Algoritmo	**Tiempo**
Convolución NVIDIA SDK	1330ms
Configuración Pipeline	625.92ms

Tabla 5.1: Comparición entre dos algoritmos de convolución en GPU

5.1.1. Señales de audio

Comprobadas las prestaciones del algoritmo de convolución *pipeline*, es necesario evaluar el rendimiento que tendría dicho algoritmo en una aplicación de audio en tiempo real. Para ello tendremos que tener en cuenta lo analizado en las secciones 3.4 y 4.4.

La sección 4.4 recogía un algoritmo de convolución donde se hablaba de 4 Matriz-Buffer, donde las Matriz-Buffer A, B, C y D pasaban por cuatro estados diferentes desde que eran enviadas hacia la GPU hasta que sus datos volvían a la CPU, donde eran de nuevo reutilizadas con nuevas muestras procedentes de las fuentes sonoras.

Para que el sistema de audio en tiempo real funcione, es necesario que la Matriz-Buffer A haya vuelto a la CPU antes de que la Matriz-Buffer D haya terminado de ser rellenada con muestras audio. Teniendo en cuenta que en audio, la frecuencia de muestreo es de 44.1KHz, cada 1/44100 s llegará una muestra nueva por cada canal. Utilizando la misma configuración que en el apartado anterior, se determinó que una Matriz-Buffer tarda 9.37 ms desde que es enviada a la GPU hasta que está de vuelta a la CPU con las señales ya filtradas.

Por tanto es fácil analizar el número de canales de audio que puede llegar a albergar dicha aplicación y que se presenta en la Tabla 5.1.1.

Número de canales	Ocupación de filas por canal	Tiempo empleado llenado Matriz-Buffer	Uso de GPU (%)	Viabilidad
1	32	212.6ms	4.4 %	Si
2	16	106.3ms	8.8 %	Si
4	8	53.15ms	17.6 %	Si
8	4	26.9ms	35.2 %	Si
16	2	13.2ms	70.5 %	Si
32	1	6.6ms	141 %	No

Tabla 5.2: Numero de posibles canales en una aplicación de audio utilizando una Matriz-Buffer con L=512 columnas y R=32 filas

Estos resultados aparecen también en [18].

5.2. Convolución Masiva

Una vez analizada la viabilidad de las GPU para implementar aplicaciones de audio en tiempo real, el siguiente paso consiste en optimizar el código que se ejecuta en el $kernel$ para ser implementado en una aplicación real y poder realizar la convolución masiva de forma generalizada.

Atendiendo a las especificaciones de programación en GPU, analizadas en el capítulo 2, debemos definir un tamaño de grid y un tamaño de bloque de threads. Siguiendo las pautas marcadas en [8], es conveniente tener 256 threads por bloque. Al empezar la ejecución, los bloques son repartidos entre los distintos SM (ver capítulo 2). De esos bloques se escogen grupos de 32 threads que son ejecutados de forma paralela entre todos los nucleos CUDA. Además, a la hora de acceder a la memoria global, interesa que threads con identificadores consecutivos accedan a posiciones de memoria consecutivas. Es por esto que una de la dimension x del **blockDim** debe ponerse a 32, es decir, **blockDim.x**=32. Como no queremos superar la cifra de 256 threads por bloque, la otra debemos ponerla a 8 **blockDim.y**=8.

Para escoger el valor de **gridDim** usaremos los datos nombrados en el capítulo 3 que indicaban que la longitud de L era bien la siguiente potencia de dos de la longitud de los filtros o bien 512 elementos. Teniendo en cuenta que L es el numero de columnas y que los filtros con los que tratamos tienen del orden de 200 coeficientes, tomaremos L=512.

El $kernel$ que se va a desarrollar a continuación se basa en la eficiencia de la multiplicación punto a punto entre un vector y las filas de una matriz tal y como se apreciaba en la figura 4.4.

Para favorecer la reutilización de datos en las operaciones, hacemos que las filas dedicadas a cada uno de los canales sea **blockDim.y**. De esta manera, los threads de un mismo bloque compartirán los mismos datos y podrán acceder fácilmente a la zona de memoria compartida para hacer uso de ellos. De esta manera, las operaciones que se realizan en la GPU son:

1. Obtener, tal y como se ha nombrado en capítulos anteriores la FFT de los $h_i[n]$ usando la CUFFT obteniendo $H_i[k]$. Esto sólo se hará al principio del algoritmo y quedarán los valores almacenados en la memoria global de la GPU.

2. Calcular la FFT de cada una de las filas de la matriz de datos.

3. Multiplicamos cada $H_i[k]$ con los canales correspondientes, para ello:

    - En la figura 5.1 se puede ver como los bloques se encuentran distribuidos sobre la matriz de datos. Cada thread calculará el valor de una matriz.

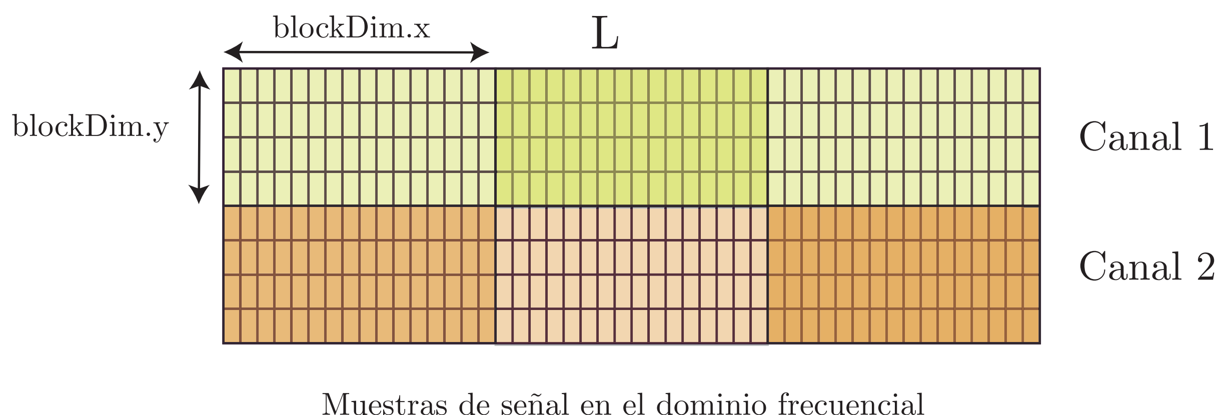


Figura 5.1: Distribución de los bloques de threads sobre la matriz de muestras de señal

- Se reserva blockDim.x memoria compartida por bloque. Los *threads* con **blockIdx.y**=0 copiarán los valores de la memoria global de $H_i[k]$ a la memoria compartida figura 5.2

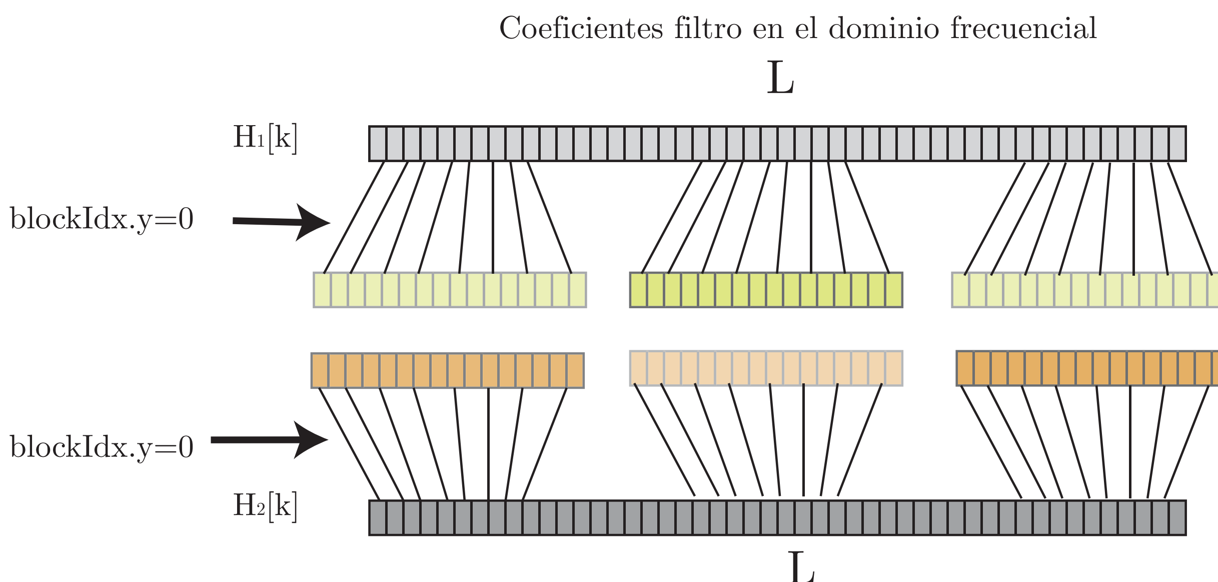


Figura 5.2: Copia de los valores del filtro de la memoria global a la memoria compartida

- Ahora hacemos que cada *thread* dentro de cada bloque multiplique por el valor que le corresponde en la memoria compartida. Los *threads* de la misma columna dentro de cada bloque multiplicarán por el mismo valor en la memoria compartida figura 5.3

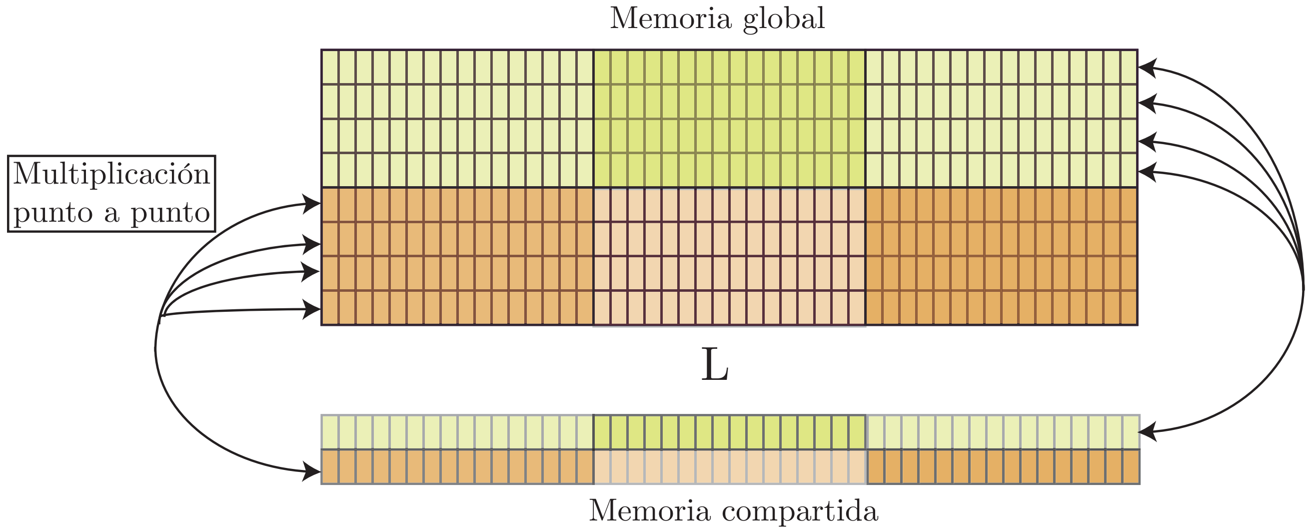


Figura 5.3: Multiplicación punto a punto entre los valores de bloque con los valores de la memoria compartida

4. Ya con las muestras filtradas en el dominio de la frecuencia, sólo tenemos que calcular la IFFT en cada una de las filas, utilizando la biblioteca CUFFT y obtenemos las muestras en el dominio temporal. Sólo queda ya trasladarlas de nuevo a la CPU.

5.3. Aplicación Multicanal

Con la convolución masiva tenemos el paso fundamental para poder implementar aplicaciones multicanal. La aplicación multicanal generalizada es aquella en la que existe un número M de fuentes $\{x_0, x_1, x_2, x_3, x_4, ..x_{M-1}\}$ y un número N de altavoces $\{y_0, y_1, y_2, y_3, y_4, ..y_{N-1}$
De manera que:

$$y_i[n] = \sum_{j=0}^{M}(h_{ij}[n] * x_j[n]) \tag{5.1}$$

Observando la ecuación anterior, podemos observar que para obtener la señal de salida de un altavoz, se deben dar tantas convoluciones como entradas existan. La implementación del nuevo *kernel* estará basado en el anterior más una reducción de todos los bloques.

Por ejemplo, si contáramos con 2 fuentes de entrada y 3 altavoces de salidas, y queremos calcular la salida del primer altavoz:

$$y_0[n] = h_{00}[n] * x_0[n] + h_{01}[n] * x_1[n] \tag{5.2}$$

Ahora añadimos a la figura 5.3 una reducción de los bloques y ya tendremos la salida del primer altavoz figura 5.4.

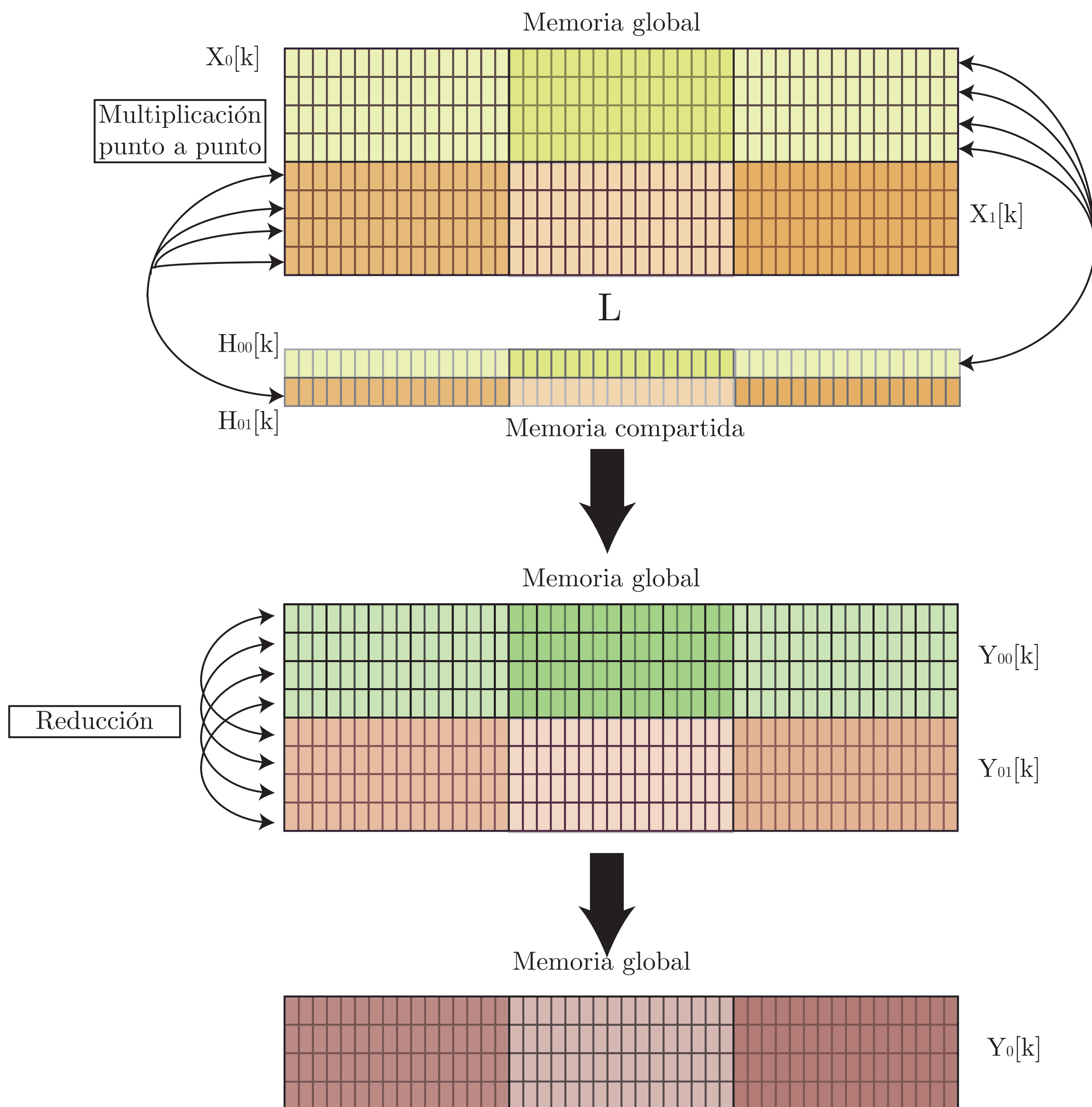


Figura 5.4: Multiplicación punto a punto entre los valores de bloque con los valores de la memoria compartida

El resto de salidas de altavoces se obtiene siguiendo el mismo esquema. Por tanto, las ejecuciones descritas anteriormente se repetirán por cada altavoz existente.

5.4. Análisis de Prestaciones

Una vez presentad el desarrollo de una aplicación multicanal, se han realizado algunos tests para alcanzar las prestaciones que nos ofrece dicha aplicación utilizando una tarjeta GTS-360 integrada en un computador personal ASUS Notebook G60 Series con un procesador Intel(R) Core(TM) i7 CPU Q720 @1.60GHz y 8 GB. En estas pruebas se ha intentado valorar, la eficiencia del procesado y la latencia. Al mismo tiempo se ha comparado con la CPU y para ver su ganancia.

5.4.1. Análisis temporal GPU-CPU

En esta primera prueba, se han lanzado distintas ejecuciones variando el número de fuentes y el número de altavoces. Se han combinado las diferentes configuraciones, variando cada una de las variables, fuentes y altavoces, entre 3 y 51, con saltos de 3 en 3. Se han escogido estos valores pues la tarjeta gráfica sobre la que se han desarrollado los diferentes tests posee 12 SM [11] y como se puede leer en [3], se recomienda que el tamaño del grid con el que se lanzan las ejecuciones sobre la GPU sea proporcional al número de SM que posee la tarjeta. Teniendo un tamaño de bloque (**blockDim.x**=32 y **blockDim.y**=8), y eligiendo a L=1024 (columnas de la Matriz-Buffer), obtenemos que las dimensiones del grid son **gridDim.x**=32 y **gridDim.y**=$Num_Canales$. Para que el tamaño de grid sea proporcional a 12, debemos escoger la variable $Num_Canales$ múltiplo de 3 para obtener las mejores prestaciones.

Las medidas de tiempo realizadas tratan de medir tiempo de ejecución de una matriz de datos, en una sola secuencia (en un único $stream$) de acuerdo a la figura 4.10. Para ello, se ha determinado el número de $streams$=4, es decir, 4 ejecuciones en modo $pipeline$ que solaparán transferencia de datos, con ejecución de $kernels$. Se va a repetir la ejecución del $pipeline$ 60 veces, de forma que una vez hayan terminado todos los $threads$ sus ejecuciones, dividiremos los tiempos por 240, (60 x 4), y obtendremos la ejecución de una secuencia.

Los tiempos obtenidos se pueden contemplar en la Tabla 5.3:

Si representamos gráficamente los datos anteriores figura 5.5, podemos observar que el tiempo de ejecución aumenta tal y como aumenta el número de fuentes y altavoces:

Alt \ Fnt	**3**	**9**	**15**	**21**	**27**	**33**	**39**	**45**	**51**
3	1.202	2.038	2.783	3.701	4.631	5.768	6.803	7.689	8.735
9	1.501	3.387	4.417	5.091	8.166	8.276	9.637	10.99	12.38
15	1.762	4.073	6.368	7.908	10.58	10.83	12.48	14.30	16.11
21	2.021	4.735	7.463	10.02	12.95	14.29	15.93	17.55	19.81
27	2.335	5.430	8.458	11.42	15.36	18.10	20.17	22.27	24.32
33	2.608	6.081	9.497	12.85	17.14	21.25	24.01	26.52	28.99
39	2.897	6.764	10.56	14.25	18.96	23.68	27.78	31.19	34.10
45	3.186	7.428	11.62	15.66	20.83	26.11	30.65	35.40	39.09
51	3.452	8.108	12.66	17.09	22.58	28.57	33.55	38.44	43.40

Tabla 5.3: Tiempo en milisegundos en realizar una ejecución del algoritmo variando el número de fuentes y el número de altavoces.

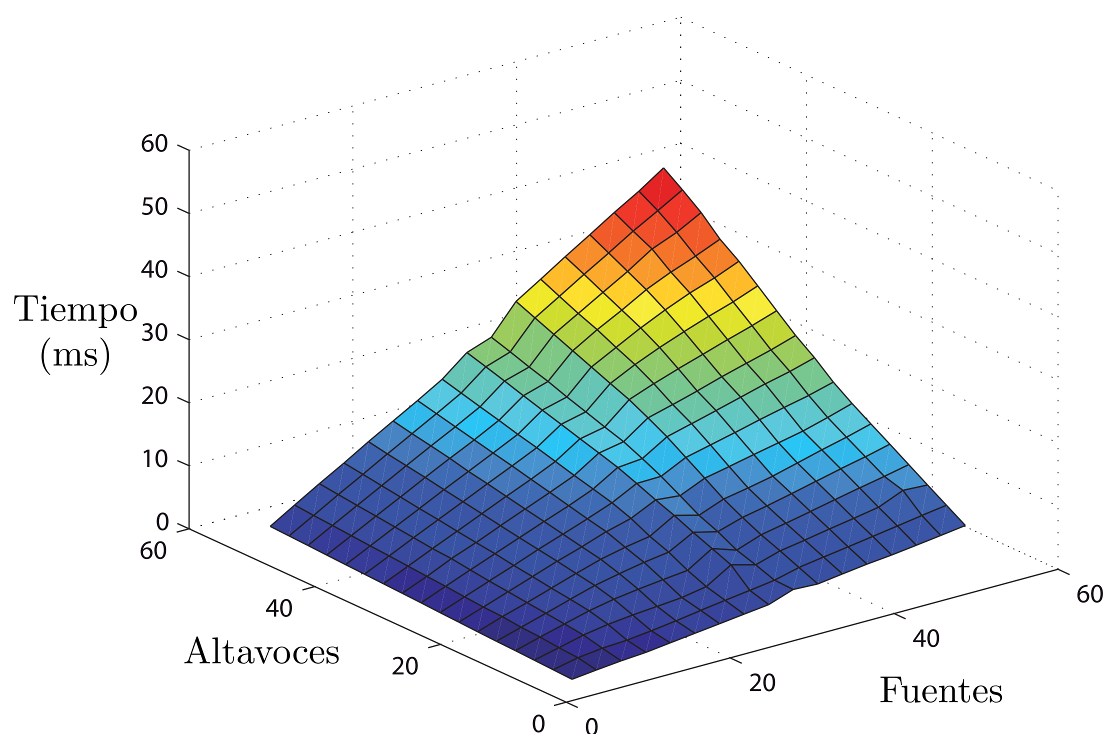


Figura 5.5: Tiempo en ms de ejecución variando el número de fuentes y altavoces

Comparación CPU

En CPU, se ha implementado el mismo algoritmo siguiendo las mismas pautas: Formación de la matriz, realización de FFT de filtro y matriz de datos, multiplicación punto

a punto y realización de la IFFT. Se han ejecutado pruebas para las siguientes configuraciones Tabla 5.4:

Fuentes	**Altavoces**	Tiempo-GPU(ms)	Tiempo-CPU (ms)	Speed-Up
3	**15**	1.762	21.383	12.135
3	**33**	2.608	44.143	16.920
3	**39**	2.897	51.800	17.880
3	**45**	3.186	60.034	18.841
9	**21**	4.735	72.348	15.277
9	**51**	8.108	123.155	15.188
15	**33**	9.497	170.760	17.979
15	**45**	11.626	228.207	19.628
21	**15**	7.908	79.583	10.061
21	**21**	10.024	155.375	15.499
27	**15**	10.582	141.088	13.332
51	**51**	43.406	554.936	12.784

Tabla 5.4: Comparación de tiempos entre diferentes ejecuciones en GPU y CPU.

La Tabla 5.4 muestra el speedup relativo obtenido, proporcionando una orientación sobre los resultados que se pueden llegar a obtener.

5.4.2. Audio multicanal en tiempo real

En una aplicación de audio multicanal nos llegan muestras por canal cada 1/44100 segundos. Todas las $h_{ij}[n]$ utilizadas tienen una longitud de M=201 coeficientes, luego en cada fila de la matriz, habrán 1024-(201-1)=824 muestras. Al tener 8 filas por canal, en total cada canal rellenará un buffer de 6592 muestras. Teniendo en cuenta la tasa de llegada de muestras, cada canal llenará su buffer de muestras en 149.48 ms. Este tiempo de relleno dependerá exclusivamente de la tasa de muestras del sistema de audio, siendo por tanto independiente del número de canales.

El valor alcanzado por el llenado de la matriz supera en tiempo al requerido por las pruebas realizadas anteriormente que involucraban 51 fuentes y 51 altavoces. Así que, continuamos ejecutando el algoritmo del apartado anterior aumentando número de fuentes y de altavoces, probando con diferentes combinaciones. Algunos de los valores temporales se encuentran recogidos en la Tabla 5.5:

Paramos ahora de ejecutar el algoritmo a pesar de no haber superado los 149.48ms pues la llamada a la librería CUFFT cuando trata de realizar la FFT de una matriz de 64 fuentes x 8 filas/fuente, es decir, 512 FFT en paralelo, se satura y da error interno. Por tanto, hemos encontrado un límite en el número de las fuentes, 63. Sin embargo podemos seguir aumentando el número de salidas (Tabla 5.6):

Alt \ Fnt	**54**	**57**	**60**	**63**
54	48.100	50.271	52.501	54.556
63	53.204	57.097	59.899	62.998
72	60.034	63.225	66.608	69.943
81	65.615	69.449	73.268	76.529
90	71.693	75.678	79.489	83.428
99	77.463	81.772	86.049	90.224

Tabla 5.5: Tiempo en milisegundos en realizar una ejecución del algoritmo variando el número de fuentes y el número de altavoces.

Alt \ Fnt	**63**
101	91.674
111	99.376
121	106.948
131	114.540
141	122.301
151	129.591
161	137.123
171	145.190
175	148.039
176	**148.840**
177	149.640
179	151.056
187	157.086
197	164.863

Tabla 5.6: Tiempo en milisegundos en realizar una ejecución del algoritmo con 63 fuentes variando el número de altavoces.

Por tanto, de acuerdo con la configuración realizada y tal y como muestra la Tabla 5.6, nuestra aplicación multicanal será capaz de soportar hasta 63 fuentes y 176 altavoces, siempre y cuando exista un flujo continuo de muestras en calidad CD, es decir, con fs=44.1 kHz.

5.4.3. Conclusiones

En este capítulo hemos implementado sobre un hardware físico el algoritmo desarrollado en el capítulo anterior. Hemos analizado como se han ido realizando diferentes implementaciones, de forma que el algoritmo iba mejorando de implementación en implementación culminando con el análisis de las prestaciones de una aplicación multicanal.

De este análisis hemos concluido que el speed-up obtenido por una GPU sobre una CPU puede estar relacionado con la cantidad de SM que tiene dicha GPU. Por otra parte, desde el punto de vista de señal, hemos comprobado que la GPU GTS-360 es capaz ejecutar aplicaciones multicanal donde intervengan hasta 63 fuentes y 176 altavoces, mientras que en una CPU, estaríamos en torno a 21 fuentes posibles (Tabla 5.4). Por eso, la GPU es una herramienta que se puede explotar para realizar aplicaciones de audio multicanal, tal y como veremos en el capítulo siguiente donde se implementará una de ellas.

Capítulo 6

Implementación de un Cancelador de Crosstalk

Examinadas la prestaciones de una aplicación de audio multicanal, en este capítulo se pretende implementar un caso particular de éstas, un Cancelador de Crosstalk. Esta aplicación está formada por dos fuentes y dos altavoces y trata, como veremos a lo largo del capítulo, de eliminar la influencia entre diferentes canales auditivos.

6.1. Introducción

Dentro del procesado de audio, uno de los campos con aplicaciones más interesantes es la reproducción de audio envolvente. En este campo, una de las técnicas más empleadas es la reproducción binaural. Su principio básico consiste en reproducir la misma presión sonora a cada oído del oyente que la que éste tendría si se encontrase en el lugar original de la grabación, dejando sin alterar las características de la señal que permiten una localización espacial de las fuentes. Dadas las características especiales de grabación que se requieren para estas señales binaurales, a la hora de reproducirlas se deben usar auriculares si se desea preservar la información espacial contenida en ellas. Si las reproducimos a través de altavoces, estaremos sometidos al consecuente crosstalk de los sistemas estéreo, donde a cada oído llega información de su altavoz correspondiente y también del opuesto. Con la finalidad de evitar esta situación, se pueden diseñar unos filtros canceladores que permitan al oyente tener presente en cada oído sólo la señal deseada.

Una vez calculados estos filtros mediante alguna de las diversas técnicas existentes, implementaremos el sistema sobre una GPU.

6.2. Planteamiento del problema

En un sistema estéreo de reproducción mediante altavoces, el oyente recibe señales en cada oído procedentes de ambos, tal y como podemos apreciar en la figura 6.1

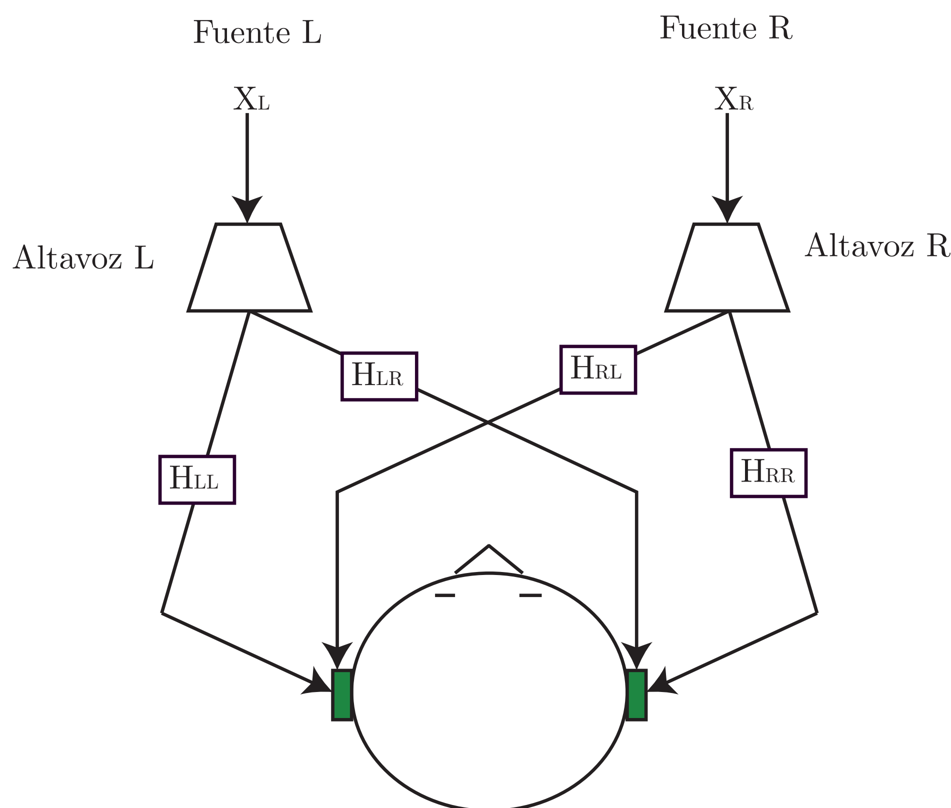


Figura 6.1: Especificación del problema crosstalk

Generalmente el sistema se modela con cuatro filtros, de forma que la señal de cada altavoz pasa por dos de ellos. Uno de estos filtros nos dará la respuesta directa al oído correspondiente a ese altavoz y el otro proporcionará el término cruzado.

Estos términos cruzados que se presentan en la figura son los causantes del efecto de crosstalk, el cual se manifiesta impidiendo que las fuentes virtuales de sonido sean situadas espacialmente. Además, este efecto también degrada la imagen virtual sonora limitándola a los confines de la ubicación espacial física de los altavoces, mediante el efecto de precedencia o de Haas [19]

La figura 6.2 muestra un Cancelador de Crosstalk implementado mediante cuatro filtros, tal y como se comenzaron a describir en la patente del año 1962, de Atal y Schroeder, publicado más tarde en [20]

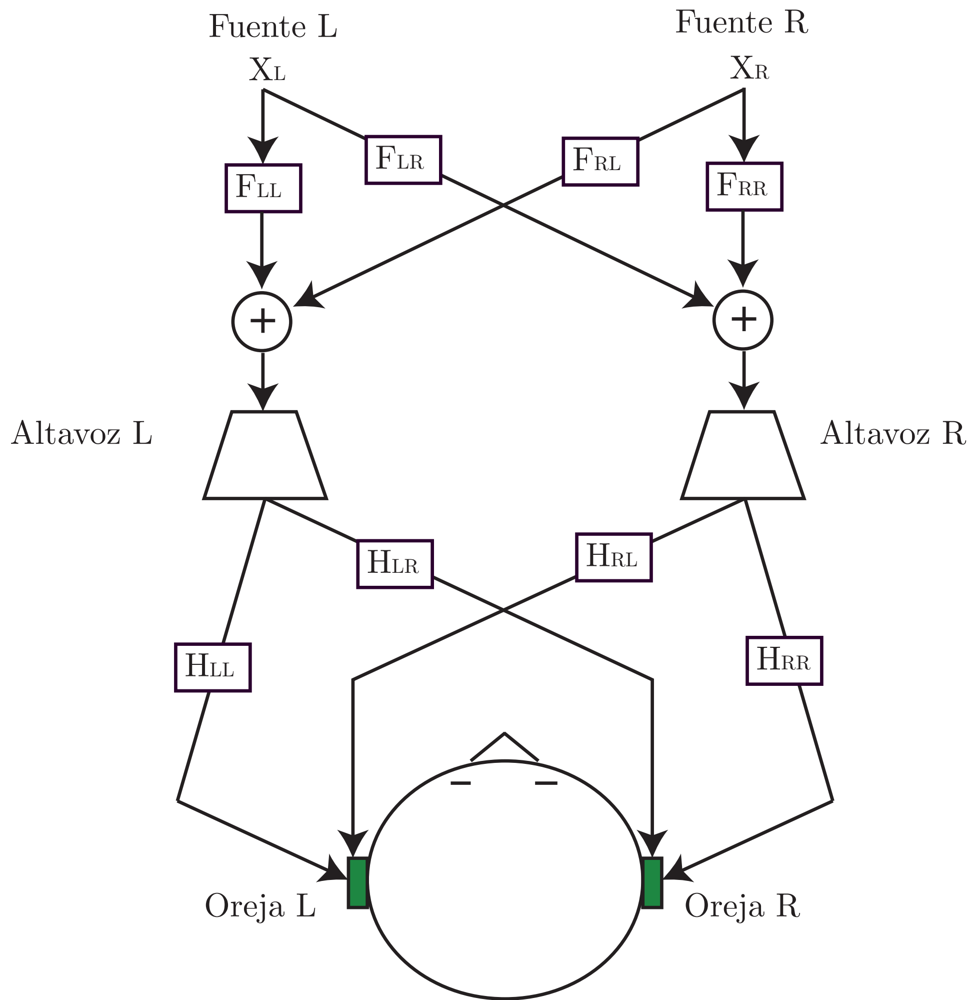


Figura 6.2: Especificación del problema crosstalk

Analizando para la figura 6.2 para la Oreja Izquierda, observamos que la señal que llega a ésta es:

$$L = (X_L * F_{LL} + X_R * F_{RL}) * H_{LL} + (X_L * F_{LR} + X_R * F_{RR}) * H_{RL} \tag{6.1}$$

Poniendo la ecuación anterior en función de las dos fuentes:

$$L = X_L * (F_{LL} * H_{LL} + F_{RL} * H_{RL}) + X_R * (F_{RL} * H_{LL} + F_{RR} * H_{RL}) \tag{6.2}$$

Una aplicación crosstalk, consistirá en diseñar un banco de filtros F_{RL}, F_{RL}, F_{RL} y F_{RL} de forma que:

$$F_{LL} * H_{LL} + F_{LR} * H_{RL} = 1 \tag{6.3}$$
$$F_{RL} * H_{LL} + F_{RR} * H_{RL} = 0 \tag{6.4}$$

De forma análoga se operará con la Oreja derecha

6.3. Herramientas lógicas y físicas utilizadas

Una vez planteado el problema, debemos afrontar la implementación. Para ello, debemos medir las respuestas al impulso de la sala donde vayamos a ejecutar la aplicación crosstalk utilizando el hardware cuyos altavoces serán utilizados para la reproducción del sonido binaural. Posteriormente, se programará dicha aplicación utilizando bibliotecas de audio que permitirá acceder a la tarjeta de sonido del propio computador.

6.3.1. Medida de las respuestas al impulso

Para medir las respuestas al impulso, hemos utilizado el software diseñado por el Grupo de Tratamiento de Audio y Comunicaciones [21], del Instituto de las Telecomunicaciones y Aplicaciones Multimedia [22], en la Universidad Politécnica de Valencia. Este software, llamado Soundtenaz, se encargará de medir la respuesta del camino de la señal sonora, que sale de los altavoces hasta que ésta es captada por un micrófono. Para conseguir esto, utilizamos un maniquí que lleva un micrófono en cada oreja. Colocamos a éste a una distancia de 90 cm del computador donde se encuentran los altavoces (figura 6.3).

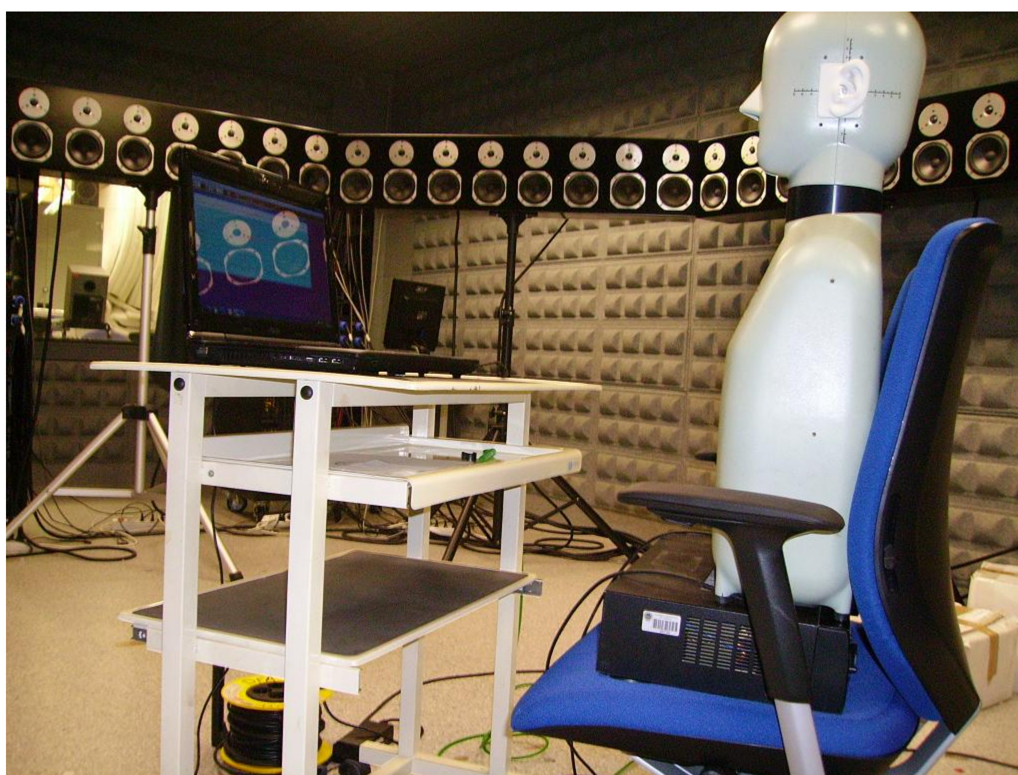


Figura 6.3: Medida de las respuestas al impulso

Con el programa Soundtenax, reproducimos por el altavoz derecho un barrido de frecuencias (figura 6.4) y lo grabamos primero con el micrófono de la oreja izquierda, y

posteriormente con el de la oreja derecha. De esta manera, habremos medido el H_{LR} y el H_{RR} (figura 6.5).

Figura 6.4: Barrido en frecuencia

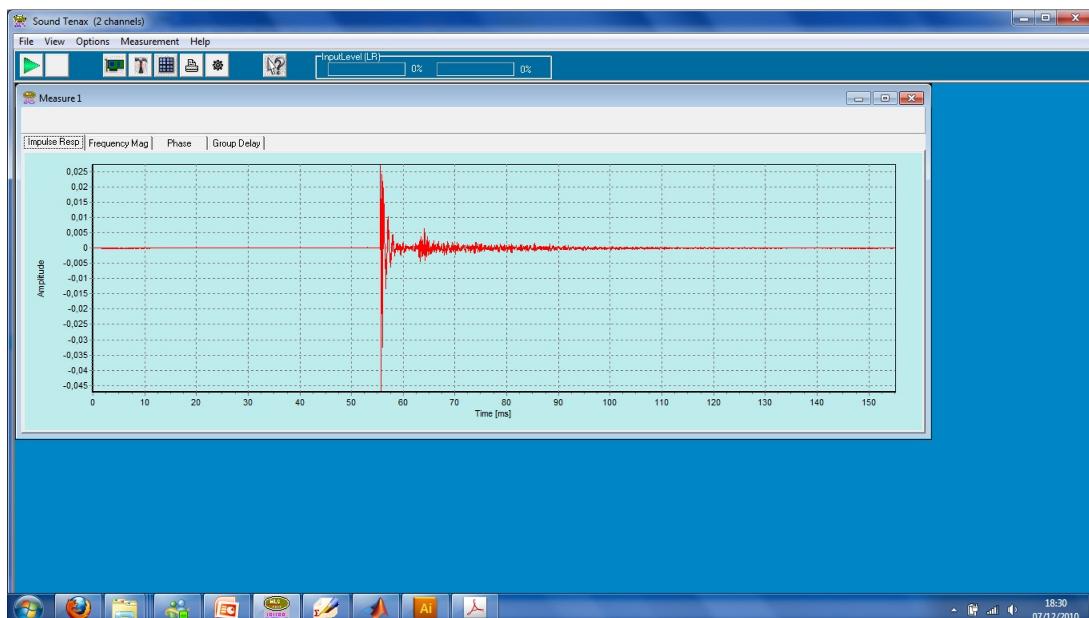


Figura 6.5: Respuesta al impulso medida

Repetimos el mismo proceso con el altavoz izquierdo y obtendremos la medida de H_{RL} y el H_{LL}. Posteriormente calculamos los filtros inversos en matlab [23], y simulamos para ver el nivel de señal que llegaría a cada oído (figura 6.6 y figura 6.7) , realizando la convolución de todos los caminos posibles para comprobar la eficiencia de los filtros.

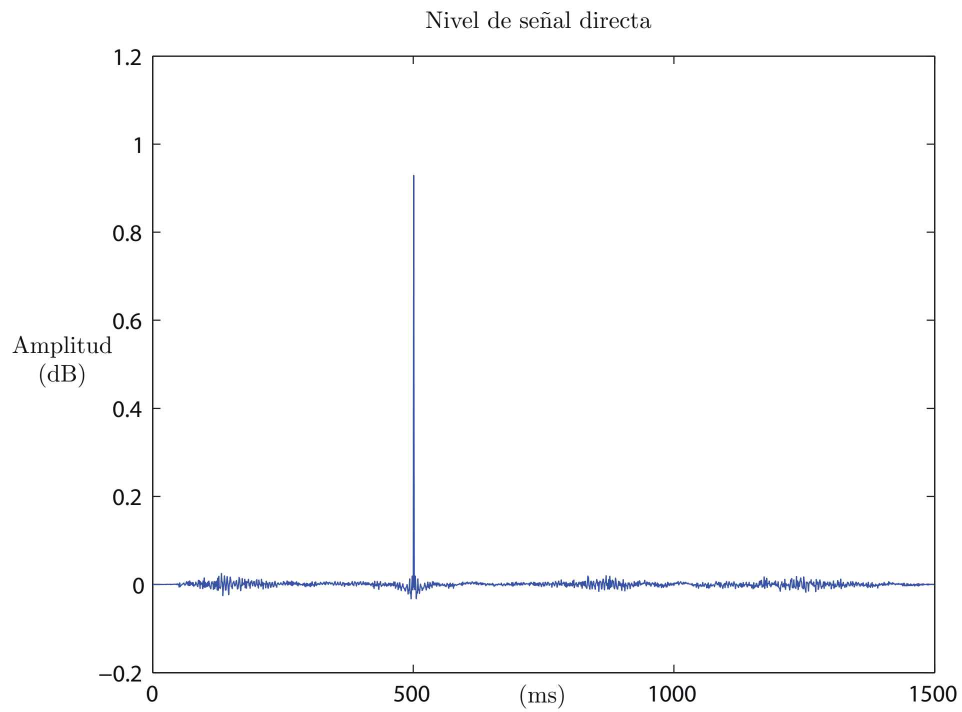


Figura 6.6: Nivel de señal con altavoz-oreja directa

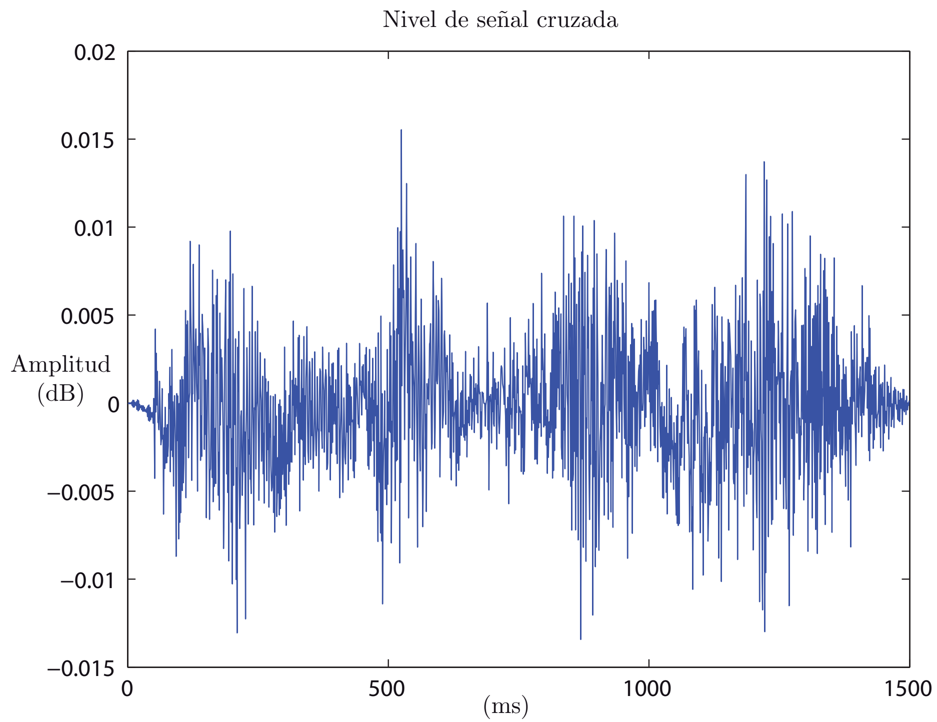


Figura 6.7: Nivel de señal con altavoz-oreja cruzada

Se puede observar que al las contribuciones de señal directa (altavoz izquierdo - oreja izquierda) llegan a un nivel de señal alto en torno a 1 dB, mientras que las contribuciones cruzadas (altavoz derecho - oreja izquierda), son de un nivel mucho más bajo, en torno a 0.015 dB.

6.3.2. Preparación del sistema

Audio Stream Input/Output (ASIO) es un protocolo de ordenador para audio digital, que provee una baja latencia y una interfaz de alta fidelidad entre el software, es decir, la aplicación, el hardware y la tarjeta de sonido. ASIO fue creado por Steinberg [24] para poder comunicar con cualquier tarjeta de sonido que funcionara a través de drivers ASIO. La ventaja de usar ASIO es que el propio Steinberg elaboró un SDK (*Software development Kit*) (figura 6.8) que puede ser descargado gratuitamente desde su página web, y que permite poder realizar aplicaciones que manipulen las salidas y entradas de sonido de un hardware que funcionara con drivers ASIO.

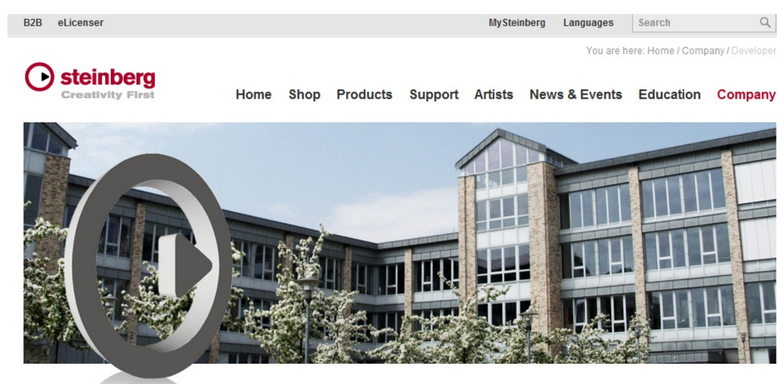


Figura 6.8: http://www.steinberg.net/en/company/developer.html

La manera de funcionar de ASIO consiste en 4 buffers, 2 de entrada y 2 de salida. Los buffers de entrada están relacionados con micrófonos mientras que los de salida con altavoces. La tarjeta de sonido permitirá que el programador escriba en el buffer de salida 1 y/o lea en el buffer de entrada 1, mientras la tarjeta de sonido reproduce lo escrito en el buffer de salida 2 y/o captura nuevas muestras en el buffer de entrada 2. Cada cierto tiempo (parámetro programable), la tarjeta interrumpe el programa y se produce un intercambio de de buffers.

En la aplicación crosstalk interesa observar el comportamiento de los buffers de salida, pues deberemos escribir las muestras ya filtradas por la GPU en el buffer de salida para que la tarjeta de driver ASIO pueda reproducirlas.

Sin embargo, no todos los computadores personales cuentan con tarjetas que tengan un modus operandi como ASIO. De hecho, lo normal, es que cada tarjeta de sonido tenga su propio driver y tenga su propia manera de funcionar. Por eso, existen unos drivers, llamados ASIO4ALL que hacen de capa intermedia, es decir, permiten al programador

manipular las entradas y salidas de cualquier dispositivo como si fuera un dispositivo ASIO.

Precisamente, a la hora de preparar nuestra aplicación Cancelador de Crosstalk, necesitaremos estas Herramientas: SDK de Steinberg y ASIO4ALL.

6.4. Demostración y Pruebas

Para hacer funcionar nuestra aplicación multicanal sobre una GPU y hacerla funcionar en tiempo real, debemos utilizar tanto el SDK de Steinberg [24] como el SDK de proporcionado por NVIDIA [15], y trataremos de hacer un proyecto único.

En nuestro caso, no estaremos interesados en las entradas, es decir, en los micrófonos, pues esta aplicación trata de cancelar las contribuciones procedentes de otros altavoces, aunque no se descarta poder utilizar una entrada de voz en tiempo real para futuras aplicaciones. Las fuentes sonoras a reproducir serán leídas de diferentes archivos wavs, serán trasladados a la GPU, filtradas de acuerdo a los coeficientes de los filtros calculados anteriormente, devueltas a la CPU y por último, escritas en el buffer de escritura para que sean reproducidas posteriormente por los altavoces del computador (figura 6.9).

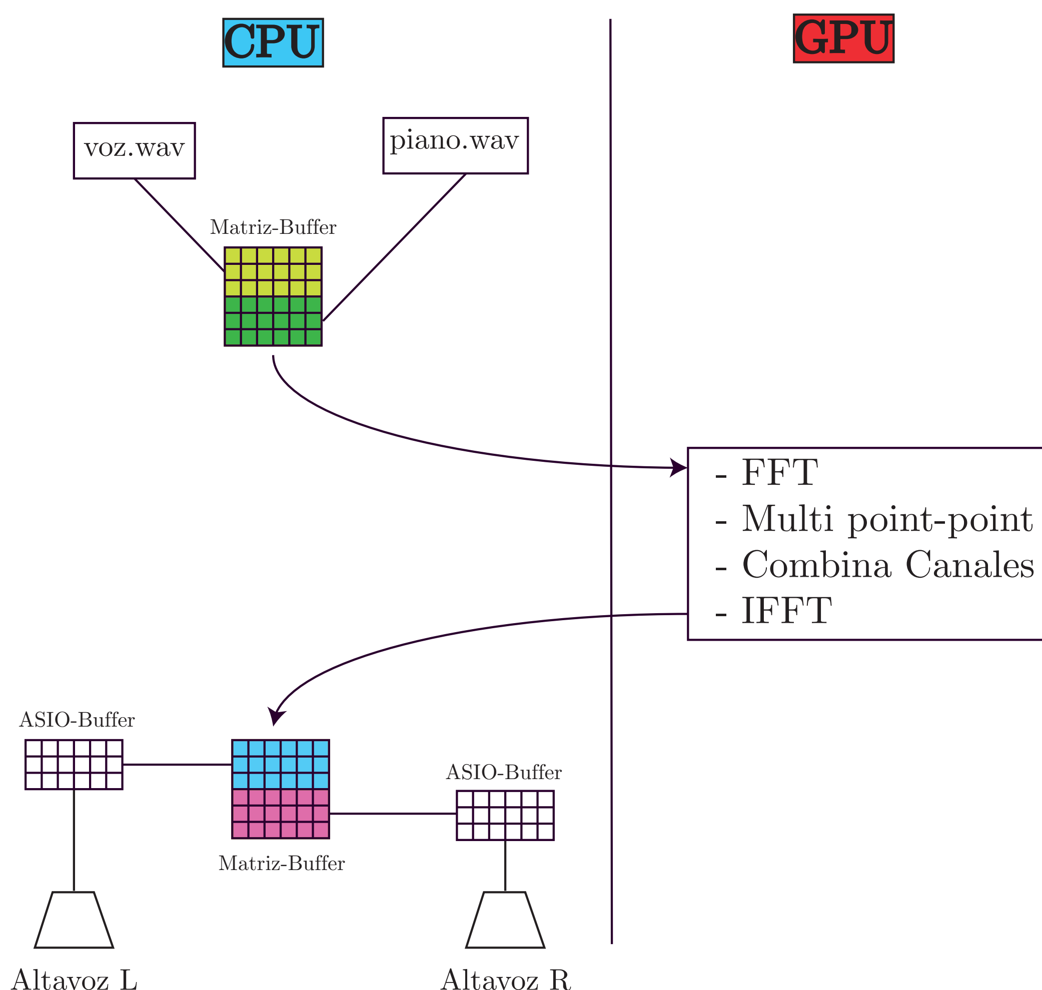


Figura 6.9: Aplicación con 2 fuentes sonoras

6.4.1. Convolución multicanal

La primera prueba que será mostrada en la lectura de la tesis de máster consistirá en comprobar como la GPU hace el procesado paralelo de los dos canales independientes en tiempo real. Se utilizarán dos fuentes: voz.wav y piano.wav . Posteriormente, ejecutaremos el mismo algoritmo sobre CPU y comprobaremos a través del Administrador de Tareas, como el uso de la GPU reduce en un 10 % aproximadamente los recursos de la CPU como puede verse en [25].

6.4.2. Cancelador de Crosstalk

Se utilizará las mismas fuentes y colocándose a una distancia de 90 cm, el oyente podrá comprobar como el sonido de la fuente izquierda alcanza la oreja izquierda, mientras que el sonido de la fuente derecha, alcanza la oreja derecha. Es importante resaltar, que el cambio de sala provocará que no se pueda apreciar totalmente el efecto del sonido

binaural, pues el efecto rebote en las paredes influirá en las contribuciones de señal que alcanzan las orejas.

Capítulo 7

Conclusiones

Esta tesis de máster demuestra que el uso de las GPU sirve como herramienta para desarrollar aplicaciones que giran en torno al procesado de audio multicanal. Se ha implementado un algoritmo de convolución masiva sobre una tarjeta gráfica que puede funcionar en tiempo real, así como un algoritmo que sirve como base para diferentes aplicaciones de sonido espacial, como pueda ser Sonido 3D, Wave Field Síntesis, Cancelación Crosstalk. En esta implementación se ha logrado alcanzar prestaciones en sistemas de audio difíciles de conseguir, como es el hecho de utilizar una tarjeta gráfica de un computador portatil, la GTS-360M, para poder llevar a cabo una aplicación multicanal donde intervengan hasta 63 altavoces. Por otra parte, se ha logrado implementar una aplicación crosstalk que funciona en tiempo real y que resuelve el problema del sonido binaural, reduciendo aproximadamente en un 10 % los recursos usados por una CPU.

7.1. Líneas Futuras

Para un futuro, quedan varios aspectos con los que seguir trabajando, como puedan ser: uso de una tarjeta FERMI que permite totalmente el solapamiento entre transferencia de datos y ejecución del kernel, permitiendo incluso transferencia de datos en ambos sentidos a la vez. Esto permitirá obtener mejores prestaciones ,pues podremos lanzar varios $kernels$ concurrentemente, y llegar incluso a realizar una aplicación de sonido 3D para un escenario de mayores dimensiones como pueda ser un teatro o un parque de atracciones.

Por otra parte, desde el punto de vista de computación en altas prestaciones, está previsto realizar un estudio más exhaustivo de la ganancia proporcionada entre la GPU y la CPU, utilizando para ello bibliotecas FFT optimizadas en GPU.

7.2. Aportaciones

El estudio realizado en esta tesis de máster, ha quedado reflejado en varias publicaciones de carácter nacional e internacional (Artículos completos en Anexos):

7.2.1. Revistas Internacionales

Autores: J.A.Belloch, A. Gonzalez, F.J.Martínez-Zaldívar, A.M.Vidal
Título: Real-Time massive convolution for audio applications on GPU
Ref.: Journal of Supercomputing.
Submitted

7.2.2. Revistas Nacionales

Autores: A.Gonzalez, J.A.Belloch, F.J.Martínez, P.Alonso, V.M.García, E.S.Quintana-Ortí, A.Remón, A.M.Vidal
Título: The Impact of the Multi-core Revolution on Signal Processing.
Ref.: Waves (iTeAM Research Journal) ISSN 1889-8297.
Clave: A Volumen: 2 Páginas: 74-85 Fecha: 2010

Autores: A.Gonzalez, J.A.Belloch, G.Piñero, J.Llorente, M.Ferrer, S.Roger, C.Roig, F.J.Martínez, M. De Diego, P.Alonso, V,M.García, E.S.Quintana-Ortí, A.Remón, A.M.Vidal
Título: Applications of Multi-core and GPU Architectures on Signal Processing: Case Studies.
Ref.: Waves (iTeAM Research Journal) ISSN 1889-8297.
Clave: A Volumen: 2 Páginas: 86-96 Fecha: 2010

7.2.3. Congresos Internacionales

Autores: J.A.Belloch, A. González, F.J.Martínez-Zaldívar, A.M.Vidal
Título: Multichannel acoustic signal processing on GPU
Congreso: 10th International Conference on Computational and Mathematical Methods in Science and Engineering– CMMSE'10
Ref.:ISBN: 13:978-84-613-5510-5 Volumen: 1 Páginas: 181-187 Fecha: Junio 2010
Lugar de Realización: Almería

Autores: J.A.Belloch, A. Gonzalez, F.J.Martínez-Zaldívar, A.M.Vidal
Título: Real-time Multichannel Audio Convolution
Congreso: GPU Technology Conference 2010
Lugar de Realización: San Jose (California) Fecha: septiembre 2010

Autores: J. Lorente, G. Piñero, A. M. Vidal, J. A. Belloch, A. González
Título: Parallelization of Beamforming Design and Filtering for Microphone Array Applications

Congreso:: Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP 2011).
Lugar celebración: Praga, República Checa. Fecha: a realizar en Mayo 2011
Submitted

Bibliografía

[1] E. Torick. Highlights in the history of multichannel sound. *J. Audio. Eng. Soc.*, 46(5):27–31, 1998.

[2] F. Orduña-Bustamante P.A. Nelson and H. Hamada. Multichannel signal processing techniques in the reproduction of sound. *J. Audio. Eng. Soc.*, 44(11):973–989, November 1996.

[3] Y. Huang and Eds J. Benesty. *Audio Signal Processing for the Next-Generation Multimedia Communication Systems*. 2004.

[4] Samir S. Soliman and Mandyam D. Srinath. *Continuous and Discrete Signals and Systems*. 1997.

[5] A. Gonzalez, J.A. Belloch, F.J. Martinez-Zaldivar, P. Alonso, V. Garcia, E.S. Quintana-Orti, A Remon, and A.M.Vidal. The impact of the multi-core revolution on signal processing. *Waves ITEAM*, (2):64–75, November 2010.

[6] What is CUDA? *online at: http://www.nvidia.es/object/what_is_cuda_new_es.html*.

[7] NVIDIA CUDA C Best Practices Guide. *online at: http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf*.

[8] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors*. 2010.

[9] What is gpgpu? *online at: http://gpgpu.org/*.

[10] CUDA Enabled GPU computing products. *online at: http://www.nvidia.com/object/cuda_gpus.html*.

[11] NVIDIA CUDA C Programming Guide. *online at: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs /CUDA_C_Programming_Guide.pdf*.

[12] NVIDIA CUDA C Programming Guide release 2.3. *online at: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/ NVIDIA_CUDA_Programming_Guide_2.3.pdf*.

[13] NVIDIA's Next Generation: FERMI. *online at: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA _Fermi_Compute_Architecture_Whitepaper.pdf*.

[14] Alan S. Willsky with S. Hamid Nawab Alan V. Oppenheim. *Signals and Systems*. 1996.

[15] NVIDIA DEVELOPER ZONE. *online at: http://developer.nvidia.com/object/ cuda_3_2_downloads.html*.

[16] NVIDIA library CUFFT. *online at: http://developer.download.nvidia.com/compute/ cuda/3_2_prod/toolkit/docs/CUFFT_Library.pdf*.

[17] J.A. Belloch, A. M. Vidal, F.J.Martínez-Zaldívar, and A. Gonzalez. Multichannel acoustic signal processing on gpu. *Procedings of the 10th International Conference on Computational and Mathematical Methods in Science and Engineering*, 1:181–187, June 2010.

[18] J.A. Belloch, A. Gonzalez, F.J. Martínez-Zaldívar, and A. M. Vidal. Real-time massive convolution for audio applications on gpu. *Journal of Supercomputing*, Submitted 2010.

[19] M.B. Gardner. Historical background of the haas and/or precedence effect. *J. Acoust. Soc. Am.*, 43(6):1243–1248, 1968.

[20] M.B. Schroeder. Models of hearing. *Proc. IEEE*, 9(63):1332–1350, 1975.

[21] Grupo de Tratamiento de Audio y Telecomunicaciones. *online at: http://www.gtac.upv.es/*.

[22] Instituto de las Telecomunicaciones y Aplicaciones Multimedia. *online at: http://www.iteam.upv.es/*.

[23] Matlab. *online at: http://www.mathworks.com/products/matlab/*.

[24] Steinberg Company. *online at: http://www.steinberg.net/en/company/developer.html*.

[25] J.A. Belloch, A. M. Vidal, F.J. Martínez-Zaldívar, and A. Gonzalez. Real-time multichannel audio convolution. *GPU Technology Conference 2010*, September 2010.

Noname manuscript No.
(will be inserted by the editor)

Real-Time massive convolution for audio applications on GPU

Massive convolution on GPU

Jose A. Belloch · Alberto Gonzalez ·
F.J. Martínez-Zaldívar · Antonio M. Vidal

Received: date / Accepted: date

Abstract Massive convolution is the basic operation in multichannel acoustic signal processing. This field has experienced a major development in recent years. One reason for this has been the increase in the number of sound sources used in playback applications available to users. Another reason is the growing need to incorporate new effects and to improve the hearing experience [1]. Massive convolution requires high computing capacity. GPU offers the possibility of parallelizing these operations. This allows us to obtain the processing result in much less time and to free up CPU resources. One important aspect lies in the possibility of overlapping the transfer of data from CPU to GPU and vice versa with the computation, in order to carry out real-time applications. Thus, a synthesis of 3D sound scenes could be achieved with only a peer-to-peer music streaming environment using a simple GPU in your computer, while the CPU in the computer is being used for other tasks. Nowadays, these effects are obtained in theaters or funfairs at a very high cost, requiring a large quantity of resources. Thus, our work focuses on two mains points: to describe an efficient massive convolution implementation and to incorporate this task to real-time multichannel-sound applications.

Keywords Massive convolution · Multichannel audio processing · FFT · GPU ·

1 Introduction

A basic operation in multichannel acoustic signal processing is Massive Convolution. It consists of carrying out simultaneously many convolutions of different audio channels. This provides a multichannel convolution that allows to achieve with different filters well known acoustic effects like: 3D spatial sound, crosstalk cancellation, room compensation [2], loudspeakers equalization, etc. [3].

Jose Antonio Belloch, Alberto Gonzalez, F.J. Martínez-Zaldívar
Institute of Telecommunications and Multimedia Applications
Universidad Politecnica de Valencia Tel.: +34-96-3877007 ext 73008
E-mail: jobelrod@iteam.upv.es, {agonzal,fjmartin}@dcom.upv.es

Antonio M. Vidal
INCO2-DSIC, Universidad Politecnica de Valencia (Spain)
E-mail: avidal@dsic.upv.es

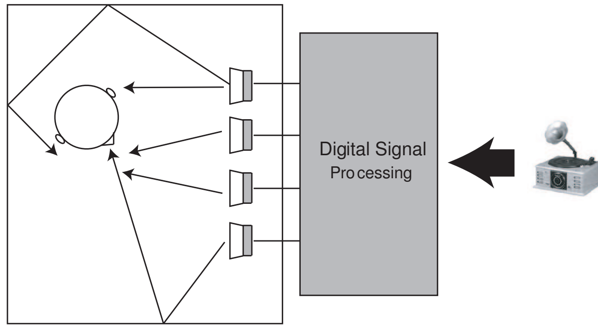


Fig. 1 Different filters applied to a sound source for audio reproduction through loudspeakers in a room

Up to now, most of these effects could be achieved only in theaters or funfairs, always using very powerful computers and consuming a large amount of energy. The use of GPU (Graphics Processing Unit) makes it possible to achieve these amazing effects saving energy, and also, to obtain them in a personal computer even faster, as can be seen at [4] and [5], where some experiments comparing performance convolution in CPU and GPU have already been carried out using OpenGL [6].

However, in spite of obtaining good performance using GPU, the fact of transferring data from/to the CPU to/from GPU prevents the running of real-time applications. In this article, an algorithm with a pipeline structure is developed, which allows to perform a massive acoustic real-time convolution. As analyzed in this article, massive convolution requires the calculation of several FFT simultaneously. There are various libraries that implement efficient FFT algorithms. They allow to obtain the Discrete Fourier Transform of a signal either in a CPU (like MKL [7] or IPP [8]) or in a GPU (like CUFFT [9] from NVIDIA whose performances have been analyzed in [10]).

CUFFT NVIDIA library [9] offers good parallelization, however, in order to implement multichannel convolution, it is important to configure a data structure suitable for exploiting parallel operations. Without that, efficient multichannel convolution in real-time would be impossible to implement. The paper is organized as follows. Section 2 describes the convolution algorithm and how it can be developed over a GPU. In Section 3, an efficient GPU implementation of massive convolution is presented. Section 4 analyzes the performance of a possible real-time application. Finally, some conclusions are presented in Section 5.

2 Multichannel convolution on GPU

Multichannel convolution consists of carrying out many convolutions of different channels simultaneously. Depending on the desired audio effect, different combinations can be required: different filters applied to a sound source (Figure 1), one filter applied to several sound sources, or different filters applied to different sound sources. In order to understand how multichannel convolution is organized, it is important to describe the one channel convolution first.

Let us consider x and input audio signal, h an acoustic filter (unit-impulse response) and y the desired output audio signal of our system. N, M and $L = N+M-1$ [11] will be the lengths of x, h and y respectively The execution of the convolution using a GPU can be illustrated in Figure 2. In spite of the parallelism in operations that GPU offers,

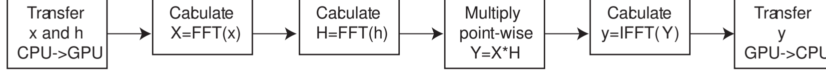


Fig. 2 Steps in order to calculate convolution of signals x and h on GPU.

the transfer time penalty prevents us from running a real time application in a GPU. Moreover, if the signal x consists of several channels, then multiple convolutions would be required. On the other hand, if a CPU is used to implement a massive convolution, all our resources would be used and no more applications could be run at the same time.

2.1 Algorithm for long signals

In a real-time application, the length of signal x can not be known a priori. Techniques are available to fragment the signal, and obtain the convolution of the whole signal from the convolution of each fragment. One of these techniques is called overlap-save [13] and it performs as follows:

1. Fragments of L samples are taken, where L will be either the next power of two, larger than M (length of h) or 512.
2. In the first fragment, the first $M-1$ samples will be padded with zeros.
3. From the second and following fragments, the first $M-1$ samples will be duplicated from the last $M-1$ samples of the previous fragment, see at the top of Figure 5.
4. Following the steps of the previous subsection, $y_0[n]$, $y_1[n]$, $y_2[n]$, ..., are obtained as the result of the convolution of $x_0[n]$, $x_1[n]$, $x_2[n]$, ..., with h respectively, see Figure 3.

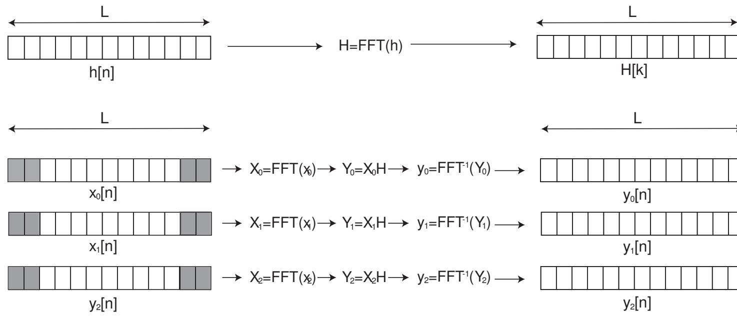


Fig. 3 Convolution of each fragment is calculated following the convolution theorem [11].

5. From each fragment result, the first $M-1$ samples will not be valid values and will therefore be eliminated, see Figure 4.

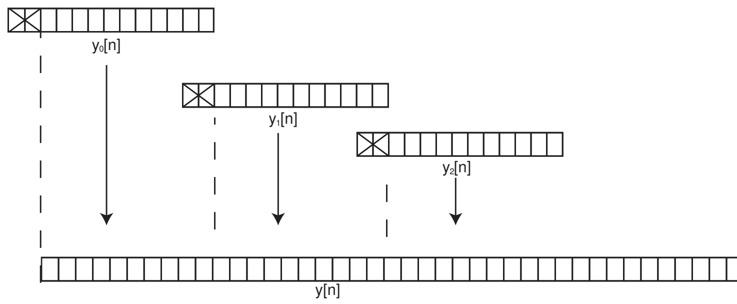


Fig. 4 As long as convolutioned fragments are obtained, output signal y is being formed.

3 Implementation on GPU

The operation described in Section 2 is applied over every signal fragment. CUFFT NVIDIA FFT library, allows multiple FFT 1D to be run at the same time, a matrix signal can be configured with all the signal parts in order to carry out as many FFTs as rows of this signal matrix. Figure 5 illustrates the formation of this signal matrix. Recent CUDA toolkit versions [12] enables the use CUFFT [9] with the property *concurrent copy and execution*. Therefore, the latency of transferring data from the CPU to the GPU and vice versa can be overlapped by computations. This will enable not only high speedup of the convolution, but also, the use of real-time applications. Therefore, the configured signal matrix of Figure 5 with R rows and L columns could be considered as a buffer, which is being built as the incoming audio samples arrive. The first $M-1$ values of one row will coincide with the last $M-1$ values of the previous row, except for the first configured matrix at the algorithm beginning whose first $M-1$ values from the first row will be zeros. The last $M-1$ samples from the last row of the

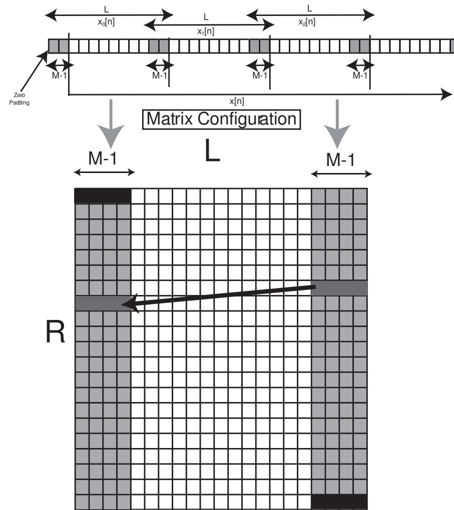


Fig. 5 A signal matrix is built from signal parts.

matrix will be kept in an internal buffer in order to occupy the first $M-1$ positions of the next matrix to be filled. The unit-impulse response h will have been sent to the GPU before sending the first matrix. As shown in Figure 3, and described in sections 2 and 3, vector h will be padded with zeros until L samples (length of each fragment of signal x), then a FFT will be carried out obtaining vector H, and finally a point-wise multiplication with each fragment of X (x in a frequency domain) will be done.

To carry out operations on GPU, since signal x is configured as a matrix, an h-matrix must be also configured. It consists of R replications of vector h. Over the GPU, FFT function from CUFFT library is applied to both matrixes, then a point-wise multiplication is done between them (Figure 6), and finally, inverse FFT function from CUFFT is applied again over the result matrix. Thus, time samples of output signal are obtained.

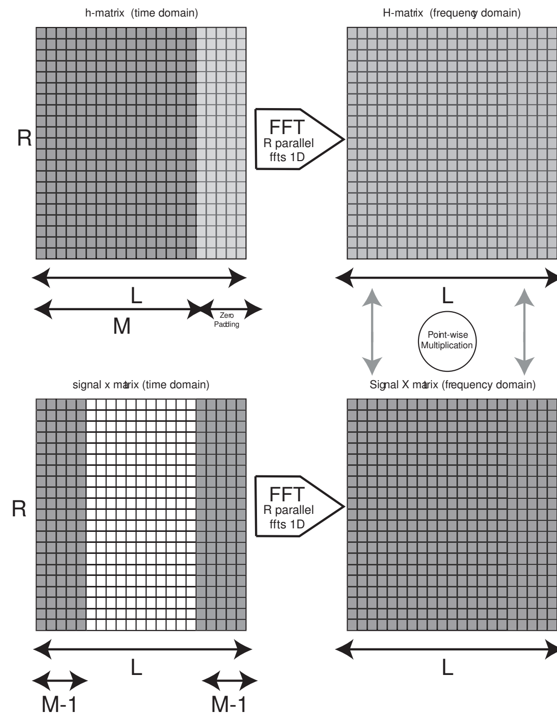


Fig. 6 FFT function from CUFFT library is applied to signal matrix and h-matrix, then a point-wise multiplication is done between them.

3.1 Scalability from one channel to multichannel

It is obvious that the hearing effects explained previously cannot be represented by either one filter or a single signal. Thus, when dealing with a stereo signal (two audio channels) or maybe with a four-channel audio signal, resources will be shared, as shown in Figure 7.

Fig. 7 Signal matrix on the left shows 2-channel resource sharing, signal matrix on the right shows 4-channel resource sharing

Table 1 Comparison between basic and pipelined algorithm

Type of Algorithm	Time
Basic convolution on GPU	13330ms
Using a pipeline configuration	625.92ms

3.2 Pipelined Algorithm

The *concurrent copy and execution* property enables multichannel convolution using a four-step pipelined model. This model uses the asynchronous transfer of matrix signals from CPU to GPU and vice versa while other tasks are executed in parallel.

At the beginning of the algorithm, vectors h are sent to the GPU where h-matrix is configured. Then, the first buffer begins to be built. We will call this buffer: A-buffer. Using asynchronous transfer, while A-buffer is sent to GPU, another buffer, B-buffer, is built simultaneously. When these two steps end, the computations described in previous subsections are carried between h-matrix and A-Buffer (signal matrix), while B-buffer is transferred from CPU to GPU, and a new buffer (C-buffer) is built. When all the previous tasks end, then a new D-buffer is built, while C-buffer is transferred from CPU to GPU, execution in GPU is carried out over B-Buffer and A-buffer is transferred back to CPU. Finally, when D-buffer is transferred, A-buffer is built again. Thus, four buffers are being used cyclically. It is important to appreciate that in order to keep the continuity of the application, A-buffer must be waiting for samples at the CPU before D-buffer is transferred to GPU, while taking into account the sample incoming rate. All these steps are illustrated in Figure 8.

4 Results

Two main tests have been carried out to verify massive convolution on GPU. The first test concerns the speed-up achieved when the pipelined algorithm of Figure 8 is compared with a basic convolution algorithm, shown in Figure 2, using a signal x and an impulse-response h made up of 176400 samples and 220 coefficients respectively. The size of the configured signal matrix x was 32 x 512 elements. Results are shown in Table 1. With a pipeline structure, the time spent can be halved.

The most significant test resolves around the number of audio channels that can be managed by a GPU to carry out a massive convolution. In a real time audio application, transfer and computation on GPU must spend less time than filling the sample's data

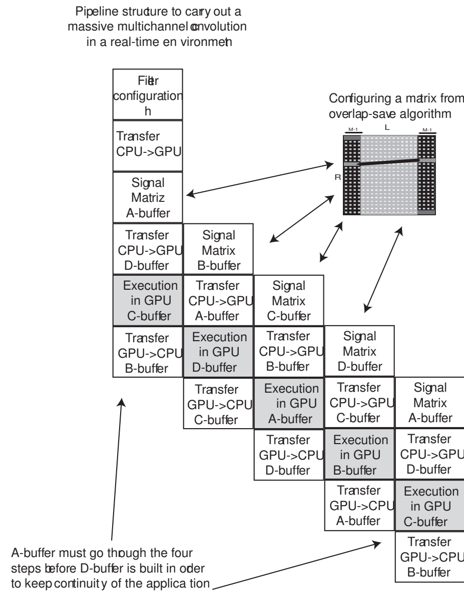


Fig. 8 Four buffer matrixes are needed in order to carry out a pipelined algorithm.

Table 2 Number of possible audio channels in the application using a matrix buffer of 512 x 32

Number of channels	Occupacy of rows per channel	Time employed filling buffer	Use of GPU (%)	Availability
1	32	212.6ms	4.4%	Yes
2	16	106.3ms	8.8%	Yes
4	8	53.15ms	17.6%	Yes
8	4	26.9ms	35.2%	Yes
16	2	13.2ms	70.5%	Yes
32	1	6.6ms	141%	No

buffer. This time depends on the rate of the incoming samples, what is known as sample frequency. CD quality has an audio sample frequency of 44.1 KHz. It means that 44100 samples per channel arrive within one second. Taking into account that one sample of one channel arrives in 1/44100 s. Depending on the buffer size, different numbers of channels will be managed by a GPU.

Using the same test signals, and taking the typical buffer size 512 x 32 elements, the time spent for transferring one matrix buffer from CPU to GPU, computed in GPU and transferred back to CPU is 9.37 ms.. Real samples in one row of the buffer matrix will be $L-(M-1)$ because first $M-1$ will be zero or duplicated. In our test, 293 samples, which arrives at 1/44100 s each. Table 2 shows that processing on GPU allows managing until 16 audio channel simultaneously using a matrix buffer of 32 x 512. If one row of the buffer were dedicated to one channel, then, executing time 9.37 would be larger than filling buffer 6.6ms. and that would not be possible.

5 Conclusions

The *concurrent copy and execution* CUDA property allows to configure a pipelined algorithm, which can be used for carrying out a massive convolution. This algorithm offers much better performance than the classical algorithm of the convolution over GPU. The main advantage is that it is a scalable algorithm, even when incoming signal x has several channels or there is more than one filter or effect to be carried out over the signals.

As the results show, dealing with 16 audio channels would require the use of many CPU-resources. With this pipelined algorithm it is clear that with only one GPU, applications like 3D spatial sound, which could need around many channels, can be achieved. Moreover the use of a single GPU provides energy saving, because the large computers used nowadays in funfairs or theaters would no longer be required to develop audio applications.

Furthermore, using GPU frees up CPU resources, providing better performance and more importantly, opening up a new way of implementing audio applications where GPUs have not previously been used before.

Acknowledgements This work was partially supported by the Spanish Ministerio de Ciencia e Innovacion (Projects TIN2008-06570-C04-02 and TEC2009-13741), Universidad Politecnica de Valencia through PAID-05-09 and Generalitat Valenciana through project PROMETEO/2009/2013

References

1. E. Torick, Highlights in the history of multichannel sound, J. Audio. Eng. Soc. Vol. 46 pag 27-31 (1998).
2. S. Spors, R. Rabenstein, W. Herbordt, Active listening room compensation for massive multichannel sound reproduction system using wave-domain adaptive filtering, J. Acoust Soc. Am., vol 122, pag 354-369, (2007)
3. Y. Huang, J.Benesty and J. Chen, Generalized crosstalk cancellation and equalization using multiple loudspeakers for 3D sound reproduction at the ears of multiple listeners, IEEE Int. Conference on Acoustics, Speech and Signal Processing page 405-408, Las Vegas, USA (2008)
4. B. Cowan, and B. Kapralos. Spatial sound for video games and virtual environments utilizing real-time GPU-based convolution. In Proceedings of the ACM FuturePlay 2008 International Conference on the Future of Game Design and Technology. Toronto, Ontario, Canada, November 3-5, (2008).
5. Brent Cowan and Bill Kapralos. GPU-Based One-Dimensional Convolution for Real-Time Spatial Sound Generation. Scholary Journals, Vol 3, No 5 (2009), ISSN 1923-2691
6. openGL: "http://www.opengl.org/"
7. MKL library: "http://software.intel.com/en-us/intel-mkl/"
8. MKL library: "http://software.intel.com/en-us/intel-ipp/"
9. CUFFT library: "http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs /CUFFT_Library_3.1.pdf"
10. P. Alonso, J.A. Belloch, A. Gonzalez, E.S. Quintana-Orti, A. Remon and A.M. Vidal, Evaluacion de bibliotecas de altas prestaciones para el calculo de la FFT en procesadores multinúcleo y GPUs, II Workshop en Aplicaciones de Nuevas Arquitecturas de Consumo y Altas Prestaciones, Freeman, Mostoles (Madrid), (2009).
11. S.S. Soliman, D.S.Mandyam and M.D. Srinath, Continuous and Discrete Signals and Systems, Prentice Hall, ISBN:0135184738 (1997)
12. CUDA Toolkit 3.1: "http://developer.nvidia.com/object/cuda_3_1_downloads.html"
13. A.V. Oppenheim A.S.Willsky and S.Hamid Nawab, Signals and Systems, Prentice Hall, ISBN:0138147574

The Impact of the Multi-core Revolution on Signal Processing

Alberto González[1], José A.Belloch[1], Francisco J. Martínez[1], Pedro Alonso[2], Víctor M. García[2], Enrique S. Quintana-Ortí[3], Alfredo Remón[3], and Antonio M. Vidal[2]
Correspondence author: agonzal@dcom.upv.es
Audio and Communications Signal Processing Group[1] (GTAC) iTEAM, Universidad Politécnica de Valencia
Department of Information Systems and Computation[2] (DSIC) Universidad Politécnica de Valencia
Department of Computer Science and Engineering[3] (ICC) Universidad Jaume I de Castellón

Abstract

This paper analyzes the influence of new multi-core and many-core architectures on Signal Processing. The article covers both the architectural design and the programming models of current general-purpose multi-core processors and graphics processors (GPU), with the goal of identifying their possibilities and impact on signal processing applications.

Keywords: Signal Processing, Multi-core processors, GPU, High Performance Computing, Parallel programming

1. Introduction

The current conception of Signal Processing is intimately linked with the type of computation required to perform the "Processing". In a recent issue of the IEEE Signal Processing Magazine [1], José F.M. Moura, president of the Signal Processing Society, noted: "As for processing, it comprises operations of representing, filtering, coding, transmitting, estimating, detecting, inferring, discovering, recognizing, synthesizing, recording, or reproducing signals by digital or analog devices, techniques, or algorithms, in the form of software, hardware, or firmware".

This definition emphasizes the strong dependence between signal processing and the computational media (digital or analogical, algorithms, hardware devices, software, etc) used to conduct it. In particular, if we focus on Digital Signal Processing, processors (in a wide sense) represent the most widespread digital devices in applications within this field.

The increase of processors performance and other digital devices has opened the possibility of addressing increasingly complex problems in a short period of time. This has been exploited both in real-time applications that are common in Signal Processing as well as other Signal Processing applications that require the management of very large data sets and which cannot be tackled within a reasonable time without the help of advanced computational tools. In summary, the advances of the hardware architecture of digital devices, including digital processors, strongly influence the techniques used and results produced in the field of Signal Processing. Considering the computational media, the following systems can be identified as the most used in Signal Processing during the past years:

General-purpose microprocessors (as those present in desktop computers, servers or high performance computers): The versatility, availability and ease of programming of these architectures have made them particularly useful in the field of Signal Processing, especially in intensive off-line applications.

Digital Signal Processors (DSP): They yield high performance as specific hardware for computationally intensive applications. They are especially appealing as components for the embedded market: devices that require intensive computing with small size, light weight, low cost and low consumption chips (GPS, mobile phones, etc.) [2].

Field Programmable Gate Arrays (FPGA): These are especially useful for real-time applications that require low weight, inexpensive, specific chips, with limited clock frequency, for highly repetitive operations (FPGA are used, for example, in space vehicles to cope with cosmic radiation), although it is difficult to use FPGAs as a general-purpose tool in a large variety of Signal Processing problems.

In the last five years, explicitly parallel systems are being accepted in all segments of the industry, including Signal Processing, in the form of multi-core processors and many-core hardware accelerators. The triple hurdles of power dissipation and consumption of air-cooled chips, little instruction-level parallelism (ILP) left to be exploited, and unchanged memory latency, combined with the desire to transform the increasing number of transistors dictated by Moore's Law into faster computers, has led the major hardware manufacturers to design multi-core processors as the primary means of increasing the performance of their products. General-purpose four-core chips from Intel and AMD are nowadays common in desktop machines, there exist six- and eight-core designs from these same vendors for the server market, and the ITRS Roadmap [3] predicts that by 2022 the number of general-purpose cores per chip will increase 100x with respect to current designs.

On the other hand, specialized (many-core) hardware with hundreds of simple cores is already available in the form of cheap, widely-spread NVIDIA and AMD/ATI graphics processors (GPU) incorporated in any standard graphics card. For example, 240 cores are embedded in NVIDIA GeForce GTX280 and, in the first quarter of 2010, the number of cores is promised to double in the upcoming NVIDIA Fermi design.

General-purpose multi-core processors (which we will refer here after as just multi-core processors) and specialized many-core accelerators like the GPU will surely impact current and future signal processing applications. First, these new hardware designs deliver an enormous high-performance computing capability at a remarkable low price, and programmers of signal processing applications will naturally want to exploit this. Second, as predicated by Herb Sutter in 2005, "The free lunch is over" [4]: Till 2004-2005 most classes of applications enjoyed free and regular performance gains, because the hardware manufacturers and computer architects reliably designed and produced ever-faster CPU. That enjoyable period is over and, although new processors yield higher performance, only those application developers who embrace parallel programming will benefit from it. In particular, Signal Processing is surely one of these applications that will be affected by the multi-core revolution.

As important as the hardware revolution may seem, it is the software that will determine the success or failure of the new products. A recent example is the IBM/Sony/Toshiba Cell B.E. processor, an innovative heterogeneous multi-core solution that did not succeed in the HPC arena mainly due to the lack of an appropriate, easy-to-use SDK-software development kit. Thus, the major hardware multi-core and many-core manufacturers dedicate considerable part of their efforts to develop and help others to create a varied ecosystem of low-level and high-level programming tools, which ease the task of software developers and, in the end, allow their designs to reach a larger number of customers.

INCO2 (www.inco2.upv.es) is a group created with the specific goal of tackling the software challenge in Signal Processing applications. INCO2 has been recognized as a research group in the Comunidad Valenciana (Spain) by the local government (PROMETEO 2009/013 project award). The research lines of INCO2 address problems of Signal Processing from an interdisciplinary perspective, providing solutions based on high performance hardware and developing algorithm design techniques that imply a modern and advanced software conception. Researchers of INCO2 have a vast experience in using parallel computing as a means of accelerating the time-to-solution and focus mainly on computers with multi-core and many-core architectures. The researchers of INCO2 are also part of the Partnership Program of NVIDIA Company, the world's leading manufacturer of graphics processing units.

The rest of the paper is organized as follows. The following two sections offer a brief description of the architectural characteristics of multi-core processors and GPU. Next, in Section 4, we discuss the possibilities of applying these architectures to the solution of Signal Processing problems, and we state logical needs and appropriate strategies needed to effectively tackle the problems. The final section of the article gathers our conclusions.

Probably the best form of appreciating the impact of the new architectures on signal processing is to analyze "possible application" and the performance reached in their solution when multi-core/GPU architectures are used. As a continuation of this work, in the paper "Applications of Multi-core/GPU architectures in signal processing: some case studies" [5] we describe several case studies that show how parallelization on multi-core/many-core architectures can be applied to specific problems

2. The multi-core approach to parallelism

A **multi-core processor** or chip multi-processor (CMP) is an integrated circuit composed of two (dual core), four (quad-core) or more independent cores. Each core is an individual processor, but the cores in a chip may share certain resources as, e.g., a given level of the cache memory; see Figure 1.

The current conception of Signal Processing is intimately linked with the type of computation required to perform the "Processing".

The increase of processors performance and other digital devices has opened the possibil-ity of addressing increasingly complex problems in real time.

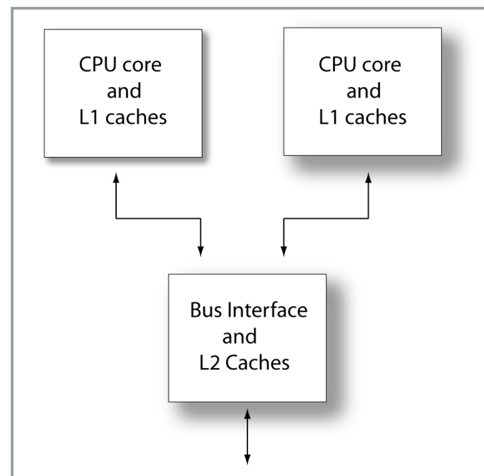


Figure 1. *Diagram of a generic dual-core multiprocessor*

A brief motivation of the multi-core development

Since the 1980s, microprocessors have dominated all computer markets, from embedded systems to servers, desktop computers and high-performance systems. Till 2004, the increasing number of transistors dictated by Moore's Law was exploited by system developers and computer architects to (respectively) reduce the scale of the chips (therefore, increasing their clock frequency) and produce more elaborated designs (e.g., with larger caches layered in multiple levels, more functional units, and, in general, capable of dynamically exploiting, i.e., at run-time, a higher amount of the instruction-level parallelism exiting in the codes).

In 2004, Intel joined all the other major hardware vendors (AMD, IBM and Sun) and declared the multi-core design as the main track to transform the gains dictated by More's Law into higher performance. The major reason for this is the limitation of the current semiconductor technology in terms of power consumption/dissipation, also known as the Power Wall. The acceleration of the clock frequency was a constant during this period: a VAX 8700 operated at 12.5 MHz while, 20 years later, an Intel Xeon reached 3.6 GHz (a factor of 290x). However, given the quadratic/cubic dependence between frequency and power dissipation of current CMOS technology, this trend came to an end: A chip operating at 5 GHz would simply melt!

Moving into the multi-core arena is not free as parallel programming must be explicitly addressed; however, this is currently recognized as the only way of pushing the performance of computer hardware, due to the combined effects of the power wall, the increasing gap between the processor and the memory speeds (the memory wall), and difficulties of finding enough parallelism in a single instructions stream to keep a single processor busy (ILP wall). Consider, e.g., that an increase of the clock frequency by 15% translates into a 2x power consumption but a potential increase in performance of only 15%. Whether this potential gain is real also depends on the ability of the programmer to hide the memory latency and the availability of more ILP in the program. On the other hand, by decreasing slightly the clock frequency, it is possible to double the number of cores in a design, maintaining the overall power consumption, and potentially doubling the potential performance gain. In this case, the potential gain is resulting from doubling the number of cores in a design is not hampered by the memory/ILP walls.

The multi-core solution is 10+ years old in the embedded market. Specific designs for mobile phones and network chips have included multiple cores for many years now. The big change is in the adoption of multi-core designs for the general-purpose market as well. Current multi-core chips for the server market include six-core AMD Opteron (model 2435, 45 nm scale, 75 W, 2.6 GHz, 128 KB L1 cache, 512 KB L2 cache, 6144 KB L3 cache), six-core Intel Xeon (model X7460, 45 nm, 130 W, 2.66 GHz, 9 MB L2 cache, 16 MB L3 cache), 8-core Sun UltraSPARC T1 "Niagara" (0.09 micron, 72 W, 1.2 GHz, 16I+8D KB L1 cache, 3 MB L2 cache), and AMD and Intel have announced, respectively, 12-core and 8-core designs for the first quarter of 2010. The number of cores is expected to double with each reduction in the integration scale (roughly, every two years), as long as Moore's Law holds.

CPU architecture

Current general-purpose multi-core processors feature basically the ISA (Instruction Set Architecture) of the corresponding uni-processor designs, with minor additions of synchronization instructions. The major consequence and advantage of sharing the ISA is the availability of legacy codes and a vast amount of programming tools for traditional uni-processor chips.

To increase performance, the processor datapath of current general-purpose processors is pipelined, so that the execution of multiple instructions can be overlapped. By splitting the processing of an instruction into a series of independent stages, with storage at the end of each step, instructions can be issued (to execution) at the processing rate of the slowest step, which is much faster than the time needed to perform all steps at once. Thus, pipelining improves the throughput of the datapath, but it does not decrease the execution time of a single instruction. The classic, simple pipelined datapath consists of four steps:

1. Fetch instruction from memory (IF).
2. Decode instruction while, simultaneously, fetch the operands from the registers (ID).
3. Execute the operation (EX).
4. Write the result back in a register (WB).

The operation of such pipelined processor is illustrated in Figure 2. Intel stressed the concept

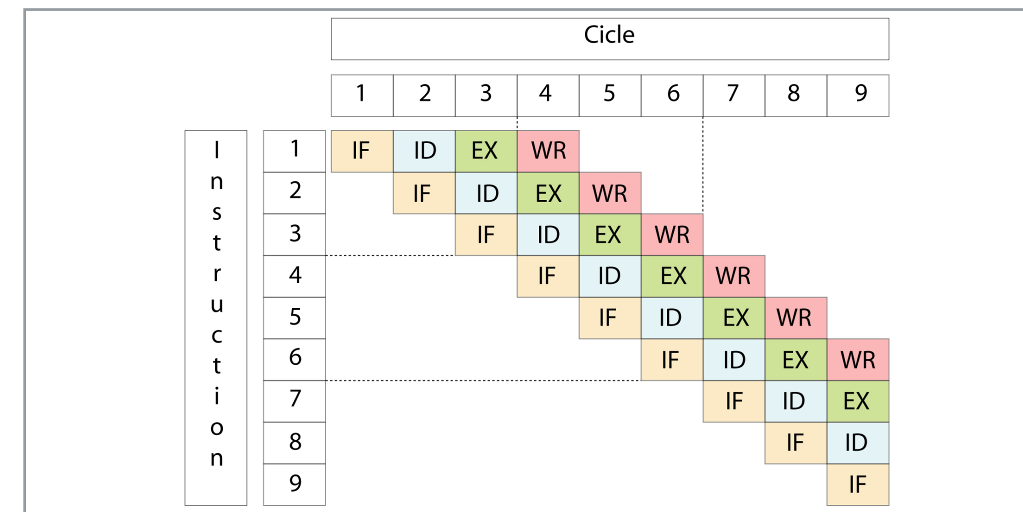


Figure 2. *Operation of a basic four-stage pipeline.*

of pipelining with 31 stages in the Prescott microarchitecture (February 2004).

The peak instruction issue rate yield by pipelined processors is 1 (instruction per cycle). To improve this performance, current processors issue more than one instruction per cycle; see Figure 3. Superscalar processors, like the Intel Xeon and the AMD Opteron multi-core designs, are the most spread class of multiple-issue processors. (VLIW processors, like the Intel Itanium2 are also multiple-issue architectures, but they issue a fixed number of operations encoded within one large instruction which explicit the parallelism among operations). Most general-purpose processors today are four and six-issue designs.

Superscalar processors detect and exploit ILP at run-time (dynamic scheduling), reordering the flow of instructions (out-of-order) to overcome the stalls due data hazards (i.e., data dependencies in the instruction flow). To be effective, this needs to be combined with a hardware-based speculation mechanism, which hides the stalls due to control hazards (due to branches in the instruction flow). The result is a complex hardware design, which requires substantial die area, and often is not power efficient. Because out-of-order multiple-issue processors are large and power hungry, few of them can be combined in a single chip. Thus, the current trend in multi-core design is to use simpler cores, with limited issue (e.g., 2-issue), with in-order scheduling, and moderate clock frequency.

Memory system

The memory system plays an important role in multi-core processors, as the problem of feeding the processing units in the cores (memory bandwidth) is multiplied by the number of cores with respect to that of a uni-processor design. In general-purpose designs, caches are often made as big as the die area and power budget allow. As the number of transistors inside the chip increased, the number of levels in on-chip caches has increased with current processors from Intel and AMD featuring now a third level of on-chip cache. The first level of cache is usually (divided into data and instruction caches) small, fast and private to each core. Subsequent levels are (shared for data and instructions,) larger, lower, and in general shared by the cores.

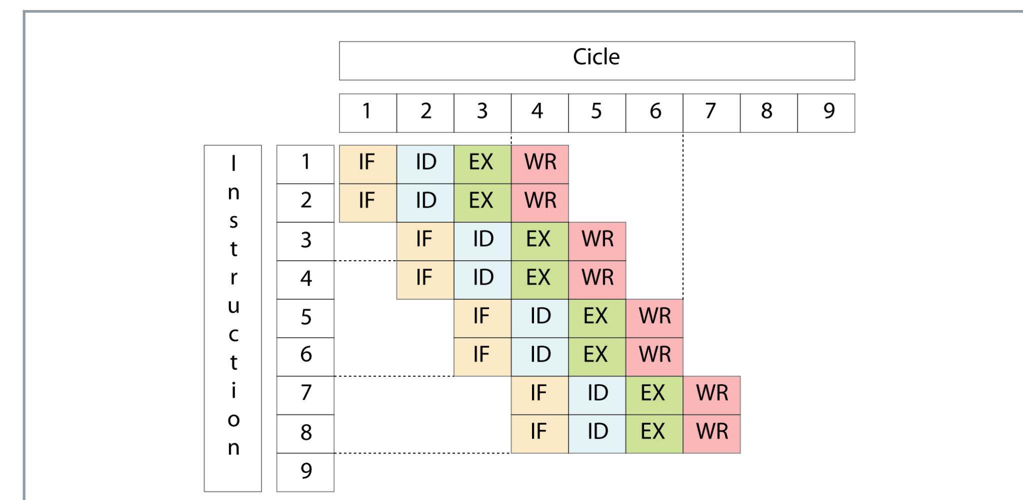


Figure 3. *Operation of a basic four-stage two-issue pipeline.*

The computational power of multi-core processors and GPU outweighs by a large factor that of the computers from past generations.

Interconnect

Multi-core processors include a fast intrachip interconnect that provides the required communication path among cores and is responsible for maintaining cache coherence (if present). Simple, bus-based interconnect designs exhibit serious limitations in both bandwidth and latency and, therefore, cannot scale with the number of cores. Alternative network-on-chip (NoCs) designs, like the crossbar, overcome these limitations at the cost of a more complex design.

Cache coherence maintains a single image of the memory system (including the different caches and the main memory) and is a key issue as it determines the programming model that is natively supported. Broadcast-based coherence is simple and provides a solution, e.g., for up to eight cores in the Intel Core i7. Directory-based coherence allows multiple coherence messages to proceed concurrently and thus scales to a larger number of cores. In summary, we can provide the following list of advantages and drawbacks of the multi-core approach.

Advantages

- Existence of a large scopus of programming environments, libraries, tools and applications.
- Compatible with x86 ISA codes.
- Truly general-purpose.
- A restricted programming model.
- Moderate power consumption.

Drawbacks

- Suboptimal for many applications, specially data-parallel ones.
- High cost of large clusters (high price-performance ratio).
- High power-performance ratio.

3. The GPU approach to parallelism

A bit of history

Two interesting phenomena happened in the early twenty-first century: the video game market was positioned among the most vibrant ones and graphic processors were delivering an important computational performance. Graphic processors are very specific hardware in design and functionality. They yield high performance in applications for which they are designed, but the initial programming techniques in this class of processors were closely tied to the hardware. However, although graphic processors were and are hardware devices specially designed to carry out video rendering (vertex shader, primitive assembly, rasterizer, pixel shader, etc.), many of their features can be extrapolated with high efficiency to other applications.

When CUDA (Compute Unified Device Architecture) appeared in 2006, the development of GPU software changed significatively, becoming more accesible to non-specialized developers. In 2007, the functional units of the GPU turned into more general-purpose units. In the next two years, a large number of applications were addressed using GPU in a wide variety of fields [6]. Nowadays, we are attending to the generalized spread of GPU hardware, including multiprocessor systems built from GPU, the evolution of CUDA towards the OpenCL standard, etc. Nowadays general-purpose GPU (GPGPU) has become a powerful tool to the service of science and technology community.

Structure, Functionality and Programmability of GPUs

We can now view a GPU as a number of multiprocessors embedded in a chip. Each multiprocessor is made up of several fine-grain processors (or functional units). Each of these simple processors plays the role of a core in the current multi-core architectures. Although the clock frequency of the system is relatively low, the number of cores can be rather high, for example, 240 in the NVIDIA GT280. All multiprocessor cores run simultaneously a set of threads called warp and all of them execute, in principle, the same instruction (SIMT: Single Instruction, Multiple Thread), but each one on its own data (SIMD model: Single Instruction Multiple Data), as shown in Figure 4.

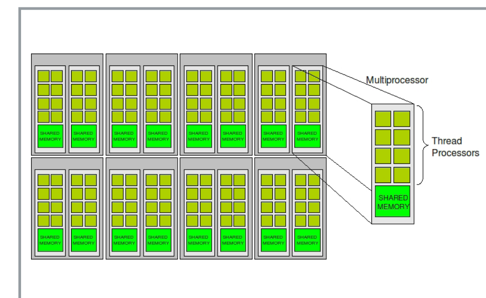


Figure 4. *Many-core architecture.*

There are several classes of memory that can be accessed by the processors of the GPU: shared memory (accessible by all cores within a multiprocessor), global memory (read/write memory accessible by any core in any multiprocessor with a relatively high access cost) and constant and texture memory (read-only memory, closely related to the graphics processing). Communication between processors can be carried out through various types of memory, depending on the context.

The GPU is designed to operate in association with a CPU that plays the role of the "master processor (Figure 5)". The GPU is often connected with the master processor via the PCI-Express bus and all the communications between the GPU and the "outside" world happens through this bus. Thus, CPU and GPU form a dual system, where the GPU acts as a coprocessor or hardware accelerator.

Programming of GPU as general purpose machines is relatively complex, as it is partly tied to the low level aspects of the system (assembly language/hardware). However, the high performance delivered by these machines partially compensates for the difficult programming.

Folo[wing Flynn's classification [7], a GPU can be considered, from a conceptual point of view, as an SIMD machine (Single Instruction, Multiple Data); that is, a computer in which a single set of instructions is executed on different data sets. Implementations of this model usually work synchronously, with a common clock signal. An instruction unit sends the same instruction to all the processing elements, which then execute simultaneously this instruction on their own data, contained in a shared or local memory. This model differs from SPMD (Single Program, Multiple Data), which involves the simultaneous execution of the same program by several processors but not the same instruction. A SPMD program can have conditional statements (if... then...else) producing the execution of different operations on different processors depending on the index of the processor. This is not the case of SIMD machines.

The GPU programmer is in charge of generating the instructions to be executed in the GPU, sending them from the CPU along with data and, finally collecting the results. This requires a suitable programming environment that allows to easily implement such actions.

CUDA: an approach to a CPU-GPU architecture

Since 2006, GPUs are mostly programmed using CUDA (Compute Unified Device Architecture). According to NVIDIA (visit [6]): "CUDA™ is a general-purpose parallel computing architecture that leverages the parallel compute engine in NVIDIA graphics processing units (GPU) to solve many complex computational problems in a fraction of the time required on a CPU. It includes the CUDA Instruction Set Architecture (ISA) and the parallel compute engine in the GPU. In order to program to the CUDA architecture, developers can, today, use C, one of the most widely used high-level programming languages, which can then be run at great performance on a CUDA enabled processor. Other languages will be supported in the future, including FORTRAN and C++".

CUDA provides instructions to transfer data and programs from the CPU to the GPU and to retrieve data back from the GPU to the CPU. It also provides a set of instructions for generating kernels (programs that run on the GPU only) which are arranged in the form of threads that are mapped onto the GPU cores.

CUDA has greatly simplified the job of programmers; however, its current development is not comparable to that achieved by standard compilers for other high-level languages/general-purpose architectures. The development and use

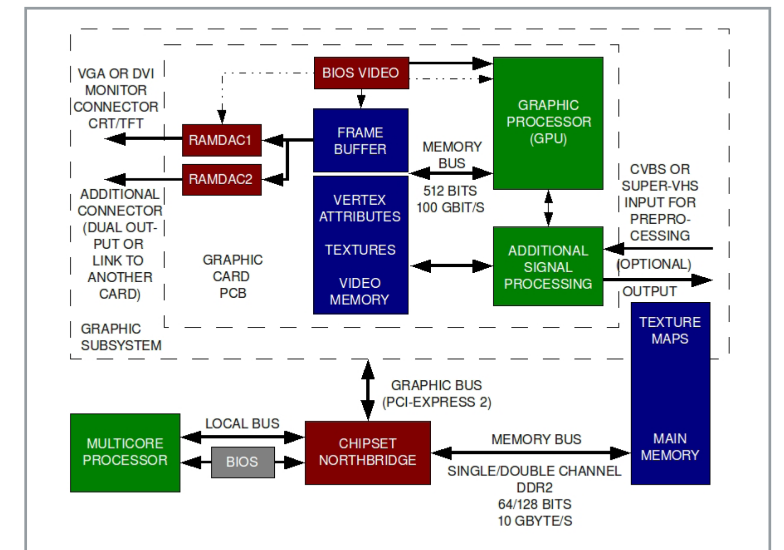


Figure 5. *GPU and CPU subsystems.*

of higher-level tools is strongly recommended. There exist several libraries that can address specific problems without having to write CUDA cores. This offers the programmer a high level programming style, similar to that commonly used in C or FORTRAN, hiding the tasks related with the implementation of GPU kernels inside library functions. While the degree of optimization has not yet reached that of standard libraries for general-purpose parallel computers, these preliminary tools represent an important aid in a not-too-friendly programming environment.

We can mention, for example, the following libraries: CUBLAS (implementation of the BLAS, Basic Linear Algebra Subprograms [www.netlib.org]), CUFFT (FFT package [6]), CULA [8] (implementation of the LAPACK [www.netlib.org] library), JACKET [9] (varied functionality of MATLAB), etc. There are also Integrated Development Environments that try to alleviate the programmer's task. One of the most significant is Parallel Nsight [6], developed by NVIDIA for the MS Windows programming environments (Visual Studio 2008). It allows debugging, profiling and analyzing GPU code using standard workflow and tools. Parallel Nsight supports CUDA C, OpenCL, Direct Compute, Direct3D, and OpenGL.

Performance

The performance of GPUs can be spectacular, especially if one only considers the peak performance of these machines. A proper use of the cores allows full concurrency, thus maximizing the whole power of parallelism (for example, 240 cores in the case of the GTX 280 card). This can potentially reduce execution times by an order of magnitude when compared with those achieved on a CPU; see Figure 6 obtained from [6].

However, several remarks are due here. Performance is much higher when using single-precision arithmetic. For example, on 2009 NVIDIA GPU processors, there is a single double-preci-

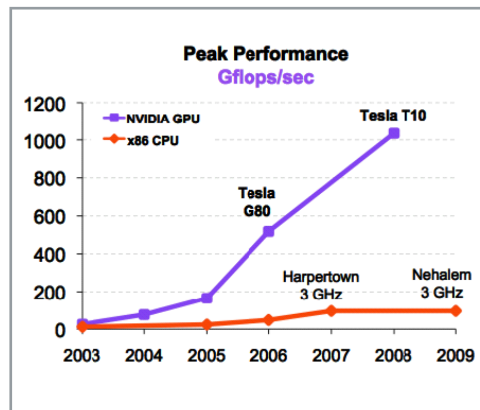


Figure 6. *GPU vs CPU GFlops.*

sion unit per multiprocessor; thus, e.g., only 30 double-precision units are present in a GT280. Furthermore, in a general application, the GPU attains a real performance that is typically much lower than its peak performance. To conclude this review of GPU, the following advantages and drawbacks can be remarked:

Advantages:

- Very high benefits in terms of Gigaflops/second.
- Excellent Price/Performance ratio.
- Existence of programming environments (CUDA, OpenCL...)
- Existence of libraries and tools.
- Many possible applications (see [5]).

Drawbacks

- A restricted programming model (SIMD model).
- CPU-GPU and I/O communications.
- Low-level programming.
- Insufficient tools.
- High power consumption.

4. Multi-core/many-core architectures in Signal Processing

Possibilities

From the discussion of multi-core processors and GPU in the two previous sections, it should be clear that the computational power of multi-core processors and GPU outweights by a large factor that of the computers from past generations. The new architectures also exhibit a more favourable power/price ratio, which may greatly facilitate their adoption and use in many application, even in those where the price may be a critical point. A preliminary conclusion is that the immediate future of computing, also in Signal Processing applications, seems tied to these architectures.

The Signal Processing field cannot remain indifferent to the computational advantages offered by the multi-core/many-core architectures. Indeed, these new systems can be an appealing alternative to the more traditional approach based on DSPs and FPGAs, as some practical Signal Processing applications have already shown; see, for example, [6]. Nevertheless, it must be yet established whether these architectures/tools are going to be widely incorporated as the primary choice in Signal Processing.

The adoption of a technology in a field of science or engineering may be influenced by factors other than the mere computational power provided by the hardware. For instance, programming models can strongly influence the pace and success of adoption. Also the nature and/or the scope of the problems may represent a limiting factor. As an example, some applications do not fit in the SIMD model, so that the use of GPU may not be appropriate or even viable; in some of these cases, the more flexible multi-core approach can solve the problem. Finally, the existence of a large scopus of legacy software or the lack of efficient software tuned for the new architectures can be a conditioning factor as well.

Only after a detailed analysis of these factors, it is possible to determine the usefulness of the multi-core/ many-core architectures in Signal Processing. Let us thus review the most popular programming tools and models available nowadays for the multi-core processors and GPU.

Multi-core is about running two or more actual CPUs (cores) on one chip. While these designs are not fundamentally different from previous multiprocessor architectures, the fundamental turning point lies in software development for applications targeting general-purpose desktop computers and low-end servers. In particular, the greatest software revolution in the past was the move from structured programming to object-oriented programming. The current "concurrency" revolution is an equally fundamental and far-reaching change in software development: Applications will only benefit from the continued exponential throughput advances in new processors if they are rewritten in terms of efficient concurrent (usually multithreaded) codes.

Luckily, there are many tools to help us in adapting software to the new architectures, especially with regular codes that are intensive in floating-point arithmetic like those frequently arising in signal processing applications. These tools can be classified in three major groups: compilers, languages/environments, and libraries.

Current compiler technology can expose a large fraction of the ILP providing a highly efficient base code for a single core. However, when dealing with multiple cores, compilers still need to be combined with some other tool (a language or an environment) that allows the programmer to pass additional information to the compiler. One such clear example is OpenMP [www.openmp.org], the current standard for shared-memory parallel programming valid for multi-core processors. OpenMP combines three elements: a high-level application programming interface (API), a compiler which transforms a program annotated with OpenMP directives into a multithreaded code, and a runtime environment combined with a library to assist in the parallel execution of the code.

OpenMP appeared in 1997 in response to the lack of a standard for parallel programming in shared-memory architectures that played a similar role to that of MPI for distributed-memory (message-passing) architectures. Version 3.0, released mid of 2008, includes the concept of tasks and the task construct, specially designed for multi-core processors. The new architectures have also given rise to a large number of contenders to OpenMP: UPC, TBB, Cilk, Chapel, etc. It still remains tobe seen whether any of these alternative solutions could become a real challenger to the acceptance of OpenMP as standard approach to programm multi-core processors.

CUDA is both NVIDIA's GPU architecture and the corresponding programming environment. Programmers use "C for CUDA" (C with NVIDIA extensions), compiled through NVIDIA C compiler, to code algorithms for execution on the GPU. CUDA architecture supports a range of computational interfaces including the new standard OpenCL [10]. High performance libraries for numerical computations, on the other hand, are much more mature. This is no surprising, as dense linear algebra kernels and the FFT have been traditionally employed by hardware vendors as the primary demonstrators of the performance attained by their designs. Current libraries for dense linear algebra include tuned multi-threaded implementations of BLAS by most hardware manufacturers (Intel, AMD, IBM, Sun, etc.), and higher level libraries as LAPACK and libflame. It is interesting to note that both LAPACK and libflame routines initially relied in BLAS to extract parallelism. However, the increase in the number of cores did require a redesign of these libraries, to extract a higher degree of (data) parallelism. The FFT has also received special attention over the last decades and, specially, with the multi-core revolution. FFTW, Spiral DFT and Intel MKL all include tuned implementations of the FFT. NVIDIA also provides its own libraries for dense linear algebra and FFT: CUBLAS and CUFFT. However, these are still suboptimal implementations which need to be further refined.

Applications

High Performance Computing (HPC) is broadening its scope to tackle a large variety of problems arising in many scientific and engineering areas. In Signal Processing, e.g., HPC techniques are applied to develop user applications in the promising market of processing, transmission and reproduction of multimedia content. The incorporation into the market of processors with multiple cores and the increasing use of graphics processors (GPU) in general-purpose applications, is at the same time a challenge and a great opportunity: the computing power of the new architectures may enable the solution of complex problems which require intensive computing using desktop computers, provided appropriate high performance algorithms are developed. The result is the availability of applications for the non-expert user that until recently were unthinkable in the consumer market.

Nowadays, signal processing has become a basic tool in many applications such as (re) creation and transmission of virtual environments, multichannel audio applications (recording and reproduction), wireless mobile communications systems with multiple antennas, to name just a few related with applications traditionally developed within the INCO2 research group. These applications often give rise to problems of high computational cost, even when using common signal processing algorithms, mainly due to the application of these algorithms to multiple signals and with real-time requirements.

The implementation of advanced algorithms for multichannel signal processing on new platforms based on computation-intensive architectures such as GPU and multi-cores is a scientific and technological challenge, of growing interest but unresolved at present, which will incorporate tools and possibilities currently available only in research to the user applications. In this line, the implementation of user systems require tools for massive signal processing: fast multichannel convolution, adaptive multichannel processing, MIMO channel equalization, and so on; as well as its interaction with computer algebra tools traditionally used in signal processing algorithms such as: solution of optimization problems with/out constraints on structured matrices, matrix decompositions (QR, SVD, etc), FFT, etc.

Only five years ago GPU supported a limited fixed number of functions, mainly addressed to the implementation of 3D graphics. Since then, GPU have evolved (both in its hardware implementation as in its programming interface - CUDA) to a very powerful processor, capable to carry out general tasks. Many references in the literature illustrate the generalized adoption of GPU, GP-GPU, also in applications other than image processing.

Concerning the use of GPU for applications in digital audio processing, the oldest references date back to 2004 [11]. However, only very recently (2007 and 2008), the use of GPU has been employed in this area. The reason for this should be attributed to the GPU programming tools, quite complex in the beginning and with the constraint of using graphic processing procedures and terms: rendering, textures, etc. A second factor against the general adoption of GPU was that, for some time, the computational power provided by a general-purpose uni-processor was enough to give support for real-time applications.

Currrent proposals for possible applications of audio and acoustics on GPGPU include (www.gpgpu.org):

The number of cores is expected to double with each reduction in the integration scale (roughly, every two years), as long as Moore's Law holds.

General-purpose multi-core processors and GPU represent the future for many scientific applications, including signal processing as well.

- Mixing audio signals.
- Modelling the acoustics of rooms and the Head Related Transfer Function (HRTF) for virtual environments.
- Adding sound effects (www.monalisa-au.org).

Other potential applications of digital signal processing on GPU can be found in the last paragraphs of [12] and [13]. An abbreviated list includes:

- Classical processing algorithms: FFT, convolution algorithms for solving differential equations, pattern recognition, sequence alignment (general algorithms using hidden Markov models), tracking.
- Algorithms for matrix massive computation: QR decomposition, Cholesky, SVD, etc.
- Wireless Applications: Implementation of some blocks of the physical layer, very suitable for standards based on OFDM, where FFT should be calculated (WiFi, WiMAX).

All the signal processing strategies developed to deal with a single signal or a few of them can be addressed when tackling multiple signals, taking advantage of its inherent parallel nature and the characteristics of the new hardware and software tools. One example of this is multichannel acoustic signal processing. This field has experienced a large growth in the last years, due to the increase in the number of sound sources used in new commercial applications for sound reproduction, and in the growing needs to include innovative effects and capabilities to the listening experience [14][15]. Moreover, the increasing market of advanced multimedia contents for home users creates the necessity of new multichannel sound processing tools, capable of extracting all the features that can be included in these contents. The creation of these contents requires as well multichannel signal processing tools, for stage analysis, signal filtering, noise reduction, etc.

The main multichannel recording-reproduction problem can be modelled as a discrete MIMO (Multiple Input, Multiple Output) system of sound signals. The problem of sound recording presents some analogies (in terms of multichannel signal processing) with the reproduction system; however, there are some distinct features. Particularly, it is possible to extract certain parameters from the analysis of the sound scene: number of sources, location of these, etc. A large number of applications of spatial sound can be derived from the scheme displayed in Figure 7, in applications that perform reproduction either through speakers or through headphones. Usually, we focus on reproduction through speakers (and recording by arrays of microphones) because this is the problem that poses harder scientific and technological challenges, and has a larger number of potential applications. In any case, it is always possible, from a generic viewpoint, to extrapolate the results obtained in reproduction by loudspeakers to headphones reproduction, if the physical phenomena and effects caused by the propagation of waves in the listening room are not taken into account.

From the acoustic man-machine interface depicted in Figure 7, which uses multiple channels for sound reproduction and acquisition, and in general can serve multiple mobile sources and listeners, the fundamental problems of signal processing can be identified. Some of these problems can be: multichannel acoustic echo cancellation, processing of signals from microphone arrays for beamforming, interference cancellation, signal separation, source localization, room equalization, active noise cancellation and spatial source location.

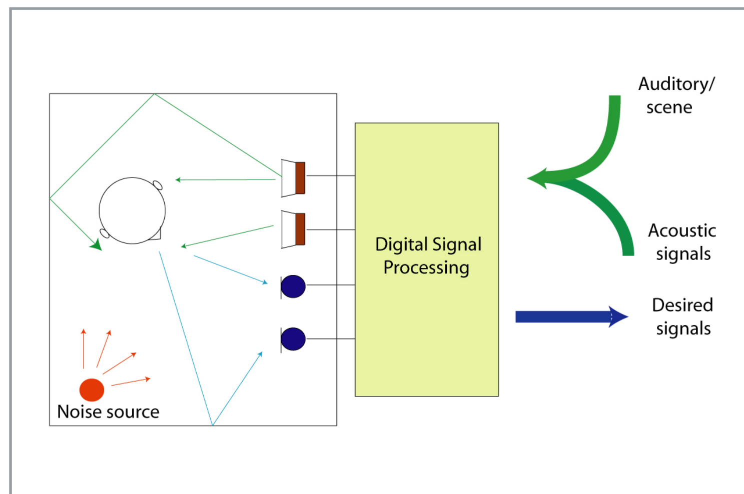


Figure 7. *Basic Scheme of a multichannel recording-reproduction system.*

5. Conclusions

General-purpose multi-core processors and GPU will surely impact future signal processing applications, with the reason for this being twofold. First, these new hardware components exhibit a vast high-performance computing capability with a much favourable price-performance ratio, and programmers of signal processing applications will naturally want to exploit this. Second, only those application developers who embrace the explicit parallel programming model intrinsic to these architectures will benefit from the multi-core revolution.

It seems that GPU may not represent the optimal model for general-purpose parallel machines, but they state an important trend that other existing architectures (for example multi-cores) cannot ignore. In this sense GPU architectures are here to stay, either in its current form or as part of a hybrid design. As a special-purpose machines, GPU have assured their presence in the short and medium term. The market for video games and graphic applications is an appropriate field for GPU that provides the necessary economic support. Nonetheless, there are also important scientific and engineering problems which exhibit a considerable degree of data parallelism, which can benefit much from the important and cheap source of computational power in GPU. There is a considerable amount of literature that delves into this line of research, to which this paper is intended to contribute.

Multicore represents a good alternative for general-purpose parallel machine, with a simple and widespreath programming model and high performance in its application scope. However, it may not be the best approach for many real-time or fine-grain signal processing applications that require real time.

Both types of architectures represent the future for many scientific applications, including signal processing as well. In this paper we have reviewed the rationale and the state-of-the-art of both architectures. In a second part, we offer a brief description of some case studies developed within the INCO2 group that illustrate the use of the new architectures [5]. These examples aim at covering issues of low/medium level, with applicability in multiple Signal Processing applications.

Acknoledgments

This work has been supported by Generalitat Valenciana Project PROMETEO/2009/013 and partially supported by Spanish Ministry of Science and Innovation through TIN2008-06570-C04 and TEC2009-13741 Projects.

References

[1] José M. F. Moura, "What is Signal Processing?," in IEEE Signal Processing magazine, vol. 26, no. 6, pp. 6-6, November 2009
[2] R. Schneiderman, "DSPs Are Helping to Make It Hard to Get Lost," in IEEE Signal Processing magazine, vol. 26, no. 6, pp. 9-13, November 2009
[3] International Technology Roadmap for Semiconductors, "International roadmap for semiconductors-System drivers," in http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_SysDrivers.pdf, 2009
[4] Herb Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," in Dr. Dobb's Journal, vol. 30, no. 3, March 2005
[5] A. Gonzalez, J. A. Belloch, G. Piñero, J. Lorente, M. Ferrer, S. Roger, C. Roig, F. J. Martínez, M. de Diego, P. Alonso, V. M. García, E. S. Quintana-Ortí, A. Remón and A. M.Vidal "Application of Multi-core and GPUs Architectures on Signal Processing: Case Studies," in Waves, vol. 2, 2010.
[6] http://www.nvidia.com/object/cuda_home_new.html
[7] M. J. Flynn, "Some computer organizations and their Effectivness," in IEEE Transactions on Computers, vol. 21 pp. 948–960, 1972
[8] http://www.culatools.com/
[9] http://www.accelereyes.com/
[10] Khronos Group, "OpenCL - The open standard for parallel programming of heterogeneous systems", in http://www.khronos.org/opencl , 2009
[11] E. Gallo and N. Tsingos, "Efficient 3D Audio Processing with the GPU", in ACM Workshop on General Purpose Computing on Graphics Processors, Los Angeles, August 2004.
[12] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone and J.C. Phillips, "GPU Computing", in Proc. of the IEEE, vol. 96, no. 5, pp. 879-899, May 2008.
[13] M.D. McCool, "Signal Processing and General-Purpose Computing and GPUs," in IEEE Signal Processing Magazine, vol. 24, no 3, pp. 109-114, May 2007.
[14] P.A. Nelson, F. Orduña-Bustamante, and H. Hamada, "Multichannel signal processing techniques in the reproduction of sound," in J. Audio Eng. Soc., vol.44, no 11, pp.973-989, November 1996.
[15] Y. Huang, J. Benesty, G.W. Elko, "Source localization, in Audio Signal Processing for Next-Generation Multimedia Communication Systems, Y. Huang, J. Benesty (Eds.), Kluwer Academic, Boston, MA, 2004 (Chapter 9).

Biographies

Antonio M. Vidal receives his M.S. degree in Physics from the Universidad de Valencia, Spain, in 1972, and his Ph.D. degree in Computer Science from the Universidad Politécnica de Valencia, Spain, in 1990. Since 1992 he has been in the Universidad Politécnica de Valencia, Spain, where he is currently a full professor in the Department of Computer Science. He is the coordinator of the project High Performance Computing on Current Architectures for Problems of Multiple Signal Processing", currently developed by INCO2 Group and financed by the Generalitat Valenciana, in the frame of PROMETEO Program for research groups of excellence. His main areas of interest include parallel computing with applications in numerical linear algebra and signal processing.

Alberto Gonzalez was born in Valencia, Spain, in 1968. He received the Ingeniero de Telecomunicacion degree from the Universidad Politecnica de Catalonia, Spain in 1992, and Ph.D degree from de Universidad Politecnica de Valencia (UPV), Spain in 1997. His dissertation was on adaptive filtering for ac-

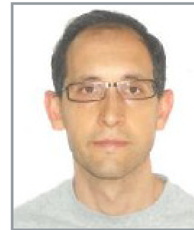
tive control applications. From January 1995, he visited the Institute of Sound and Vibration Research, University of Southampton, UK, where he was involved in research on digital signal processing for active control. He is currently heading the Audio and Communications Signal Processing Research Group (www.gtac.upv.es) that belongs to the Institute of Telecommunications and Multimedia Applications (i-TEAM, www.iteam.es). Dr. Gonzalez serves as Professor in digital signal processing and communications at UPV where he heads the Communications Department (www.dcom.upv.es) since April 2004. He has published more than 70 papers in journals and conferences on signal processing and applied acoustics. His current research interests include fast adaptive filtering algorithms and multichannel signal processing for communications, 3D sound reproduction and MIMO wireless systems.

Francisco José Martínez Zaldívar
was born in Paiporta, Spain, in 1966. He received the Licenciado en Informática and Ph.D. degrees from the Universidad Politécnica de Valencia, Spain, in 1990 and 2007 respectively. He is currently Lecturer at the Departamento de Comunicaciones, Universidad Politécnica de Valencia. His current research interests include parallel computing in signal processing.

Pedro Alonso
was born in Valencia, Spain, in 1968. He received the engineer degree in computer science from the Universidad Politecnica de Valencia, Spain, in 1994 and the PhD degree from the same University in 2003. His dissertation was on the design of parallel algorithms for structured matrices with application in several fields of digital signal analysis. Since 1996 he has been a senior lecturer in the Department of Computer Science of the Universidad Politecnica de Valencia. He is a member of the High Performance Networking and Computing Research Group of the Universidad Politecnica de Valencia. His main areas of interest include parallel computing for the solution of structured matrices with application in digital signal processing.

Alfredo Remón
was born in Valencia, Spain. In 2001 he received the B.S. in Computer Science from the Polytechnic University of Valencia, and the Ph.D. in 2008 from the Jaume I University of Castellón. He is currently an assistant researcher in the University Jaume I. His research interest include high performance computing of serial and parallel codes applied to dense linear algebra

Víctor M. García
obtained a degree in Mathematics and Computer Science (Universidad Complutense, Madrid) in 1991, later an MSc degree in Industrial Mathematics (University of Strathclyde, Glasgow) in 1992 and a Ph. D. degree in Mathematics (Universidad Politécnica de Valencia) in 1998. He is a T.U. (senior lecturer) in the Universidad Politécnica de Valencia, and his areas of interest are Numerical Computing, parallel numerical methods and applications.



Enrique S. Quintana-Ortí
is professor in Computer Architecture at the University Jaume I of Castellón, Spain. His research interests are in parallel and high-performance computing and numerical linear algebra. Enrique has a PhD in Computer Science from the Polytechnic University of Valencia.

Sandra Roger
was born in Castellón, Spain, in 1983. She received the degree in Electrical Engineering from the Universidad Politécnica de Valencia, Spain, in 2007 and the MSc. degree in Telecommunication Technologies in 2008. Currently, she is a PhD grant holder from the Spanish Ministry of Science and Innovation under the FPU program and is pursuing her PhD degree in Electrical Engineering at the Institute of Telecommunications and Multimedia Applications (iTEAM). In 2009, she was a guest researcher at the Institute of Communications and Radio-Frequency Engineering of the Vienna University of Technology (Vienna, Austria) under the supervision of Prof. Gerald Matz. Her research interests include efficient data detection, soft demodulation and channel estimation for MIMO wireless systems.

José Antonio Belloch
was born in Requena, Spain, in 1983. He received the degree in Electrical Engineering from the Universidad Politécnica de Valencia, Spain, in 2007. In 2008, he worked for the Company Getemed (Teltow, Germany) as a software developer. His interest in parallel programming for Signal processing led him to enroll in a PhD program in 2009 with the Audio and Communications Signal Processing Group (GTAC). Currently, he is finishing a Master in Parallel and Distributed Computing at the UPV. He shares his studies in the Master with his research on multichannel Audio-Signal Processing onto the CUDA environment.

Jorge Lorente
was born in Algemesí, Spain in 1985. He received the Ingeniero Técnico de Telecomunicación degree from the Universidad Politécnica de Valencia, Spain, in 2007 and the MSc. degree in Telecommunication Technologies in 2010. Currently, he is working at the Institute of Telecommunication Technologies and Multimedia Applications (iTEAM). His research focuses on microphone-array beamforming algorithms and parallelization of signal processing problems on the different cores of a CPU and also on GPU.

Application of Multi-core and GPU Architectures on Signal Processing: Case Studies

Alberto Gonzalez[1], José A. Belloch[1], Gema Piñero[1], Jorge Lorente[1], Miguel Ferrer[1], Sandra Roger[1], Carles Roig[1], Francisco J. Martínez[1], María de Diego[1], Pedro Alonso[2], Víctor M. García[2], Enrique S. Quintana-Ortí[3], Alfredo Remón[3] and Antonio M.Vidal[2]
Correspondence author: agonzal@dcom.upv.es
Audio and Communications Signal Processing Group [1] (GTAC) iTEAM, Universidad Politécnica de Valencia
Department of Information Systems and Computation[2] (DSIC) Universidad Politécnica de Valencia
Department of Computer Science and Engineering[3] (ICC) Universidad Jaume I de Castellón

Abstract

In this article part of the techniques and developments we are carrying out within the INCO2 group are reported. Results follow the interdisciplinary approach with which we tackle signal processing applications. Chosen case studies show different stages of development: We present algorithms already completed which are being used in practical applications as well as new ideas that may represent a starting point, and which are expected to deliver good results in a short and medium term.

Keywords: Multi-core/GPU Architectures, Structured linear systems, FFT, Convolution, MIMO detection, LDPC codes, Array processing, Adaptive algorithms.

1. Introduction

INCO2 [1] is a group of excellence in the Comunidad Valenciana (Spain), recognized as such by the local government through the PROMETEO 2009/013 project award. The members of the INCO2 group address problems arising in Signal Processing applications from an interdisciplinary perspective, designing solutions based on high performance hardware and developing algorithmic techniques with a modern and advanced software conception. In [2], both the architectural design and programming models of current general-purpose multi-core processors and graphics processors (GPU) were covered, with the goal of identifying their possibilities and impact on signal processing applications. Probably, the best form of appreciating the effect of these new architectures on signal processing is to analyze the performance attained by multi-core/GPUs architectures in the solution of a variety of applications. As a natural continuation of that work, in this paper we present several case studies that show how parallelization on multi-core/many-core architectures can be applied to specific problems and the outcome of it.

The rest of the paper is organized as follows. In Section 2 we show how to parallelize a detection method for MIMO digital communications systems on multi-core architectures. An evaluation of several packages to compute the FFT is presented in Section 3. Section 4 is devoted to the solution of Toeplitz linear systems on GPU and the parallelization of a beamforming algorithm for microphone arrays in Section 5. In Section 6 adaptive algorithms in digital signal processing systems with parallel convex combinations are presented. We dedicate Section 7 to present two potencial applications to be developed in GPU in the near future by INCO2: Multichannel convolution and the decoding of LDPC codes. Finally some concluding remarks are reported in Section 8.

2. Direct search methods for MIMO Systems

An emerging technology for communication is transmitting through many input and output systems, which are known as MIMO systems [3]. This technology provides, among other advantages, an increase in the bandwidth and reliability of communications [4]. In this section, we will focus on the efficient detection of digital symbols transmitted through a MIMO system.

A wireless MIMO communication can be modeled by a system composed of M transmitting antennas and N receiving antennas. A complex signal $\mathbf{s}=[s_0,\ldots s_{M-1}]^T$, $\mathbf{s} \in C^M$ is sent, where the real and imaginary parts of each component belong to a discrete and finite set A (the constellation or alphabet), and a signal $\mathbf{x} \in C^N$ is received. Signal $\mathbf{x}$ is a linear combination of the transmitted signal $\mathbf{s}$, perturbed with additive white Gaussian noise $\mathbf{v} \in C^N$; therefore, $\mathbf{x}$ can be written as $\mathbf{x} = \mathbf{H}\ \mathbf{s} + \mathbf{v}$, where the entries of the NxM (channel) matrix $\mathbf{H}$ are complex. The optimal or maximum-likelihood (ML) detection of the sent signal means that, for each signal, the following discrete minimization problem must be solved: $\min_s \|\mathbf{x}\text{-}\mathbf{H}\ \mathbf{s}\|_2$. Further details about MIMO detection can be found in [4].

When the dimensions of the problem and/or the size of the constellation grow, the computation of the optimal solution becomes very expensive [5]. In response to this, many heuristic techniques have been examined as alternatives. Our research group has studied the application of parallel computing to the different existing solvers. An approach is to use standard discrete minimization algorithms and adapt them to the problem, such as the Direct Search methods, which were first described in [6] and more recently revisited in [7]. These methods can be parallelized with two different goals in mind; following a common practice, we could use parallelism to reduce the computing time; alternatively, it can also be used to increase the probability of obtaining the optimal (ML) solution. This can be achieved by performing several searches in parallel using different initial points. We have adapted these methods to the MIMO detection problem, first with sequential versions and later with parallel versions of the sequential algorithms [8].

One of the most popular techniques for MIMO decoding is the Sphere Decoding algorithm [9]. This algorithm restricts the search to a sphere centered in the received signal x and with a given radius; it can be described as a search in a tree of solutions. The parallelism in this case can be exploited by assigning different branches of the tree to different processors. Several versions of this algorithm have been parallelized by the authors, using different parallel schemes, and different technologies (OpenMP and a hybrid method) [10]. The different versions were tested in a multi-core cluster composed of two PRIMERGY RXI600, each one with four Dual-Core Intel Itanium2 processors (1.4 GHz; 4 GB of shared RAM). The versions were tested with different problems, of increasing size (total number of nodes in the solution tree). The result is reported in terms of speed-up, which is the ratio between the time obtained with p processors and the best execution time obtained using a single processor. Figure 1 shows the speed-up attained with the parallel version based on OpenMP.

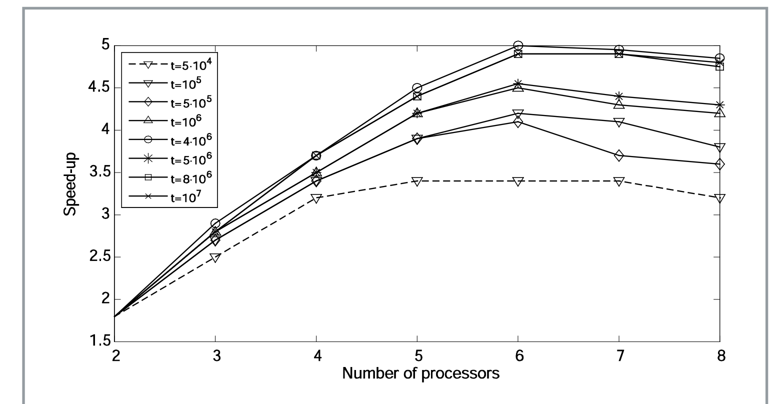


Figure 1. *Speed-up obtained using OpenMP.*

For all the problems tested, the best speedup is achieved with six processsors: compared with the time consumed by the serial version (one processor), the execution time is reduced by a factor of 5. Of course, these results strongly depend on the problem, and the results are comparatively better when the dimension of the problem is increased. Nevertheless, these results offer an idea of the possibilities of using parallel computing for this problem.

3. FFT on multi-core/many-core architectures

The discrete Fourier transform is one of the most important operations in Digital Signal Processing. Given a vector $\mathbf{x}=[x\ldots x_{n-1}]^T$ its DFT is defined as the matrix-vector product: $y_k = \sum_{j=0}^{n-1} w_n^{kj} x_j$, where $w_n = \exp\left(-2\pi i / n\right)$ and i^2=-1. The DFT can be used, among others, to obtain the frequency spectrum of a signal.

In many applications, the cost of computing the DFT [11] is too high; this is the case, e.g., of real-time applications. In those cases, the fast Fourier transform (FFT) can alleviate this problem of calculating the DFT. In particular, given a vector of size n, the computational cost of the DFT is $O(n^2)$ flops (floating-point arithmetic operations) while FFT requires only O(n log n) flops. In several experiments, we have evaluated some implementations of the FFT from different libraries on two different parallel architectures based on a multi-core processor and a GPU (see Table 1). Specifically, on the multi-core processor three libraries have been used: MKL (Intel), IPP (Intel)

Among the most widespread options in signal processing, the new multi-core / GPU architectures will be likely present in the next few years

Processors	*#cores*	*Frequency (GHz)*
Intel Xeon Quadcore E5405	8	2.33
NVIDIA Tesla C1060	240	1.3

Table 1. *Characteristics of the arquitectures used in the experiments.*

and FFTW (Massachusetts Institute of Technology). On the GPU, CUFFT (NVIDIA) and Volkov (an implementation coded by Vasily Volkov) have been evaluated, see Table 2.

The experiments comprised the computation of several FFT of a vector using single-precision arithmetic, with the size of the input vector varying from 8 to 8200 elements. The number of FFT computed in each experiment is proportional to the vector size, so the product between the vector size and the number of executions equals 8388608 (this number ensures more than 1000 executions with the biggest vector size used and is also a multiple of all the employed vector sizes). The performance (in terms of GFLOPS or 10^{15} flops per second) is computed using the same reference cost $5n\log_2 n$ for all experiments.

Library	*Version*	*Architecture*
FFTW3	3.2.1	Multi-core
Intel MKL	10.1	Multi-core
Intel IPP	6.0.2.076	Multi-core
NVIDIA CUFFT	2.3	GPU
Volkov		GPU

Table 2. *Libraries evaluated in the experiments.*

Figure 2 shows the performance obtained when the number of elements of the input vector is a power of two. As it can be seen, the performance of the kernels that operate on the GPU is notoriously higher than that of the multi-core counterparts. Other experiments were carried out for instance taking a prime number of elements of the input vector. In this case, all the FFT kernels suffered an important degradation, with the decrease being especially important for CUFFT, which yields the lowest-performance.

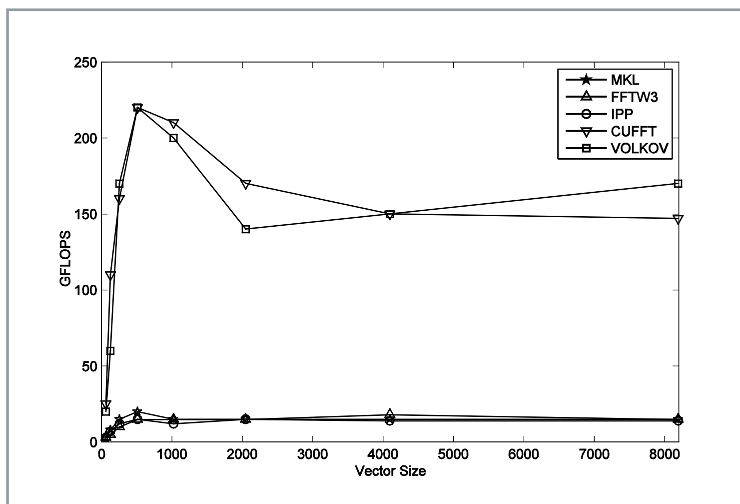


Figure 2. *Relation between GFlops and Vector Size when the number of elements of the input vector is a power of two.*

A preliminary conclusion from this study is that the FFT kernels in current libraries for the GPU clearly outperform those in libraries for the multi-core processors. However, much work remains to be done to fully optimize both types of kernels.

4. Solving structured systems on GPUs

Structured linear systems can be defined as:

$$\mathbf{TX}=\mathbf{B} \quad [1]$$

where $\mathbf{T}\in\mathbb{R}_{nxn}$ is a structured matrix, $\mathbf{B}\in\mathbb{R}_{nxn}rhs$ contains the right-hand side vector, and $\mathbf{X}\in\mathbb{R}_{nxn}$ is the sought-after solution vector.

Some structured matrices, like Toeplitz, are characterized by an external structure (e.g., in the Toeplitz matrices all elements along diagonals are equal). Hankel and Vandermonde are also examples of structured matrices with an explicit external structure. The field of structured matrices also includes some classes with non-external structure, like the inverse of a Toeplitz matrix or Cauchy-like matrices. A formal definition of structured matrices is based on the property known as displacement structure [12], which basically sets that there exist one (symmetric case) or two (non-symmetric case) matrices of nxr (r<<n) containing the same information as an nxn structured matrix.

Structured matrices appear in a wide range of engineering applications as, i.e., digital signal processing. Other appealing feature of structured matrices deals with the existence of fast algorithms which allows to solve problem (1) with an order of magnitude lower than classical algorithms that does not exploit its special structure.

We will describe next our approach to solve problem (1) where $\mathbf{T}$ is a symmetric Toeplitz matrix using a GPU. To tackle this problem, we first searched for algorithms with an intrinsic parallelism, which allowed the use of a large number of threads working concurrently on the problem. One such an algorithm performs the triangular decomposition ($\mathbf{LDLT}$, for $\mathbf{L}$ lower triangular and $\mathbf{D}$ diagonal) of symmetric Cauchy-like matrices. This algorithm works on a so called generator matrix $\mathbf{G}\in\mathbb{R}_{nx2}$, with the property that, with only two columns, it contains all the information of a given symmetric Cauchy-like matrix. The following is a scheme of the algorithm:

```
for j=1,n
   for i=j+1, n
      Use ith row of G to compute li,j
entry.
   end for
end for
```

The outer loop processes the columns of $\mathbf{L}$ while the inner loop inspects the i-th row of the j-th column. All the entries of a given column j (inner loop) can be computed concurrently. This motivates the use of a linear array of blocks of threads to compute all these entries in parallel on a GPU.

The solution of (1) using the previous algorithm is carried out by transforming the Toeplitz matrix $\mathbf{T}$ into a Cauchy-like one. This is performed by means of the Discrete Sine Transform, an FFT-related transformation that is carried out first in the CPU. When applied to symmetric Toeplitz matrices, this transform exhibits the property that it results into two independent Cauchy-like matrices of order $n/2$. This benefit allows the solution of larger problems, by solving the two independent systems in turns, overcoming the memory limits of GPU (4GB). Furthermore, it also enables the use of two GPUs in the solution of a single symmetric Toeplitz problem.

Figure 3 shows the performance improvement obtained by using one GPU over the CPU to solve problem (1) with different numbers of independent vectors.

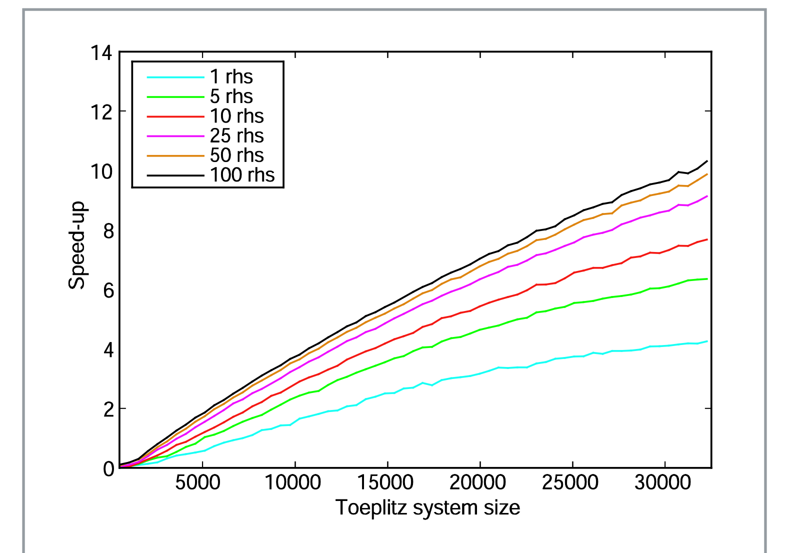


Figure 3. *GPU vs. CPU speed-up for a multiple right-hand side vectors systems.*

5. GPU array processing

A microphone array is a set of several microphones distributed in the space forming a specific pattern. Since a few decades ago, beamforming algorithms for microphone arrays have been studied and developed in order to improve the Signal-to-Noise Ratio (SNR) of the received signals, or to recover spatially separated signals considering their different Directions of Arrival (DoA) [13]. Nevertheless, one of their main limitations has been their high computational cost in practical acoustic environments where real-time sound processing must be carried out. Therefore we propose in this section an approach to the parallelization of some computations that are common to different beamforming designs.

System Model

Consider the system of Figure 4 where two loudspeakers are emitting two independent signals, $\mathbf{s}_1(k)$ and $\mathbf{s}_2(k)$, respectively, where k denotes the discrete time instant. At the other part of the room, three microphones are recording the mix of the two signals plus noise. This system can be modeled as a multichannel system with 2 inputs (loudspeakers) and 3 outputs (microphones), and the generalization to a multiple-input multiple-output (MIMO) system can be easily done.

Regarding the system of Figure 4, the problem is how to recover $\mathbf{s}_1(k)$ or $\mathbf{s}_2(k)$ by means of the signals recorded at the microphones. The approach taken herein makes use of signal processing algorithms to design the broadband beamformers (filters $\mathbf{g}_1$, $\mathbf{g}_2$ and $\mathbf{g}_1$ in Figure 4), once all the room channel responses ($\mathbf{h}_{nm}$ in Figure 4) are known. This problem is commonly known as signal deconvolution, and plays an important role in teleconferencing where the speech of interest has to be extracted from the observations of the microphone array but is usually corrupted by noise, room reverberation and other interfering sources.

According to Figure 4, the output of the n-th microphone is given by:

$$x_n(k) = \sum_{m=1}^{M}\sum_{j=1}^{Lh} h_{nm}(j)\, s_m(k-j) \quad [3]$$

where n=1,2,…, N, being N the number of microphones and M the number of source signals, that is equal to the number of loudspeakers in Figure

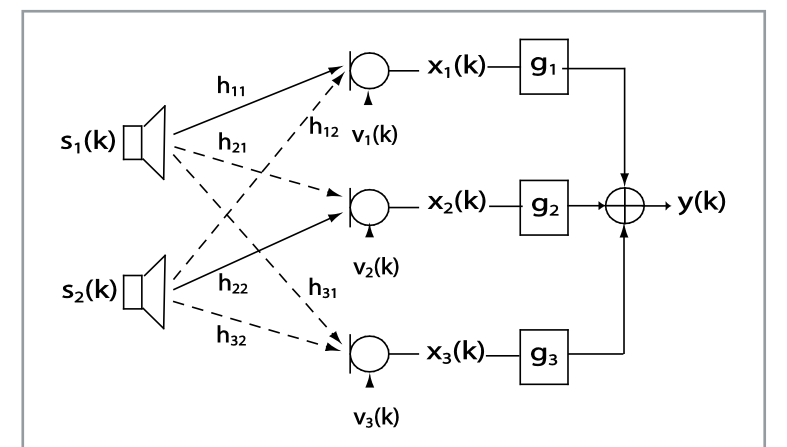


Figure 4. *System model for 2 loudspeakers (inputs) and 3 microphones (outputs).*

Structured matrices appear in a wide range of engineering applications as digital signal processing

4. L_h is the length of the longest room impulse response of all acustic channels $\mathbf{h}_{nm}$. The noise contribution has not been considered for sake of clarity.

This signal model can be rewritten in vector/matrix form as: $\mathbf{x}_n(k) = \mathbf{Hs}(k)$ where $\mathbf{x}_n(k)$ is a column vector and vector $\mathbf{s}(k)$ and matrix $\mathbf{H}$ are defined as $\mathbf{s}(k) = [\mathbf{s}_1^T(k) \quad \mathbf{s}_2^T(k)]^T$, where $\mathbf{s}_m(k)=[s_m(k) \; s_m(k\text{-}1) \ldots s_m(k\text{-}L_n\text{+}1)]^T$, and $\mathbf{H}=[\mathbf{H}_1 \; \mathbf{H}_2 \; \mathbf{H}_3]^T$, where $\mathbf{H}_n = [\mathbf{h}_{n1}^T \quad \mathbf{h}_{n2}^T]^T$, and $\mathbf{h}_{nm}=[h_{nm,0} \; h_{nm,1} \ldots h_{nm,Lh\text{-}1}]^T$ for n=1,2,3 and m=1,2; $(\cdot)^T$ denotes the transpose of a vector or a matrix and L_h is the length of the longest channel impulse response. Once the recorded signals $\mathbf{x}_n(k)$ have been modeled, the broadband beamformers (filters $\mathbf{g}$) have to be designed and calculated. Benesty et al. [14] present an excellent state-of-the-art of the main algorithms used in audio applications. Some of them make use of the channel matrix $\mathbf{H}$ exclusively and calculate filters $\mathbf{g}_i$ based on its inverse (or pseudo-inverse), whereas other methods take also into account the correlation matrix of the recorded signals. Under perfect estimation of channel impulse responses, both types of filters show similar good performance, but in practical experiments, where the $\mathbf{h}_{nm}$'s are estimated under some uncertainties, filters based on the estimated correlation matrix outperforms those based on the channel inversion.

The estimated correlation matrix of spatially sampled signals, $\mathbf{x}_n(k)$, is commonly known in the literature as the Sample Correlation Matrix (SCM), and its expression is given by:

$$\hat{\mathbf{R}}_x = \frac{1}{N}\sum_{k=1}^{N}\mathbf{x}(k)\cdot\mathbf{x}^T(k) \qquad [4]$$

where $\mathbf{x}(k) = [\mathbf{x}_1^T(k) \quad \mathbf{x}_2^T(k) \quad \mathbf{x}_3^T(k)]^T$.

Regarding the dimensions of SCM, $[3L_g \text{ x } 3L_g]$, L_g depends on the length of the room impulse response L_h and is usually greater than 150. Considering that (3) has to be recalculated at short time intervals due to the non-stationary nature of sound signals, and that $N \geq 3L_g$ to assure that $\hat{\mathbf{R}}_x$ is full-rank and invertible, then an efficient parallelization of the computation in (4) is required. Three different implementations have been considered in order to obtain the matrix correlation as fast as possible. In one hand the sequential implementation and in the other hand two different parallel implementations: one in multiple cores of CPU and the other in the GPUs.

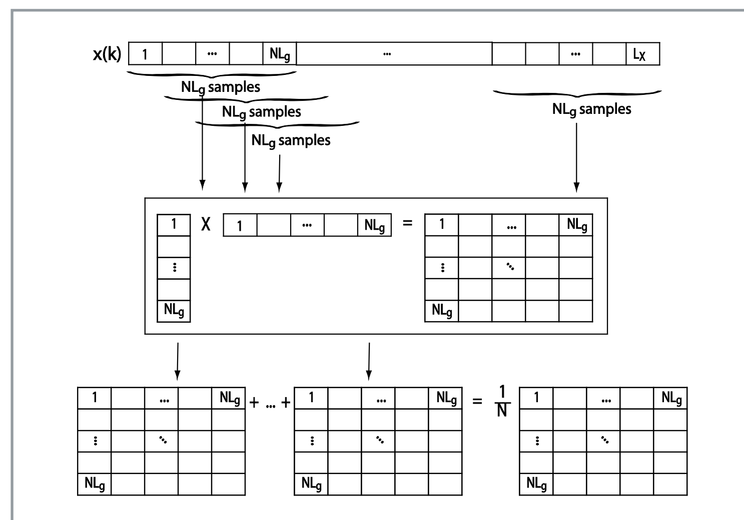


Figure 5. *Illustration of parallel method implemented on CPU.*

The sequential implementation

The sequential implementation of the Sample Correlation Matrix of (3) is iteratively done by a 'for' loop which calculates one vector outer multiplication and one matrix sum correspondent to index k-th of the total sum at each iteration. Its implementation can be seen schematically in Figure 5, where vector $\mathbf{x}(k)$ is split in smaller overlapping vectors of $3L_g$ length each.

Parallel implementations of the Sample Correlation Matrix of (3)

1) Parallelization in CPU multi-core:

In this case the parallelization consists in dividing the sequential tasks described above in different CPU cores. To achieve that the Matlab toolbox for parallel computing has been used, more specifically the functions *matlabpool* and *spmd* [15].

2) Parallelization on GPU:

In this case, the parallelization is performed at a lower level than in CPU. For this purpose, the software interface called Jacket [16], which allows running code in the GPU through Matlab, has been tested. The following steps have been taken:

- First, to send microphone array signals $\mathbf{x}(k)$ to the GPU using Jacket function *gdouble*().
- Second, to parallelize (3) so no iteration of 'for' loop must depend on a previous result. Then parallelization of the 'for' loop is done using the Jacket function *gfor*().

The step 2 has been carried out splitting each vector $\mathbf{x}_n(k)$ of (4) in basic blocks of variable length Z. The performed parallelization in GPU for $Z=L_g/2$ can be seen in Figure 6. Let us denote $\mathbf{x}_n^{(i)}(k)$ as the i-th block. Considering that $\mathbf{x}_n(k)$ has length NL_g, the number of available blocks $\mathbf{x}_n^{(i)}(k)$ is $NL_g/(L_g/2)=2N$. Therefore, the single outer product $\mathbf{x}_n(k)\mathbf{x}_n^T(k)$ of (4) is now computed in parallel at the GPU though $(2N)^2$ outer products $\mathbf{x}_n^{(i)}(k)\left(\mathbf{x}_n^{(i)}(k)\right)^T$.

Figure 6 shows the case for $N=3$ microphones, so there are $2N=6$ basic blocks available, and $\mathbf{x}_n(k)\mathbf{x}_n^T(k)$ will be computed whith $(2N)^2=36$ outer products in parallel.

		Lg=110	Lg=130	Lg=150	Lg=170	Lg=190
Sequential		0.6084	0.9121	1.2269	1.7909	2.1333
CPU parallel		0.3908	0.5823	0.7853	1.0628	1.6098
GPU parallel	9 parts	**0.1353**	**0.2135**	**0.2702**	0.3702	0.5166
	36 parts	0.1866	0.2303	0.2749	**0.3656**	**0.4232**
	81 parts	0.3012	0.3557	0.4016	0.4980	0.5819

		Lg=210	Lg=260	Lg=300	Lg=330	Lg=360
Sequential		2.5883	6.0940	9.1885	11.264	14.681
CPU parallel		1.9913	192.14	585.32	*overflow*	*overflow*
GPU parallel	9 parts	0.6714	*overflow*	*overflow*	*overflow*	*overflow*
	36 parts	**0.4808**	**0.7813**	**2.1767**	4.6818	6.8380
	81 parts	0.6642	0.9183	2.3748	**3.5623**	**4.9330**

Table 3 *Table of times used in calculating the Sample Correlation Matrix (SCM) of equation (4).*

Signal deconvolution plays an important role in teleconferencing where the speech of interest has to be extracted from the observations of the microphone array usually corrupted by noise

Three different lengths of basic blocks have been tested in GPU: $Z=L_g$, $Z=L_g/2$ and $Z=L_g/3$, which results in N^2, $(2N)^2$ and $(3N)^2$ outer products of block vectors $\mathbf{x}_n^{(i)}(k)$. For the system depicted in Figure 4 whith $N=3$, the different sizes of Z give a parallelization of 9, 36 and 81 independent outer products for step 2, respectively.

Testing Results

Sequential implementation and both parallel implementations explained above for 3 recorded signals $x_n(k)$ at sampling frequency of 11 kHz have been tested with an i7 CPU of Intel and a NVIDIA GPX285 GPU. Results obtained for signals of duration 4 seconds (44 $k$samples) can be seen in Table 3. The CPU parallel method has been carried out with 3 cores, it has been proved that for this kind of computation it was the best configuration. As we can see in Table 3, the parallel implementation with multiples cores of a CPU only obtain speed up greater than 1 when $L_g \leq 210$ comparing to sequential implementation, achieving almost double velocity in the best case of L_g=110. An explanation for this low performance may be that too much time is lost distributing tasks into the different cores of the CPU, whereas the code to be distributed consists only in a few lines. Moreover, results of Table 3 show that computational time grows exponentially when L_g exeeds L_g=210; we suppose that in this cases, length of filters g_i is very large and buffers memory of the cores collapses: big amounts of data are replicated in all buffers, and that makes such a significant time increase. For L_g>300, Matlab returns a memory error because there is not enough memory to allocate matrices with such big dimensions.

Regarding GPU implementation, Jacket performs with matrix of maximum 65.536 elements. As we see in figure 6, dimensions of SCM depends on L_g, so working whithin GPU configuration divided in 9 parts, when L_g=260 dimensions of SCM exceeds 65.536 elements, so Jacket program returns a memory error. To solve this problem we divide the calculation of SCM in more number of parts, and as table 3 show, as L_g grows, the number of parts used in the calculation of the matrix correlation must be incremented to reach the best time results. Otherwise, if we took the most efficient case for each length of filters, L_g, it can be shown that the speed-up in all the cases is near to 4, which means a significant time saving. Same results of Table 3 are depicted in Figure 7, where the graph at right shows those methods whose computation times are below 2 seconds. It should be noted that GPU outperforms sequential and multi-core implementations in all cases.

Finally it should be noted that, considering the duration of the recorded signals, 4 seconds, a delay in calculating the matrix correlation of one to three tenth of a second is attainable for real-time applications.

6. Adaptive algorithms with parallel combinations on multi-core platforms

During last years, adaptive systems [17] have been the objective of many studies due to their

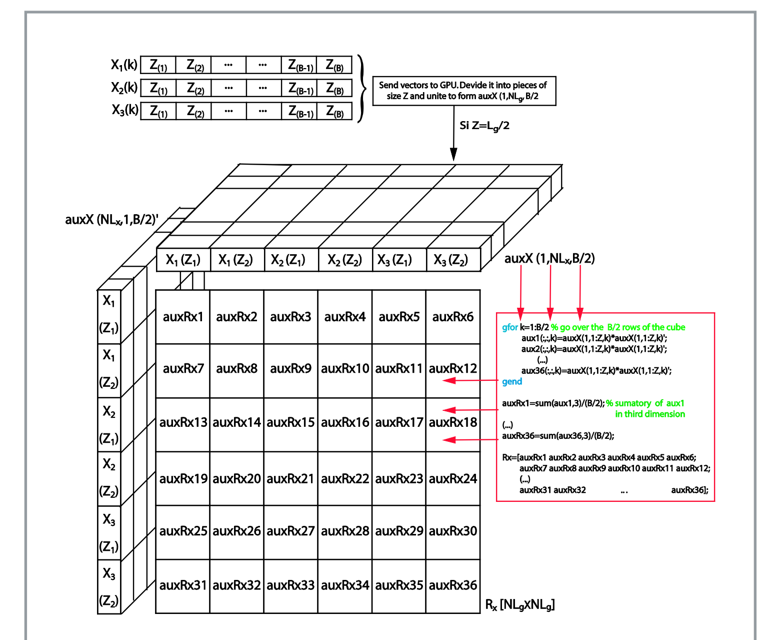


Figure 6. *Illustration of parallel method implemented on GPU.*

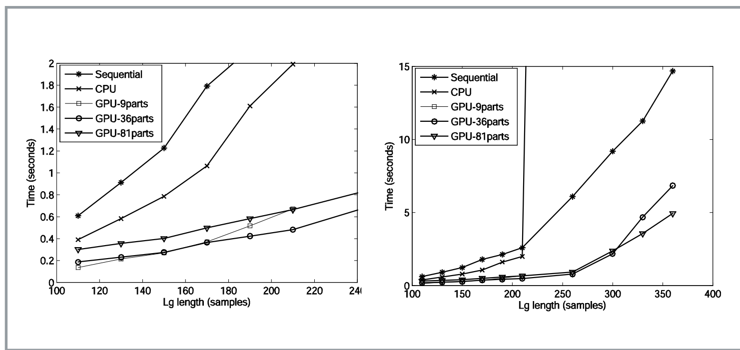


Figure 7. *Evolution of computational time when L_g grows.*

multiple applications in digital processing systems. Applications like channel identification, channel equalization or channel inversion, used for sound or communications systems, echo cancellation, noise cancellation, among others, are based on adaptive systems. There is a big amount of adaptive algorithms in order to control adaptive systems like: LMS, RLS, FTF, AP, etc. A complete description of each can be found in [18], whose conclusion says that none of them is globally better than the others, but also, the algorithms which achieve the best performances are the ones which have greater computational cost. Also, the ones which have good behavior in a permanent regime are worse than others if we compare the convergence speed. This is the reason why there are different adaptive strategies. In order to improve the performance of different adaptive algorithms, new parallel combining strategies have appeared, like convex [19]. These strategies allow to combine the strengths of two adaptive algorithms which present complementary features (for instance, one with fast convergence and the other with low residual error level in permanent regime), achieving the combination of the best performances from each one in a separated way. As it can be checked in [20], using this strategy is possible to achieve both objectives at the same time, high convergence speed and low residual error level in permanent regime. This kind of strategies can be used successfully in active noise control applications [21], obtaining a really good performance: fast convergence and low residual noise level. However, this better behavior appears at the expense of doubling the computational cost, since two algorithms have to be executed at the same time, in parallel. The parallel nature of this structure allows the distribution of the computation within parallel hardware like the multi-core systems, where the computational load can be easily dealed out among different cores and thus the execution time reduced. Therefore, the adaptive algorithm could be used in systems working at a higher sampling rate. The computations needed could be carried out in two kernels, using a third kernel to combine both algorithms, or using one of the first kernels to combine the signals if there are only two kernels. Next, Figure 8 shows the block diagram of the convex structure executed over a multi-core platform.

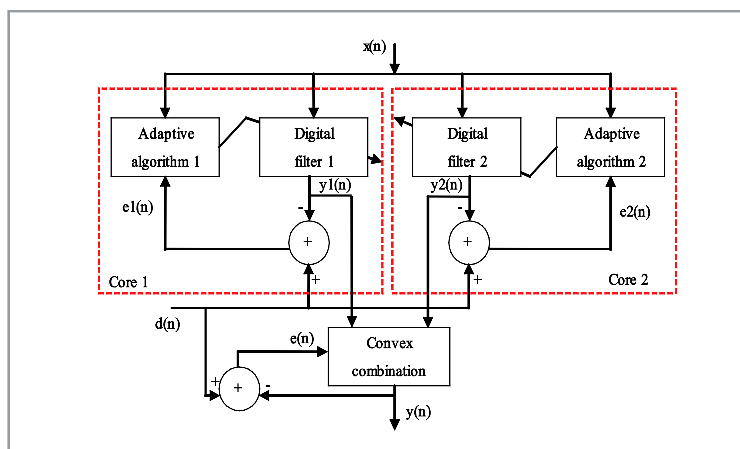


Figure 8. *Scheme of the multi-core convex combination.*

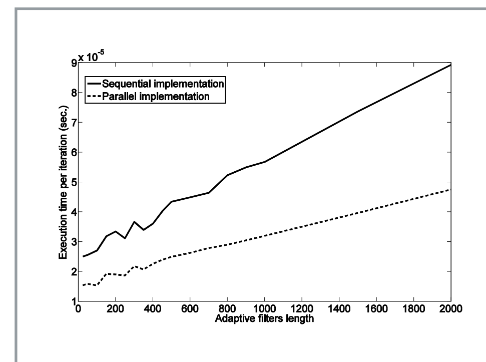


Figure 9. *Runtime per iteration for multi-core system and simple core system.*

As it can be checked in Figure 8, apart from the convex combinations, the rest can be executed in a parallel way. Therefore, the execution time has been reduced to the time that a single filters needs to carry out the computations. In other words, thank of this structure and the use of two kernels, the time required by the process becomes the time needed by one single kernel, instead of the double time required by a sequential implementation using monocore structures. Figure 9 exhibits the algorithm runtime per iteration and the comparison with the sequential version executed in a single core system. It shows the relation between the execution time and the length of the adaptive filters used in the convex structure when LMS algorithm is used as a controller of the adaptative filters. This test has been carried out on an Intel Core i7 CPU 920 @ 2.67GHz, and the algorithm was run in a Matlab R2009b platform using Parallel Computing Toolbox V4.2.

As it can be seen in Figure 9, the reduction of the algorithm runtime using a platform of two kernels is really significant. This structure only needs half runtime of the sequential one. The most important conclusion is that it will be possible to work with higher sampling frequencies in order to deal with signals with higher bandwidth, or just to have adaptive algorithms which require high computational load needing less time to carry out this operation.

7. Future Prospects.

In this section, we present two potential applications in Signal Processing which are focused on the implementation of the multichannel convolution and the decoding of LDPC codes using the capabilities of GPU computing.

Multichannel convolution

It can be shown that the computation of the convolution operation consists of several scalar multiply and add operations [22], where a certain parallelism can be identified. In order to compute the convolution, the architecture of the GPUs allows different levels of parallelism. At a first level, a single convolution operation of two signals can be efficiently implemented in parallel inside a GPU. The second level of parallelism allows carrying out different convolutions of different channels parallelly. Note that, obviously, the benefits of using a GPU increase when both levels of parallelism are exploited simultaneously.

The possibilities that GPUs offer are varied. However, the main challenge when implementing an algorithm on GPU relies on adapting the resources of the GPU to obtain the best performance depending on the properties of the signals (monochannel, multichannel, etc.) and, of course, the type of processing that wants to be carried out: Convolution of all the signals either by the same $h(k)$ or with different $h_i(k)$ with $i \in [0 \ldots n\text{-}1]$, convolution of some signals by $h_1(k)$ and others by $h_2(k)$ and all of them at the same time, etc.

Recently, the new CUDA toolkit 3.0 lets use CUFFT [11] with the property concurrent copy and execution and therefore, implementing real-time applications where the latency of transfering the samples from the CPU to the GPU for processing and vice versa overlapped by computation.

LDPC Codes on GPU

Low-Density Parity-Check codes (LDPC codes) are linear block channel codes for error control coding with a sparse parity-check matrix (a matrix that contains few ones in comparison to the amount of zeroes). They have recently been adopted by several data communication standards such as DVB-S2 and WiMax. The concept of LDPC coding was first developed by Robert G. Gallager in his doctoral dissertation at the MIT in the begining of sixties [23] but quickly forgotten due to its impractical implementation at that moment and the introduction of Reed-Solomon codes. They were rediscovered by MacKay and Neal in 1996 [24].

These codes provide a performance very close to the Shannon capacity limit of the channel, low error floor, and linear time complexity for decoding (lower than turbocodes). We can find simple tutorials to understand the basics of these kind of codes in [25], [26], and software to test them in [27]. LDPC codes are inherently suited for parallel hardware and software implementations as we can read in [28], [29] and [30] using CUDA and other GPU programming tools.

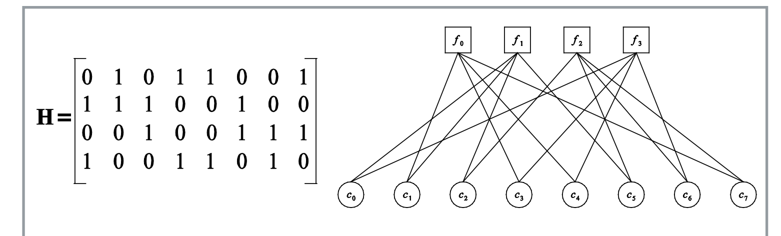


Figure 10. *Tanner graph of a linear block code parity-check matrix* **H**.

LDPC codes can be represented graphically by a Tanner graph [31] (an undirected bipartite graph with variable nodes, c_i, and check nodes, f_j). An example is shown in Figure 10, that corresponds with the parity-check matrix on its left **H**:

LDPC decoders are based on variations of belief propagation, sum-product or message passing algorithms. In any of these algorithmic denominations, information flows to/from variable nodes and from/to check nodes until the algorithm converges to a stable state, finding the most likelihood transmitted codeword. An easy example can be observed in the bit flipping algorithm (hard decision decoding). The iterations are divided in two dependent steps:

1. Each variable node sends the majority voted bit to all its connected check nodes (at the beginning, the only information available is the received bit)

2. Each check node estimates each connected variable node bit as the parity-check matrix dictates (using the estimations of the rests of the connected variable nodes and excluding the value that is estimating) and send this information to this connected variable node.

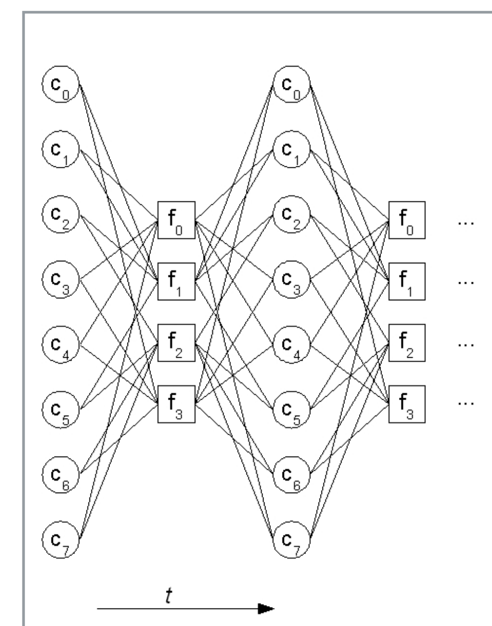


Figure 11. *Computation and message passing in parallel algorithm.*

LDPC codes are inherently suited for parallel hardware and software using CUDA and other GPU programming tools

These steps are executed iteratively until the estimated word is a codeword. Better results are obtained when soft decision is used [32]. It can be observed that the computations within the check nodes and within the variable nodes are alternated and interdependent in time, so they must be executed one after another because of their inherent sequentiallity. The computations in every check node are independent, so they are perfectly parallelizable; the same happens with the variable nodes computations. Within a check node, it must be computed a different result to every variable node that is connected to it. Something similar is observed regarding to the variable node computations. Figure 11 shows the dependency graph of the parallel algorithm.

We are focusing our implementations on concentrating the operations within every node because each result in a node shares nearly all the multiplication factors that it contains. Another important question is how the accesses to the global and shared memory are arranged in order to make a coalesced access and to avoid conflicts in the memory banks. This will ensure a good speedup in a real time environment.

9. Conclusions

Throughout this article it has become obvious the impact of new multi-core / GPU architectures in the field of signal processing. Among the most widespread options in signal processing, these new architectures will be likely present in the next few years. However, it is also very likely that FPGA devices keep a good share of the market, as they cover a large part of very specific applications.

The purpose of this work was to serve as a showcase of different signal processing applications in which new Multi-core/GPU architectures can be competitive. Different applications, in which researchers of INCO2 Group are working, have been used as case studies. The aim of this group is precisely the application of high performance computing and next-generation parallel architectures (particularly multi-cores and GPUs) in the solution of problems in signal processing. We believe this option is a sure bet in one of the most promising areas of current technology, in general, and the Information Technology area, in particular, where the duo computer-communications can not be dissociated.

Acknowledgements

This work was supported by Generalitat Valenciana Project PROMETEO/2009/013 and partially supported by Spanish Ministry of Science and Innovation through TIN2008-06570-C04 and TEC2009-13741 Projects.

References

[1] www.inco2.upv.es
[2] A. Gonzalez, J. A. Belloch, G. Piñero, F. J. Martínez, P. Alonso, V. M. García, E. S. Quintana-Ortí, A. Remón, and A. M.Vidal "The Impact of the Multi-core Revolution on Signal Processing"; Waves, vol. 2, 2010.
[3] A. J. Paulraj, D. A. Gore, R. U. Nabar, and H. Bölcskei, "An overview of MIMO communications - a key to Gigabit wireless," Proceedings of the IEEE, vol. 92, no. 2, pp. 198–218, Feb. 2004.
[4] S. Roger, F. Domene, C. Botella, G. Piñero, A. Gonzalez, and V. Almenar, "Recent advances in MIMO wireless systems", Waves, vol. 1, pp. 115-123, 2009.
[5] J. Fink, S. Roger, A. González, V. Almenar, and V. M. García, "Complexity Assessment of Sphere Decoding Methods for MIMO Detection", IEEE International Symposium on Signal Processing and Information Technology (ISSPIT), Ajman, UAE, December 2009.
[6] R. Hooke and T. A. Jeeves, "Direct Search solution of numerical and statistical problems", Journal of the Association for Computing Machinery, pp. 212–229, 1961.
[7] T. G. Kolda, R. M. Lewis, and V. Torczon, "Optimization by Direct Search: New perspective on some Clasical and Modern Methods", SIAM Review, vol. 3, pp. 385–442, 2003.
[8] R. A. Trujillo, A. M. Vidal, and V. M. García, "Decoding of signals from MIMO communication systems using Direct Search methods", 9th International Conference Computational and Mathematical Methods in Science and Engineering (CMMSE), Gijón, Spain, July 1-3 2009.
[9] M. Pohst, "On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications", ACM SIGSAM Bull., vol. 15, pp. 37–44, 1981.
[10] R. A. Trujillo, A. M. Vidal, V. M. García, and Alberto González, "Parallelization of Sphere-Decoding Methods," Lecture Notes in Computer Science, vol. 5336/2008, pp. 2-12, 2008.
[11] C. Van Loan, "Computational Frameworks for the Fast Fourier Transform," SIAM Press, Philadelphia, 1992.
[12] T. Kailath and Ali H. Sayed, "Displacement Structure: Theory and Applications", SIAM Review, vol. 37, pp. 297-386, Sept. 1995.
[13]H. Branstein and D.B. Ward, "Microphone Arrays: Signal Processing Techniques and Applications", Springer, Berlin (Germany), 2001.
[14] J. Benesty, J. Chen, Y. Huang, and J. Dmochowski, "On Microphone-Array Beamforming From a MIMO Acoustic Signal Processing Perspective", IEEE Trans. Audio, Speech, and Language Processing, vol.15, no.3, pp.1053-1065, 2007.
[15] Parallel Computing Toolbox 4.2 Product Page, The Mathworks, online at: www.mathworks.com/products/parallel-computing
[16] Jacket for MATLAB Product Page, AccelerEyes, online at: www.accelereyes.com/resources/literature.
[17] B. Widrow and S. D. Stearns, Adaptive Signal Processing, Prentice-Hall, Englewood Cliffs, N.J., 1985.
[18] S. Haykin, Adaptive Filter Theory, Prentice-Hall, Ed., Upper Saddle River, NJ, Fourth edition 2002.
[19] J. Arenas-García, A. R. Figueiras-Vidal, and Ali H. Sayed, "Mean-Square Performance of Convex Combination of two Adaptive Filters", IEEE Transactions on Signal Processing, vol 54, no. 3, March 2006.
[20] M. Ferrer, M. de Diego, A. González, and G. Piñero, "Convex combination of affine projection algorithms adaptive filters for ANC", 17th European Signal Processing Conference, Glasgow, Scotland, August 2009.
[21] M. Ferrer, M. de Diego, A. González, and G. Piñero, "Convex combination of adaptive filters for ANC", 16th International Congress on Sound and Vibration, Cracow, Poland, July 2009.
[22] S.S. Soliman, M. D. Srinath, "Continuos and discrete Signals and Systems", Ed. Prentice Hall.
[23] R.G. Gallager, "Low-Density Parity-Check Codes", MIT Press, Cambridge, MA (USA), 1963.
[24] D.J.C. MacKay and R.M. Neal, "Near Shannon limit performance of low density parity check codes", IEEE Electronics Letters, vol. 33, no. 6, pp. 457-458, 1997.
[25] M.J. Bernhard, "LDPC Codes -a brief Tutorial", users.tkk.fi/~pat/coding/essays/ldpc.pdf
[18] Overview of Low-density parity-check codes, online at: en.wikipedia.org/wiki/Low-density_parity-check_code.
[19] R. M. Neal, online at: www.cs.utoronto.ca/~radford/homepage.html
[20] G. Falcão, L. Sousa, and V. Silva, "Massive parallel LDPC decoding on GPU", Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Salt Lake City, Ut (USA), February 20 - 23, 2008.
[21] G. Falcão, V. Silva, and L. Sousa, "How GPUs can outperform ASICs for fast LDPC decoding", Proceedings of the 23rd International Conference on Supercomputing, Yorktown Heights, NY (USA), 2009.
[22] S. Cheng, "A Parallel Decoding Algorithm of LDPC codes using CUDA", tulsagrad.ou.edu/samuel_cheng/articles.html
[23] R. Tanner, "A recursive approach to low complexity codes", IEEE Transactions on Information Theory, vol. 27, no. 5, pp: 533-547, 1981.
[24] T. Richarson and R. Urbanke, Modern Coding Theory, Cambridge University Press 2008.

Biographies

Antonio M. Vidal
See page 73

Alberto Gonzalez
See page 73

Gema Piñero
was born in Madrid, Spain, in 1965. She received the Ms. in Telecommunication Engineering from the Universidad Politécnica de Madrid in 1990, and the Ph.D. degree from the Universidad Politecnica de Valencia in 1997, where she is currently working as an Associate Professor in digital signal processing. She has been involved in different research projects including active noise control, psychoacoustics, array signal processing and wireless communications in the Audio and Communications Signal Processing (GTAC) group of the Institute of Telecommunications and Multimedia Applications (iTEAM). Since 1999 she has led several projects on sound quality evaluation in the fields of automotive and toys. Since 2001 she has been involved in several projects on 3G wireless communications supported by the Spanish Government and Telefónica. She has also published more than 40 contributions in journals and conferences about signal processing and applied acoustics. Her current research interests in the communications field include array signal processing for wireless communications, MIMO multi-user techniques and optimization of signal processing algorithms for multi-core and GP-GPU computation.

Francisco José Martínez Zaldívar
See page 74

Pedro Alonso
See page 74

Alfredo Remón
See page 74

Víctor M. García
See page 74

Enrique S. Quintana-Ortí
See page 74

Maria de Diego
was born in Valencia, Spain, in 1970. She received the Telecommunication Engineering degree from the Universidad Politecnica de Valencia (UPV) in 1994, and the Ph.D degree in 2003. Her dissertation was on active noise conformation of enclosed acoustic fields. She is currently working as Associate Professor in digital signal processing and communications. Dr. de Diego has been involved in different research projects including active noise control, fast adaptive filtering algorithms, sound quality evaluation, and 3-D sound reproduction, in the Institute of Telecommunications and Multimedia Applications (iTEAM) of Valencia. She has published more than 40 papers in journals and conferences about signal processing and applied acoustics. Her current research interest include multichannel signal processing and sound quality improvement.

Miguel Ferrer
was born in Almería, Spain. He received the Ingeniero de Telecomunicacion degree from the Universidad Politécnica de Valencia (UPV) in 2000, and the Ph.D degree in 2008. In 2000, he spent six months at the Institute of aplicated research of automobile in Tarragona (Spain) where he was involved in research on Active noise control applied into interior noise cars and subjective evaluation by means of psychoacoustics study. In 2001 he began to work in GTAC (Grupo de Tratamiento de Audio y Comunicaciones) that belongs to the Institute of Telecommunications and Multimedia Applications. He is currently working as assitan professor in digital signal processing in communications Department of UPV. His area of interests includes efficient adaptive algorithm and digital audio processing.

Sandra Roger
See page 75

José Antonio Belloch
See page 75

Jorge Lorente
was born in Algemesí, Spain in 1985. He received the Ingeniero Técnico de Telecomunicación degree from the Universidad Politécnica de Valencia, Spain, in 2007 and the MSc. degree in Telecommunication Technologies in 2010. Currently, he is working at the Institute of Telecommunication Technologies and Multimedia Applications (iTEAM). His research focuses on microphone-array beamforming algorithms and parallelization of signal processing problems on the different cores of a CPU and also on GPU.

Carles Roig
was born in Alginet, Spain, in 1986. He received the degree in Telecommunication Engineering from the Universidad Politécnica de Valencia, in 2010. Currently, he works as a research assistant within the Institute of Telecommunications and Multimedia Applications (iTEAM). His research interests include adaptive filtering and its applications to the active noise control.

*Proceedings of the 10th International Conference
on Computational and Mathematical Methods
in Science and Engineering, CMMSE 2010
27–30 June 2010.*

Multichannel acoustic signal processing on GPU

Jose A. Belloch[1], Antonio M. Vidal[2], Francisco J. Martínez-Zaldívar[3] and Alberto Gonzalez[3]

[1] *Audio and Communications Signal Processing Group. Instituto de Telecomunicaciones y Aplicaciones Multimedia, Universidad Politécnica de Valencia (Spain)*

[2] *Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia (Spain)*

[3] *Departamento de Comunicaciones, Universidad Politécnica de Valencia (Spain)*

emails: `jobelrod@iteam.upv.es`, `avidal@dsic.upv.es`, `fjmartin@dcom.upv.es`, `agonzal@dcom.upv.es`

Abstract

Massive convolution is the basic operation in multichannel acoustic signal processing. Dealing with multichannel signals takes a big computational cost requiring the use of multiple resources from the CPU. Graphical Processor Units (GPU), a high parallel commodity programmable co-processors, can carry out a multichannel convolution faster. However, the fact of transferring data from/to the CPU to/from the GPU prevents to carry out a real-time application. In this paper, an algorithm with a pipeline structure is developed, what allows to perform a massive real-time convolution.

Key words: Massive convolution, Multichannel audio processing, FFT, GPU, CUDA

1 Introduction

Multichannel acoustic signal processing has experienced a great development in recent years, due to an increase in the number of sound sources used in playback applications available to users, and the growing need to incorporate new effects and to improve the experience of hearing [1].

Several effects, as the synthesis of 3D sound, are achieved through multichannel signal processing, with an efficient implementation of the massive convolution. It consists of carrying out different convolutions of different channels in a parallel way. All these operations require high computing capacity.

GPU offer us the possibility of parallelizing these operations, letting us not only to obtain the result of the processing in much less time, but also to free up CPU resources.

The paper is organized as follows. Section 2 describes the convolution and the problem of its implementation on GPU. In Section 3, an efficient GPU implementation of massive convolution is presented. Section 4 is reserved for the results of different tests on GPU. Finally Sections 5 is devoted to the conclusions, and the paper closes with some references.

2 Convolution on GPU

2.1 Convolution Algorithm in GPU

The convolution describes the behavior of a linear, time-invariant discrete-time system with input signal x and output signal y [2]:

$$y[n] = \sum_{j=0}^{N-1} x[j]h[n-j], \tag{1}$$

Signal x will the input to the system, in our case, samples from audio signal. The known signal h is the response of the system to a unit-pulse input. The output signal y contains the samples of the desired acoustic effects. N, M and $L = N + M - 1$ will be the lenghts of x, h and y respectively.

Convolution theorem [2] states that if x and h are padded with zeros to the length L, then the Discrete Fourier Transform of y is the point-wise product of the Discrete Fourier Transforms of x and h. In other words, convolution in one domain (e.g., time domain) equals point-wise multiplication in the other domain (e.g., frequency domain). This way of computing the convolution is advantageous because the number of operations is smaller than implementing the convolution in the time domain.

There exist different libraries that implement efficient FFT algorithms. They allow to obtain the Discrete Fourier Transform of a signal, in a CPU (like MKL [7] or IPP [8]) or in a GPU (like CUFFT [5] from NVIDIA whose performances have been analized in [9]).

The use of a GPU may offer two benefits: less execution time due to a high level of parallelization of the computations and the freeing up resources of the CPU.

Let us consider x and input audio signal, h an acoustic filter and y the desired output audio signal of our system. The execution of the convolution using a GPU can be enumerated in the next steps: first, the lenghts of x and h must be checked; then both signals must be transferred from the CPU to the GPU; next, the FFT (from CUFFT) is applied to each signal obtaining X and H; the frequency domain output Y is obtained multiplying point-wise X and H; the time domain output y is obtained applying the IFFT to Y; Finally, y is transferred from the GPU to the CPU. Figure 1 shows this process.

We can observe that:

1. Long time of the algorithm is spent in transfers between the CPU and the GPU.

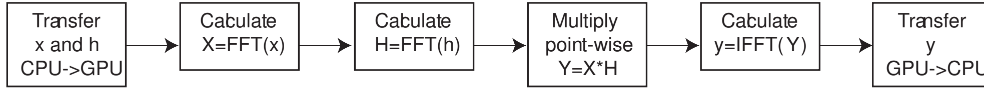


Figure 1: Steps in order to calculate convolution of signals x and h on GPU.

2. Signals must be sent to the GPU before beginning the operations, and the whole output signal must be received at the CPU to be reproduced.

In spite of the parallelism in operations that offered by the GPU, the transfer time penalty prevents us to carry out a real time application in a GPU. More even, if the signal x is compound by several channels, then a multiple convolutions would be required. On the other hand, if a CPU is used to make a massive convolution, all our resources would be used and no more applications could be run at the same time.

2.2 Convolution of Large Signals

In a real-time environment, the length of signal x can not be known a priori. There exist techniques that allow us to cut the signal in chunks, and from the convolution of each chunk we can obtain the convolution of the whole signal. One of these techniques is called overlap-save [3] and it consists of:

1. Chunks of L samples are taken, where L will be either the next power of two, bigger than M (length of h) or 512.
2. In the first chunk, the first $M-1$ samples will be padded with zeros.
3. From the second and following chunks, the first $M-1$ samples will be duplicated from the last $M-1$ samples of the previous chunk.
4. Following the steps of the previous subsection, $y_0[n]$, $y_1[n]$, $y_2[n]$, ..., are obtained as the result of the convolution of $x_0[n]$, $x_1[n]$, $x_2[n]$, ..., with h respectively.
5. From each chunk result, the first $M-1$ samples will not be valid values so they will be eliminated.

3 Pipelined Algorithm of convolution on GPU

Recently, the new CUDA toolkit 3.0 [4] lets use CUFFT [5] with the property *concurrent copy and execution*. Therefore, the latency of transferring data from the CPU to the GPU and vice versa can be overlapped by computations. That fits perfectly with the steps described in the previous section.

In fact, in order to maximize the overlapping of the computations in the GPU and the communications between CPU and GPU, a matrix can be configured with each chunk obtained with the signal samples.

In this matrix, the first $M-1$ values of one row will coincide with the last $M-1$ values of the previous one, except the first configured matrix at the start of the algorithm whose first $M-1$ values from the first row will be zeros. This matrix will have the following shape with R rows and L columns:

The last $M-1$ samples from the last row of the matrix will be kept in an internal buffer in order to occupy the first $M-1$ positions of the next matrix to be filled.

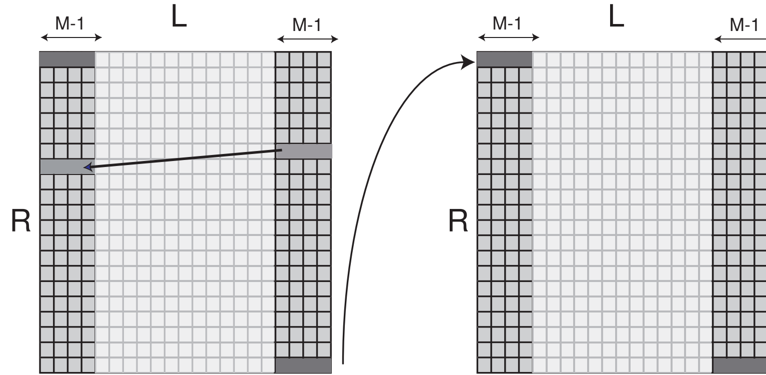


Figure 2: Samples are sent to GPU in a chunks-matrix- configuration. $matrix_{i-1}$ (left side) shares a $M-1$ samples with, previously sent to $matrix_i$ (right side).

The *concurrent copy and execution* property lets sending the $matrix_i$ using the asynchronous transfer while carrying out the other tasks in parallel:

1. Beginning to configure the next matrix $matrix_{i+1}$ with new samples.
2. Execution of the convolution algorithm at the GPU with the matrix which was previously sent $matrix_{i-1}$. So, R chunk-convolutions (the matrix sent to the GPU has R rows) will be executed in a parallel way.
3. Chunk-convolutions results from $matrix_{i-2}$ will be sent back from GPU to the CPU

The unit-impulse response h will have been sent to the GPU before sending the first matrix. h will be kept in the GPU memory and reused over and over with all the convolutions.

All the previous tasks can be viewed in a pipeline configuration as shown in Figure 3.

3.1 Extrapolation to a multichannel signal: Massive Convolution

Dealing with a multichannel-signal will be totally scalable due to an equal distribution of the resources. So, the matrix that contains the chunks will be divided in the number of channels of the signal.

In the same way if more than one effect is going to be applied, each of the impulse responses would be sent and kept in the GPU.

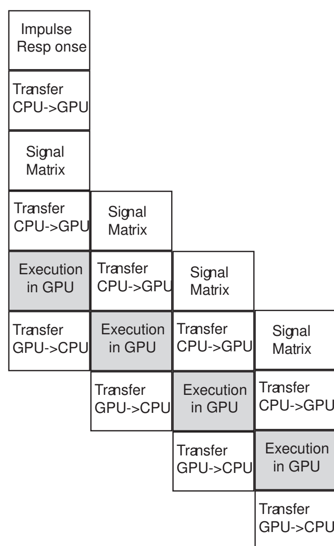


Figure 3: Pipeline Configuration.

The parallel architecture of the GPU give us freedom to configure several possibilities such as: Apply the same effect to all the channel signals. Apply one specific effect to one channel and other effect to the rest. Even, apply one effect to a determinate number of channels.

So, all the combinations are possible, and therefore, the possibilities of mixing several acoustic effects, as well.

4 Results

Many are the tests that are being carried out in order to know the achievement of the massive convolution. One of the most significative resolves around the comparison between the convolution algorithm in GPU, shown at Figure 1 (implemented, for example, in [6]), and the pipeline algorithm. In this case, as it can be shown in table 2, the second achieves the convolution of the signal in half of the time than the first one.

Test has been carried out on a signal x and a impulse-response h compound by 176400 samples and 220 coefficients respectively. Results are shown in Table 1. A time comparison with the basic convolution algorithm is shown in Table 2.

As it can be appreciated, performance are improved if an algorithm in convolution with a pipeline configuration is used.

5 Conclusions

With this article, it has been revealed that GPU can be used for carrying out a massive convolution of multichannel-acoustic signals in real-time. It has been possible thanks to the pipeline configuration that is now available with the new CUDA Toolkit 3.0. It

must be pointed that one of the advantages of using a GPU, is the fact of freeing up CPU resources, letting us to run more applications in the CPU.

R//L	**512**	**1024**	**2048**	**4096**	**8192**
32	625.92	833.28	802.83	836.83	870.70
64	730.80	745.64	809.62	890.21	876.21
128	761.43	819.39	865.55	906.63	981.27
256	870.89	913.35	1094.1	995.02	1111.4
512	937.24	951.26	949.20	994.04	1206.3
1024	1005.2	110.51	1080.4	1278.7	1622.8
2048	1089.1	1274.8	1436.7	1603.1	1969.1

Table 1: Time in miliseconds of the pipelined algorithm varying number of rows (R) and columns (C) of the matrix.

Type of Algorithm	**Time**
Convolution Algorithm in GPU	1330ms
Configuration Pipeline (Best Performance)	802.83ms

Table 2: Comparison between basic and pipelined algorithm.

Acknowledgements

This work was financially supported by the Spanish Ministerio de Ciencia e Innovacion(Projects TIN2008-06570-C04-02 and TEC2009-13741), Universidad Politecnica de Valencia through "Programa de Apoyo a la Investigacion y Desarrollo (PAID-05-09)", Regional Government Generalitat Valenciana through grant PROMETEO/2009/013 and NVIDIA through CUDA Community program.

References

[1] E. TORICK, *Highlights in the history of multichannel sound*, J. Audio. Eng. Soc. **46** (1998) 27–31.

[2] S. S. SOLIMAN, D. S.MANDYAM AND M. D. SRINATH, *Trade paperback, Prentice Hall*, ISBN:0135184738, 1997.

[3] Overlap-save method: “http://en.wikipedia.org/wiki/Overlap-save-method”

[4] CUDA Toolkit 3.0: “http://developer.nvidia.com/object/cuda-3-0-downloads.html”

[5] CUFFT library: "http://developer.download.nvidia.com/compute/cuda/3-0/toolkit/docs/CUFFT-Library-3.0.pdf"

[6] GPU Computing SDK code samples: "http://developer.nvidia.com/object/cuda-3-0-downloads.html"

[7] MKL library: "http://software.intel.com/en-us/intel-mkl/"

[8] MKL library: "http://software.intel.com/en-us/intel-ipp/"

[9] P. Alonso, J. A. Belloch, A. Gonzalez, E. S. Quintana-Orti, A. Remon and A. M. Vidal, *Evaluacion de bibliotecas de altas prestaciones para el calculo de la FFT en procesadores multinúcleo y GPUs*, II Workshop en Aplicaciones de Nuevas Arquitecturas de Consumo y Altas Prestaciones, Freeman, Mostoles (Madrid), 2009.

Session Outline

Real-time Multichannel Audio Convolution

Dealing with multichannel signals takes a high computational cost requiring the use of multiple resources from the CPU. Massive convolution is the basic operation in multichannel acoustic signal processing. Graphics Processing Unit (GPU) can carry out a multichannel convolution faster and also freeing up resources from CPU. However, the fact of transferring data from/to the CPU to/from the GPU has prevented to carry out meaningful real-time audio applications. In this presentation, an algorithm with a pipeline structure is developed, which allows performing a scalable massive real-time convolution. In order to obtain special sound effects, such as a synthesis of 3D sound, different filters must be applied to multichannel audio signal. Latency time spent in transfers CPU <-> GPU damages the good performance obtained with the GPU parallelism. The algorithm we present is focused on the overlap-save technique to carry out the convolution. It is based on the convolution of signal fragments with additional processing. Moreover, the new CUDA toolkit 3.1 lets use CUFFT with the concurrent copy and execution property. Therefore, the latency of transferring data from the CPU to the GPU and vice versa can be overlapped by computations. This allows obtaining the best performance of the algorithm and at the same time to exploit the parallelism of the CUDA architecture.

During the presentation, a basic demo using a laptop with an NVIDIA GTS 360M will be carried out. In that demo, it will be showed how a real-time audio application is developed on a GPU. In that case, as a laptop has only two loudspeakers, a two channels signal will be used. The application example will be a cross-talk application. Different filters will be applied to the multichannel signal in a real-time environment. Different combinations of the output signals let us achieve different acoustic effects.

PARALLELIZATION OF BEAMFORMING DESIGN AND FILTERING FOR MICROPHONE ARRAY APPLICATIONS

Jorge Lorente[1], *Gema Piñero*[1], *Antonio M. Vidal*[2], *Jose Antonio Belloch*[1], *Alberto González*[1]

[1] Institute of Telecommunications and Multimedia Applications (iTEAM)
[2] Interdisciplinary Group of Computer and Communications (INCO2)
Universidad Politécnica de Valencia
{jorlogi, gpinyero}@iteam.upv.es

ABSTRACT

For a few decades beamforming algorithms for microphone arrays have been studied and developed in order to improve the signal-to-noise ratio of the received signals, or to recover spatially separated signals considering their different directions of arrival. Additionally, beamforming algorithms have been used for the recovery of acoustic signals from their observations when they are corrupted by noise, reverberation and other interfering signals, but due to its limited performance further research regarding this matter is still needed. One of the main limitations of microphone arrays algorithms for audio applications has been their high computational cost in real acoustic environments where real-time signal processing is absolutely required. Our research focuses on the computational cost of the classical Linear Constrained Minimum Variance (LCMV) method. Specifically, we propose a new approach for the design and filtering of LCMV beamformers on Graphic Processing Unit (GPU) platforms, whose reduced computational time allows to run the LCMV algorithm in real-time audio applications.

Index Terms— Audio Signal Processing, Beamforming, Microphone Arrays, GPU

1. INTRODUCTION

While digital signal processors and other computational devices have substantially increased their performance, we have been able to address increasingly complex problems and to run solutions in a short time. This has benefited both typical real-time applications that are common in signal processing, and other signal processing applications that imply the management of very large data sets which cannot be addressed within a reasonable time without the help of advanced computational tools.

Recently, specialized hardware with hundreds of simple cores (many-core) are available in the form of cheap, widely-spread NVIDIA and AMD/ATI Graphic Processor Units (GPU) incorporated in any standard graphics card. For example, 448 cores are embedded in the newest NVIDIA arquitecture, called Fermi. The signal processing field should start to take into account the computational advantages offered by the multicore / manycore architectures. In fact, some signal processing applications are already taking advantage of these opportunities [1–4] but a much bigger effort is still needed.

This work has been supported by Spanish Government through grant TEC2009-13741, Regional Government Generalitat Valenciana through grant PROMETEO/2009/013 and NVIDIA through CUDA Community program.

Although the programming of classical signal processing algorithms in many core architectures is not trivial [5], there are many tools to help software developers to adapt their programs to the new architectures [6], particularly with regular codes that are intensive in floating-point arithmetic like those frequently arising in signal processing applications. We can cite, for example, the following libraries: CUBLAS and CUFFT, implementations on CUDA of the well-known BLAS computational kernels [7] and FFT algorithm [8] respectively; CULA, implementation of the LAPACK library for GPU [9], and JACKET, which offers the functionality of MATLAB on GPU [10]. The work presented herein focuses on the GPU implementation of the Linear Constrained Minimum Variance (LCMV) method, proposing new strategies for the design and filtering of this beamforming algorithm in order to allow to be used in real-time audio applications.

This paper is organized as follows. Section 2 describes the signal model used in the microphone array application. Section 3 introduces classical LCMV algorithm. Section 4 outlines two efficient implementations of LCMV algorithm on GPU and section 5 explains multichannel filtering with CUDA. Finally, sections 6 and 7 show some performance results and the main conclusions respectively.

2. SIGNAL MODEL

Consider the system of Figure 1 where two loudspeakers are emitting two independent signals, $s_1(k)$ and $s_2(k)$, respectively. At the other part of the room, three microphones are recording the mix of the two signals corrupted by noise and room reverberation. The problem is how to recover $s_1(k)$ or $s_2(k)$ by means of the signals recorded at the microphones. The approach taken herein makes use of signal processing algorithms to design the broadband beamformers (filters g_n in Figure 1), once all the room channel responses (h_{nm} in Figure 1) are known. This system can be modelled as a multichannel system with 2 inputs (loudspeakers) and 3 outputs (microphones), and the generalization to a Multiple Input Multiple Output (MIMO) system can be easily addressed [11].

According to Figure 1, the output of the nth microphone is given by:

$$x_n(k) = \sum_{m=1}^{M} \sum_{j=1}^{L_h} h_{nm}(j) s_m(k-j) + v_n(k), \qquad (1)$$

where $n = 1, 2, \ldots, N$, being N the number of microphones and M the number of source signals, that is equal to the number of loudspeakers in Figure 1. L_h is the length of the longest room impulse response of all the acoustic channels h_{nm} and $v_n(k)$ is the noise

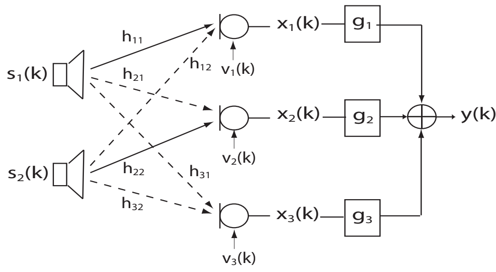


Fig. 1. Signal model for $M = 2$ loudspeakers (inputs) and $N = 3$ microphones (outputs).

signal. For the sake of clarity the noise term $v_n(k)$ of (1) will not be considered in the following signal model.

In order to improve computation efficiency, equation (1) can be rewritten in vector/matrix form as:

$$x_n(k) = \sum_{m=1}^{M} \mathbf{h}_{nm}^T \mathbf{s}_m(k), \tag{2}$$

where $\mathbf{s}_m(k)$ is the column vector defined as $\mathbf{s}_m(k) = [s_m(k) \quad s_m(k-1) \quad \cdots \quad s_m(k - L_h + 1)]^T$, $\mathbf{h}_{nm}$ is the $\mathcal{R}^{L_h \times 1}$ acoustic channel vector from loudspeaker m to microphone n and $()^T$ denotes the transpose of a vector or a matrix.

Considering now the problem of recovering source signals $s_m(k)$ from the recorded observations $x_n(k)$, beamforming filters g_n of Figure 1 have to be designed in such a way that the output signal $y(k)$ is a good estimate of $s_m(k)$, that is, $y(k) = \hat{s}_m(k - \tau)$ with minimum error. Given a maximum length of L_g taps for each of the N g_n filters, the broadband beamforming output signal is expressed in a similar form as in (2):

$$y(k) = \sum_{n=1}^{N} \mathbf{g}_n^T \mathbf{x}_n(k), \tag{3}$$

where $\mathbf{g}_n$ is the $\mathcal{R}^{L_g \times 1}$ vector containing the ordered taps of beamforming filters g_n of Figure 1, and $\mathbf{x}_n(k)$ is the column vector defined as $\mathbf{x}_n(k) = [x_n(k) \quad x_n(k-1) \quad \cdots \quad x_n(k - L_g + 1)]^T$. In order to compute in matrix form the whole vector $\mathbf{x}_n(k)$ used in (3), equation (2) has to be rewritten in compact form as:

$$\mathbf{x}_n(k) = \mathbf{H}_n \mathbf{s}(k), \tag{4}$$

where column vector $\mathbf{s}(k) = [\mathbf{s}_1^T(k) \quad \cdots \quad \mathbf{s}_M^T(k)]^T$ has now dimensions $[ML \times 1]$ due to the new length $L = L_g + L_h - 1$ of vectors $\mathbf{s}_m(k)$. Accordingly, definition of matrix $\mathbf{H}_n$ is given by $\mathbf{H}_n = [\mathbf{H}_{n1} \quad \cdots \quad \mathbf{H}_{nM}]$ where $\mathbf{H}_{nm}$ are Sylvester matrices of size $[L_g \times L]$ defined as follows:

$$\mathbf{H}_{nm} = \begin{bmatrix} \mathbf{h}_{nm}^T & 0 & 0 & \cdots & 0 \\ 0 & \mathbf{h}_{nm}^T & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & \mathbf{h}_{nm}^T \end{bmatrix}$$

Finally, the global signal model of Figure 1 is expressed by:

$$\mathbf{x}(k) = \mathbf{H}\mathbf{s}(k), \tag{5}$$

where $\mathbf{x}(k) = [\mathbf{x}_1^T(k) \quad \cdots \quad \mathbf{x}_N^T(k)]^T$ has length NL_g, and the global MIMO acoustic channel matrix $\mathbf{H}$ is given by $\mathbf{H} = [\mathbf{H}_1^T \cdots \mathbf{H}_N^T]^T$.

Once the vector model of the microphone signals $\mathbf{x}_n(k)$ has been stated by means of (4) and (5), the broadband beamformer filters $\mathbf{g}_n$ have to be designed and calculated.

3. LCMV BEAMFORMING ALGORITHM

In [11], Benesty et al. present an excellent state-of-the-art of the main algorithms used in audio applications. Due to its better performance, we have focused our study on LCMV (Linearly Costrained Minimum Variance) algorithm, which calculates beamforming filters as:

$$\mathbf{g}^{\mathrm{LCMV}} = \mathbf{R}_x^{-1} \mathbf{H}_{:m} [\mathbf{H}_{:m}^T \mathbf{R}_x^{-1} \mathbf{H}_{:m}]^{-1} \mathbf{u}_m, \tag{6}$$

where $\mathbf{g}^{\mathrm{LCMV}}$ is formed by the concatenation of filters $\mathbf{g}_n$, that is, $\mathbf{g}^{\mathrm{LCMV}} = [\mathbf{g}_1^T \quad \cdots \quad \mathbf{g}_N^T]^T$, matrix $\mathbf{H}_{:m}$ is a partition of the channel impulse matrix that only includes the impulse responses from mth source to the N microphones, matrix $\hat{\mathbf{R}}_x$ is the correlation matrix of the recorded signals and $\mathbf{u}_m$ is a vector of zeros except for a one at the proper place in order to compensate the room impulse response delay.

The estimated correlation matrix of spatially sampled signals, $x_n(k)$, is commonly known in the literature as the Sample Correlation Matrix (SCM), and its expression is given by:

$$\hat{\mathbf{R}}_x = \frac{1}{K} \sum_{k=1}^{K}, \; \mathbf{x}(k)\mathbf{x}^{\mathbf{T}}(k) \tag{7}$$

where vector $\mathbf{x}(k)$ is the same as in (5). Regarding the squared size of the SCM, $[NL_g \times NL_g]$, L_g is the number of taps of the largest filter $\mathbf{g}_n$ and depends on the maximum length of room impulse responses, L_h.

In order to avoid loops, the calculation of SCM is done redifining $\hat{\mathbf{R}}_x$ of (7) as:

$$\hat{\mathbf{R}}_x = \mathbf{X} \cdot \mathbf{X}^T, \tag{8}$$

where $\mathbf{X} \in \Re^{[NL_g \times K]}$ is defined as:

$$\mathbf{X} = \frac{1}{\sqrt{K}} \begin{pmatrix} \mathbf{x}_1(k) & \mathbf{x}_1(k+1) & \cdots & \mathbf{x}_1(k+K-1) \\ \mathbf{x}_2(k) & \mathbf{x}_2(k+1) & \cdots & \mathbf{x}_2(k+K-1) \\ \vdots & \vdots & \dots & \vdots \\ \mathbf{x}_N(k) & \mathbf{x}_N(k+1) & \cdots & \mathbf{x}_N(k+K-1) \end{pmatrix} \tag{9}$$

4. EFFICIENT IMPLEMENTATION OF LCMV ALGORITHM

This section presents two efficient implementations of the LCMV algorithm based on algebra solutions, thus avoiding inverse computation (6) and SCM calculation (8).

4.1. Direct method

Considering $\hat{\mathbf{R}}_x$ defined in (8), let us denote matrix $\mathbf{W}$ as:

$$\mathbf{W} = \hat{\mathbf{R}}_x^{-1} \mathbf{H}_{:m}. \tag{10}$$

Consequently, the LCMV beamformer filter can be obtained from (6) through the equivalent expression using $\mathbf{W}$:

$$\mathbf{g}^{\mathrm{LCMV}} = \mathbf{W}[\mathbf{H}_{:m}^T \mathbf{W}]^{-1} \mathbf{u}_m. \tag{11}$$

We can also consider $\mathbf{A} = \mathbf{H}_{:m}^T \mathbf{W}$ and hence $\mathbf{b}_m = \mathbf{A}^{-1} \cdot \mathbf{u}_m$, so the LCMV beamformer filter $\mathbf{g}^{\mathrm{LCMV}}$ in (11) can be obtained from the equivalent expression using $\mathbf{W}$ and $\mathbf{b}_m$:

$$\mathbf{g}^{\mathrm{LCMV}} = \mathbf{W} \cdot \mathbf{b}_m. \tag{12}$$

4.2. QR decomposition based method

This method is based on a QR decomposition of matrix $\mathbf{X}^T$. Consider now $\mathbf{X}^T = \mathbf{Q} \cdot \mathbf{L}$, where $\mathbf{Q}$ is an orthogonal matrix (its columns are orthogonal unit vectors meaning $\mathbf{Q}^T \cdot \mathbf{Q} = \mathbf{I}$) and $\mathbf{L}$ is an upper triangular matrix.

After QR decomposition, we can redefine (8) as:

$$\hat{\mathbf{R}}_x = \mathbf{X}\mathbf{X}^T = \mathbf{R}^T\mathbf{Q}^T\mathbf{Q}\mathbf{L} = \mathbf{L}^T\mathbf{L}, \tag{13}$$

and now $\mathbf{W}$ of (10) can be defined as:

$$\mathbf{W} = \hat{\mathbf{R}}_x^{-1}\mathbf{H}_{:m} = (\mathbf{L}^T\mathbf{L})^{-1}\mathbf{H}_{:m} = \mathbf{L}^{-1}\mathbf{Z}, \tag{14}$$

where $\mathbf{Z} = \mathbf{L}^{T^{-1}}\mathbf{H}_{:m}$. Using matrix $\mathbf{Z}$, matrix $\mathbf{A}$ defined in direct implementation can be redefined as:

$$\mathbf{A} = \mathbf{H}_{:m}^T\mathbf{W} = \mathbf{H}_{:m}^T\mathbf{L}^{-1}\mathbf{Z} = \mathbf{Z}^T\mathbf{Z}. \tag{15}$$

Therefore $\mathbf{g}^{\mathrm{LCMV}} = \mathbf{W} \cdot \mathbf{b}_m = \mathbf{L}^{-1}\mathbf{Z} \cdot \mathbf{b}_m$, and defining vector $\mathbf{y} = \mathbf{Z} \cdot \mathbf{b}_m$, we can express the LCMV beamformer filter $\mathbf{g}^{\mathrm{LCMV}}$ as follows:

$$\mathbf{g}^{\mathrm{LCMV}} = \mathbf{L}^{-1}\mathbf{y}. \tag{16}$$

4.3. Implementation

Seeking the most efficient implementation, it is necessary to note that both methods have to solve linear equations of type $\mathbf{C} \cdot \mathbf{u} = \mathbf{v}$, where $\mathbf{u} = \mathbf{C}^{-1}\mathbf{v}$ as for instance in (10). It is well known in the computing literature that matrix inversion is not the most efficient way to solve the problem, so left matrix division, also called backslash, has been used in the above solutions. Therefore, both methods avoid matrix inverse calculations of (6) and improve consequently its computational time.

Regarding specifically QR method, it also avoids SCM calculation (8), which improves even more LCMV computational time. Furthermore, since $\mathbf{L}$ is a triangular matrix, linear equations resolution of type (16) performs faster than if full matrices were involved.

5. CUDA FILTERING

Once the filters $\mathbf{g}_n^{\mathrm{LCMV}}$ have been calculated the desired signal can be recovered through system depicted in Figure 1. However, multichannel filtering has a large computational cost and regarding our goal of a whole GPU implementation to free up resources from the CPU, the GPU implementation of multichannel filtering must be considered. The algorithm we present is based on the overlap-save technique [12] and carries out a multichannel convolution.

Regarding the transfer of data from/to the CPU to/from the GPU, CUFFT of CUDA toolkit 3.0 owns the concurrent copy and execution property. Therefore, the latency of transferring data from the CPU to the GPU and vice versa can be overlapped by computations. This allows to obtain the best performance of the algorithm and at the same time to exploit the parallelism of the CUDA architecture. For this purpose, a Matrix-Signal $\mathbf{x}$ is configured with the chunks of the signal. The filters $\mathbf{h}$ are sent only once to the GPU. As long as the matrix signal is filled, this matrix will be sent to the GPU and a new matrix will begin to be filled. In the same way, as long as operations end at the GPU, the result matrix will be sent back to the CPU and a pipeline configuration can be achieved easily. See reference [13] for further details.

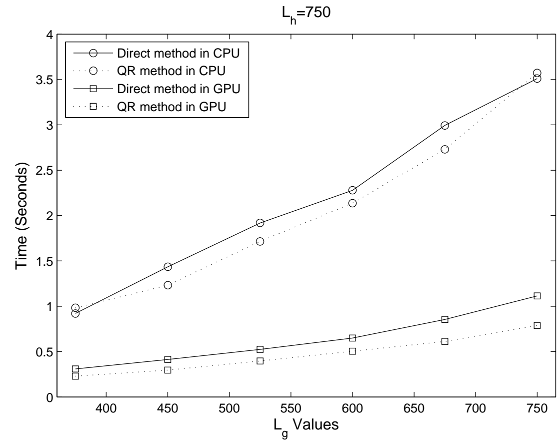


Fig. 2. Computational time of Direct and QR methods for $L_h = 750$, for different L_g values.

6. TESTING RESULTS

Regarding efficient implementations of sections 4.1 and 4.2, both algorithms have been run on CPU (Intel i7 2.67Ghz, 4Gb of RAM) and on GPU (GeForce GTX285) using respectively as programming interfaces Matlab and CUDA libraries (CULA and CUBLAS) over Microsoft Visual Studio. Both methods have also been tested for a simulated room with impulse responses of maximum length $L_h = 750$ and different number of beamformer coefficients, L_g.

Results are shown in Figure 2. It can be seen that QR based method outperforms direct method in both processing platforms. Furthermore, GPU implementation can reach a reduction (speedup) about six times CPU time execution for any of the methods.

6.1. Achieving beamforming design and filtering in real-time

In order to achieve beamforming design and filtering in real-time, several blocks of microphone observations must be stacked to calculate SCM in the direct method, or to perform QR decomposition in QR-based method. On the other hand, an unavoidable but not critical initial delay must be taken into account when working by blocks (see Figure 3). Under these assumptions, the minimum block size must be set by the minimum number of microphone observations needed to make the SCM to have full rank. Let us call this the *full-rank requirement*. This minimum size depends of L_g, which at the same time depends on L_h. For example, for a $L_h = 750$, L_g must be grater than $L_h/2$ [11], so the minimum length of the filters must be $L_g = 375$, and the minimum number of microphone observations to make a block should be $L_g + N \cdot Lg - 1 = 1499$. Working with this block size and using 44100 Hz as the sample frequency, a block should be processed in less time than $1499/44100 = 0{,}0340$ seconds (see figure 3).

Notice that the whole real-time processing of a block includes the design of the beamformers and the following filtering. As it can be appreciated in figure 2, beamformers are calculated in $0{,}23$ seconds, which is almost 7 times longer than the time allowed in *full-rank requirement*. The proposed solution then is to take largest blocks but use the same SCM / filter for processing them. In particular, blocks 8 times larger than the minimum for *full-rank requirement* have been taken. Consequently a slot of $8 * 1499/44100 = 0{,}272$

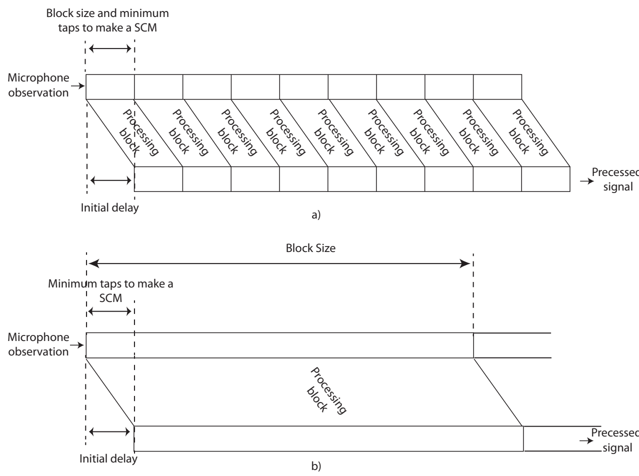


Fig. 3. a) Working with minimum size block of microphones observations. b) Working with eight times minimum size block of microphones observations.

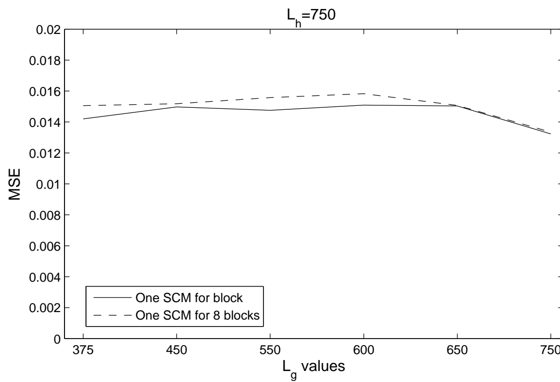


Fig. 4. MSE of the recovered signal with filters designed for each block and with filters designed every eight blocks.

seconds is allowed to design the beamformers and to successively filter the 8 blocks. Given the filter design computational time of figure 2, there are $0{,}272 - 0{,}23 = 0{,}042$ seconds left to do the filtering. Whereas the filtering is performed in CUDA and the three channels are filtered in less than 0.009 seconds [14], the whole design and filtering process can be achieved in real-time.

Regarding the solution adopted to work in real-time, the error introduced by calculating the correlation matrix for several blocks instead of calculating it for each block has been analyzed. The Mean Square Error (MSE) of the recovered signal with filters designed for each block and the MSE obtained with filters designed every eight blocks have been compared and depicted in figure 4. It can be noticed that for the different values of L_g the error does not increase significantly between the two ways of working with blocks.

7. CONCLUSIONS

The LCMV microphone-array algorithm has been studied in order to implement it in real-time applications. Beamformer design has been developed for GPU using CULA and CUBLAS, and the corresponding filtering using CUDA has also been proposed. Results have shown that LCMV algorithm implemented in GPU is near 6 times faster than using CPU, so audio applications could be performed in real-time with the additional advantage of freeing up resources from CPU. Furthermore, the error of the recovered signal when designing the filter every 8 blocks does not seem critical for the beamforming performance, resulting in a good approach for real-time beamforming implementations.

8. REFERENCES

[1] E. Gallo and N. Tsingos, "Efficient 3D audio processing with the GPU", in *ACM Workshop on General Purpose Computing on Graphics Processors*, Los Angeles, USA, vol. 1, pp. C–42, August 2004.

[2] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone and J.C. Phillips, "GPU computing", *Proc. of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[3] M.D. McCool, "Signal processing and general-purpose computing and GPUs", *IEEE Signal Processing Magazine*, vol. 24, no. 3, pp. 109–114, May 2007.

[4] Rob V. van Nieuwpoort and John W. Romein, "Building Correlators with Many-Core Hardware", *IEEE Signal Processing Magazine*, Vol. 27, Issue 2, pp. 108–117, March 2010.

[5] David Patterson, "The Trouble with Multi-Core", *IEEE Spectrum*, Vol. 47, Issue 7, pp. 28–32 and 52–53, June 2010.

[6] Trista P. Chen and Yen-Kuang Chen, "Challenges and Opportunities of Obtaining Performance from Multi-Core CPUs and Many-Core GPUs", *ICASSP 2009*.

[7] CUBLAS Library 3.0, available online: "http://developer.download.nvidia.com"

[8] CUFFT Library 1.1, available online: "http://developer.download.nvidia.com"

[9] CULA Library, available online: "http://www.culatools.com"

[10] JACKET for MATLAB Product Page, available online: "http://www.accelereyes.com/resources/literature"

[11] J. Benesty, J. Chen, Y. Huang and J. Dmochowski, "On microphone-array beamforming from a MIMO acoustic signal processing perspective", *IEEE Trans. on Audio, Speech and Lenguage Processing*, vol. 15, no. 3, pp. 1053–1065, March 2007.

[12] A.V. Oppenheim, R.W. Schafer, "Discrete-Time Signal Processing. 3rd.", Prentice Hall Press, 2009

[13] J.A. Belloch, A.M. Vidal, F.J. Martinez-Zaldivar, A. Gonzalez, "Multichannel acoustic signal processing on GPU", *10th International Conference on Computational and Mathematical Methods in Science and Engineering*, Vol 1, pag 181-187,June 2010.

[14] J.A. Belloch, A. Gonzalez, A.M. Vidal, F.J. Martinez-Zaldivar, "Real-time Multichannel Audio Convolution", *GPU Technology Conference (GTC 2010)*, *http://nvidia.com/content/GTC-2010/flvs/2116_GTC2010.mp4*