UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Performance Improvement of Multichannel Audio by Graphics Processing Units

DOCTORAL THESIS

by
José Antonio Belloch Rodríguez

Supervisor:
Prof. Antonio M. Vidal Maciá
Prof. Alberto González Salvador

Valencia, Spain
September 2014

To Lola

Abstract

Multichannel acoustic signal processing has undergone major development in recent years due to the increased complexity of current audio processing applications. People want to collaborate through communication with the feeling of being together and sharing the same environment, what is considered as Immersive Audio Schemes. In this phenomenon, several acoustic effects are involved: 3D spatial sound, room compensation, crosstalk cancelation, sound source localization, among others. However, high computing capacity is required to achieve any of these effects in a real large-scale system, what represents a considerable limitation for real-time applications.

The increase of the computational capacity has been historically linked to the number of transistors in a chip. However, nowadays the improvements in the computational capacity are mainly given by increasing the number of processing units, i.e expanding parallelism in computing. This is the case of the Graphics Processing Units (GPUs), that own now thousands of computing cores. GPUs were traditionally related to graphic or image applications, but new releases in the GPU programming environments, CUDA or OpenCL, allowed that most applications were computationally accelerated in fields beyond graphics. This thesis aims to demonstrate that GPUs are totally valid tools to carry out audio applications that require high computational resources. To this end, different applications in the field of audio processing are studied and performed using GPUs. This manuscript also analyzes and solves possible limitations in each GPU-based implementation both from the acoustic point of view as from the computational point of view. In this document, we have addressed the following problems:

Most of audio applications are based on massive filtering. Thus, the first implementation to undertake is a fundamental operation in the audio processing: the convolution. It has been first developed as a computational kernel and afterwards used for an application that combines multiples convolutions concurrently: generalized crosstalk cancellation and equalization. The proposed implementation can successfully manage two different and common situations: size of buffers that are much larger than the size of the filters and size of buffers that are much smaller than the size of the filters.

Two spatial audio applications that use the GPU as a co-processor have

been developed from the massive multichannel filtering. First application deals with binaural audio. Its main feature is that this application is able to synthesize sound sources in spatial positions that are not included in the database of HRTF and to generate smoothly movements of sound sources. Both features were designed after different tests (objective and subjective). The performance regarding number of sound source that could be rendered in real time was assessed on GPUs with different GPU architectures. A similar performance is measured in a Wave Field Synthesis system (second spatial audio application) that is composed of 96 loudspeakers. The proposed GPU-based implementation is able to reduce the room effects during the sound source rendering.

A well-known approach for sound source localization in noisy and reverberant environments is also addressed on a multi-GPU system. This is the case of the Steered Response Power with Phase Transform (SRP-PHAT) algorithm. Since localization accuracy can be improved by using high-resolution spatial grids and a high number of microphones, accurate acoustic localization systems require high computational power. The solutions implemented in this thesis are evaluated both from localization and from computational performance points of view, taking into account different acoustic environments, and always from a real-time implementation perspective.

Finally, This manuscript addresses also massive multichannel filtering when the filters present an Infinite Impulse Response (IIR). Two cases are analyzed in this manuscript: 1) IIR filters composed of multiple second-order sections, and 2) IIR filters that presents an allpass response. Both cases are used to develop and accelerate two different applications: 1) to execute multiple Equalizations in a WFS system, and 2) to reduce the dynamic range in an audio signal.

Keywords: Multichannel filtering, Spatial Sound, Sound Source localization, Graphics Processing Units.

Resumen

El procesado de audio multicanal ha experimentado un gran desarrollo en los últimos años y como consecuencia se ha producido un aumento notable de la complejidad computacional en las nuevas aplicaciones. Actualmente, se pretende que la telecomunicación ofrezca una sensación de cercanía, compartiendo incluso el mismo entorno entre usuarios distantes. Es lo que llamamos: Esquemas de Audio Inmersivo. En este fenómeno intervienen varios efectos acústicos: sonido espacial 3D, compensación de salas, cancelación crosstalk, localización de fuentes sonoras, entre otros. Pero a su vez, para llevar a cabo cualquiera de estos efectos en un sistema real, se necesita una alta capacidad computacional, lo que representa una severa limitación cuando se trata de ejecutar dichas aplicaciones en tiempo real.

El aumento de la capacidad computacional ha ido históricamente unido al número de transistores en un chip. Actualmente, las mejoras en la capacidad computacional están íntimamente ligadas al número de unidades de proceso que tiene un computador, lo que permite un alto grado de paralelismo en computación. Este es el caso de las Unidades de Procesamiento Gráfico (GPUs, *Graphics Processing Units*), que poseen actualmente miles de núcleos computacionales. Las GPUs se han relacionado tradicionalmente con la computación gráfica o el tratamiento de imágenes, pero con la aparición de nuevos entornos de programación para GPUs (CUDA o OpenCL) muchas aplicaciones de otros campos científicos han podido ser aceleradas mediante su implementación en las GPUs. Esta tesis tiene como objetivo desarrollar aplicaciones de audio que necesiten gran cantidad de recursos computacionales, demostrando con ello que las GPUs son herramientas totalmente válidas para llevarlas a cabo. Para ello, se han implementado y evaluado sobre el entorno de programación CUDA diferentes aplicaciones del campo de procesado de señales de audio. También se han analizado y resuelto las posibles limitaciones surgidas durante el proceso de implementación, tanto desde el punto de vista acústico como desde el punto de vista computacional.

En la tesis se han abordado los siguientes problemas:

La primera operación a implementar en GPU era la operación fundamental en el procesado de audio: la convolución, ya que la mayoría de aplicaciones de audio multicanal están basadas en el filtrado masivo. En

principio, la convolución se ha desarrollado como un núcleo computacional, que posteriormente se ha usado para desarrollar una aplicación que combina múltiples convoluciones concurrentemente: cancelación *crosstalk* generalizada y ecualización. La implementación propuesta es capaz de gestionar dos situaciones comunes en el filtrado multicanal: buffers para muestras de audio de tamaños mayores que los tamaños de los filtros; y buffers para muestras de audio de tamaños menores que los tamaños de los filtros.

Se han desarrollado dos aplicaciones de audio espacial a partir del filtrado masivo multicanal que usan las GPUs como co-procesadores. La primera aplicación gira en torno al sonido binaural. Esta aplicación presenta dos características importantes: 1) es capaz de sintetizar fuentes sonoras en posiciones espaciales que no estén incluidas en las bases de datos de los filtros HRTFs, y 2) genera movimientos continuos entre diferentes posiciones. Estas características se han obtenido en la implementación después de diversas pruebas tanto objetivas como subjetivas. Posteriormente, se ha estudiado el máximo número de fuentes sonoras que pueden ser gestionadas por diferentes arquitectura GPU. El mismo estudio se ha llevado a cabo en un sistema de síntesis de onda *Wave Field Synthesis* (segunda aplicación de sonido espacial) compuesto por 96 altavoces. La implementación de este sistema en GPU es capaz de reducir los efectos de sala durante la reproducción.

Otro problema que se ha abordado en esta tesis es la localización de fuentes sonoras en entornos ruidosos y con mucha reverberación. Para este problema se ha propuesto una implementación basada en el algoritmo de localización *Steered Response Power with Phase Transform* (SRP-PHAT) en un sistema multi-GPU. La exactitud en la localización de las fuentes sonoras está íntimamente ligada a una malla espacial de puntos donde se busca la fuente, y al número de micrófonos utilizados en el algoritmo. En esta tesis, se han evaluado las capacidades de las GPUs cuando éstas implementan el algoritmo SRP-PHAT bajo condiciones de tiempo real, atendiendo a diferentes parámetros: tamaño de malla, número de micrófonos, reverberación en la sala, y relación señal a ruido.

Finalmente, esta tesis trata el problema del filtrado masivo multicanal cuando los filtros presentan una respuesta al impulso infinita (*Infinite Impulse Response*, IIR). Se han analizado dos casos particulares: 1) Filtros IIR compuestos de múltiples secciones paralelas de orden dos, y 2) Filtros IIR que presentan una respuesta plana en frecuencia (allpass filters). Ambas estructuras se usan para desarrollar y acelerar dos aplicaciones de audio

diferentes: 1) implementar múltiples ecualizaciones en un sistema WFS, y 2) reducir el margen dinámico en señales de audio.

Palabras Clave: Filtrado Masivo Multicanal, Síntesis de Campo de Ondas, Localización de Fuentes Sonoras, Sonido Espacial, Unidades de Procesamiento Gráfico.

Resum

El processament d'àudio multicanal ha experimentat un gran desenvolupament en els darrers anys i com a conseqüència s'ha produït un augment notable de la complexitat computacional en les noves aplicacions. Actualment, es pretén que la telecomunicació ofereixi una sensació de proximitat, compartint fins i tot el mateix entorn entre usuaris distants. És el que s'anomena: Esquemes d'Àudio Immersiu. En aquest fenomen intervenen diversos efectes acústics: so espacial 3D, compensació de sales, cancel.lació crosstalk, localització de fonts sonores, entre altres. Però, per dur a terme qualsevol d'aquests efectes en un sistema real, es necessita una alta capacitat computacional, el que representa una severa limitació quan es tracta d'executar aquestes aplicacions en temps real.

L'augment de la capacitat computacional ha anat històricament unida al nombre de transistors en un xip. Actualment, les millores en la capacitat computacional estan íntimament lligades al nombre d'unitats de procés que té un ordinador, el que permet un alt grau de paral.lelisme en computació. Aquest és el cas de les Unitats de Processament Gràfic (GPUs, *Graphics Processing Units*), que posseeixen actualment milers de nuclis computacionals. Les GPUs s'han relacionat tradicionalment amb la computació gràfica o el tractament d'imatges, però amb l'aparició de nous entorns de programació per a GPUs (CUDA o OpenCL) moltes aplicacions d'altres camps científics han pogut ser accelerades mitjançant la seua implementació a les GPUs. Aquesta tesi té com a objectiu desenvolupar aplicacions d'àudio que necessiten gran quantitat de recursos computacionals, demostrant amb això que les GPUs són eines totalment vàlides per dur-les a terme. Amb aquest objectiu, s'han implementat i avaluat diferents aplicacions del camp de processament de senyals d'àudio sobre l'entorn de programació CUDA. També s'han analitzat i resolt les possibles limitacions sorgides durant el procés d'implementació, tant des del punt de vista acústic com des del punt de vista computacional.

En la tesi s'han abordat els següents problemes:

La primera operació a implementar a la GPU era l'operació fonamental en el processament d'àudio: la convolució, ja que la majoria d'aplicacions d'àudio multicanal estan basades el filtrat massiu. Primerament, s'ha desenvolupat la convolució com un nucli computacional que posteriorment

s'ha utilitzat per desenvolupar una aplicació que combina concurrentment múltiples convolucions: cancel.lació generalitzada i equalització. La implementació proposta pot gestionar dues situacions comuns en el filtrat multicanal: buffers de mostres d'àudio de major tamany que el dels filtres, o per contra buffers de mostres d'àudio de menor tamany que el dels filtres.

S'han desenvolupat dues aplicacions d'àudio espacial a partir del filtrat massiu multicanal que fan servir les GPUs com a co-processadors. La primera aplicació gira entorn al so binaural. Aquesta aplicació presenta dues característiques importants: 1) Pot sintetitzar fonts sonores en posicions espacials que no estan incloses a les bases de dades dels filtres HRTFs, i 2) genera moviments continus entre diferents posicions. Aquestes característiques s'han obtingut en la implementació després de diverses proves tant objectives com subjectives. Posteriorment, s'ha estudiat el màxim nombre de fonts sonores que poden ser gestionades per diferents arquitectures GPU. El mateix estudi s'ha dut a terme a un sistema de síntesi d'ona Wave Field Synthesis (segona aplicació de so espacial) compost per 96 altaveus. La implementació d'aquest sistema en GPU pot reduir els efectes de sala durant la reproducció.

Un altre problema que s'ha abordat en aquesta tesi és la localització de fonts sonores en entorns sorollosos i amb molta reverberació. Per a aquest problema s'ha proposat una implementació basada en l'algoritme de localització *Steered Response Power with Phase Transform* (SRP-PHAT) en un sistema multi-GPU. L'exactitud en la localització de les fonts sonores esta íntimament lligat a una malla espacial de punts on es busca la font i el nombre de micròfons utilitzats en l'algorisme. En aquesta tesi, s'han avaluat les capacitats de les GPUs quan aquestes implementen l'algoritme SRP-PHAT sota condicions de temps real, tenint en compte diferents paràmetres: tamany de la malla, nombre de micròfons, reverberaciò a la sala, i relaciò senyal a soroll.

Finalment, aquesta tesi tracta el problema del filtrat massiu multicanal quan els filtres presenten una resposta a l'impuls infinita (*Infinite Impulse Response*, IIR). S'han analitzat dos casos particulars: 1) Filtres IIR compostos de múltiples seccions paral.leles d'ordre dos, i 2) Filtres IIR que presenten una resposta plana en freqüència (allpass filters). Ambdues estructures es fan servir per a desenvolupar i accelerar dues aplicacions d'àudio diferents: 1) implementar múltiples equalitzacions a un sistema WFS, i 2) reduir el marge dinàmic a les senyals d'àudio.

Paraules Clau: Filtrat Massiu Multicanal, Síntesi d'Ona, Localització de

Fonts Sonores, So Espacial, Unitats de Processament Gràfic.

Acknowledgements

It is a pleasure for me to thank those who made this thesis possible. First and foremost, I offer my sincerest gratitude to my supervisors, Prof. Alberto González and Prof. Antonio M. Vidal Maciá, who supported me throughout this thesis with their knowledge and advice whilst allowing me the room to work in my own way.

I am very grateful to Prof. Rudolf Rabenstein from the University of Erlangen- Nüremberg, Dr. Pedro Vera from the University of Jaen and Dr. Leroy Anthony Drummond from Lawrence Berkeley National Laboratory for serving as reviewers of this thesis and for providing me with very useful comments that helped to improve the final manuscript. Special thanks also to Prof. José Ranilla Pastor from University of Oviedo and Prof. Pedro Juan López Rodríguez from Technical University of València for acting as members of the committee.

I would like to thank Prof. Vesa Välimäki, who hosted me at the Department of Signal Processing and Acoustics of the Aalto University, Espoo, Finland. I really appreciate the opportunity he gave me and his kind support during the months I spent working with his team. I want to thank also Prof. Lauri Savioja, Dr. Balázs Bank, and Dr. Julian Parker for their collaboration in the work that I conducted at Aalto University. Special mention goes to Pekka and Annelie Välimäki for making me spend a wonderful time in Helsinki. I will never forget the finnish meals and the good moments that I spent at your home.

I owe my deepest gratitude to Dr. Paco Martínez, Dr. Miguel Ferrer and Dr. Máximo Cobos for their continuous support and collaboration, which have contributed in a very important way to the development of this thesis.

I would like to show my gratitude to all the people at the Universitat Politècnica de València that shared my daily work at the Institute of Telecommunications and Multimedia Applications (iTEAM). In particular, I would like to thank Prof. Jose J. López, Dr. Gema Piñero, and Dr. María

de Diego. Thanks also to my current and former colleagues at the iTEAM: Emanuel Aguilera, Amparo Martí, Fernando Domene, Luis Maciá, Jorge Lorente, Carla Ramiro, Laura Fuster, Marian Simarro, Eliseo Villanueva, Pablo Gutierrez, Cristian Antoñanzas, and Csaba M. Józsa.

Special thanks go to my close friend Sandra Roger for all the advice, support, and help during the good and difficult moments we have been sharing since we met at the Faculty of Telecommunications Engineering of the UPV in 2001. You advised me to start a research career and this is the first result. Thank you.

Thanks to my dear friends Choni, Bea, Paco, Dani, Bego, Lucas, Elena Bernal, Elena Fdez, Andreu, María and specially to the dentists Alberto Albero and Vicente Ejarque for being always by my side and for making me smile even in my worse days. I would like to acknowledge the encouragement given by family friends Ángel, Rosa, and Marta.

I am deeply indebted to my parents, Toni and Pili, and to my sister Mapi, for their endless confidence and fondness.

Last, but not least, I would like to express my sincere gratitude to Lola, whose continuous support has made it possible for me to complete this thesis. Finding you was the best thing that could have ever happened to me during my research career. I love you.

José Antonio Belloch Rodríguez
July 2014

Contents

Abstract iii

Resumen v

Resum ix

Acknowledgements xiii

List of symbols xxv

Abbreviations and Acronyms xxxi

1 Introduction 1
- 1.1 Background . . . 3
- 1.2 Motivation . . . 5
- 1.3 Objectives . . . 7
- 1.4 Organization of the Thesis . . . 8

2 Preliminaries and Tools 11
- 2.1 Introduction . . . 13
- 2.2 Frequency domain . . . 15
  - 2.2.1 Discrete Fourier Transform . . . 18
  - 2.2.2 Fast Fourier Transform . . . 19
- 2.3 Convolution . . . 19
  - 2.3.1 Convolution Theorem . . . 20
  - 2.3.2 Convolution in Audio Signals . . . 21
  - 2.3.3 Convolution with long sequences . . . 22
  - 2.3.4 Overlap-save . . . 22
  - 2.3.5 Overlap-add . . . 23
  - 2.3.6 Other operations in Digital Signal Processing . . . 25
  - 2.3.7 Real-time processing . . . 27
- 2.4 Traditional Hardware for Digital Signal Processing . . . 28
  - 2.4.1 Digital Signal Processors . . . 28
  - 2.4.2 Field-Programmable Gate Arrays . . . 28

2.5 Multi-core Architectures and Graphic Processing Units (GPUs) 29
2.5.1 Multi-core and GPUs Origin 30
2.6 GPU and CUDA 31
2.6.1 Streams on GPU 35
2.6.2 Multi-GPU programming with multicore 36
2.7 Tools used for the development of the thesis 36
2.7.1 ASIO protocol 39

3 State-of-the-Art 41
3.1 Generalized crosstalk cancellation and equalization (GCCE) 43
3.2 Headphone-base spatial audio 45
3.3 Wave Field Synthesis 46
3.4 Sound source localization 47
3.5 GPU computing in other research inside audio field 49
3.6 Conclusion 49

4 Massive Multichannel Filtering 51
4.1 Convolution 53
4.1.1 Pipelined algorithm in a multichannel system 56
4.2 Crosstalk Cancellation using a stereo signal 60
4.2.1 Definition of the problem 60
4.2.2 GPU Implementation 63
4.2.3 Test system and Results 66
4.3 Multichannel massive audio processing for a GCCE application 67
4.3.1 Definition of the problem 68
4.3.2 GPU data structure for efficient convolution 70
4.3.3 GPU data structure for GCCE applications 72
4.3.4 Performance and Results 80
4.3.5 Conclusions 82

5 Headphone-based spatial sound system 87
5.1 Introduction 90
5.2 Processing Head-Related Transfer functions 91
5.3 Switching technique 93
5.3.1 Evaluation of the switching technique 95
5.4 Interpolation technique 98
5.4.1 Evaluation of the interpolation technique 101
5.5 GPU-based implementation of a head-phone audio application 105
5.5.1 Emulating a source movement 111

5.5.2 Interaction with the user 114
5.6 Results 115
5.7 Conclusions 119

6 Wave Field Synthesis system 121
6.1 Theory of a WFS system 123
6.1.1 Room Compensation in a WFS system 126
6.1.2 Practical Implementation of a WFS system 127
6.2 Test system 129
6.2.1 System Setup 130
6.2.2 Computational kernels implemented on GPU 133
6.3 Performance and results 140
6.4 Conclusion 141

7 Sound Source Localization 143
7.1 Introduction 146
7.2 Sound Source Localization using SRP-PHAT Algorithm . . 146
7.2.1 SRP-PHAT Implementation 149
7.2.2 Computational Cost 149
7.3 Algorithm Parallelization for real-time GPU implementation 151
7.3.1 Considerations in code of CUDA kernels 23 and 24 . 158
7.3.2 Multi-GPU Parallelization 159
7.3.3 Basic Implementation using two GPUs 160
7.4 Experiments and Performance 161
7.4.1 Localization Performance 163
7.4.2 Computational Performance 166
7.5 Conclusion 167

8 Multichannel IIR Filtering 169
8.1 Definition of the problem 171
8.1.1 Fixed-pole parallel filters 172
8.1.2 Filter design 173
8.2 Implementations on Many-core architectures (GPU and multi-cores) 174
8.2.1 GPU-based parallel implementation 175
8.2.2 Multicore-based parallel implementation 175
8.3 Results 179
8.4 Conclusion 182

9 Massive Multiple Allpass filtering **183**
9.1 Definition of the problem 185
9.2 Test Setup 187
9.3 GPU-based Implementation 188
9.4 Results 192
9.4.1 Computational Performance 192
9.5 Conclusion 195

10 Conclusion **197**
10.1 Main Contributions 199
10.2 Further Work 201
10.3 List of Publications 203
10.4 Institutional Acknowledgements 207

Bibliography **208**

A Appendix **225**
A.1 Alternative Multi-GPU Parallelization strategy 227
A.1.1 Basic Implementation using two GPUs 228
A.1.2 Comparison between strategies 228

List of Figures

1.1 Relation among objectives, computational kernels on GPU, and applications to develop through this dissertation. 9

2.1 Sound wave that is captured by a transducer and converted to an electrical signal. 14
2.2 Signal $x(t)$ and the samples signal $x(kT_s)$. 15
2.3 The square periodic signal $g(x)$ can be decomposed in multiple sum of sines and cosines. 17
2.4 Measure of the reflection coefficient using a probe that emits a step signal. 19
2.5 Overlap-save: Split the signal x in blocks of size l_o. 23
2.6 Overlap-save: Each block x^i together with h are Fourier transformed and element-wise multiplied. 23
2.7 Overlap-save: To configure output signal y, the first $l_h - 1$ of every block y^i are discarded. 24
2.8 Overlap-add: To configure output signal y, the last $l_h - 1$ samples of block y^i must be added to the first $l_h - 1$ samples of the block y^{i+1}. 25
2.9 An application composed of four inputs and two outputs. . 27
2.10 Block diagram of the FPGA Virtex IV. 29
2.11 Evolution of the different Nvidia architectures through the time line. 31
2.12 A GPU has multiple Stream Multiprocessor (SM) that are composed of multiple pipelined cores (SP). The number of SPs depends on the compute capability and the number of SMs depends on the kind of the device. A GPU device has off-chip device memories and on-chip memories *(in devices with compute capability 2.x and 3.x) **(only in devices with compute capability 3.x). 33
2.13 Distribution of the threads inside the cuda grid. 34
2.14 CUDA features that depend on the capability of the GPU device. 34

2.15 The UVA feature reduces data-transfer time among GPUs by using peer-to-peer communication (bottom). 37
2.16 http://www.steinberg.net/en/company/developer.html . . . 40

4.1 Matrix **S** is built from signal blocks. 55
4.2 P FFTs are applied to matrices **S** and **H**. Afterwards, both matrices are element-wise multiplied. 57
4.3 Matrices **S** is composed of samples of four different signals, and **H** is composed of coefficients of two different filters. . . 58
4.4 Four matrices are needed in order to carry out a pipelined algorithm. 59
4.5 Crosstalk canceller filters. 61
4.6 Measurement of the transmission path filters. 62
4.7 Acquired signals in the left ear: 1) only direct path (signal goes through filters f_{LL} and f_{RL}), and 2) only the cross path (signal goes through filters f_{LR} and f_{RR}). 63
4.8 Matrices **S**, **F**, $\mathbf{S}_{res}$ used in CUDA kernel 2 and CUDA kernel 3. 66
4.9 Required operations on a application that performs a crosstalk cancelation by using the CPU and the GPU. 67
4.10 Task Manager on Windows operating system in both cases: a) GPU-based implementation, and b) CPU-based implementation. 68
4.11 $2 \cdot Z$ desired signals are set to each ear of Z listeners in a room. Cross paths and room effects are canceled by means of the use of the *Crosstalk canceler and Equalizer* block. . . 69
4.12 The signal at loudspeaker y_n is composed of a combination of all the sources x_m filtered through their respective f_{mn}. . 70
4.13 (a) shows Scheme 1 where matrix **S** is located in *global-memory* and matrix **F** in *shared-memory*; (b) shows the opposite case, Scheme 2 where matrix **F** is located in *global-memory* and matrix **S** in *shared-memory*. 71
4.14 (a) shows matrices **S** and **F** in GPU. Then, frequency-domain transform and element-wise multiplication are applied; (b) shows that the resulting matrix is stored at the same memory position. 73
4.15 Addition of all the planes to obtain the different outputs (in this case, Y_0 and Y_1). 76

4.16 (a) shows matrices **S** and **F** in GPU. Then, frequency-domain transform and element-wise multiplication are applied; (b) shows that the resulting matrix $\mathbf{R}_{\mathrm{v}}$ is stored in a different memory position. 77
4.17 Element-wise sum between $\mathbf{R}_{\mathrm{v}}$ and $\mathbf{R}_{\mathrm{v}-1}$. Row 0 of $\mathbf{R}_{\mathrm{v}}$ is element-wise sum with the row indicated by *PointOut*; row 1 is element-wise sum with the row indicated by *PointOut+1*; and so on. 78
4.18 Copy of the row indicated by *PointOut* in $\mathbf{R}_{\mathrm{v}}$ to **OV**, which is later set to 0. *PointOut* increases incrementally and gets prepared for the next input-data buffer. 79
4.19 Important parameters in a real-time multichannel application, with M=4, N=2 and C_{tot}=8. 81
4.20 t_{proc} in a multichannel application fragmenting the input-buffer in different overlap-save blocks: (a) for 2 loudspeakers; (b) for 4 loudspeakers; (c) for 32 loudspeakers; and (d) for 64 loudspeakers. 84
4.21 t_{proc} used by GPU in a GCCE for different values of sources M and loudspeakers N, using a sampling frequency of f_s=44.1 kHz with: t_{buff}=2.9 ms in (a), t_{buff}=5.8 ms in (b), t_{buff}=11.6 ms in (c), and t_{buff}=23.2 ms in (d). 85

5.1 The HRIR filtering allows a person to perceive a piano sound as if it were located in a virtual position in the space given by the coordinates (θ, ϕ, r). 92
5.2 L Core samples that are used to obtain the total energy. . . 96
5.3 Percentage of the energy out of the band when the different fading vectors are applied. The right side of the figure corresponds to the percentage computed for the right channel, whereas the left side corresponds to the percentage for the left channel. 97
5.4 Percentage of preference obtained with the paired comparison test when RAMP, SQRT, TRI, FOURIER, and SIMPLE fading vectors were compared. 99
5.5 The star represents the position to be synthesized in the elevation plane ϕ_S. This position is synthesized by combining the two nearby azimuth positions using the weighted factors w_A and w_B. 100

5.6 The percentage of preference obtained when melodies that switched from virtual positions that were separated by 1, 7, 15, 30, and 45 degrees were compared. 104
5.7 GPU diagram of a head-phone audio application. 106
5.8 Operations carried out by CUDA kernel 8 and CUDA kernel 9. Each thread is responsible for the computation of a sample. 111
5.9 The number of filters is double in CUDA kernel 8. The processing is carried out for the old position and for the new position. 112
5.10 CUDA kernel 10 groups the buffers that belong to a position and a source for the particular case $M = 2$. 112
5.11 Flowchart from a spatial audio application whose audio processing is totally carried out on the GPU. Tridimensional application processes are executed on the GPU and symbolize the use of multiple threads that are launched in parallel. 116
5.12 Developed headphone-based spatial application running on a notebook with the GPU GTS360M. 117
5.13 Number of sound sources that can be managed by our proposed spatial sound application when all the sound sources stay static in real time. 118
5.14 Number of sound sources that can be managed by our proposed spatial sound application when all the sound sources are moving in real time. 119

6.1 Geometry of a WFS system where it is appreciated the sound source m, the N loudspeakers, and the different distances among sound source, loudspeakers and a listener. 125
6.2 Multichannel inverse filter bank, where every driving signal is convolved by N filters. The signal that is reproduced by a loudspeaker is a combination of all the filtered signals. . . 127
6.3 Configuration of the array at the laboratory of the GTAC at UPV 131
6.4 Data structures used for storing in the *global-memory* of the GPU: the delay factors and the amplitude factors in (a), and the output-data buffers in (b). 132
6.5 Flowchart of the processing executed on the GPU. 132

6.6 Processing time for different number of sound sources that are rendered in a spatial audio system (WFS + RC) without fading processing (a) and with fading processing (b) on different GPUs. 142

7.1 Intersecting half-hyperboloids for $M = 3$ microphones. Each half-hyperboloid corresponds to a TDOA peak in the GCC. 148
7.2 Accumulated SRP-PHAT values for a 2-D spatial grid (4×6 m and $M = 6$ microphones) with different spatial resolutions. (a) $r_{sp} = 0.01$ m. (b) $r_{sp} = 0.1$ m. 151
7.3 Computational cost when for different number of microphones M and spatial resolutions r_{sp}. 152
7.4 Operations that are carried out by CUDA kernel 21 in case M=4. 154
7.5 Operations that are carried out by CUDA Kernel 22. 156
7.6 Operations that are carried out by CUDA kernel 23. 156
7.7 Steps of the GPU-based SRP-PHAT implementation using two GPUs and *openMP*. 161
7.8 Microphone set-ups for $M = 6$, $M = 12$, $M = 24$ and $M = 48$. The black dots denote the actual active microphones in each configuration. 163
7.9 Localization accuracy for different wall reflection factors ($\rho \in \{0, 0.5, 0.9\}$) as a function of the SNR and the number of microphones M. Each row presents results for different spatial resolutions ($r_{sp} = 0.01$ and $r_{sp} = 0.1$ m). 165
7.10 Time t_{proc} for different resolutions and number of microphones. 167

8.1 Structure of the parallel second-order filter. 173
8.2 Structure of the second-order section used in Fig. 8.1 173
8.3 GPU-based Parallel Implementation of one IIR filter processing. 175
8.4 Performance comparison between multi-core CPU and GPU implementations for parallel filters composed of 1024 and 128 second-order sections with a buffer size of 32 samples. . . . 180
8.5 Maximum number of IIR filters that can be realized in real time for the multi-core and GPU implementation for filters composed of 1024 and 128 second-order sections. 181

9.1 Block diagram of the M parallel allpass filter chains. 187

9.2 Two-dimensional CUDA grid configuration. One thread performs an allpass filter using a delay line combination with a coefficient combination. **Col** defines the delay-line lengths and the **Row** determines the lookup in the coefficient table. 189
9.3 Maximum peak value obtained for the 28,966,400 combinations for all the signals. 194
9.4 Waveforms of the five isolated musical sound, before and after being processed (the work in [1] and third test: 28,966,400 combinations). The horizontal dashed lines show the positive and negative peaks of the original waveform whilst the solid horizontal lines show the positive and negative peaks after processing. 195

10.1 Developed and future applications that require massive multichannel signal processing. 202

A.1 Distribution of the audio buffers in order to compute the rows of the **GCC** matrix when N_{GPU} is 1,2,3 and 4. 230

List of symbols

$\mathbf{X}$	Matrix
$\mathbf{x}$	Vector
p	Scalar
$(\cdot)^{\mathrm{T}}$	Transpose
$\overline{p}$	Conjugation of complex value p
$(\cdot)^{\mathrm{H}}$	Conjugate transpose
$\lvert \cdot \rvert$	Absolute value
$\otimes$	Element-wise convolution
a	Coefficient of allpass filter 1 in the allpass filter chain. Chapter 9
a_{mn}	Amplitude factor that depends on the positions of the sound source m, and the loudspeaker n in a WFS system. Chapter 6
$a_{r,2}$	Denominator coefficient 2 of the r-th second-order section in IIR filtering. Chapter 8
b	Coefficient of allpass filter 2 in the allpass filter chain. Chapter 9
$b_m(t)$	Time domain signal extracted from microphone m in a localization system. Chapter 7
$b_{r,1}$	Numerator coefficient 1 of the r-th second-order section in IIR filtering. Chapter 8
c	Coefficient of allpass filter 3 in the allpass filter chain. Chapter 9
C	Geometry dependent constant in a WFS system. Chapter 6
C_{tot}	Number of filters implemented in a GCCE application
d_0	FIR path in IIR filtering. Chapter 8
d_1	Embedded delay length of allpass filter 1 in the allpass filter chain. Chapter 9
d_2	Embedded delay length of allpass filter 2 in the allpass filter chain. Chapter 9
d_3	Embedded delay length of allpass filter 3 in the allpass filter chain. Chapter 9

d_{max}	Maximum delay length of an allpass filter. Chapter 9
d_{zL}	Desired signal of the listener z-th (z is an integer value) at the Left ear. Chapter 4
d_{zR}	Desired signal of the listener z-th (z is an integer value) at the Right ear. Chapter 4
$\mathbf{f}$	Fading Vector. Chapter 5 and Chapter 6
f_s	Sampled Frequency
f_{ij}	Impulse Response of the filter implemented between the i-th source and the j-th loudspeaker. Chapter 4 and 6.
$\mathbf{f}_m$	Vector of frequency bins obtained from audio samples that are captured by microphone m in a localization system. Chapter 7
$\mathbf{g}$	Fading Vector. Chapter 5 and Chapter 6
$\mathcal{G}$	Spatial Grid in a localization system. Chapter 7
$h(t)$	Impulse Response in time domain
h	Impulse Response in discrete time domain
h_{RL}	Impulse Response of the crossed path from the Right Loudspeaker to the Left Ear. Chapter 4
h_{LR}	Impulse Response of the crossed path from the Left Loudspeaker to the Right Ear. Chapter 4
h_{LL}	Impulse Response of the direct path from the Left Loudspeaker to the Left Ear. Chapter 4
h_{RR}	Impulse Response of the direct path from the Right Loudspeaker to the Right Ear. Chapter 4
$\mathbf{h}_{\mathrm{r}}(\theta, \phi, r)$	Impulse responses HRIRs corresponding to position (θ, ϕ, r) for the Right Ear. Chapter 5
$\mathbf{h}_{\mathrm{l}}(\theta, \phi, r)$	Impulse responses HRIRs corresponding to position (θ, ϕ, r) for the Left Ear. Chapter 5
$\mathbf{h}$	Column vector in IIR filtering
$\mathbf{H}$	Matrix of filters coefficients in Section 4
k	Discrete time index
K	Number of second-order sections in IIR filtering. Chapter 8.
l	Index of the microphone used in the SRP-PHAT algorithm for the sound source localization. Chapter 7
l_f	Length/size of filter/impulse response f_{ij}

l_h	Length/size of filter/impulse response h
l_o	Length/size of the processing block. In chapter 4, its size is equal to an overlap-save block. In chapters 5,6 and 7, its size is $2L$
l_x	Length/size of a discrete finite signal x
l^x,l^y,l^z	Dimensions of the shoe-box-shaped room used in a localization system. Chapter 7
L	Size of buffers composed of audio samples and provided by audio cards
m	Sound source index in Chapters 4,5 and 6; and microphone index in Chapter 7
M	Number of Inputs in the Audio system. In Chapter 5 and Chapter 6, M means virtual sound sources to be rendered, but in Chapter 7, M means number of microphones in a Localization system.
M_A	Number of Allpass filters chains in parallel. Chapter 8
$\mathbf{M}$	Modeling matrix. It contains all-poles transfer functions. Chapter 8
n	Loudspeaker index
N	Number of Loudspeakers
N_{GPU}	Number of GPUs that are presented in the localization system. Chapter 7
N_f	Number of frames which are composed of speech audio samples that are used in the localization system. Chapter 7
N_p	Number of positions that are tested in the localization system. Chapter 7
$\mathbf{OV}$	Output vector in Chapter 4, scheme 2, FIR filtering
p_r	Pole of the r-th second-order section in IIR filtering. Chapter 8
P	Number of rows of matrices $\mathbf{S}$ and $\mathbf{H}$
ρ	Reflection factor in the walls of a room. Chapter 7
P^x	Number of points a given spatial grid $\mathcal{G}$ in the dimension that is labeled as x-axis. Chapter 7
P^y	Number of points a given spatial grid $\mathcal{G}$ in the dimension that is labeled as y-axis. Chapter 7
P^z	Number of points a given spatial grid $\mathcal{G}$

	in the dimension that is labeled as z-axis. Chapter 7
$\mathbf{p}$	Column vector in IIR filtering design. Chapter 8
Q	Number of microphone pairs in a localization system. Chapter 7
$Q_n(\mathbf{x}_m, \omega)$	Driving Signal of the loudspeaker n generated from virtual sound source m that is located in $\mathbf{x}_m$ in the frequency domain. Chapter 6
r	Index inside Sum
r_{sp}	Spatial resolution in a given spatial grid $\mathcal{G}$, Chapter 7
r_{sp}^x	Spatial resolution in a given spatial grid $\mathcal{G}$ in the dimension that is labeled as x-axis, Chapter 7
r_{sp}^y	Spatial resolution in a given spatial grid $\mathcal{G}$ in the dimension that is labeled as y-axis, Chapter 7
r_{sp}^z	Spatial resolution in a given spatial grid $\mathcal{G}$ in the dimension that is labeled as z-axis, Chapter 7
R	Specific listening point inside listening area in a WFS system. Chapter 6
$\mathbf{R}(r)$	Complex factor that aims to virtualize distance in the sound source. Chapter 5
S_c	Number of processors in a multi-core computer. Chapter 8
$\mathbf{S}$	Matrix of samples
t_{buff}	Elapsed time between audio sample buffers provided by the audio cards: $\frac{L}{f_s}$ s.
t_{proc}	Processing time since input-data buffers begin to be processed till output-data buffers are available in the application.
τ_{mn}	Delay in samples that is proportional to the distance between the virtual sound source m and the loudspeaker n in WFS system. Chapter 6
T_s	Sampling Interval
T_L	Duration in seconds of a block of the signal $b_m(t)$. Chapter 7
$\mathbf{T}_{GPU}$	A GPU buffer that is used to store audio samples. Chapter 7
U	Number of threads that configure a thread block in IIR filtering. Chapter 8
v	Reference to the v-th input-data buffer

ν	Total number of functional evaluations in a localization system based on SRP-PHAT algorithm. Chapter 7
ϑ_n	Angular frequencies in IIR filtering. Chapter 8
w	frequency (radians)
w_n	frequency bins (radians)
w_A	Weighting factor. Chapter 5
w_B	Weighting factor. Chapter 5
w_C	Weighting factor. Chapter 5
w_D	Weighting factor. Chapter 5
$x(t)$	Time-domain signal
x_a	Discrete-Time signal labeled with a letter, in this case a. Chapter 2
x_j	Discrete-Time signal that represents the signal in the j-th virtual sound source/microphone.
x^i	Block i-th composed of samples of signal x and i is an integer value.
x	Discrete-Time signal (Input in a system)
x_R	Signal that is planned to be rendered through the Right Loudspeaker. Chapter 4
x_L	Signal that is planned to be rendered through the Left Loudspeaker. Chapter 4
$\mathbf{x}_R$	Coordinates of the point R at a listening area in a WFS system. Chapter 6
$\mathbf{x}_m$	Coordinates of the virtual sound source m in a WFS system. Chapter 6
$\mathbf{x}_n$	Coordinates of the loudspeaker n in a WFS system. Chapter 6
$\mathbf{x}_{\text{buff}_i}$	Input Audio buffer composed of $2L$ samples of the sound source i. Chapter 5 and Chapter 6
$\mathbf{X}_{\text{buff}_i}$	Input Audio buffer in frequency domain composed of $2L$ frequency bins of the sound source i
y	Discrete-Time signal (Output in a system)
y^i	Discrete-Time signal that belongs to the i-th loudspeaker.
y_R	Signal that is reproduced by Right Loudspeaker in an stereo system. Chapter 4
y_L	Signal that is reproduced by Left Loudspeaker in an stereo system. Chapter 4

$\mathbf{y}_{\text{buff}_i}$	Output Audio buffer composed of $2L$ samples of the sound source i. Chapter 5 and Chapter 6
$\mathbf{y}(\theta_S, \phi_S, r_S)$	Output signal in time domain that represents virtual sound source that is rendered at position (θ_S, ϕ_S, r_S) in a binaural system. Chapter 5
$\mathbf{Y}(\theta_S, \phi_S, r_S)$	Output signal in frequency domain of $\mathbf{y}(\theta_S, \phi_S, r_S)$
Z	Number of listeners in a reproduction scenario. Chapter 4

Abbreviations and Acronyms

FT	Fourier Transform
iFT	inverse Fourier Transform
DFT	Discrete Fourier Transform
iDFT	inverse Discrete Fourier Transform
FFT	Fast Fourier Transform
iFFT	inverse Fast Fourier Transform
HRIR	Head Related Impulse Response
HRTF	Head Related Transfer Function
FIR	Finite Impulse Response
IIR	Infinite Impulse Response
MAE	Mean Absolute Error
SDR	Signal to Distortion Ratio
SRP	Steered Response Power
SSL	Sound Source Localization
TDOA	Time Delay Of Arrival
WFS	Wave Field Synthesis
RC	Room Compensation
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Arrays
GPU	Graphics Processing Units
CPU	Central Processing Unit
CPU	Central Processing Unit
ALU	Arithmetic and Logic unities
SIMD	Single Instruction Multiple Data machine
SM	Stream Multiprocessor
SMX	Modern Stream Multiprocessor
CUDA	Compute Unified Device Architecture
CUFFT	GPU library that performs multiple FFTs concurrently
SDK	Software development kit
UVA	Unified Virtual Adressing
P2P	peer-to-peer
PCI-E	Peripheral Component Interconnect Express
ASIO	Audio Stream Input/Output

1 Introduction

Introduction 1

1.1 Background

The field of multichannel audio signal processing has undergone major development in recent years due to the increased complexity of current audio processing applications. People want to collaborate through communication with the feeling of being together and sharing the same environment, what is considered as Immersive Audio Schemes [2] [3]. In this phenomenon, several acoustic effects are involved: 3D spatial sound [4], room compensation [5], crosstalk cancelation [6], sound source localization [7], among others. To achieve these amazing effects, the processing of multiple sources, channels, or filters are required.

Everyday life is full of three-dimensional sound experiences. Natural sounds are perceived in terms of their location, since they contain cues in all three dimensions (width, height, depth) that help our brain to identify the localization of the sounds [8]. We can define a spatial sound as a rendered sound that contains the localization cues.

Attending to the transducer used for spatial sound rendering, we can consider two kind of systems: headphone-based systems (binaural sound),

and loudspeaker array-based systems. In this last case, there are mainly two ways of achieving spatial sound: by using the Phantom effect [9], or by Sound Field Synthesis [10].

Binaural sound is based on headphones and allows a listener to perceive the virtual position of a sound source [11]. This kind of sound is obtained by filtering sound samples through a collection of special filters whose coefficients shape the sound with spatial information. In the frequency domain, these filters are known as Head-Related Transfer Functions (HRTFs). The response of HRTFs describes how a sound wave is affected by properties of the body shape of the individual (i.e., pinna, head, shoulders, neck, and torso) before the sound reaches the listeners eardrum [9] [12]. Each pair of HRTF filters is related to a specific virtual position. A set of HRTFs of different spatial fixed positions configure an HRTF database. When multiple sound sources in different spatial positions move around the scene, fantastic audio effects that provide more realism to the scene are achieved. These spatial sounds are usually added to video games, video conference systems, movies, music performances, etc. However, if a CPU processor were used to perform these tasks, the CPU processor would be overloaded and the whole application would slow down. When this happens, spatial sound information is usually avoided and, unfortunately, is not added to the applications.

Regarding Sound Field Synthesis, one of the most popular systems is the *Wave Field Synthesis* (WFS) [13], a spatial reproduction system capable of synthesizing an acoustic field in an extended area by means of loudspeaker arrays. This makes the reproduced sound scene independent from the listening position, and therefore the relative acoustic perspective perceived by a listener changes as he or she moves. WFS systems require high computational capacity since they involve multiple loudspeakers, such as the WFS system at the Universitat Politècnica de València (UPV) (shown in [14]) that has 96 loudspeakers, or the IOSONO WFS system (shown in [15]) that has 120 loudspeakers. Realistic scenes are achieved with high number of sound sources, which demands high computational needs. One of the problems to put WFS in practise is related to the interaction of the array with the listening room. The listening room introduces new echoes that are not included in the signal to be reproduced, thus altering the synthesized sound-field and reducing the spatial effect. One block that can be added to the WFS system is a Room Compensation (RC) block. The

purpose of this block is to minimize the undesirable interaction of the array with the listening room. A common RC block is based on a multichannel inverse filter bank that corrects the room effects at selected points within the listening area, such as those in [16] and [17]. This formula is validated by [18], where it is presented meaningful improvements in the acoustic field when a RC block is applied to a WFS system. However, the application of this spatial audio system (WFS + RC) in real environments (theaters, cinemas, etc.) requires a real-time solution which requires even more computational resources.

Sound source localization can be also considered as an important aspect in the inmersive audio schemes. Realistic teleconferencing systems are able to render accurate sound source localization. This indicates that sound source positions must be previously computed to be lately transmitted. The Steered Response Power with Phase Transform (SRP-PHAT) algorithm is a well-known approach for sound source localization due to its robust performance in noisy and reverberant environments [19]. This algorithm analyzes the sound power captured by an acoustic beamformer on a defined spatial grid, estimating the source location as the point that maximizes the output power. Since localization accuracy can be improved by using high-resolution spatial grids and a high number of microphones, accurate acoustic localization systems require also high computational power.

1.2 Motivation

Taking into account the above context, the performance of the described applications are totally dependent from the computational resources available.

In the last decade, 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 Moores 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.

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 [20], the development of GPU (Graphics Processing Units) 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 years, a large number of applications were addressed using GPU in a wide variety of fields [21]. Nowadays, GPUs have become highly parallel programmable co-processors that provide massive computation when the needed operations are properly parallelized.

The conception of Signal Processing is intimately linked with the type of computation required to perform the Processing. In 2009, José F.M. Moura, president of the Signal Processing Society, noted in [22]: "*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*". Furthermore, some personalities from the Signal Processing Society have recently highlighted in [23] the need for more application oriented works connecting to other scientific domains such as Computer Science. For instance, Professor Mos Kaveh (President, IEEE Signal Processing Society 2010-2011) stated: "*Signal processing is thriving. Submissions to the publications and conferences of the IEEE Signal Processing Society have been increasing, apparently with no end in sight. To more effectively brand what we do, we have no choice but to connect with applications that are meaningful to the public. And, to have real impact beyond our own circles, we must actively engage and collaborate with domain experts, for example, in biology, medicine, energy, and business.*"

In this context, Professor Li Deng (Editor-in-Chief of the IEEE/ACM TASL and ICASSP 2013 General Chair), says in [23]: *Within the next ten years, I expect a great deal of and growing interplay between the community of signal processing and those from artificial intelligence, machine learning, computer science, and applied mathematics (e.g., optimization).*

Analyzing, implementing, and performing multichannel audio signal processing-based applications by using GPUs is a challenge that can change the way in which processing is carried out in future audio systems.

1.3 Objectives

Considering all the motivation aspects, the main objective of this thesis is as follows:

To analyze different multichannel audio signal processing-based applications that require high computational resources. To study the GPU architecture in order to use it as co-processor that computes and accelerates the massive processing tasks that demand the cited applications. To solve possible application drawbacks both from the acoustic point of view as from the computational point of view. To perform the GPU-based applications focusing on the possibility of carrying out the application under real-time conditions. Finally, To develop real prototypes of the applications.

As most of the audio applications are based on multichannel massive filtering, some particular aims emerge from this main scope:

- To analyze and to implement on a GPU the operation that is massively carried out in an audio processing system: the convolution by using filters with a Finite Impulse Response (FIR).
- To implement on GPU a multichannel filter bank that allows not only to carry out multiple filtering concurrently, but also to be able to combine their convolution results. To apply this filter bank to develop the application: generalized crosstalk cancellation and equalization.
- To use the multichannel filter bank to develop a headphone-based spatial sound application. In this context, different problems regarding spatial sound rendering are studied and analyzed, in order to

consequently propose solutions. A real prototype is designed and proposed.

- To develop a Wave Field Synthesis system that aims to reduce the room effects by applying a multichannel filter bank that plays the role a Room Compensation block. A real prototype is designed and proposed.
- To study the computational requirements of a sound source localization system in order to implement it on a GPU and assess the influence of the computational capacity in the localization accuracy under real-time conditions. To this end, a computational kernel that carries out multiple correlations must be also developed.
- To implement on GPU multichannel filtering for the case that the filters present an Infinite Impulse Response (IIR) and are composed of multiple second-order sections. To apply this massive filtering to the application: Equalization of a WFS system.
- To implement on GPU multichannel filtering for the case that the filters present an allpass infinite impulse response, and to apply this massive filtering to the application: Dynamic Range reduction in the audio signals.

Figure 1.1 shows how the above objectives are related. It can be observed as most of the audio applications that demands high computational needs are related with the massive multichannel filtering.

1.4 Organization of the Thesis

The remainder of this thesis describes the research that has been undertaken to develop the aims stated above. It is important to remark that this thesis involves two different disciplines: Audio Signal Processing and Computational Science. Thus, we have dedicated chapter 2 of this dissertation to introduce some concepts of the mentioned disciplines, which could be considered fundamental concepts within a mono-disciplinary thesis. However, these concepts aims to help in the global understanding from a multi-disciplinary point of view. The chapters are organized and presented as follows:

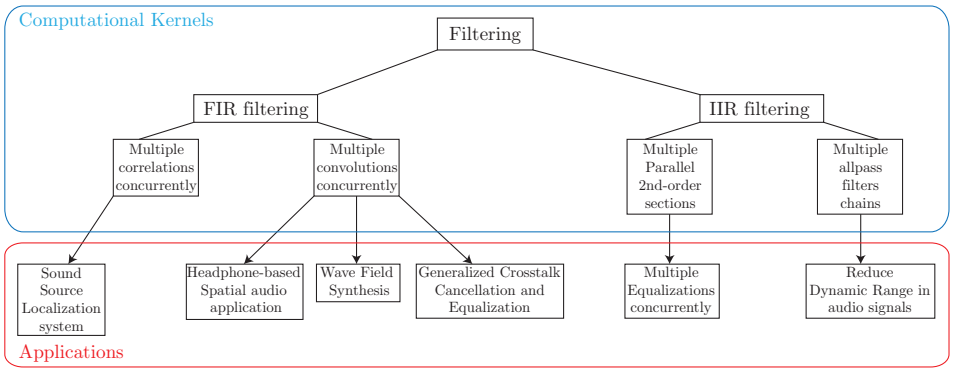


Figure 1.1. Relation among objectives, computational kernels on GPU, and applications to develop through this dissertation.

- Chapter 2: This chapter describes a large number of necessary concepts for the understanding of this dissertation. It contains an introduction to the topic of Multichannel Signal Processing and presents the hardware and software tools that have been used to perfom the audio applications.

- Chapter 3: This chapter deals with the state of the art of the developed applications through this dissertation. It overviews the advances in these applications both acoustical aspect as computational aspect.

- Chapter 4: This chapter describes GPU-based implementations of massive multichannel filtering. The chapter describes, firstly, the implementation of a single convolution on a GPU; then this implementation is extrapolated to carry out multiple convolutions. Finally, it is presented an application that requires a large number of concurrent filtering: generalized crosstalk cancellation and equalization. Two common situations are properly managed in this application: size of buffers that are much larger than the size of the filters and size of buffers that are much smaller than the size of the filters.

- Chapter 5: This chapter deals with a headphone-based spatial audio application that aims to recreate a multisource spatial sound scene. The first part of the chapter describes the problems that arise when a total rendering in the space is intended. Objective and subjective analysis are carried out to solve the problems. Computational per-

formance of this application in different GPU architectures is also presented. Furthermore, a real prototype of this application is developed in a notebook GPU.

- Chapter 6: In this chapter, an overview of the fundamentals of a wave field synthesis system is presented. Some concepts involving the wavefront generation are also discussed. Two aspects are considered when the Computational performance is assessed in different GPU architectures: a WFS system with a multichannel filter bank that plays the role of Room Compensation block, and without it.

- Chapter 7: This chapter develops a GPU-based sound source localization system. Firstly, a description of the SRP-PHAT algorithm (used for the sound source localization) is presented. Sound source localization accuracy is assessed when different levels of reverberation and noise are given in a room. A multi-GPU implementation is presented.

- Chapter 8: This chapter deals with multichannel filtering when the filters are composed of second-order IIR filters. Parallelization of this kind of filters is analyzed and studied. Implementations on GPU and on a multicore CPU are assessed.

- Chapter 9: This chapter deals with multichannel filtering when the filters are composed of allpass filter chains. The GPU is used in this case for launching multiple allpass filter chains concurrently in order to reduce dynamic range of the audio signals. A comparison with a multicore CPU is also assessed in this chapter.

- Chapter 10: Finally, the conclusions obtained throughout this thesis are presented, including some guidelines for future research lines. A list of published work related to this thesis is also given.

2 Preliminaries and Tools

2 Preliminaries and Tools

This chapter describes many necessary concepts for the understanding of this dissertation. It contains an introduction to the topic of Multichannel Signal Processing and presents the hardware and software tools that have been used to perform audio applications. Some of these concepts could be considered as fundamental concepts. However, they are included in this thesis because of its multi-disciplinary nature.

2.1 Introduction

What do we mean by signal?. Ages ago, signal referred to some physical manifestation of information that changed with time and/or space. By signal we may still be referring to a physical manifestation but we might also be dealing with other symbolic or abstract attributes of sequenced information: cold, hot, high, low [22]. Examples of signals include audio, video, speech, language, image, multimedia, sensor, communication, geophysical, sonar, radar, biological, among others. In this dissertation we focus on the audio signal.

An audio signal is obtained if we capture the vibrations of a mechanical wave of pressure and displacement. This capture can be easily carried out

by a microphone which acts as an acoustic-to-electric transducer or sensor that converts sound into an electrical signal. Figure 2.1 shows this process.

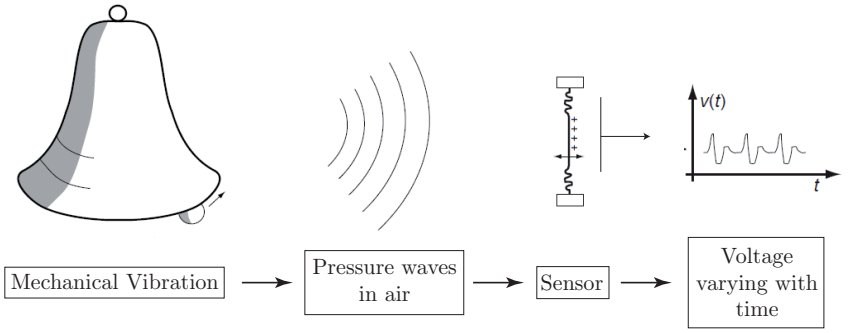


Figure 2.1. Sound wave that is captured by a transducer and converted to an electrical signal.

Multiple operations can be performed with this electrical signal, such as: storage, level compression, data compression, transmission, enhancement (e.g., equalization, filtering, noise cancelation, echo or reverb removal or addition, etc.). These operations are very common at the field Audio Signal Processing.

Historically, this processing was carried out analogically, i.e. altering the continuous signal by changing the voltage or current or charge via various electrical means. However, as computers and software became more advanced, audio signal processing tasks started to be carried out by computers and dedicated digital devices such as microprocessors. These devices use digital circuits, what implies that the analogical audio signal has to be previously converted to a digital signal. This is achieved by sampling the continuous signal.

Let $x(t)$ be an audio signal to be sampled, and let sampling be performed by measuring the value of the continuous function every T_s s, which is called the sampling interval. Thus, the sampled function is given by the sequence $x(kT_s)$, where k is an integer value. The sampling frequency or sampling rate f_s is defined as the number of samples obtained in one second (samples per second), thus $f_s = 1/T_s$. Figure 2.2 illustrates the continuous and the sampled signal.

Along with discretization of the audio signal, a quantization must be carried out. This means to approximate the discrete sample values by

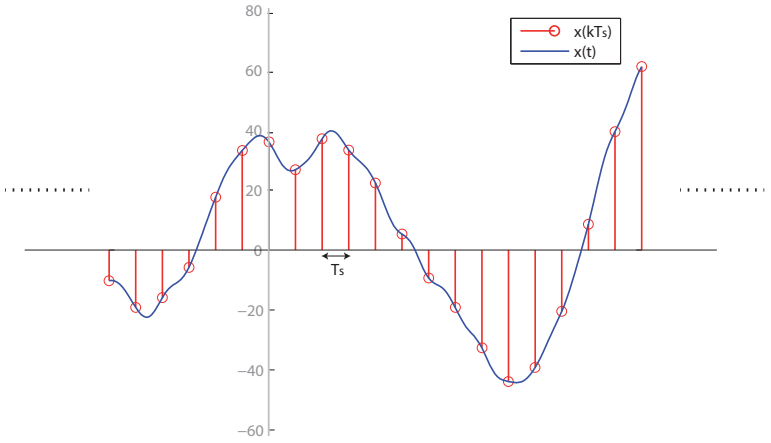


Figure 2.2. Signal $x(t)$ and the samples signal $x(kT_s)$.

values from a finite set that use usually any computational device. In this dissertation, we assume that the quantification is ideally performed and there is not any error in this process.

Mathematically, we obtained a vector of samples that ideally has an infinite length. Thus, the values $x(kT_s)$ are Fig. 2.2:

$$x = [\cdots -10.253 \quad -10.162 \quad -10,345 \ -10,742 \quad -11,413 \cdots].$$

2.2 Frequency domain

There are different domains in which a digital signal can be studied. The above presented domain is called the time domain, but there are other domains where the signal can offer more interesting information such as the frequency domain.

To connect the time domain and the frequency domain, we use the Fourier transform [24]. This transform comes from the study of Fourier series. In this study, periodic functions $f(x)$ with period 2π ($f(x) = f(2\pi + x)$) are written as an infinite sum of sines and cosines.

$$f(x) = \frac{a_0}{2} + \sum_{m=1}^{\infty} (a_m \cos(mx) + b_m \sin(mx)). \tag{2.1}$$

To find the coefficients a_m, b_m, and a_0, it is necessary to multiply the above equation by $cos(mx)$ or $sin(mx)$ and integrate it over interval $-\pi < x < +\pi$. By the orthogonality relations of sin and cos functions [25], we can get

$$a_m = \frac{1}{\pi}\int_{-\pi}^{+\pi} f(x)\cos(mx)dx, \tag{2.2}$$

$$b_m = \frac{1}{\pi}\int_{-\pi}^{+\pi} f(x)\sin(mx)dx, \tag{2.3}$$

$$a_0 = \frac{1}{\pi}\int_{-\pi}^{+\pi} f(x)dx.$$

Figure 2.3 shows how the square periodic signal $g(x)$

$$g(x) = \begin{cases} 1 & 0 < x < \pi \\ -1 & -\pi < x < 0 \end{cases}$$

can be decomposed in multiple sum of sines and cosines. Thus, signal $g(x)$ can be also written with its Fourier coefficients such as

$$g(x) = \frac{4}{\pi}(\frac{\sin x}{1} + \frac{\sin 3x}{3} + \frac{\sin 5x}{5} + \cdots). \tag{2.4}$$

In case of using any function $f(t)$ with arbitrary period T, Fourier series decomposition can be also carried out. To this end, a simple change of variables can be used to transform the interval of integration from $[-\pi, +\pi]$ to $[-T/2, T/2]$ as

$$x = \frac{2\pi}{T}t \qquad dx = \frac{2\pi}{T}dt \tag{2.5}$$

The $f(t)$ can be described by Fourier series as

$$f(t) = \frac{a_0}{2} + \sum_{m=1}^{\infty}(a_m\cos(\frac{2\pi mt}{T}) + b_m\sin(\frac{2\pi mt}{T})). \tag{2.6}$$

Taking into account the Euler formulae

$$\left.\begin{aligned} e^{ix} &= \cos x + i\sin x, \\ e^{-ix} &= \cos x - i\sin x, \end{aligned}\right\} \Rightarrow \begin{aligned} \cos x &= \frac{(e^{ix} + e^{-ix})}{2}, \\ \sin x &= \frac{(e^{ix} - e^{-ix})}{2}, \end{aligned} \tag{2.7}$$

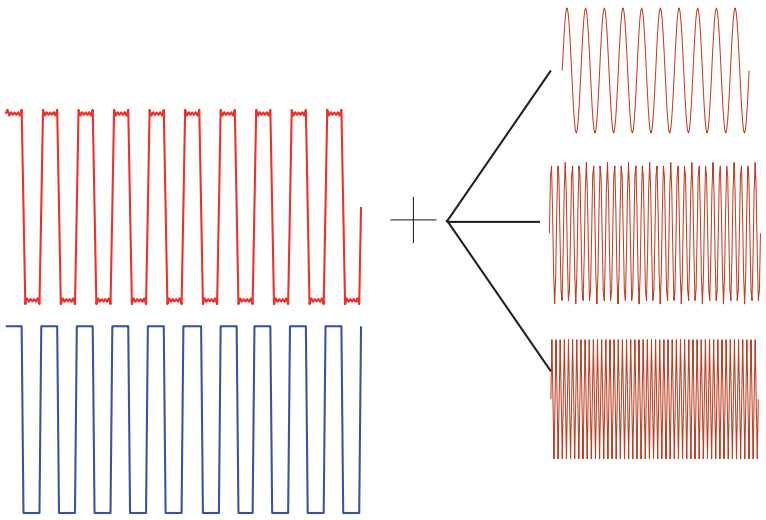


Figure 2.3. The square periodic signal $g(x)$ can be decomposed in multiple sum of sines and cosines.

and denoting $w{=}\frac{2\pi}{T}$, equation (2.1) can be rewritten as

$$f(t) = \sum_{k=-\infty}^{\infty} c_k e^{ikwt}. \tag{2.8}$$

where

$$c_k = \frac{2}{T}\int_{-T/2}^{+T/2} f(t)e^{-ikwt}dt. \tag{2.9}$$

In case of a non-periodic function, we can rewrite (2.8) assuming that $T \to \infty$

$$f(t) = \sum_{k=-\infty}^{\infty} (\frac{2}{T}\int_{-T/2}^{+T/2} f(\xi)e^{-ikw\xi}d\xi)e^{ikwt} \tag{2.10}$$

Considering $T{=}\frac{2\pi}{w}$, (2.10) can be written as

$$f(t) = \frac{1}{\pi}\sum_{k=-\infty}^{+\infty} w\int_{-T/2}^{+T/2} f(\xi)e^{-ikw(t-\xi)}d\xi = \frac{1}{2\pi}\int_{-\infty}^{+\infty} e^{iwt}dw\int_{-\infty}^{+\infty} f(\xi)e^{-iw\xi}d\xi. \tag{2.11}$$

Analyzing (2.11), we can define the *Fourier Transform* (FT) $F(w)$ of $f(t)$ as

$$\boxed{F(\omega) = \int_{-\infty}^{+\infty} f(t)e^{-i\omega t}dt.} \tag{2.12}$$

whilst $f(t)$ can be also obtained as the *inverse Fourier Transform* (iFT) of $F(w)$ by the following equation

$$\boxed{f(t) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} F(w)e^{-i\omega t}dw.} \tag{2.13}$$

2.2.1 Discrete Fourier Transform

When digital devices are being used to perform the Fourier analysis, we need to use the Discrete Fourier Transform (DFT). In this case, both the time and the frequency variables are discrete. Discrete Fourier Transform can be defined as a numerical approximation to the Fourier transform.

To convert the integral Fourier Transform (FT) into the Discrete Fourier Transform (DFT), we can do following steps:

1. Taking a T s. from the signal $x(t)$, and the number of sampling points l_x, the sample interval $T_s{=}\frac{T}{l_x}$.
2. This T_s allows to define $t_k{=}k \cdot T_s$ for $k = 0, 1, \ldots, N-1$, and the signal values at these points $s_k{=}s(t_k)$.
3. Define the frequency sampling points $\omega_n{=}\frac{2\pi n}{T}$, where $\frac{2\pi n}{T}$ is termed as the fundamental frequency and n is an integer value.
4. Considering the problem of approximating the FT of f at the points $\omega_n{=}\frac{2\pi n}{T}$, we obtain
$$F(\omega_n) = \int_{-\infty}^{+\infty} f(t)e^{-i\omega_n t}dt, \qquad n = 0, 1, \ldots, N-1, \tag{2.14}$$
5. Finally, Discrete Fourier Transform (DFT) is obtained by approximating the previous integral by Riemann sum approximation using the points t_k since $f \neq 0$ for $t > T$
$$\boxed{F(\omega_n) = \sum_{k=0}^{N-1} f(t_k)e^{-i\omega_n t_k}.} \tag{2.15}$$

The inverse Discrete Fourier Transform (iDFT) is defined as

$$\boxed{f(t_k) = \frac{1}{l_x} \sum_{n=0}^{N-1} F(w_n) e^{-i\omega_n t_k}.} \tag{2.16}$$

2.2.2 Fast Fourier Transform

Fast Fourier Transform (FFT) is an effective algorithm for computing the Discrete Fourier Transform. It was developed by Cooley and Tukey in 1965 [26]. This algorithm reduces the computation time of DFT for l_x points from l_x^2 to $l_x \log_2(l_x)$. It is also called the Butterfly algorithm and is based on divide-and-conquer algorithms. It consists of dividing the transform into two pieces of size $\frac{l_x}{2}$ recursively, and is therefore limited to power-of-two sizes. In case l_x is not a power of two, a zero padding at the end of the data can be carried out in order to employ more efficiently the algorithm. The inverse Fast Fourier Transform (iFFT) corresponds to the effective algorithm of the inverse Discrete Fourier Transform.

2.3 Convolution

One of the most important concepts in Signal Processing is the convolution concept. For example, if we want to measure the reflection coefficient of a porous medium, we can use an oscilloscope that is connected to a probe that emits a step signal. The own probe collects an output signal whose waveform is determined by the reflection coefficient of this medium. Fig. 2.4 shows this example.

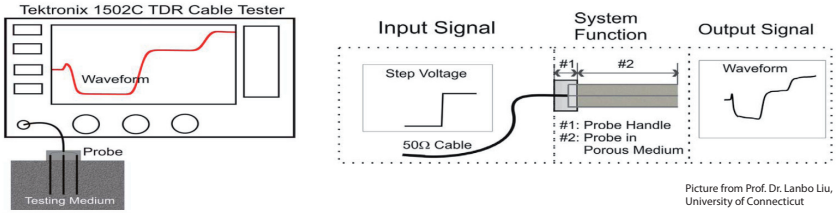


Figure 2.4. Measure of the reflection coefficient using a probe that emits a step signal.

An input signal, $x(t)$ (step signal), is passed through a system char-

acterized by an impulse response, h(t) (probe in porous medium + probe in handle), to produce an output signal, y(t) (reflection coefficient). This operation is called convolution and can be written in the familiar mathematical equation

$$y(t) = x(t) * h(t) \tag{2.17}$$

Mathematically, the convolution is defined as the integral over the time of one function at τ times another function at $t - \tau$ [27]. The integration is taken over the variable τ, typically from minus infinity to infinity. The following equation shows the convolution operation.

$$y(t) = \int_{-\infty}^{+\infty} x(\tau)h(t-\tau)d\tau \tag{2.18}$$

Using discrete signals, the convolution is defined as

$$y[k] = \sum_{r=-\infty}^{+\infty} x[r]h[k-r] \tag{2.19}$$

where $y[k]$ is the k-th sample of the output signal y.

As can be appreciated, discrete convolution consists of multiple samples multiplications and sums. In case of signals with limited lengths, the output signal y has a length of $l_x + l_h - 1$, where l_x and l_h are the lengths of x and h, respectively.

Related to the convolution concept is the circular convolution. This one is obtained when two periodic sequences x_a and x_b, with period l_x, are convolved using the following expression

$$y[k] = \sum_{k=0}^{N-1} x_a[r]x_b[k-r] \tag{2.20}$$

It is important to note that the length of the signal y is also l_x. To difference both kind of convolution, (2.19) is also called linear convolution, since it represents a linear system response.

2.3.1 Convolution Theorem

The convolution theorem states that the circular convolution of two periodic sequence can be computed by using the Discrete Fourier Transform [27].

This theorem states that sequence y is obtained as the inverse discrete Fourier transform of Y, which is obtained as the element-wise multiplication of the DFTs of x_a and x_b, X_a and X_b.

$$\begin{aligned} y &= \text{iDFT}(Y) \\ Y &= X_a \otimes X_b, \\ X_a &= \text{DFT}(x_a), \\ X_b &= \text{DFT}(x_b), \end{aligned} \tag{2.21}$$

where $\otimes$ represents element-wise multiplication.

According to (2.19), this operation requires l_x arithmetic operations per output value and l_x^2 operations for N outputs, being its complexity $O(l_x^2)$. With the help of the convolution theorem and the Fourier Transform, the complexity of the circular convolution is reduced to $O(l_x \log(l_x))$.

Convolution theorem is also applied to linear convolution in (2.19). To this end, both sequences, x and h must be zero-padded up to a size of $l_x + l_h - 1$ as a minimum. In this case, the result of the linear convolution matches with the result of the circular convolution. If we want to apply FFT to the sequences, then the zero-padded must be increased from $l_x + l_h - 1$ to the following power of two [27].

2.3.2 Convolution in Audio Signals

Convolution is used in audio signals to add/delete acoustical effects in the signals. As was mentioned in Section 2.2, a signal can be expressed as a sum of sinusoids, which resulted in a function called the Discrete Fourier transform, whose values depending on ω_n . These ω_n variables are related with the signal frequencies. Specifically, $\omega_n = 2\pi f_n$ where f_n is the frequency of the sinus or cosinus and is measured in Hz.

The human hearing range is on average from 20 to 20,000 Hz, although there is considerable variation between individuals, especially at high frequencies, where a gradual decline with age is considered normal. Range of frequencies between 16 Hz and 256 Hz produce bass sounds, whilst treble sounds corresponds to a frequency higher than 2 kHz. Between 256 Hz and 2 kHz are the middle frequencies. A common use of the convolution operation in music consists of filtering an specific range of frequencies. In case we are interested in bass sound, we must remove frequencies higher than 256 Hz by using a low frequency filter. To this end, we must only

convolve the audio signal $x(t)$ with the low pass filter whose coefficients are given by an impulse response $h(t)$.

Convolution allows also to add different effects to audio signals. For example, we can add a room effect. To this end, it is measured the impulse response of a room [28], which takes the role of $h(t)$ in Eq.(2.17). Then, an audio signal $x(t)$, that has been previously recorded under anechoic conditions, is convolved through this impulse response $h(t)$. The resulting signal $y(t)$ is composed of audio signal $x(t)$ plus the room effects.

2.3.3 Convolution with long sequences

Up to now, we have been dealing with finite sequences of signals, but when these sequences are long, it is required a large-scale memory to store all the samples in order to do first the FFT, the element-wise multiplications and the iFFT. To deal with inefficiently long FFT sizes two methods exists for splitting up the signal into blocks and perform the convolution operations in blocks: *Overlap-save* and *Overlap-add*.

2.3.4 Overlap-save

Considering that h and x have a size of l_h and l_x samples, respectively with $l_x \gg l_h$, we can convolve both sequences by breaking the long signal into blocks of l_o samples with the peculiarity that each block has an overlap of $l_h - 1$ samples with the previous block. The size l_o is the first power-of-two integer that is $l_o \geq l_x + l_h - 1$, as mentioned in Section 2.3.1. First block of x will be zero-padded at the beginning with with $l_h - 1$ samples. Although we take $l_o = l_x + l_h - 1$ and an overlap of $l_h - 1$ samples, most of the authors in literature take a value of $l_o = 2l_h$ with an overlap of l_h samples, which gives a better efficiency [29]. Figure 2.5 illustrates this block division where x^i represents the block i-th of signal x.

Sequence h is also zero-padded up its size is l_o. Thus, each block x^i and h have the same size. The FFT is applied to each block x^i and the sequence h, then each block is element-wise multiplied by the H (Fourier transformed of h) and gives as a result another block Y^i composed of l_o samples. Figure 2.6 illustrates these operations where X^i and Y^i denote the Fourier transforms of the blocks x^i and y^i, respectively.

Finally, the first $l_h - 1$ samples of every block y^i are discarded. To

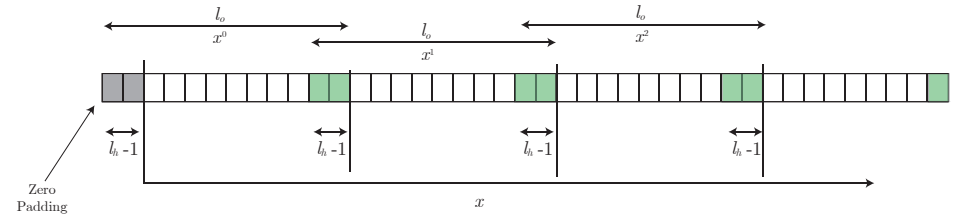


Figure 2.5. Overlap-save: Split the signal x in blocks of size l_o.

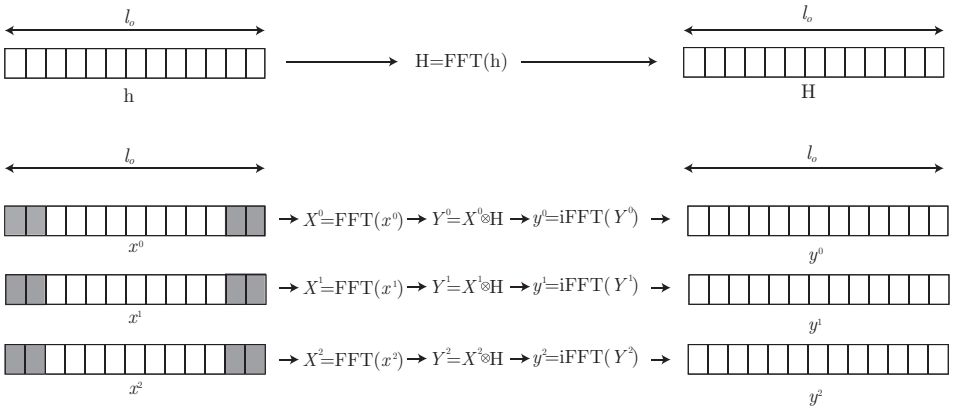


Figure 2.6. Overlap-save: Each block x^i together with h are Fourier transformed and element-wise multiplied.

configure the output signal y, all the blocks are afterwards concatenated, as is shown in Fig. 2.7.

2.3.5 Overlap-add

Overlap-add works also in blocks of size l_o, but with the peculiarity that the last $l_x - 1$ samples of the block are zero padded. Thus, each block x^i is configured by $l_o - l_h + 1$ samples of the signal x and $l_h - 1$ zeros. The sequence h is again zero-padded up its size is l_o. Fourier transforms and element-wise multiplications between the blocks x^i and the sequence h are carried out in the same way as in overlap-save, as shown in Fig. 2.6.

However, to concatenate all the resulting y^i blocks, the last $l_h - 1$ samples of block y^i must be added to the first $l_h - 1$ samples of the block

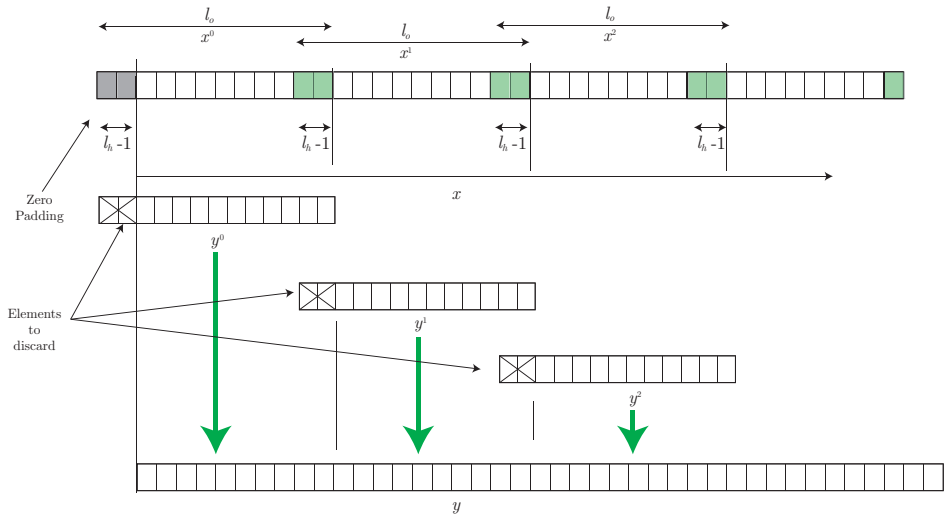


Figure 2.7. Overlap-save: To configure output signal y, the first $l_h - 1$ of every block y^i are discarded.

y^{i+1}. Figure 2.8 shows this processing.

Computational Cost

According to [30],the computational cost for computing the circular convolution of the described methods are:

- Direct Method in (2.20): $O(l_x^2)$.
- Using the convolutions theorem, the computational cost scale with (2.21): $O(l_x log(l_x))$.
- Using the overlap methods, the computational cost scale with: $O(l_x log(l_o))$

Overlap methods are faster for larger problems but not for small problems, since the computation overhead produced by splitting the signal can be more meaningful. They are mainly used when the length of the signal is not known, and can not be previously stored. Besides, these methods allow to reduce the latency of the system, i.e, it is not necessary to know all the signal to compute first samples of the output signal.

Overlap-save and overlap-add only compute $l_o - l_h + 1$ elements per processed block of length l_o, but overlap-add has two additional steps: One

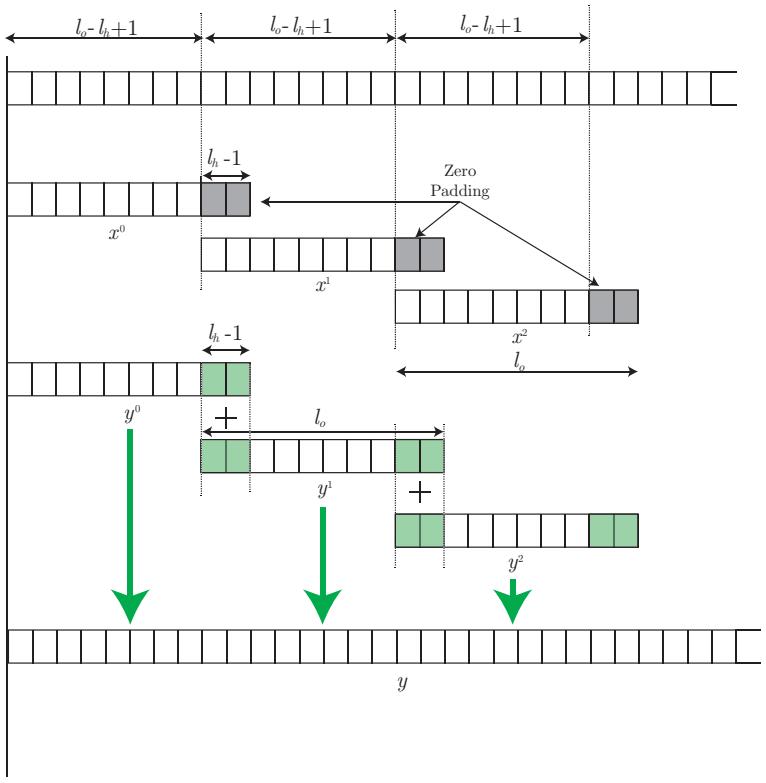


Figure 2.8. Overlap-add: To configure output signal y, the last $l_h - 1$ samples of block y^i must be added to the first $l_h - 1$ samples of the block y^{i+1}.

of zero-padding the input, and one of summing up with the overlap from the previous iteration. In practice, this will always result in worse performance for the overlap-add algorithm. Therefore, the overlap-save method should be preferred in real-time processing.

2.3.6 Other operations in Digital Signal Processing

There are other operations that can be carried out with signal samples.

Cross-Correlation

Cross-correlation is a measure of similarity of two signals. It is used for finding relevant characteristics of one signal inside the other signal. Math-

ematically, it is similar to the convolution but the inversion of one of the signals is not carried out. Equation (2.22) computes the cross-correlation between signals f and g, where $'$ denotes the complex conjugate of f. Note that an audio signal is composed of real samples. Thus, the complex conjugate has not effect in this case.

$$y = \sum_{r=-\infty}^{+\infty} f'[r]g[k+r] \tag{2.22}$$

As in the convolution case, cross-correlation can be also computed through the convolution theorem by using the DFTs: DFT(f') and DFT(g). In this case, the complex conjugate operator is important, since DFT has as a result a complex value.

Autocorrelation

Autocorrelation is the cross-correlation of a signal with itself. It is a mathematical tool for finding repeating patterns, such as the presence of a periodic signal obscured by noise. In an autocorrelation, there will always be a peak at a lag of zero unless the signal is a trivial zero signal.

Scaling and Delay

Two common operations in digital signal processing is scaling and delay. By scaling, we mean to weight the value of a samples or samples of signal by a factor α. To make a delay of γ samples in a signal implies that the value of signal in the sample index k=0 will be now in the sample index $k = \gamma$. In time domain, a delay of γ samples is expressed as the convolution of a discrete signal x with the delta function $\delta[k-\gamma]$ [27]. In the frequency domain, this convolution is converted to a complex exponential whose exponent is -$i\omega_n\gamma$. Equation (2.23) shows the equivalence of a signal when it is scaled and delayed in both domains. Note that index k in the time domain and the index ω_n in the frequency domain are not related.

$$\begin{aligned} y = \alpha \cdot X \quad &\Rightarrow \quad Y = \alpha \cdot X, \\ y = x * \delta(k-\gamma) \quad &\Rightarrow \quad Y = e^{-i\omega_n\gamma} \cdot X. \end{aligned} \tag{2.23}$$

Signals addition

In the audio signal processing field, the addition of the signals is very important, since the sound is additive. This means that, if we have two separated recorded sounds, we can sum up both signals and reproduce only the resulting signal for listening both sounds at the same time. This operation works in both time domain and frequency domain.

$$y = x_a + x_b \quad \Rightarrow \quad Y = X_a + X_b. \tag{2.24}$$

2.3.7 Real-time processing

In applications where multiple inputs and multiple outputs are involved, the time used for processing takes an important role.

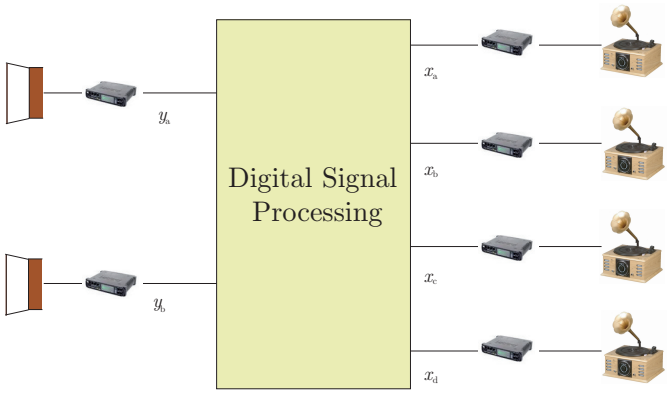


Figure 2.9. An application composed of four inputs and two outputs.

Figure 2.9 shows an application composed of four input signals and two outputs signals. Audio samples are provided by audio cards that digitize the signal from the gramophones. These ones symbolize the audio sound sources that could be obtained from an audio file, or directly from a microphone.

Most common audio cards work with "data buffers" instead of "sample by sample" acquisition. If each audio card provides L samples at sample frequency f_s, this means that each $\frac{L}{f_s}$s., a new buffer with L samples arrives. Focusing on Fig. 2.9, four input-data buffers arrive to be processed

every $t_{\text{buff}} = \frac{L}{f_s}$ s . In the block *Digital Signal Processing*, multiple operations are performed with the audio buffers such as, convolution, scaling, sum of samples, among others. Let t_{proc} denotes the processing time since the four input-data buffers are processed and resulted in two output-data buffers, which will be afterwards reproduced through the loudspeakers. As long as $t_{\text{proc}} < t_{\text{buff}}$, the application works in real time. If this does not happen, audio samples would be lost and processing would not be properly computed.

2.4 Traditional Hardware for Digital Signal Processing

In the block *Digital Signal Processing* of Fig. 2.9, different operations are performed with audio samples. These operations have been traditionally performed by DSPs (Digital Signal Processors) and, more recently, by FPGAs (Field-Programmable Gate Arrays). In the last decade, Programmable Graphics Processors (GPUs) have emerged as a low-cost, high-performance solution for general-purpose computations. In this section, we present an overview of the hardware that has been traditionally used for digital signal processing.

2.4.1 Digital Signal Processors

A Digital Signal Processor (DSP) is a specialized microprocessor that can process data in real time, making it ideal for applications that can not tolerate delays. A DSP has an architecture which is optimized for the fast operational needs of digital signal processing. A DSP is composed mainly of: a fast functional unity that allows to multiply and accumulate in the same clock cycle; various units that allow to carry out multiple operations in parallel including the address memory access computations and its corresponding access; Arithmetic and Logical Unities (ALU) that have their own registers; and various data buses.

2.4.2 Field-Programmable Gate Arrays

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing. A FPGA

contains programmable logic components called *logic blocks*, and a hierarchy of reconfigurable interconnections that allows the blocks to be wired according to the application to be developed/run. The number of interconnections depends on the different FPGA architectures. Figure 2.10 shows the block diagram of the FPGA Virtex IV of the company xilinx [31].

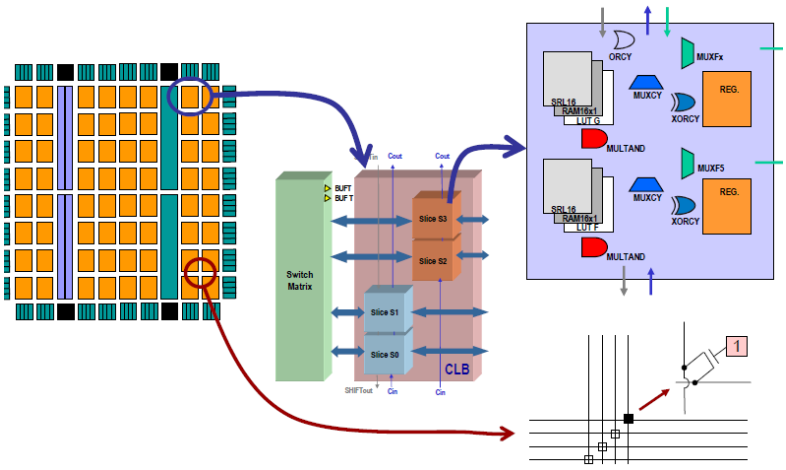


Figure 2.10. Block diagram of the FPGA Virtex IV.

2.5 Multi-core Architectures and Graphic Processing Units (GPUs)

The development of multichannel audio application is demanding high computational needs, even more when real-time conditions are required. Last section described the traditional hardware used for audio digital signal processing. However, this kind of hardware is mainly included in consumer audio hardware that is composed generally of fixed-function pipelines, that often evolved at a slower pace [32]. Nowadays, custom audio hardware is disappearing from console or PC architectures, and audio processing tasks are left to the multiple CPU cores and the powerful GPUs that have the

new workstations.

2.5.1 Multi-core and GPUs Origin

One of the historical laws about the increase in central processing unit (CPU) performance is the Moore's law [33] that states that the number of transistors in a chip doubles every one to two years. This law continues being valid if we refer to it as a computational capacity measure: not only taking into account the capacity of a single processor, but also the speed gain that is obtained by increasing the number of processing units, i.e expanding parallelism in computing [34]. In fact, actual CPUs are dual-core, quad-core, hexa-core, among others. The same is happening to the graphics processing units (GPUs) that owns now thousands of computing cores.

The massive development of the computational capacities of the GPUs was produced inside the computer game industry, in which there is a consistent demand to improve the visual appearance of games. In 1995, the games moved to three dimensions and the computing power increased greatly requiring specific-purpose hardware accelerator. The first GPUs had a fixed-function pipeline optimized for graphics that was difficult to use for other purposes. In 2001, programmable shaders were introduced in [35] being this a meaningful improvement, since it allowed to use the GPU computational capacities for nongraphics applications. Around 2003-2004 the GPUs enabled to process floating-point texture data. The final step of the evolution of GPUs into general-purpose programmable processors was produced when the programmable vertex and other processors were unified into a large collection of general-purpose stream processor units that allowed general-purpose C code to be compiled into GPU executable code.

But the breakthrough in the development of GPUs occurred in 2006, when Nvidia, one of the main manufacturers of GPUs, launches to the market a GPU with the G80 architecture that featured a total of 128 cores. Two years later, the GT200 architecture (also known as Tesla architecture) appeared with 240 cores and additional double precision units. In 2010, Nvidia launched the Fermi architecture that exhibited a total of up to 480 cores per chip. Kepler architecture appeared in 2012 and grouped more than 2000 cores. In March 2013, Nvidia announced that the successor to Kepler would be the Maxwell architecture. It is planned for release in 2014. Figure 2.11 shows the evolution of the different Nvidia architectures

through the time line and with large-scale computational capacity.

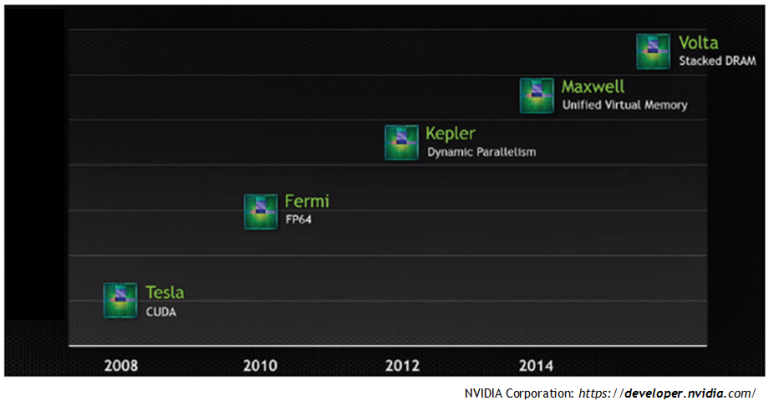


Figure 2.11. Evolution of the different Nvidia architectures through the time line.

All Nvidia architectures have a hardware common structure and works with the same programming paradigm. Thus, the same source code could be compiled for different architectures and be run in any of them. However, as every architecture has special features, it is not assured that using an architecture with more cores gives a better performance than other that has a less number of cores.

2.6 GPU and CUDA

Following Flynn's taxonomy [36], from a conceptual point of view, a GPU can be considered to be a Single Instruction Multiple Data machine (SIMD), i.e, 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 of the processing elements, which execute this instruction on their

own data simultaneously. The last generations of GPUs are composed by multiple Stream Multiprocessor (SM), where each SM consists of eight pipelined cores (SP) if compute capability is 1.2 or 1.3 (Tesla architecture), or 32 pipelined cores if it is 2.0 (Fermi architecture), or even 192 pipelined cores if it is 3.0 or 3.5 (Kepler architecture [37]).

A GPU device has a large amount of off-chip device memory (*global-memory*) and a fast on-chip memory (*shared-memory*, *registers*). The *shared-memory* is normally used when multiple threads must share data. There are also read-only cached memories called *constant-memory* and *texture-memory*. The first memory is optimized for broadcast i.e. when all the threads read the same memory location while the second one is more oriented to graphics.

GPU devices of compute capability 2.x and greater come with an L1/L2 cache hierarchy that is used to cache *global-memory*. Cache of level L1 is located on-chip memory. The same occurs to the *read-only cache* that is only present in GPU devices of compute capability 3.x. Fig. 2.12 shows how the GPU architecture is organized.

CUDA (Compute Unified Device Architecture) technology is an environment based on C language which allows the development of software intended to solve high complexity computational problems efficiently [20]. This software takes profit from the high amount of execution threads which are available on GPU. In CUDA, the programmer defines the kernel function where the code to be executed on the GPU is written. A grid configuration, which defines the number of threads and how they are distributed and grouped, must be built into the main code. Threads are grouped into thread blocks, and thread blocks configure a grid. Both thread blocks and grid are organized in three dimensions. Thus, a thread identification will be defined by a position within a block (`ThreadIdx.x`, `ThreadIdx.y`, and `ThreadIdx.z`), and this block will be defined within a grid (`BlockIdx.x`, `BlockIdx.y`, and `BlockIdx.z`). Parameters `BlockDim.x`, `BlockDim.y`, and `BlockDim.z` indicate the dimensions of blocks in the same way as `gridDim.x`, `gridDim.y`, and `gridDim.z` indicate the dimensions of grid. Fig. 2.13 shows a cuda kernel configured by a unidimensional grid composed of 3 blocks, and each block composed of 5 threads unidimensional. The compute capability of the GPU limits the maximum size in each dimension of the grid and the blocks in each cuda kernel. Figure 2.14 shows a table that is taken from the Nvidia CUDA programming guide [20] that outlines some CUDA

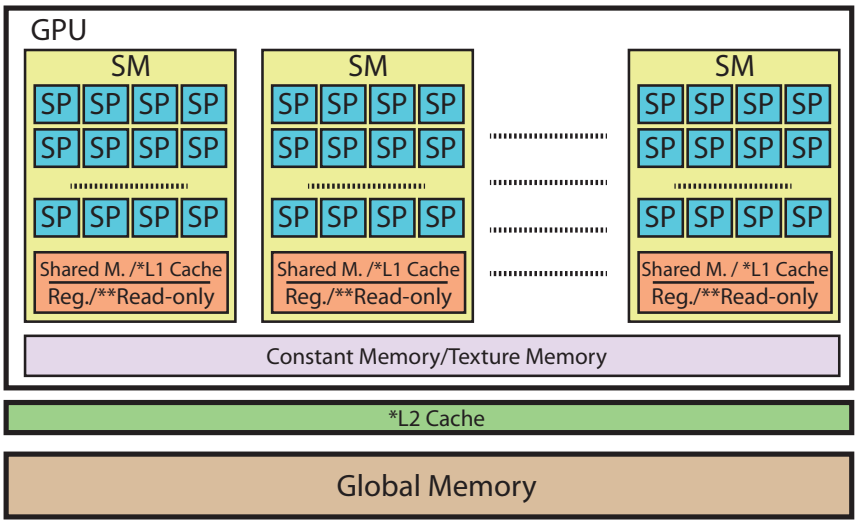


Figure 2.12. A GPU has multiple Stream Multiprocessor (SM) that are composed of multiple pipelined cores (SP). The number of SPs depends on the compute capability and the number of SMs depends on the kind of the device. A GPU device has off-chip device memories and on-chip memories *(in devices with compute capability 2.x and 3.x) **(only in devices with compute capability 3.x).

features that depend on the capability of the GPU device. In [38], the reader can find all the GPU devices that allow CUDA technology with its corresponding capability. Together with CUDA technology, Nvidia made available to the GPU developers a Nvidia GPU Computing SDK (Software development kit) [20]. A new SDK appeared with a new CUDA version release. The SDK collected different CUDA projects that could be taken as examples for the GPU developers.

An important aspect when accessing *global-memory* is to do it in a *coalesced* way. Coalescing means that the threads are writing into a small range of memory addresses having a certain pattern. For example, considering an array pointer to *global-memory* (`array`) and `idx` as the identification of a thread, if thread `idx` writes to address `array[idx]` and thread `idx+1` to address `array[idx+1]`, we achieve good coalescing.

In the CUDA model, the programmer defines the kernel function where the code to be executed on the GPU is written. A grid configuration,

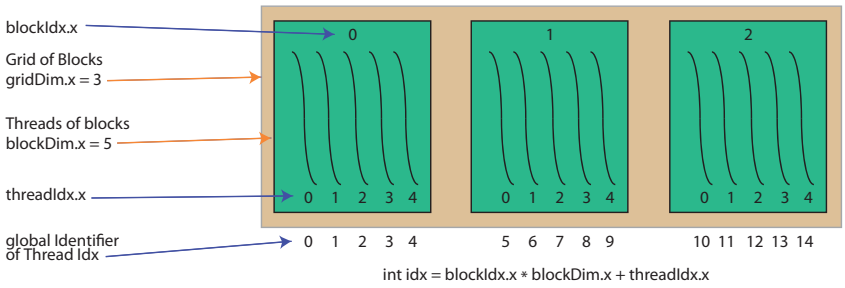


Figure 2.13. Distribution of the threads inside the cuda grid.

	Compute Capability						
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**	**3.5**
Maximum dimensionality of grid of thread blocks	2				3		
Maximum x-dimension of a grid of thread blocks	65535					2^{31}-1	
Maximum y- or z-dimension of a grid of thread blocks	65535						
Maximum dimensionality of thread block	3						
Maximum x- or y-dimension of a block	512				1024		
Maximum z-dimension of a block	64						
Maximum number of threads per block	512				1024		
Warp size	32						
Maximum number of resident blocks per multiprocessor	8					16	
Maximum number of resident warps per multiprocessor	24		32		48	64	
Maximum number of resident threads per multiprocessor	768		1024		1536	2048	
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K	

Figure 2.14. CUDA features that depend on the capability of the GPU device.

which defines the number of threads and how they are distributed and grouped, must be built into the main code (threads are grouped in thread blocks). The total number of threads launched in a kernel by means of thread blocks can exceed the number of physical cores. At runtime, the kernel distributes all the thread blocks among SMs. Each SM can host up

to 8 or 16 thread blocks depending on the CUDA capability. If there is more blocks than GPU resources can host, these blocks wait until other blocks finish in order to be hosted later. Grouping on thread blocks allows to distribute multiple threads among SMs, however each SM manages all the thread blocks jointly, and executes all the threads (regardless of the block they belong to) in groups of 32 parallel threads called *warps* that get scheduled by *warp schedulers* for execution.

It is important to have multiple *warps* in SMs, since it allows to hide latency. This means that, in case all threads of a *warp* must carry out a memory access whose access time lasts several clock cycles, the *warp schedulers* select other *warp* that is ready to execute in order to hide latency. The *warp schedulers* are responsible to switch among different *warps* in order to try full utilization of SMs. Depending on the CUDA capability, a GPU will have 1, 2 or 4 *warp schedulers* per SM.

2.6.1 Streams on GPU

Streams are virtual work queues on the GPU. They are used for asynchronous operation (i.e, the control of the program returns to the CPU immediately). Operations assigned to the same *stream* are executed in order and sequentially. Multiple *streams* can be defined on CUDA programming; however, up to 32 *streams* are available to be independently run on the GPU thanks to the *Hyper-Q* technology that is presented in hardware with 3.5 capability [39].

Different *streams* may execute their assigned operations out of order with respect to one another or concurrently. Thus, when a launched kernel does not require all the GPU resources, these could be used for another kernel that was launched from a different *stream*. Hence, *streams* allows multiple kernels to be launched concurrently. Following this idea, data transfer between CPU and GPU can also be overlapped with kernel computations and other transfers whenever they are carried out in different *streams*. If the data transfers are not assigned to any *stream* queue, they are executed synchronously and in an isolated way,(i.e. the CPU waits until all the previous operations have finished). GPU kernels are always launched asynchronously by the CPU (regardless of whether or not they are scheduled on a *stream* queue or not). Thus, data transfers are usually used as a synchronization barrier.

Although the *streams* appeared as a CUDA feature in devices with compute capability 1.3, their computational efficiency in the GPU implementations has been uncertain. In fact, most of the implementations did not use it, since there was an overhead time in configuring the *streams* that penalized the performance. It was not till the apparition of compute capability 3.5 when the *streams* began to play an important role in the CUDA implementations.

2.6.2 Multi-GPU programming with multicore

One of the standards that allows for multicore processing is openMP [40]. This standard works by using a *fork/join* pattern, that is, parallel regions are specified by the programmer. The CPU code runs sequentially and at some point hits a section where work can be distributed into several processors that perform the computations (CPU core spans several CPU threads). Afterwards, when all the computations are completed, all the CPU threads converge to a single thread again, which is called the *master* thread.

If a machine has a multicore processor and several GPUs, the parallelization can be achieved by defining a number of threads in the parallel region equal to the number of GPUs. In this sense, each CPU thread deals with a GPU. This is very important since a CPU thread is bound with a GPU context. Thus, all subsequent CUDA calls (e.g. `cudaMalloc`) allocate memory only in its corresponding GPU context [39].

Recent CUDA releases (beyond 2.x capability and CUDA SDK 4.x) allow the time employed in data transfers among GPUs to be reduced by using the UVA (Unified Virtual Addressing) feature. That means that inter-GPU communication (peer-to-peer, P2P) can also be performed without routing the data through the CPU, saving PCI-E bandwidth. Before the appearance of these recent features, communication among GPUs had to be carried out through memory space in the CPU, as shown in Fig. 2.15.

2.7 Tools used for the development of the thesis

These are the features of the different Nvidia GPU and multi-core architectures devices together with their specifications that are used for performing

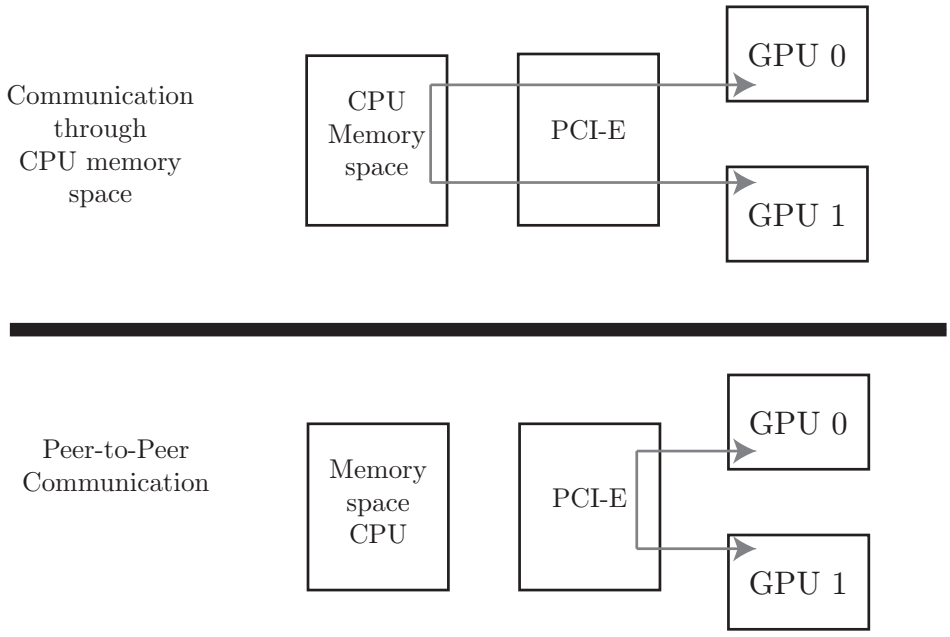


Figure 2.15. The UVA feature reduces data-transfer time among GPUs by using peer-to-peer communication (bottom).

all the implementations in this dissertation:

Parallel Multi-core computer: Golub

- CPU: Intel multi-core (4 physical cores).
- Operative System: Linux.
- GPU: Nvidia Quadro FX 5800
- Compute Capability: 1.3
- Cuda SDK: 2.3
- Architecture: Tesla
- Number of cores: 30 (SM) x 8 cores = 240 cores.

Parallel Multi-core computer: EleanorRigby

- CPU: Intel i7 processor.

- Operative System: Linux.
- GPU: Tesla C2070.
- Compute Capability: 2.0
- Cuda SDK: 2.3
- Architecture: Fermi
- Number of cores: 14 (SM) x 32 cores = 448 cores.

Parallel Multi-core computer: Notebook

- CPU: Intel i7 processor.
- Operative System: Windows (Visual Studio 2008).
- GPU: GTS 360M.
- Compute Capability: 1.2
- Cuda SDK: 3.0
- Architecture: Tesla
- Number of cores: 12 (SM) x 8 cores = 96 cores.

Parallel Multi-core computer: SalaWFS

- CPU: Intel i7 processor.
- Operative System: Windows (Visual Studio 2010).
- GPU: GTX-590.
- Compute Capability: 2.0
- Cuda SDK: 4.0
- Architecture: Fermi
- Number of cores: 16 (SM) x 32 cores = 512 cores.

Parallel Multi-core computer: GpuDSIC

- CPU: Intel multi-core.
- Operative System: Linux.
- GPU1: K20c.

- GPU2: K20c.
- Compute Capability: 3.0
- Cuda SDK: 5.0
- Architecture: Kepler
- Number of cores: 13 (SM) x 192 cores = 2496 cores.

Parallel Multi-core computer: Turbotron

- CPU: Two SMPs (Symmetric Multi-Processing) Intel Xeon CPU X5680 (Two hexacores).
- Operative System: Linux.
- GPU: GTX-690 (internally, this device is composed of two GPUs).
- Compute Capability: 3.0
- Cuda SDK: 5.0
- Architecture: Kepler
- Number of cores: 8 (SM) x 192 cores = 1536 cores.

2.7.1 ASIO protocol

An important feature to develop audio applications is how to communicate the computer with the audio card that capture and reproduce the audio signals. To this end, we employ throughout this dissertation the ASIO protocol.

Audio Stream Input/Output (ASIO) is a computer sound card driver protocol for digital audio specified by Steinberg [41], providing a low-latency and high fidelity interface between a software application and the sound card of the computer. ASIO allows musicians and sound engineers to access external hardware directly.

Steinberg developed a SDK that can be freely downloaded from the web site of his company, as shown in Fig. 2.16. This SDK allows to develop multichannel applications that manipulate output and input audio buffers of any audio card that works with ASIO drivers.

However, not all the personal computers use ASIO drivers. In fact, Windows uses its own protocol to communicate applications with the computer audio card. To solve this limitation, there are specific drivers called

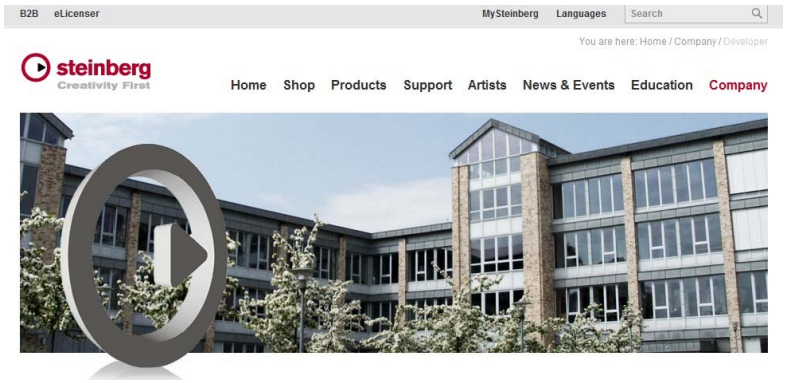


Figure 2.16. http://www.steinberg.net/en/company/developer.html

ASIO4ALL [42] that work as an intermediate layer between the programmer and the proprietary protocol of the operative system.

ASIO specifications work with four audio buffers, two input buffers (A and B) and two output buffers (A and B). Input buffers are related to microphones whereas output buffers with the loudspeakers. This allows to the programmer to write in a output buffer A and/or read from input buffer A, while audio card reproduces what it was previously written in output B and/or captures new audio samples that are stored in input buffer B. After an elapsed time, the audio card interrupts and exchanges the two buffers A for two buffers B, and so on.

State-of-the-Art 3

State-of-the-Art 3

As described in chapter 1, among the effects that are related to participate in the immersive audio schemes, we have dealt in this dissertation with: the generalized crosstalk cancelation and equalization that requires to perform massive filtering; the rendering of spatial audio both with headphones and with loudspeakers; and sound source localization. In all cases, high computational needs are required, since multichannel audio involves the processing of multiple sources, channels, or filters, sometimes in real time. The following sections overview the state-of-the-art of these effects and how they have been computed up to now.

3.1 Generalized crosstalk cancellation and equalization (GCCE)

Many applications such as computer gaming and multi-party teleconferencing make use of spatial audio systems over the IP networks [2]. The participants of this applications need to differentiate competing sounds or voices, but as Huang noted in [6]: "*wearing a tethered headphone to enjoy spatial audio is anyway inconvenient and undesirable, if not cumbersome*". Here arises the challenge of the reproduction of binaural audio without the

use of headphones. However, this is not straightforward, since binaural signals are distorted by room reverberation when arriving at the listener's two ears, which leads to the need for a generalized crosstalk cancellation and equalization (GCCE). This concept was introduced by Atal and Schroeder [43] and Bauer [44] in the early 1960's. Since then, sophisticated algorithms have been proposed, using two or more loudspeakers for rendering the binaural sounds. Most of these algorithms are based on least-squares techniques [45] [46]. One of the recent research where crosstalk cancelation takes an important role is in the format for future TV broadcasting, where it is being analyzed the Binaural Reproduction of 22.2 Multichannel Sound over Frontal Loudspeakers [47].

Up to now, most of the implementations of crosstalk cancelations were carried out with few number of loudspeaker, since the required computational capacity increased as the number of sources and loudspeakers increase [48] [49]. The use of the GPU as co-processor that carries out all audio processing is a challenge that is addressed in this dissertation. To audio engineers, it is important to know how the GPU can deal with the audio samples, and how the required operations can be parallelized in order to obtain maximum utilization of the GPU computational resources. Up to date, there are some publications that focus on the convolution operation and implement it on GPU, but none of them combines the result of multiple real-time convolutions.

Cowan and Kapralos were apparently the first ones to implement a convolution algorithm on GPU using the OpenGL shading language [50] in [51], but they do not take into account the time spent in data transfer GPU $\Leftrightarrow$ CPU, which can be critical in a real-time application. The convolution algorithm shown in [52] has the feature of reducing the latency of the system. This is achieved by subdividing the filters into several subfilters of equal length. Their GPU-implementation is able to convolve 352 channels with filters of 44100 coefficients in a time of 10.53 ms. Moreover, the study of [53] reveals that at a buffer size of 1024 samples, the maximum length for a single channel FFT on a GPU was around 4 million samples. The last two references offer excellent results for GPU performances; however, neither of them goes into the analysis of GPU-parameters.

Chapter 4 presents a complete study that extrapolates the implementation of multiple convolutions to a GCCE application on a GPU attending to different and common situations: the size of data buffers that are much

larger than the size of filters and the size of data buffers that are much smaller than the size of filters.

3.2 Headphone-base spatial audio

There are other applications where spatial sound is required to be rendered by using headphones, specially when somebody tries to be isolated from outside. This common situation can occur for example during console gaming in public transport such as busses, trains or also in airplanes.

Spatial sound by using headphones is also called binaural hearing, and offers a number of important advantages over monaural hearing. This is due to the fact that binaural hearing provides additional information, which is encoded in the differences of the input signals to the two ears [54]. Among other features, binaural sound allows to locate the position of a sound source.

Powerful gaming effects require to use binaural sound to give more realism to scene rendered by the 3D graphics and to transmit an immersive feeling inside the game. Moreover, gaming in network is becoming popular and communication among the participants is necessary to achieve the game target. Binaural sound offers the possibility to focus attention to the ear with a better signal-to-noise ratio [9] and thus, to understand better the conversations among participants during gaming.

To synthesize a binaural sound from a simple mono recording, it is necessary to use special filters that are known as *Head-Related Transfer Functions* (HRTFs). The response of HRTFs describes how a sound wave is affected by properties of the body shape of the individual (i.e., pinna, head, shoulders, neck, and torso) before the sound reaches the listeners eardrum [9]. Individualized HRTFs are traditionally obtained either through measurements and extrapolations [55] or through numerical simulation [56]. Recently, in [57], Enzner presented a specific dynamical measurement based on adaptive filtering that allows a quasi-continuous HRTF representation to be obtained. A set of HRTFs of different spatial fixed positions configure a HRTF database. There are multiple databases of HRTFs or HRIRs on Internet such as, [58] or [59].

Spatial sound through HRTFs has also made use of GPUs. The first

studies date from 2004. In [60], the use of the GPU for carrying out spatial audio processing is studied. In [61], the GPU is used for delay, gain, air absorption, and HRTF filtering in real-time auralization. An implementation using the OpenGL shading language [50] is presented in [62], where the accuracy of the filtered spatial audio signal on GPU using HRTFs is assessed. Until now, the related works have focused on evaluating the GPU performance for different environments; however, none of the previous work presented a real-time application whose audio processing is totally carried out on a GPU.

Chapter 5 describes the designing and the implementation of a binaural spatial sound application that runs on a notebook with the GPU GTS360M, ideal for computing gaming in public transport. This application interacts with a user who selects, changes, and moves the sound sources in real time. The described application faces two common problems when we deal with a HRTF database composed of a limited number of spatial fixed positions: render sound sources at any position of the space and virtualize movements of the sound sources.

3.3 Wave Field Synthesis

In the last decades, there has been increasing the interest in listening experience and more specifically in spatial audio rendering. One of the spatial audio systems available today is the Wave Field Synthesis (WFS), where sound field is synthesized in a wide area by means of arrays of loudspeakers, which are referred to as secondary sources. WFS is usually implementing discrete-time signal processing and is able to reproduce complex auditory scenes consisting of multiple acoustic objects, which are generally denoted as primary or virtual sources. WFS concept was introduced at the Delft University of Technology around 80's and 90's decades. Berkhout carried out first researchers in this field [63, 13], which were followed by different dissertations such as [64, 65, 66, 67, 68].

One of the problems to put WFS in practise is related to the interaction of the array with the listening room. The listening room introduces new echoes that are not included in the signal to be reproduced, thus altering the synthesized sound-field and reducing the spatial effect. One block that can be added to this system is a Room Compensation (RC) block.

The purpose of this block is to minimize the undesirable interaction of the array with the listening room. A common RC block is based on a multichannel inverse filter bank that corrects the room effects at selected points within the listening area, such as those in [16] and [17]. This formula is validated by [18], where it is presented meaningful improvements in the acoustic field when a RC block is applied to a WFS system. However, the application of this spatial audio system (WFS + RC) in real environments (theaters, cinemas, etc.) requires a real-time solution which demands high computational capacity.

Up to now, there were different researches that aimed to implement a WFS system. In [69], it is presented a WFS implementation that benefited of a time invariant preprocessing in order to reduce CPU load. In [70], Theodoropoulos et al. propose a minimalistic processor architecture adapted to WFS-based audio applications. They estimated that their system could render in real time up to 32 acoustic sources when driving 64 loudspeakers. The same authors presented in [71] a WFS implementation on different multi-core platforms, including a GPU-based implementation that achieved more than 64 sources when driving 96 loudspeakers. They concluded that GPUs are suitable to build inmersive-audio real-time systems.

None of the previous implementations has approached computationally the problem of the interaction of the WFS with the listening room. Chapter 6 is devoted to design, implement and assess a spatial audio system of these characteristics. The main feature of this implementation is that all audio processing is carried out by a GPU.

3.4 Sound source localization

Microphone arrays are commonly employed in many signal processing tasks, such as speech enhancement, acoustic echo cancellation or sound source separation [72]. The localization of broadband sound sources under high noise and reverberation is another challenging task in multichannel signal processing, making it being a very active research topic for applications in human-computer interfaces, teleconferencing or robot artificial audition. Algorithms for sound source localization can be broadly divided into indirect and direct approaches [7]. Indirect approaches usually follow a two-step

procedure: they first estimate the *Time Difference Of Arrival* (TDOA) [73] between microphone pairs, and, afterwards, they estimate the source position based on the geometry of the array and the estimated delays. On the other hand, direct approaches perform TDOA estimation and source localization in one single step by scanning a set of candidate source locations and selecting the most likely position as an estimate of the real source location. Although the computation of TDOAs usually requires time synchronization, new approaches are being developed to avoid this limitation [74]. Most localization algorithms are based on *Generalized Cross-Correlation* (GCC) [75], which calculates the cross-correlation function of the received signals by using the inverse Fourier transform of the cross-power spectral density of the signals, which is suitably weighted.

The *Steered Response Power - Phase Transform* (SRP-PHAT) algorithm is a direct approach that has been shown to be very robust in adverse acoustic environments [19]. The algorithm is usually interpreted as a beamforming-based approach that searches for the candidate position that maximizes the output of a steered delay-and-sum beamformer. Since localization accuracy can be improved by using high-resolution spatial grids and a high number of microphones, accurate acoustic localization systems require high computational power.

The use of GPUs for implementing sound source localization algorithms has also recently been tackled in the literature. The time performances of different localization algorithms implemented on GPU were reported in [76] and [77]. In fact, although different implementations of the SRP-PHAT in the time-domain and frequency-domain are analyzed in [76], their results mainly focus on pure computational issues and do not discuss how localization performance is affected by using different numbers of microphones or a finer spatial grid.

Chapter 7 is aimed at demonstrating how localization systems using a high number of microphones distributed within a room can perform real-time sound source localization in adverse environments by using GPU massive computation resources. We discuss how massive signal processing for sound source localization can be efficiently performed by Multi-GPU systems, analyzing different performance aspects on a set of simulated acoustic environments.

3.5 GPU computing in other research inside audio field

GPU computing has already been applied to different problems in acoustics and audio processing. Excellent survey on audio-related topics that can be performed on GPU have been presented earlier by Tsingos in [32]. Studies of computing room acoustics were carried out by Webb and Bilbao in [78], Savioja in [79], Southern in [80], and Hamilton in [81] as well as geometric acoustic modeling like ray-tracing [82] [83]. Computer-music synthesis using additive synthesis and sliding phase vocoder was developed on GPU in [84] and in [85], respectively. Parallel implementations of beamforming design and filtering using GPUs were also presented in [86].

3.6 Conclusion

As can be appreciated in this chapter, we have reviewed different effects that are related to the immersive audio schemes. The impact of these effects within the audio schemes can increase meaningfully when they are brought to the multichannel field and can be also performed in real time. The computational development of the GPUs has opened a new paradigm where the massive required processing by multichannel field effects can be carried out by the GPU. The present dissertation aims to implement applications that make use of these effects in order to assess their performances. These performance results will be helpful to audio engineers that will be able to know which are the limits of these applications. Moreover, detailed information about GPU implementation can be extracted from this dissertation that can be used in applications that use similar audio processing.

Massive Multichannel Filtering 4

4 Massive Multichannel Filtering

This chapter describes GPU-based implementations of massive multichannel filtering whose filters present a Finite Impulse Response (FIR). The chapter describes firstly the implementation of a single convolution on a GPU, then this implementation is extrapolated to carry out multiple convolutions. Finally, it is presented an application that requires a large number of concurrent convolutions: generalized crosstalk cancellation and equalization. Two common situations are properly managed in this application: size of buffers that are much larger than the size of the filters and size of buffers that are much smaller than the size of the filters.

4.1 Convolution

The key operation in a multichannel filtering block is the convolution. Together with CUDA technology, Nvidia made available to the GPU developers a Nvidia GPU Computing SDK (Software development kit) [20]. First approaches to develop the convolution on the GPU were taken from the CUDA 2.3 SDK release (in 2010). This SDK collected different cuda projects that could be taken as examples for the GPU developers. Among them, there was an example of convolution. This example made use of the

CUFFT library [87] to carry out the FFT of two vectors, which were afterwards element-wise multiplied. Finally, CUFFT library was used again to carry out the iFFT of the resulting vector of the element-wise multiplication.

However, the implementation in the SDK has large-scale limitations. The convolution is applied to the whole signal. Thus, it requires that the whole signal and the filter are first sent to the GPU, and after the convolution, the whole output signal is also transferred back to the CPU. From the signal processing point of view, this implementation prevents to carry out a real-time application. In fact, the length of the signal to convolve is not known most of the times. Besides, from an audio perspective, the latency of the system increases meaningfully, since the output signal begins to be reproduced after the convolution operation which can take several seconds.

First contribution of this thesis consists of developing a real-time convolution on the GPU. This main advantage of this implementation is that the reproduction of the convolved signal can start without having to wait for the processing of the whole signal. To this end, and attending to the mentioned in Section 2.3.5, we use the described technique in Section 2.3.4: Overlap-save. This first development aims to demonstrating that GPUs are valid for real-time audio applications.

First step in this implementation consisted in configuring a matrix of samples that is denoted in the following as **S**. This matrix is composed of signal blocks that arise from the overlap-save technique. Figure 4.1 illustrates how matrix **S** is configured from signal blocks, where x^i are the overlap-save blocks of size l_o that are obtained from signal x. The convolution is carried out between signal x and a filter h whose size is l_h ($l_h \ll l_o$). In this implementation, the number of rows of matrix **S** in the system is fixed to P. Previous to the processing, the filter h is transferred to the GPU and its length is zero-padded to l_o, then, the filter h is replicated P times in the memory in order to configure a matrix **H**. Thus, convolution operation is now reduced to an element-wise multiplication between the matrices **S** and **H**. This design aims to parallelizing the largest number of operations.

The GPU-based implementation uses firstly the CUFFT Nvidia FFT library that allows to execute multiple FFTs in one dimension at the same time. Therefore, it is possible to compute P FFTs concurrently. Thus, matrices **S** and **H** are converted in two matrices composed of complex

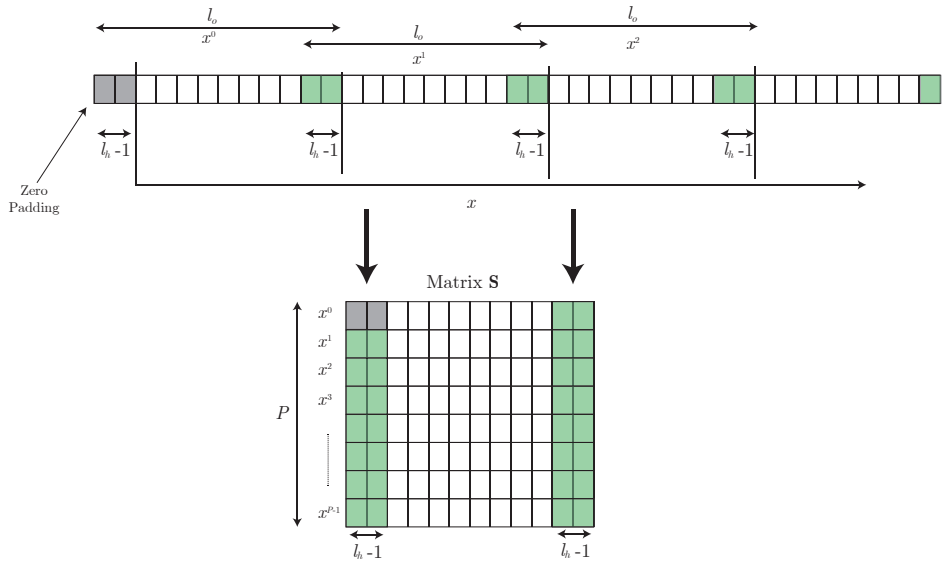


Figure 4.1. Matrix **S** is built from signal blocks.

numbers. All the operations in matrix **H** are performed previous to the real-time processing, which can be summarized in:

1. Configuration of matrix **S**.

2. Transfer of matrix **S** from CPU to GPU.

3. A FFT is applied to each row of the matrix **S**.

4. A CUDA kernel element-wise multiplies matrices **S** and **H**. To this end, we launch a CUDA kernel composed of $l_o \cdot P$ threads and each thread perfoms a simple complex multiplication (see CUDA kernel 1). The result of the multiplication is stored again in **S**. Figure 4.2 illustrates these operations.

5. An iFFT is applied to each row of the matrix **S**.

6. Transfer of matrix **S** GPU to CPU.

7. Output signals are built by discarding first $l_h - 1$ samples of each row of the matrix **S**, as overlap-save technique indicates in Section 2.3.4.

CUDA Kernel 1

In CUDA Kernel 1, it must be pointed out that number of blocks launched in this kernel is $\frac{l_o}{128} \times \frac{P}{2} \times M$, being `BlockDim.x=128` and `BlockDim.y=2`.

CUDA Kernel 1 Element-wise multiplication

Input: $\mathbf{H}, \mathbf{S}$, l_o

Output: $\mathbf{S}$

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int Row = BlockIdx.y * BlockDim.y + ThreadIdx.y;
3:  // Global Thread -> Idx;
4: int idx = Row * l_o + Col;
5:  // Complex Multiplication between two complex elements
6: S[idx] = ComplexMultiplication(S[idx],H[idx]);
7:  // Scaling the multiplication
8: S[idx] = ComplexScale(S[idx], 1/l_o);
```

In order to perform the implementation we use the *Golub* machine (see Section 2.7) which includes an Nvidia Quadro FX 5800 and, one of the first CUDA devices. The described implementation is compared with the implementation proposed in the Nvidia SDK by using a signal x composed of 176400 samples, and a filter h composed of 220 coefficients. Parameters P and l_o where set to 32 and 512, respectively. This implementation is approximately twice faster than the proposed implementation of the Nvidia SDK. More details can be found in [88]

4.1.1 Pipelined algorithm in a multichannel system

Extending this first implementation to a multichannel system is achieved by sharing the rows of the matrices $\mathbf{S}$ and $\mathbf{H}$. Figure 4.3 shows how overlap-save blocks of four different signals are filtered by two different filters.

The CUDA 2.3 SDK release introduced the *concurrent copy and execution* property that allows apparently to overlap memory transfers with the computational operations that were carried out by the CUDA kernel. This property introduced the CUDA concept of *streams* (see Section 2.6.1) that allows to execute different task in parallel. The use of *streams* in the implementation implies to use asynchronous operations Thus, it was developed a new GPU-based implementation following a four-stage pipelined

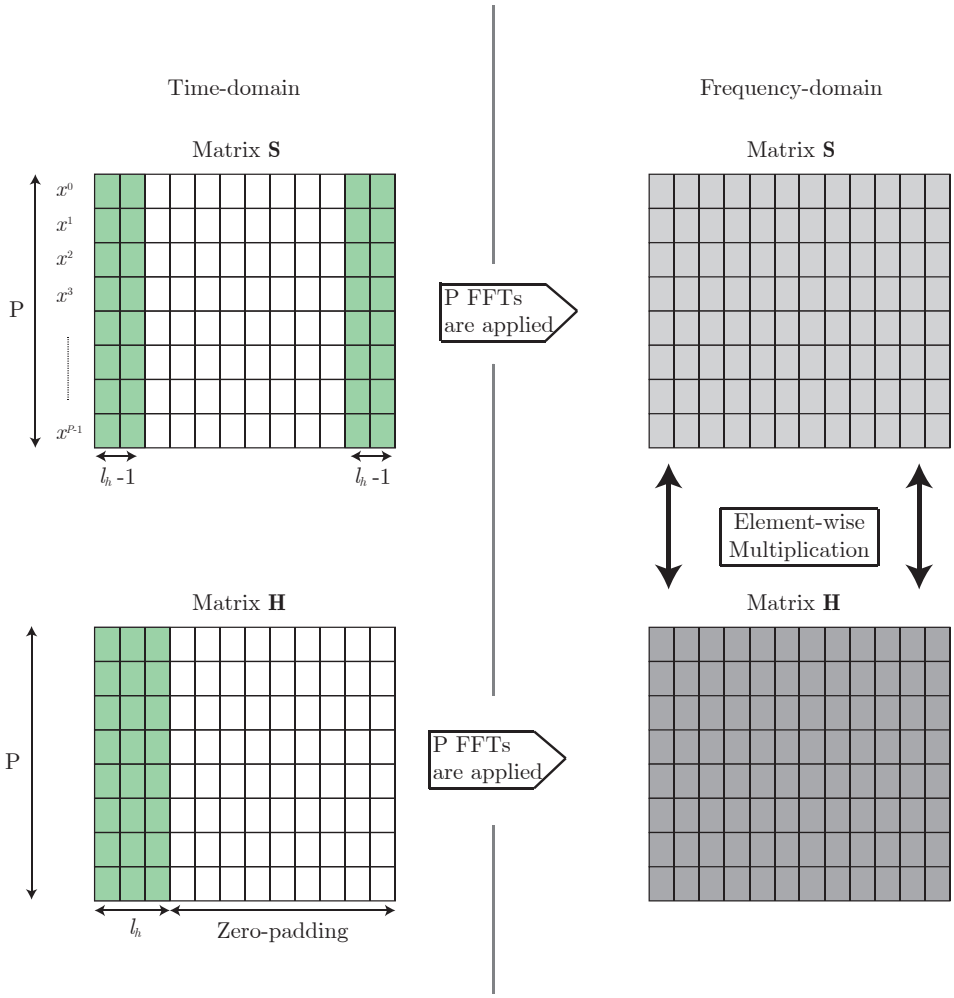


Figure 4.2. P FFTs are applied to matrices **S** and **H**. Afterwards, both matrices are element-wise multiplied.

model. This implementation is based on the following points:

1. Matrix **H** is configured and sent to the GPU.
2. First matrix **S** is configured and is referenced as the matrix A-**S**.
3. Using asynchronous transfer, while A-**S** is transferred to the GPU by *stream* 1, another matrix, B-**S** is built simultaneously by *stream* 2.

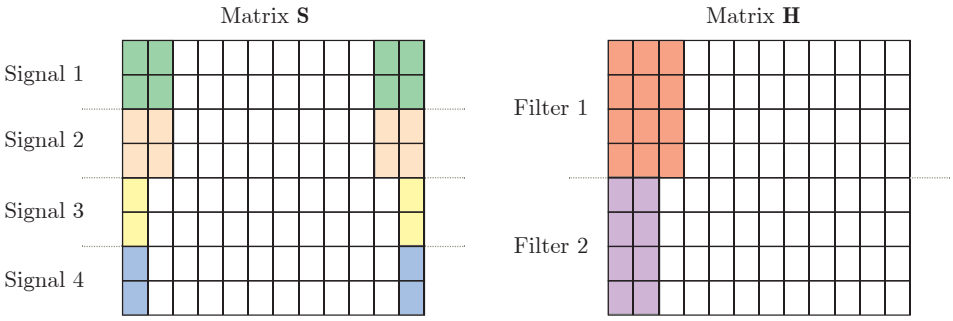


Figure 4.3. Matrices **S** is composed of samples of four different signals, and **H** is composed of coefficients of two different filters.

4. *Stream* 1 executes the CUDA kernel 1 that element-wise multiplies between the matrices **H** and A-**S**, while matrix B-**S** is transferred from CPU to GPU by *stream* 2, and a new matrix C-**S** is built by *stream* 3.

5. Finally, a new matrix D-**S** is built by *stream* 4, while matrix C-**S** is transferred from CPU to GPU by *stream* 3, execution in GPU is carried out on matrix B-**S** by *stream* 2, and matrix A-**S** is transferred back to CPU by *stream* 1.

6. Once matrix A-**S** is received on the CPU, the outputs of the signals are computed by discarding the first samples of each one of the rows, as overlap-save technique indicates in Section 2.3.4. Afterwards, the memory positions that allocated the matrix A-**S** are used again.

Thus, the four matrices **S** (A, B, C and D) are being used cyclically. Figure 4.4 shows all the stages with the time required by each of them. The test setup is the same as in the previous section: a signal x composed of 176400 samples, a filter h composed of 220 coefficients, and the *Golub* machine. It must be pointed out that the block called "Rebuilding Signals" represents the last step in the previous enumeration and begins once the whole matrix **S** is back to CPU, in order to avoid race conditions. Thus, both operations are in the same step of the implementation.

Configuring both matrices (**S** and **H**) with l_o=512 columns, and P=32 rows, the number of audio samples per row is 293 (512-220+1). If we use

this model for performing a real-time system with a sample rate f_s=44.1 kHz, the $t_{\text{buff}} = \frac{293}{44.1} = 6.6$ ms. Figure 4.4 indicates that the time $t_{\text{proc}} =$ 9.37 ms. This number comes from the sum of all the steps executed by one *stream* taking into account some conflicts among adjacent *streams* when more than a transfer between CPU and GPU exists simultaneously, as it was documented in [89] for the release CUDA 2.3 .

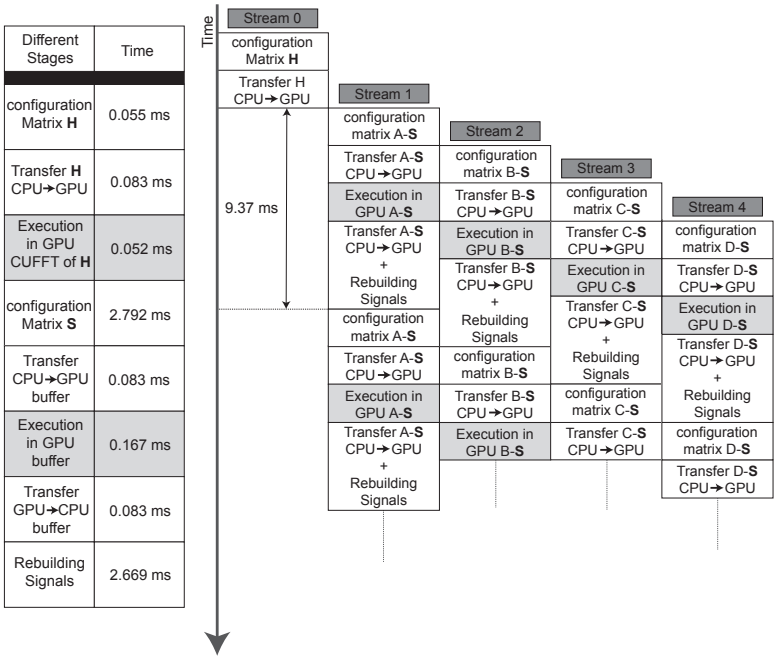


Figure 4.4. Four matrices are needed in order to carry out a pipelined algorithm.

If one row of the **S** were dedicated to one signal, then the executing time of 9.37 ms would be larger than the filling buffer time of 6.6 ms and thus the application would not work properly in real time. In this case, some of the incoming samples would not be processed because the matrix *A*-**S** (Fig. 4.4) would not be available to be filled of samples. Table 4.1 shows that this GPU implementation allows managing up to 16 audio channels simultaneously using a matrix **S** with a size of 32×512. More details can be found in [90] and [91].

Table 4.1. Number of possible audio channels in the application using a matrix $\mathbf{S}$ with a size of 32×512

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

4.2 Crosstalk Cancellation using a stereo signal

An improved implementation of the convolution on GPU was afterwards developed to carry out not only multiple convolution, but also, to combine them. This section presents a real application where multiple convolutions are used, and that runs in a GPU of a notebook computer. This application consists in a *Crosstalk Cancellation* of a stereo signal. The purpose of this implementation is to know if a GPU can be used as a co-processor that carries out all the audio processing. This is important, since the CPU resources would be released and used for other tasks when an audio application is running.

4.2.1 Definition of the problem

The general approach to a 3-D audio system is to reconstruct the acoustic pressures in the listener's ears that would result from the natural listening situation. To accomplish this by using loudspeakers requires: 1) the ear signals corresponding to the target scene are synthesized by appropriately encoding directional cues (a process known as binaural synthesis), and 2) these signals are delivered to the listener by inverting the transmission paths that exist from the speakers to the listener (a process known as crosstalk cancellation).

In a stereo system, the listener receives contributions to each ear from both loudspeakers. Cross terms (h_{RL}, h_{LR} denote the impulse response of

the crossed paths in Fig. 4.5) prevent virtual sound sources to be located spatially and degrade the perception of speech by Haas Effect [92]. In order to eliminate the cross contributions, a net of crosstalk cancelation as in [93] is designed. To implement this net, four filters denoted as f_{LL}, f_{LR}, f_{RL} and f_{RR} are computed using the fast deconvolution method with regularization, [94]. These filters are used in order to correct the effect that produce the cross paths h_{RL} and h_{LR}. Equation 4.1 shows the sound that comes to the

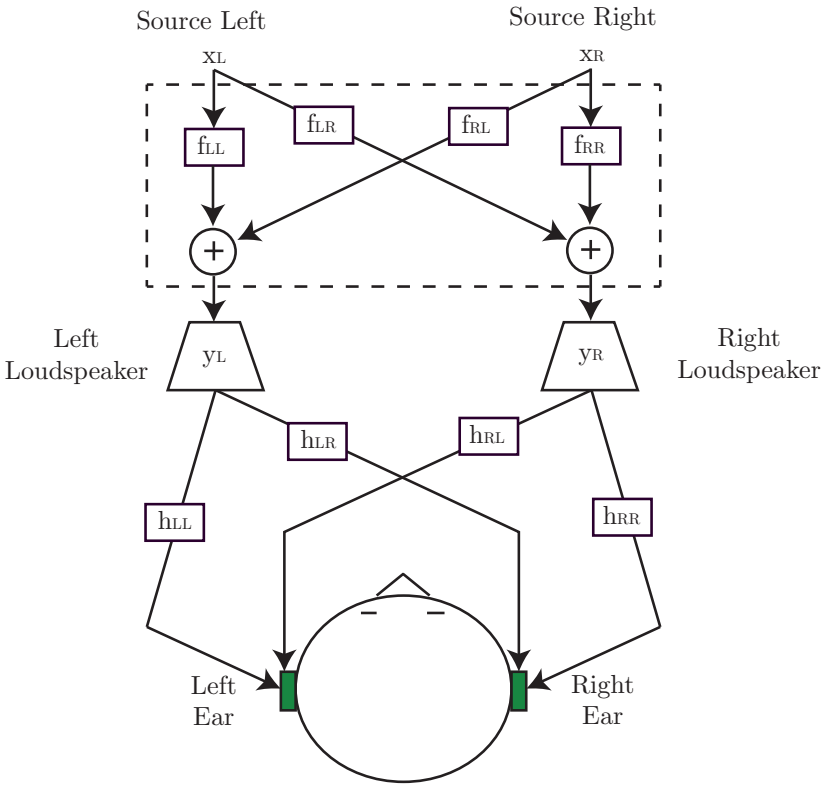


Figure 4.5. Crosstalk canceller filters.

left ear, where x_L and x_R corresponds to the rendered signals, and h_{LL} and h_{RR} are the direct paths. Figure 4.5 clarifies the rol of each variable:

$$\text{Left ear} \Rightarrow (x_L * f_{LL} + x_R * f_{RL}) * h_{LL} + (x_L * f_{LR} + x_R * f_{RR}) * h_{RL}. \quad (4.1)$$

We can also write the previous equation in terms of the sound source as:

$$\text{Left ear} \Rightarrow x_L * (f_{LL} * h_{LL} + f_{RL} * h_{RL}) + x_R * (f_{RL} * h_{LL} + f_{RR} * h_{RL}). \quad (4.2)$$

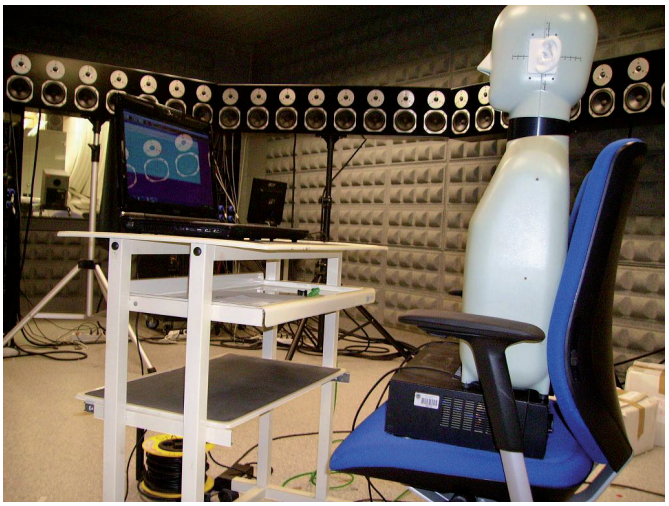


Figure 4.6. Measurement of the transmission path filters.

Considering the previous equations, the designed filters f_{RL}, f_{LL}, f_{RR}, and f_{LR} must fulfill the following equations:

$$f_{LL} * h_{LL} + f_{LR} * h_{RL} = 1, \tag{4.3}$$
$$f_{RL} * h_{LL} + f_{RR} * h_{RL} = 0. \tag{4.4}$$

The previous analysis is analogously carried out for the right ear. In the experiments, the coefficients values of h_{RR}, h_{LL}, h_{RL}, and h_{LR} were measured using a dummy with microphones in both ears and computed through MLS algorithm [95], as shown in Fig. 4.6. Afterwards, by using an impulse response as input signal, we have simulated the acquired signal by the microphone in the left ear in two situations: 1) The direct path (signal goes through filters f_{LL} and f_{RL}), and 2) the cross path (signal goes through filters f_{LR} and f_{RR}). Figure 4.7 shows how the contributions of the direct path achieve a signal level around 1 dB, whereas the contributions of the cross path hardly achieve 0.016 dB. The designed technique is sensitive to the location of the listener because filters do not change in the actual implementation. So, in order to sense the binaural sound, the listener must be located as the dummy in Fig. 4.6.

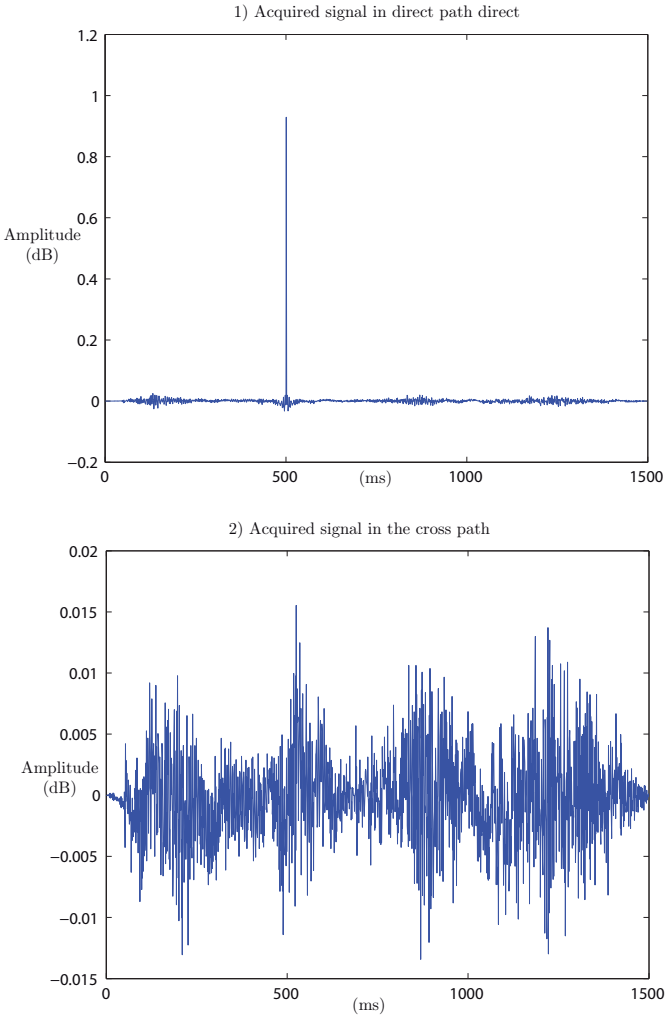


Figure 4.7. Acquired signals in the left ear: 1) only direct path (signal goes through filters f_{LL} and f_{RL}), and 2) only the cross path (signal goes through filters f_{LR} and f_{RR}).

4.2.2 GPU Implementation

The multichannel reproduction system generates the signals y_L and y_R that are reproduced through the loudspeakers. These signals are the sum of two convolutions that have to be carried out in real time. Equation 4.5 shows

the operations to implement on the GPU.

$$\begin{aligned} y_L &= f_{RL} * x_R + f_{LL} * x_L, \\ y_R &= f_{RR} * x_R + f_{LR} * x_L. \end{aligned} \tag{4.5}$$

The convolution operation is computed using the overlap-save technique using overlap-save blocks of size l_o. Thus, three kinds of operations are carried out on GPU: FFT and iFFT transformations (computed by library CUFFT [20] from NVIDIA), an element-wise multiplication of different vectors, and a vector sum. The last two operations are performed by the following CUDA kernels.

CUDA Kernel 2

In comparison with the previous implementations, we do not configure a matrix $\mathbf{H}$, i.e., filter coefficients are not any more replicated. We opt in this implementation for using the *shared-memory* whose access time is around 10 times less than the access time to *global-memory*. To this end, filters must be transferred first from CPU to GPU, then transferred from *global-memory* to *shared-memory*. Matrix $\mathbf{S}$ is set to use 16 overlap-save blocks (8 overlap-save blocks per signal).

The operations to carry out by CUDA kernel 2 consists of element-wise-multiplying all the rows of the matrix $\mathbf{S}$ by its corresponding filters. The results are stored in different memory positions that configure a matrix with twice the size of $\mathbf{S}$, denoted in CUDA kernel 2 as $\mathbf{S}_{res}$. Matrix $\mathbf{F}$ in CUDA kernel 2 is composed of 4 rows and l_o columns, and stores the four filters in the frequency domain F_{LL}, F_{LR}, F_{RL}, and F_{RR}.

The *shared-memory* allows that the same coefficient values can be used by different CUDA threads that belong to the same block of threads. Hence, this GPU implementation requires to set the CUDA parameters `BlockDim.x` and `BlockDim.y` to 32 and 8, respectively. Thus, the number of blocks that are launched is: $\frac{l_o}{32} \times \frac{16 \cdot 2}{8}$, in a 2-D cuda grid. Figure 4.8 clarifies the setting of cuda parameters and how the threads of the different blocks interact with matrices $\mathbf{H}$ and $\mathbf{S}$ in order to configure Matrix $\mathbf{S}_{res}$, which contains samples of the four convolutions.

CUDA Kernel 2

CUDA Kernel 2 Element-wise multiplication with *shared-memory*

Input: $\mathbf{F}$, $\mathbf{S}$, l_o

Output: $\mathbf{S}_{res}$

```
1: __shared__ Complex F_s[32];
2: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
3: int Row = BlockIdx.y * BlockDim.y + ThreadIdx.y;
4:  // Global Identification -> Idx
5: int idx = Row * l_o + Col;
6:  // Identification for accessing matrix S
7: int idxMod = idx & (16 * l_o-1);;
8:  // Filters to shared memory;
9: if(ThreadIdx.y== 0)
10:    F_s[ThreadIdx.x] =  F[Col + l_o*BlockIdx.y];
11: end if
12:  __syncthreads();
13:  // Complex Multiplication between two complex elements
14: S_res[idx] = ComplexMult(S[idxMod],F_s[ThreadIdx.x]);
15:  // Scaling the multiplication
16: S_res[idx] = ComplexScale(S_res[idx], 1/l_o);
```

Last step of the implementation consists of carrying out the sums of (4.5). To this end, it is launched CUDA kernel 3. This kernel has the same settings regarding CUDA parameters, but with the half of threads, in comparison with CUDA kernel 2. Thus, the number of blocks that are launched is: $\frac{l_o}{32} \times \frac{16}{8}$, in a 2-D grid.

CUDA Kernel 3

CUDA Kernel 3 Element-wise Sum

Input: $\mathbf{S}_{res}$

Output: $\mathbf{S}_{res}$

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int Row = 2 * BlockIdx.y * BlockDim.y + ThreadIdx.y;
3:  // Global Identification -> Idx
4: int idx = Row * l_o + Col;
5:  // Complex Multiplication between two complex elements
6: S_res[idx] = ComplexSum(S_res[idx],S_res[idx + 8*l_o]);
```

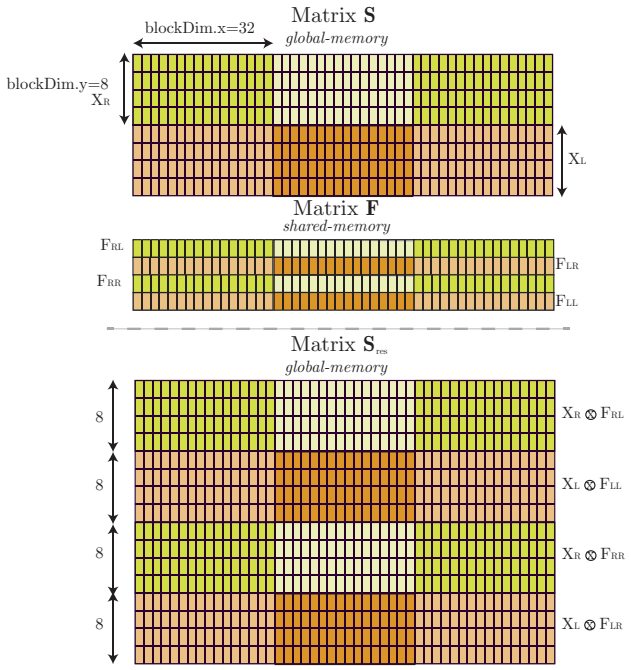


Figure 4.8. Matrices **S**, **F**, $\mathbf{S}_{res}$ used in CUDA kernel 2 and CUDA kernel 3.

4.2.3 Test system and Results

The test system is a notebook Intel Core i7 running at 1.60 GHz with a GPU Geforce GTS360M with 1.2 capability. The experiment that is performed implements a crosstalk cancellation of two individual wav files (voice.wav and piano.wav). The listener, situated at 90 cm from loudspeakers, perceives in the left ear the voice signal and in the right ear the piano signal without appreciable interference. The platform used to develop the software is Microsoft Visual Studio 2008, joined to Steinberg's ASIO interface [41] for low-latency audio streaming. Figure 4.9 summarizes the real-time operations of a crosstalk cancelation in an application that uses the CPU and the GPU. To perform the GPU-based implementation, the algorithm has also been developed on CPU. In this case, the same operations are carried out sequentially. Our test consists in running both implementations on the presented notebook, and observing the task Manager on Windows operating system to see the % of resources that are being demanded by the

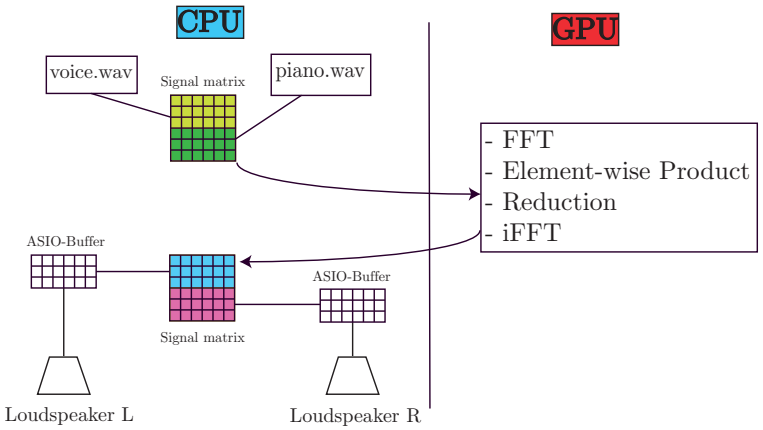


Figure 4.9. Required operations on a application that performs a crosstalk cancelation by using the CPU and the GPU.

CPU during the application running. Figure 4.10 shows the task Manager on Windows operating system in both cases: a) GPU-based implementation, and b) CPU-based implementation.

When all the processing is carried out on the CPU, the task Manager on Windows operating system shows that up to a 20% of its capacity is used, while the GPU-based implementation shows a percentage of use around 1%. It means that if a crosstalk application with only two signals is saving 20% of the CPU capacity, then in a theater, or maybe in a funfair where more signals are required, the possible reduction in resources can be highly significant. More details can be found in [96].

4.3 Multichannel massive audio processing for a GCCE application

Previous section checked that the use of the GPU as a co-processor for carrying out audio processing tasks has sense. In this section, we go a step further and extrapolate the crosstalk cancellation using two signals to a generalized crosstalk cancellation, which involves the processing of multiple sources, channels, or filters.

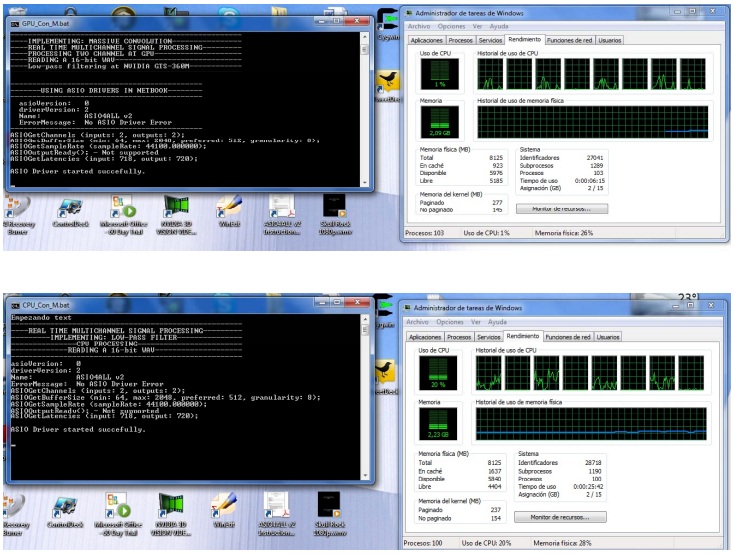


Figure 4.10. Task Manager on Windows operating system in both cases: a) GPU-based implementation, and b) CPU-based implementation.

4.3.1 Definition of the problem

One application, that is especially important in the context of multichannel acoustic signal processing, is the reproduction of binaural audio without the use of headphones. *Generalized Crosstalk Cancellation and Equalization* (GCCE) plays an important role in this phenomenon by inverting the transmission paths between loudspeakers and listeners. Assuming a reproduction scenario with Z listeners, each listener would receive contributions from every loudspeaker at both ears. The aim of the GCCE is to create a pair of desired signals that are not disturbed by these contributions at the ears of the listeners. Figure 4.11 shows the placement of $2 \cdot Z$ desired signals, one signal per ear (represented by d_{zR} and d_{zL}, $z \in [0, Z-1]$, L=Left ear and R=Right ear) in a room. Let's define M and N as the total number of sources and loudspeakers, respectively, where x_m is the m-th source, and y_n is the n-th loudspeaker signal.

An application example would be a scenario where there are several people watching a movie in the same room and each of them is capable of

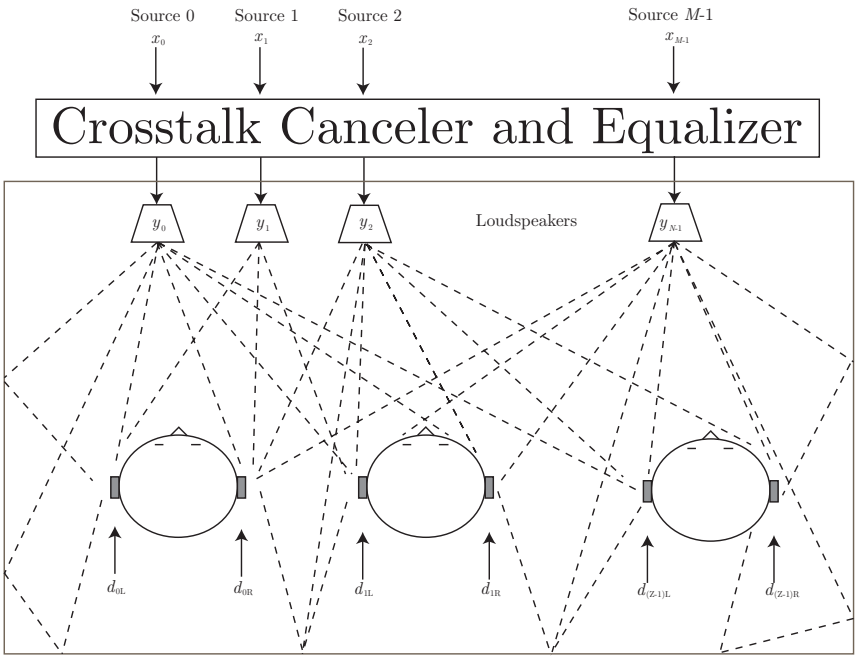


Figure 4.11. $2 \cdot Z$ desired signals are set to each ear of Z listeners in a room. Cross paths and room effects are canceled by means of the use of the *Crosstalk canceler and Equalizer* block.

listening to the audio in a different language without the use of headphones. GCCE is mainly based on combining the output signals resulting from convolution operations in such a way that a given special acoustic effect is achieved. The block *Crosstalk Canceler and Equalizer* in Fig. 4.11 is a filter bank with the structure shown in Fig. 4.12, where the filter implemented between the m-th source and the n-th loudspeaker has an impulse response given by f_{mn}, with $m = 0, \ldots, M-1$ and $n = 0, \ldots, N-1$. All operations of the multichannel reproduction system are reflected in (4.6), where $*$ denotes the convolution operation.

$$y_n = \sum_{m=0}^{M-1} (f_{mn} * x_m). \tag{4.6}$$

Moreover, parameter C_{tot} represents the number of filters involved in the application. As there is a filtering path from every source to every loud-

speaker, the number of filters implemented is $C_{\text{tot}} = M \cdot N$.

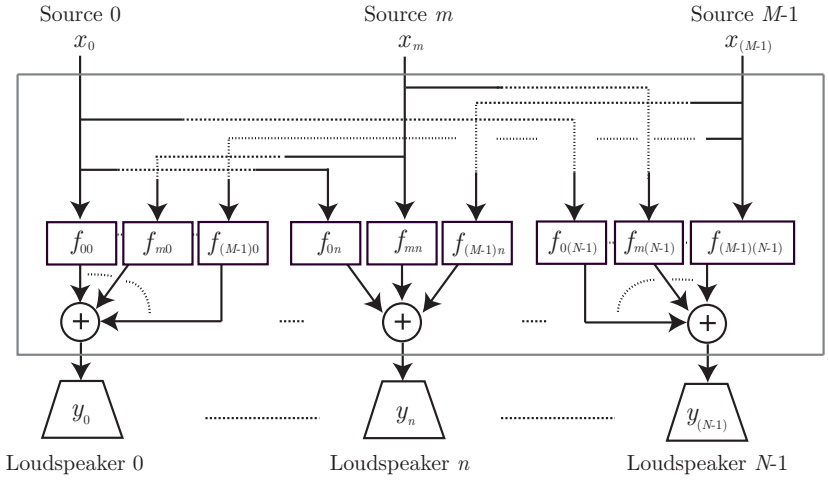


Figure 4.12. The signal at loudspeaker y_n is composed of a combination of all the sources x_m filtered through their respective f_{mn}.

4.3.2 GPU data structure for efficient convolution

The most relevant operation in a Generalized Crosstalk Cancelation is convolution. For this application, we implement the convolution on GPU focusing on two different environments based on the size of the filter (l_f represents the size of the filter) and the size of the input-data buffer (L represents the size of the input-data buffer). An implementation where the size of the input-data buffer is much larger than the size of the filter ($L \gg l_f$) is described in Scheme 1 and is based on the the fragmentation of the input-data buffer. On the other hand, Scheme 2 deals with the opposite case, ($L \ll l_f$) and is based on the fragmentation of the filter. The main goal of fragmentation is to obtain the best performance from the resources on the GPU, which maximally exploits the parallelism. Note that, although both approaches are described independently here, the user does not have to be aware of this issue since the system would choose the most efficient one in a real application for the given task. The following subsections describe both schemes in the easiest situation, a simple convolution of one source with one filter.

Scheme 1: Fragmentation of the input-data buffer.

The implementation we present is based on the overlap-save technique [97]. A matrix **S** is configured using the samples within the input-data buffer. Matrix **S** has P rows and l_o columns. The value P indicates the number of the overlap-save blocks that is configured from the L samples of the input-buffer. The value l_o is the size of the overlap-save blocks. In order to exploit GPU resources, P must be properly selected; its value determines l_o, which also depends on L. The filter must have the same size as the overlap-save blocks. Thus, the filter length will be zero-padded from l_f to l_o. In this scheme, the filter is also considered to be a matrix, which we call **F**. Hence, matrix **F** has 1 row and l_o columns. The reason for configuring data in a matrix structure is to allow the same operation to be executed with different data portions and to allow data to be reused when an element-wise multiplication is carried out between the overlap-save blocks and the filter (Convolution Theorem, see Section 2.3.1). For this operation, matrix **S** stays in *global-memory* and matrix **F** is moved to *shared-memory* on GPU, since filter values are shared for all the overlap-save blocks during the element-wise multiplication (see Fig. 4.13 (a)).

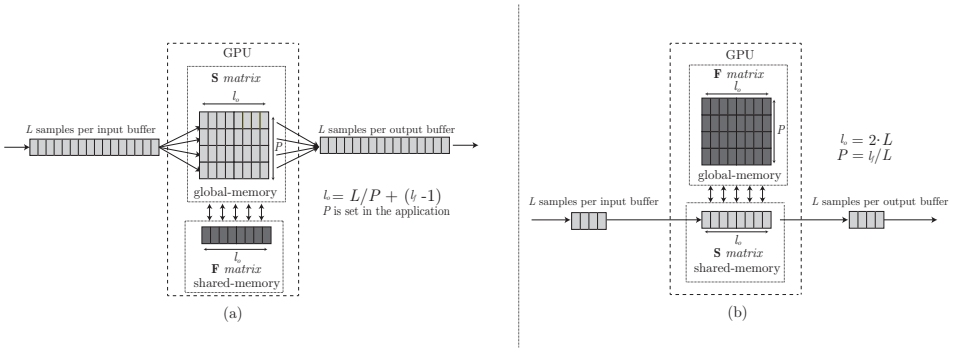


Figure 4.13. (a) shows Scheme 1 where matrix **S** is located in *global-memory* and matrix **F** in *shared-memory*; (b) shows the opposite case, Scheme 2 where matrix **F** is located in *global-memory* and matrix **S** in *shared-memory*.

Scheme 2: Fragmentation of the filter.

This scheme occurs in applications where latency plays an important role and the filter size is much larger than the size of input-data buffer. There-

fore, it is necessary to split the filter into blocks in order to obtain a fast system response. Fragmentation could be done uniformly as in [98] and [99] or non-uniformly as in [100]. For this implementation, the filter is uniformly fragmented into blocks whose size is the same as the size of the input-data buffer. The sizes of the matrices of Scheme 1 change in Scheme 2. Matrix $\mathbf{F}$ now has P rows and l_o columns, where $P = \frac{l_f}{L}$ is the number of fragments obtained from the filter and l_o is twice the size of the input-buffer $l_o = 2 \cdot L$, that is, each subfilter is zero-padded to length l_o. In this case, matrix $\mathbf{S}$ has one row and l_o columns and contains samples of the current input-data buffer and the previous one.

One of the operations of this algorithm refers to an element-wise multiplication in the frequency domain between all the fragments of the filter and the input-data buffer. Matrix $\mathbf{F}$ stays in *global-memory* and matrix $\mathbf{S}$ is moved to *shared-memory* for this operation since input samples are shared for all the element-wise multiplications with the filter fragments (see Fig. 4.13 (b)).

The way to partition the filter and convolve it with an input-data buffer is developed in [52], where the convolution algorithm is detailed. They also implement the algorithm on GPU apparently without using the resources of the *shared-memory* and, they carry out the necessary FFTs on CPU.

4.3.3 GPU data structure for GCCE applications

This section analyzes and describes in detail the implementation of the two schemes on GPU extrapolating to a multichannel system. In the case of a GCCE application, tridimensional structures are used. The implementations are generalized for any value of sources M and loudspeakers N. In order to make the configuration and the implementation on GPU more understandable, the figures presented throughout this section illustrate a multichannel application with M=4 sources, N=2 loudspeakers and, therefore, C_{tot}=8 different filters. Thus, the output signals in the two loudspeakers are:

$$\begin{aligned} y_0 &= f_{00} * x_0 + f_{10} * x_1 + f_{20} * x_2 + f_{30} * x_3, \\ y_1 &= f_{01} * x_0 + f_{11} * x_1 + f_{21} * x_2 + f_{31} * x_3. \end{aligned} \tag{4.7}$$

As in Section 4.3.2, we distinguish two schemes, but now the fragmenta-

tion will be carried out in every input-data buffer (Scheme 1: Fragmentation of multiple input-data buffers) and every filter (Scheme 2: Fragmentation of multiple filters).

Scheme 1: Fragmentation of multiple input-data buffers.

Matrix **S** turns into a tridimensional matrix whose dimensions will be ($P \times l_o \times M$) for multichannel convolution, where overlap-save blocks from the M input-data buffers are located in different layers, see Fig. 4.14 (a). The matrix **F** also turns into a tridimensional structure whose dimensions are ($1 \times N \cdot l_o \times M$). Filters f_{00} and f_{01} are placed on the same layer because their respective operations refer to different outputs. In contrast, filters f_{00}, f_{10}, f_{20}, and f_{30} are located on different layers because they take part in calculating the output y_0. The same occurs with the output y_1. This can be checked in (4.7) and Fig. 4.14 (a).

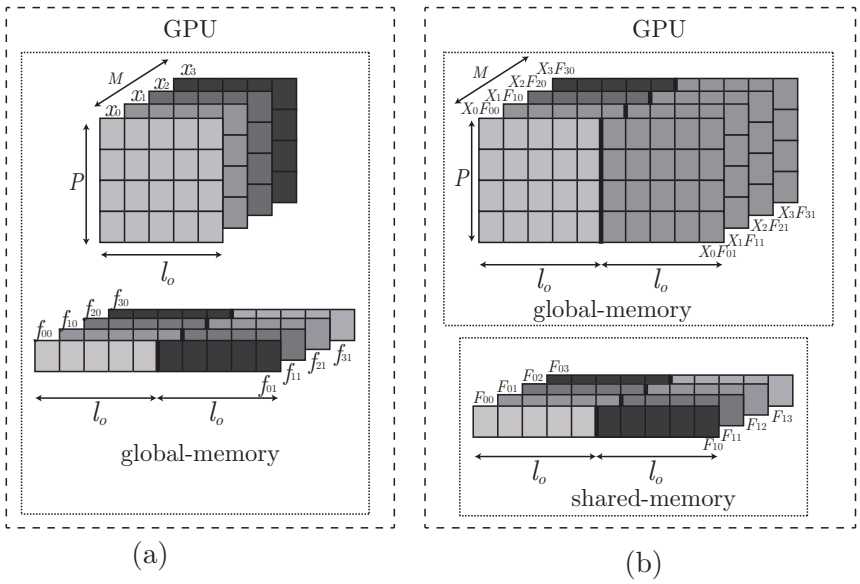


Figure 4.14. (a) shows matrices **S** and **F** in GPU. Then, frequency-domain transform and element-wise multiplication are applied; (b) shows that the resulting matrix is stored at the same memory position.

Following the overlap-save technique, the FFT of each overlap-save block in matrix **S** must be carried out (X_m represents samples of x_m in

the frequency domain). The same occurs with the filters f_{mn}, which are transformed into F_{mn} in the frequency domain. There are some recent publications about FFT in GPU as in [101], but the NVIDIA FFT library, CUFFT [87], is used for our application. This GPU library allows multiple one-dimesional FFTs to be obtained simultaneously. Thus, $M \cdot P$ FFTs are calculated for each new input-data buffer while C_{tot} FFTs of filters will be executed (one for each filter) only once at the beginning of the algorithm. Two different CUDA kernels are launched to carry out the rest of the algorithm.

CUDA Kernel 4

Once the data are in the frequency-domain, the placement of matrix **F** in the *shared-memory* allows each overlap-save block to be simultaneously element-wise multiplied by its corresponding filter. This CUDA kernel launches $M \cdot P \cdot N \cdot l_o$ threads. Each thread will only make a complex multiplication between a value of matrix **S** and its corresponding complex-component in matrix **F**. Each component of the filter is accessed P times, while each component of an overlap-save block is accessed N times. The results of the multiplications are stored at different memory positions, which are denoted as matrix $\mathbf{S}_{res}$. Note that the `BlockDim.y=BlockDim.z=1`, and the number of blocks launched in this CUDA kernel is $\frac{l_o \cdot N}{128} \times P \times M$, being `BlockDim.x=128`.

CUDA Kernel 4 Element-wise multiplication with Matrix F to *shared-memory*

Input: **F**, **S**, l_o, N

Output: $\mathbf{S}_{res}$

```
1: __shared__ Complex Fs[128];
2: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
3: int Row = BlockIdx.y * BlockDim.y + ThreadIdx.y;
4: int High = BlockIdx.z * BlockDim.z + ThreadIdx.z;
5:  // Global Identification -> Idx
6: int idx = High*P*N*lo +  Row *lo*N + Col;
7:  // Identification for accessing matrix S
8: int idxMod = idx & (lo-1);
9: idxMod = idxMod + Row*lo +  High*P*N*lo;
10:  // Filters to shared memory;
11: if(ThreadIdx.y== 0)
12:    Fs[ThreadIdx.x] =  F[Col + N*lo*High];
```

13: **end if**
14: `__syncthreads();`
15: `// Complex Multiplication between two complex elements`
16: $\mathbf{S}_{res}$[idx] = ComplexMult($\mathbf{S}$[idxMod],$\mathbf{F}_s$[ThreadIdx.x]);
17: `// Scaling the multiplication`
18: $\mathbf{S}_{res}$[idx] = ComplexScale($\mathbf{S}_{res}$[idx], $1/l_o$);

CUDA Kernel 5

The next step consists of adding up all the layers in order to calculate the outputs in the frequency domain Y_n according to the multichannel system in (4.7). In this case, we use a bidimensional grid configuration where a thread processes an output sample. Thus, $P \cdot N \cdot l_o$ threads are required to sum the layers. Each thread will make M complex sums reducing all the layers to one layer (see Fig. 4.15). The result matrix $\mathbf{S}_{res}$ has now two dimensions: $(P \times N \cdot l_o)$. Note that the number of blocks launched in this CUDA kernel is $\frac{l_o \cdot N}{128} \times P$, being `BlockDim.x=128`, as in CUDA kernel 4.

CUDA Kernel 5 Tridimensional Element-wise Sum
Input: $\mathbf{S}_{res}$,M,N,l_o
Output: $\mathbf{S}_{res}$
1: `int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;`
2: `int Row = BlockIdx.y * BlockDim.y + ThreadIdx.y;`
3: `// Global Identification -> Idx`
4: `int idx = Row*`l_o`*`N `+ Col;`
5: `// Complex Sum between two complex elements`
6: **for** $k = 1, \ldots, M-1$ **do**
7: $\mathbf{S}_{res}$[idx] = ComplexSum($\mathbf{S}_{res}$[idx], $\mathbf{S}_{res}$[idx + l_o`*`k`*`P`*`N]);
8: **end for**

Finally, the CUFFT library is applied again $N \cdot P$ times in order to obtain iFFT from all the output overlap-save blocks of all the outputs y_n. All the overlap-save blocks in the time domain are then sent back to the CPU to be reproduced.

Scheme 2: Fragmentation of multiple filters.

In this scenario, the size of the input-data buffers is much smaller than the size of filters. Hence, the filters f_{mn} are split into P fragments (as

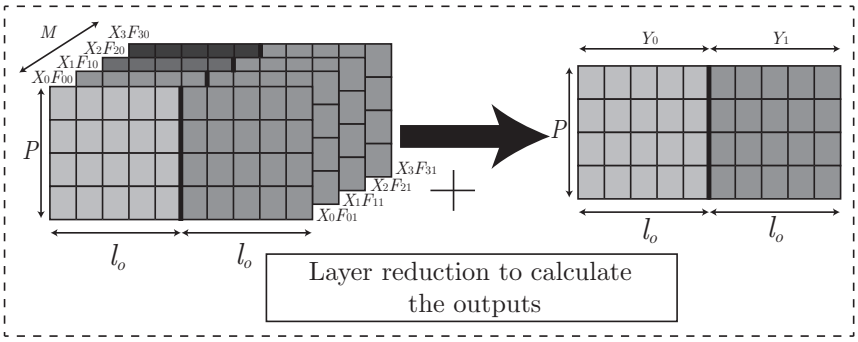


Figure 4.15. Addition of all the planes to obtain the different outputs (in this case, Y_0 and Y_1).

in Scheme 2 of section 3), each of which has the same size as the input buffers. As in Scheme 1, matrix **F** turns into a tridimensional matrix with dimensions ($P \times N \cdot l_o \times M$). All the fragments that belong to the same filter are placed within the same layer. The filters used for calculating the same output y_n remain in different layers. Matrix **S** configures another tridimensional structure with dimensions ($1 \times l_o \times M$). Figure 4.16 (a) clarifies the setting of data on GPU.

In this scheme, M FFTs are carried out every time the input-data buffers are transferred to GPU. At the beginning of the processing, $N \cdot P$ FFTs are executed in matrix **F** only once. As in the previous scheme, two different CUDA kernels are executed on GPU.

CUDA Kernel 6

Once the data are in the frequency-domain, the placement of matrix **S** in the *shared-memory* allows every fragment in matrix **F** to be simultaneously element-wise multiplied by its corresponding input-data buffer, thus obtaining a resulting matrix **R** with the same structure as matrix **F**. If the processing in GPU is carried out on the v-th input-data buffer, the resulting matrix **R** is called $\mathbf{R}_\mathrm{v}$ (see Fig. 4.16 (b)). This matrix $\mathbf{R}_\mathrm{v}$ must be accumulated with the previous one, $\mathbf{R}_{\mathrm{v}-1}$, which was obtained from the $(v-1)$-th input-data buffer. However, this element-wise sum is not straightforwardly carried out but depends on a parameter that we call *PointOut* $\in [0, P-1]$. This parameter is a modular counter that increases incrementally with each new input-data buffer. It indicates that the row 0 of $\mathbf{R}_\mathrm{v}$

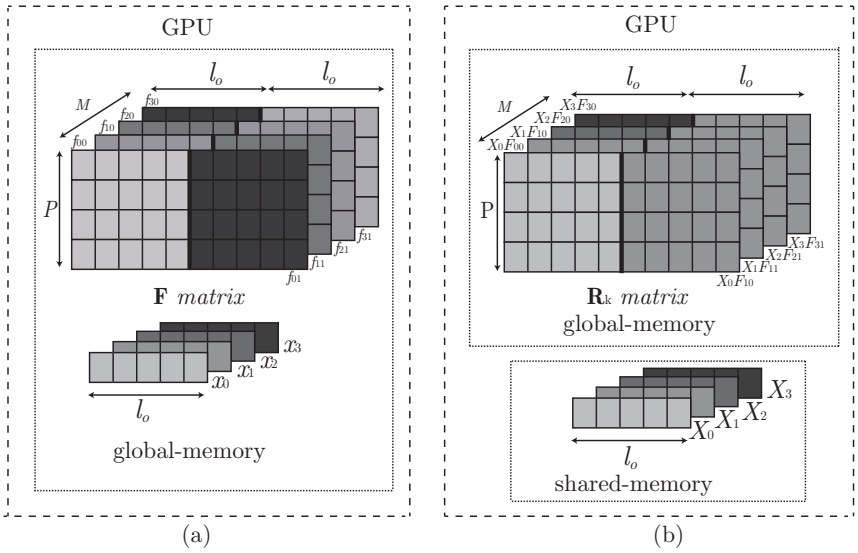


Figure 4.16. (a) shows matrices **S** and **F** in GPU. Then, frequency-domain transform and element-wise multiplication are applied; (b) shows that the resulting matrix $\mathbf{R}_v$ is stored in a different memory position.

must be element-wise sum with the row *PointOut* of $\mathbf{R}_{v-1}$, the row 1 of $\mathbf{R}_v$ must be element-wise sum with the row *PointOut+1* of $\mathbf{R}_{v-1}$, and so on (Fig. 4.17). For these operations, $M \cdot P \cdot N \cdot L$ threads are used. Each thread performs a complex multiplication between a value of matrix **F** and its corresponding complex component in matrix **S**, and then accumulates the result with the corresponding value in $\mathbf{R}_{v-1}$. As a thread per sample of every fragment is used, the same grid configuration as CUDA kernel 4 from Scheme 1 is applied.

CUDA Kernel 6 Element-wise multiplication with Matrix S to *shared-memory*

Input: **F**, **S**,l_o,N,P,*PointOut*

Output: $\mathbf{R}_v$

```
1: __shared__ Complex S_s[128];
2: Complex  cReg;
3: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
4: int Row = BlockIdx.y * BlockDim.y + ThreadIdx.y;
5: int High = BlockIdx.z * BlockDim.z + ThreadIdx.z;
```

```
 6:  // Global Identification -> Idx
 7: int idx = High*P*N*l_o +  Row *l_o*N + Col;
 8:  // Identification for accessing matrix R_v
 9: int ColMod = Col & (l_o-1);
10: int RowMod = (Row + PointOut)%P;
11: idxMod = idxMod + Row*l_o +  High*P*N*l_o;
12: int idx_gl_RowMod = Col + RowMod*N*l_o + High*N*l_o*P;
13:  // Matrix S to shared memory;
14: if(ThreadIdx.y== 0)
15:    S_s[ThreadIdx.x] =  S[ColMod + l_o*High];
16: end if
17:  __syncthreads();
18:  // Complex Multiplication between two complex elements
19: cReg = ComplexMult(F[idx],S_s[ThreadIdx.x]);
20: cReg = ComplexScale(cReg,1/l_o);
21: R_v[idx_gl_RowMod]=ComplexAdd(R_v[idx_gl_RowMod],cReg);
```

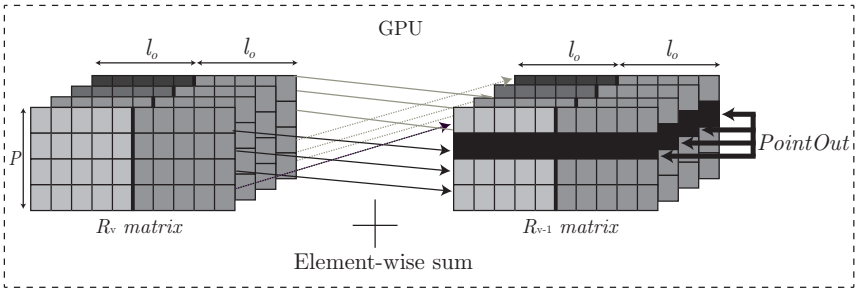


Figure 4.17. Element-wise sum between $\mathbf{R}_{\mathrm{v}}$ and $\mathbf{R}_{\mathrm{v}-1}$. Row 0 of $\mathbf{R}_{\mathrm{v}}$ is element-wise sum with the row indicated by *PointOut*; row 1 is element-wise sum with the row indicated by *PointOut+1*; and so on.

CUDA Kernel 7

The next step consists of adding up all the layers; however, in this case, only the values on the row indicated by *PointOut* are used. The resulting vector is copied to other memory positions denoted as **OV**. This vector represents the output-data buffers in the frequency-domain. After the iFFTs are applied, the outputs y_n are obtained, and are sent back to the CPU. The matrix $\mathbf{R}_{\mathrm{v}}$ takes the role of matrix $\mathbf{R}_{\mathrm{v}-1}$ for the next input-data buffer.

Nevertheless, to take this role, the row indicated by *PointOut* will be set to 0 and the parameter *PointOut* will be increased incrementally after the copy to **OV** from the matrix $\mathbf{R}_{\mathrm{v}}$. Figure 4.18 reflects all these operations. This CUDA kernel launches $N \cdot l_o$ threads. Each thread sums M complex values, saves the result in **OV**, and sets its corresponding elements to 0 on all layers of the row marked by *PointOut*. In this case, a unidimensional grid configuration is used where there is one thread for each processing sample. Note that the number of blocks launched in this CUDA kernel is $\frac{l_o \cdot N}{128} \times 1 \times 1$, being `BlockDim.x=128`, as previous CUDA kernels.

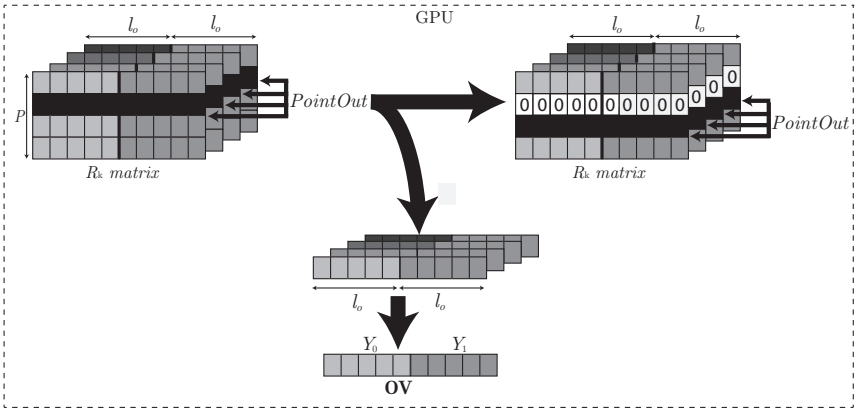


Figure 4.18. Copy of the row indicated by *PointOut* in $\mathbf{R}_{\mathrm{v}}$ to **OV**, which is later set to 0. *PointOut* increases incrementally and gets prepared for the next input-data buffer.

CUDA Kernel 7 Special Tridimensional Element-wise Sum

Input: $\mathbf{R}_{\mathrm{v}}$, *PointOut*,M,N,l_o,P

Output: **OV**

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int iCte = PointOut*l_o*N;
3:  // Complex Multiplication between two complex elements
4: for k = 0,...,M-1 do
5:    OV[Col] = ComplexSum(OV[Col], R_v[Col + iCte + k*N*l_o*P]);
6:    R_v[Col + iCte + k*N*l_o*P] = 0;  // (Zero Complex)
7: end for
8:  // PointOut is incremented at the main program;
```

4.3.4 Performance and Results

Two different schemes have been presented depending on the size of both the input-data buffer and the filter (L and l_f, respectively). When the input-data buffer is much larger than the filter size, it is fragmented into different overlap-save blocks (Scheme 1). On the other hand, when the input-data buffer is much smaller than the filter size, the filter is the one that is fragmented (Scheme 2).

This second scheme aims to reduce the latency time by reducing the time of response of the system t_{proc}. Note that t_{proc} contains not only the execution time of the CUDA kernels but also the data transfers between GPU and CPU and all the data overhead in order to carry out a real-time application. The objective that is derived from scheme 2 is to know the maximum number of filters that can be managed by a GPU in a GCCE application for different environments. To support these results, it has been measured two different audio parameters: Latency and Throughput.

Regarding scheme 1, we have proposed a GPU implementation that profits from the computational resources of the GPU by distributing the audio samples among P overlap-save blocks. The results from this scheme look for the most efficient number of blocks P in order to exploit GPU parallelism.

For performing the implementations, we used the *Eleanorrigby* machine that is composed of Nvidia TESLA C2070 GPU with 2.0 CUDA capability. Because there are variables that are used by multiple threads, the selected configuration for L1 cache is 16 kB and 48 kB for *shared-memory*. The CUDA toolkit and SDK version is 4.0. The following subsection describes the computed tests for both schemes.

Scheme 1 : $L \gg l_f$

In this case, the input-data buffer is divided into P overlap-save blocks. Among the different configurations tested in a multichannel application, we selected the one that fixes a $t_{\text{buff}} = 92.86$ ms (4096 samples) and a filter size of 129 coefficients. Figure 4.20 shows t_{proc} in multichannel applications with 2, 4, 32, and 64 loudspeakers. In each implementation, a sweep of number of sources was carried out dividing the input-data buffer into a different number of overlap-save blocks (2, 4, 8, 16, and 32). The best performances, which exploit the maximum GPU resources, are obtained

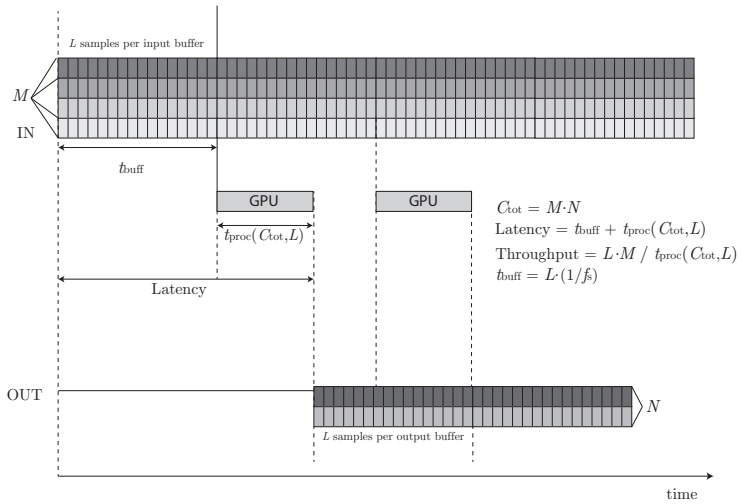


Figure 4.19. Important parameters in a real-time multichannel application, with M=4, N=2 and C_{tot}=8.

when the input-data buffer is divided into P=4 overlap-save blocks, as Fig. 4.20 shows in (a), (b), (c), and (d).

Scheme 2 : $L \ll l_f$

The most significant test in the presented work revolves around the maximum number of filters C_{tot} that a GPU, given a specific latency time t_{buff}, can manage in a real-time multichannel GCCE. Among the different tests, we detail the time t_{proc} used by the GPU to process a system configured with a different number of sources M combined with a specific number of loudspeakers N (2, 4, 8, 16, 32, 64, and 96) using filters whose size is l_f=2048 coefficients.

The first test was done setting an input-data buffer size L of 128 samples, with $t_{\text{buff}} = 2.9$ ms. The results in Fig. 4.21 (a) show that the obtained t_{proc} times increase linearly as the number of sources increases. Focusing on real-time applications, the maximum number of filters of this size that this implementation can manage is 1408 filters, which is obtained when M=22 and N=64.

The system can also carry out applications involving more filters, but they would not satisfy the real-time condition $t_{\text{proc}} < t_{\text{buff}}$. Therefore, the

configurations below the dotted line (t_{buff}) in Fig. 4.21 (a),(b),(c), and (d) allow real-time applications to be carried out. Figure 4.21 (a) also shows the maximum number of filters that can be achieved with a specific number of loudspeakers, 1344 filters for 96 sources and 32 loudspeakers among others. In any case, every configuration would work for off-line processing; even the ones that are above the dotted line. For example, as Fig. 4.21 (a) shows, for a M=38, N=64, C_{tot}=2432, the processing time t_{proc} is 4.830 ms. This means that using M=38 audio wav file sources of mono systems of 2 MB each (a mono audio wav file is composed of audio samples of `short int`, 2 bytes), the time spent to process 38 audio wav file sources with 64 loudspeakers using buffers of 128 samples would be 158.27 s. This time could be used as a processing reference for other kinds of applications that do not require real-time.

If we increase the number of input-data buffer samples to 256, as Fig. 4.21 (b) shows, the maximum number of filters increases to 3136, obtained when M=98 and N=32. By doubling the input-buffer size, the limit is achieved with 6336 filters (see Fig. 4.21 (c)).

Figure 4.21 (d) shows the maximum number of filters with input-data buffer sizes of 1024. The maximum number of filters is obtained using N=96, which achieves up to 12480 filters in a GCCE.

Figure 4.19 shows a temporal evolution of a real-time multichannel application, with M=4, N=2 and C_{tot}=8. Table 4.2 shows the latencies and throughputs from the maximum number of configurations. The latencies are calculated as $t_{\text{proc}} + t_{\text{buff}}$, where $t_{\text{buff}}=\frac{L}{44.1}$ ms. It can be observed that the latency values are approximately double the t_{proc}. Generally, the greater the number of sources M, the greater t_{proc}, and the greater throughput, whose values revolve around 1 and 10 million samples processed per second. When the input-buffer is 128 samples, maximum throughput achieves $9.977 \cdot 10^5$ samples/s; when the input-buffer is 2048 samples, maximum throughput achieves $8.185 \cdot 10^6$ samples/s.

4.3.5 Conclusions

The algorithm implemented on GPU responds to a massive convolution or a generalized crosstalk cancellation and equalization. The placement of data inside the GPU changes depending on the size of the input-data buffer and the size of the filters. When the size of filters is much larger than

Table 4.2. Latencies and Throughputs from the maximum number of C_{tot} that are obtained under real-time conditions.

Input-data buffer size	C_{tot}	t_{proc} (ms)	Latency (ms)	Throughput (input samples/s)
128	1408	2.822	5.724	$9.977 \cdot 10^5$
512	6336	11.551	23.159	$2.925 \cdot 10^6$
256	3136	5.625	11.429	$4.459 \cdot 10^6$
1024	12096	22.531	45.745	$8.271 \cdot 10^6$
2048	17472	45.537	91.966	$8.185 \cdot 10^6$

the size of input-data buffer, the filters are fragmented and the parallelism is exploited by the element-wise multiplication of the fragments with the input-data buffer. The evaluated tests show that, with only an input-data buffer of 128 samples, it is possible to achieve up to real-time multichannel applications with 1408 filters of 2048 coefficients. This number gets larger as the input-data buffer increases. Otherwise, when the size of the filters are much smaller than the size of the input-data buffers, these buffers are fragmented into ovelap-save blocks. In this case, parallelism is exploited by the element-wise multiplication of the overlap-save blocks with the filter in the frequency domain. The figures shown for this test indicate that when the input-data buffers are fragmented into four overlap-save blocks, minimun t_{proc} time is achieved.

The selection of the correct placement of data in the different GPU memories is crucial to achieving good performance. This chapter describes an efficient way to do it by exploiting parallelism and taking advantage of *shared-memory*. As a result of the good performances offered by these implementations on GPU, it has been demonstrated that a GPU can be used as a co-processor. This co-processor carries out audio processing tasks, even in a real-time environment, freeing up CPU resources in the same way the GPU is currently used for graphic tasks. More details of the presented work can be found in [102], and in [103].

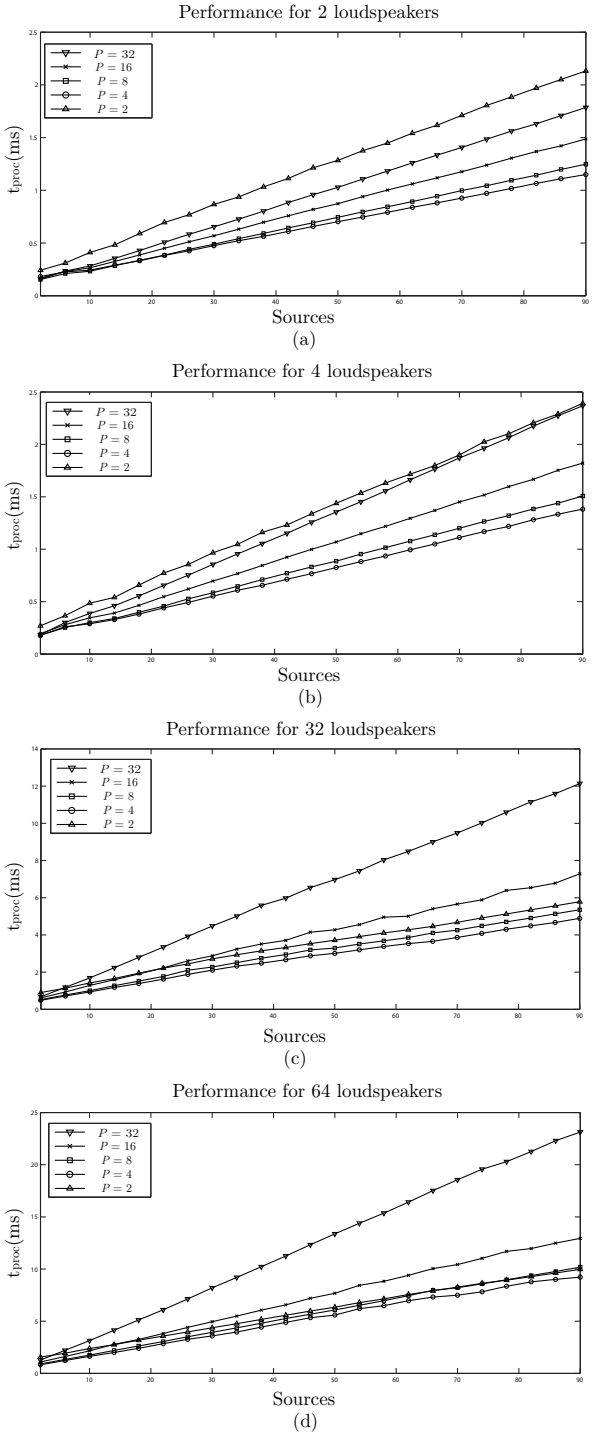


Figure 4.20. t_{proc} in a multichannel application fragmenting the input-buffer in different overlap-save blocks: (a) for 2 loudspeakers; (b) for 4 loudspeakers; (c) for 32 loudspeakers; and (d) for 64 loudspeakers.

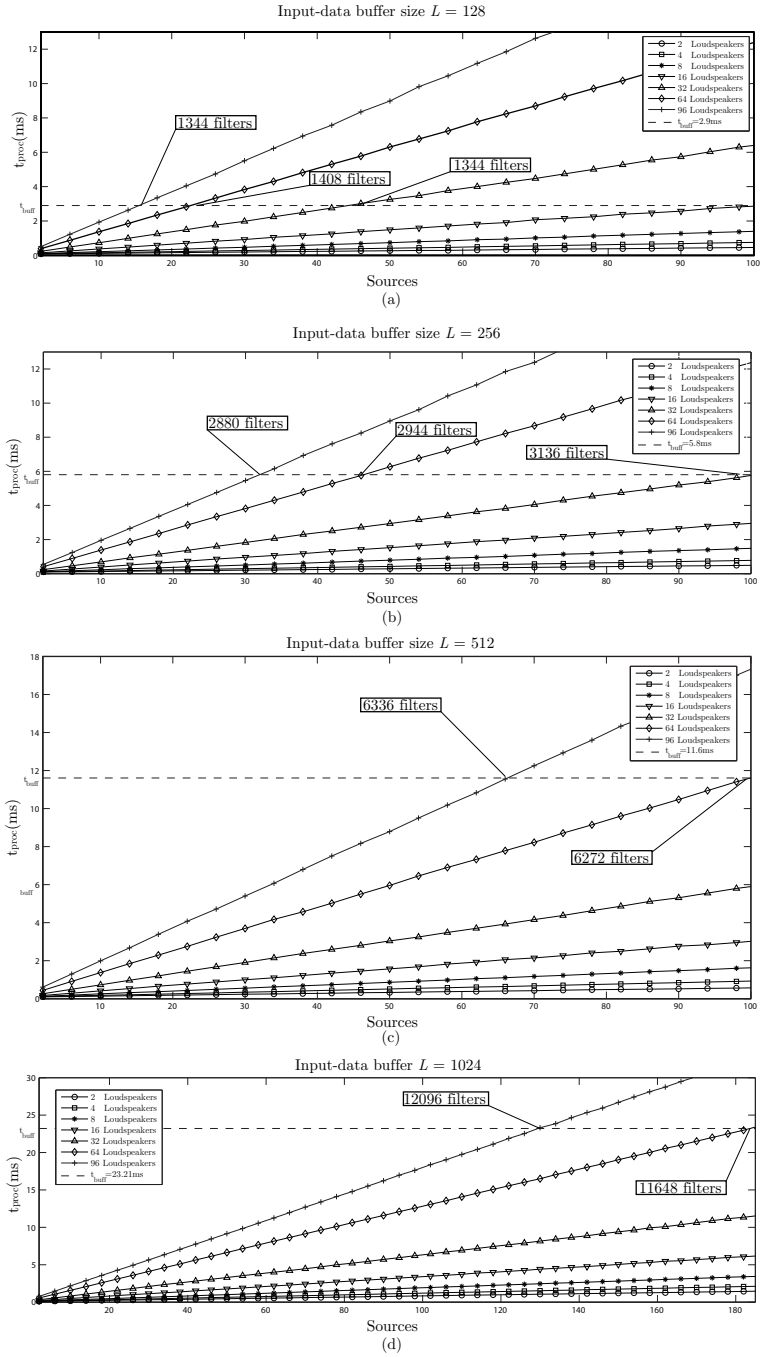


Figure 4.21. t_{proc} used by GPU in a GCCE for different values of sources M and loudspeakers N, using a sampling frequency of f_s=44.1 kHz with: t_{buff}=2.9 ms in (a), t_{buff}=5.8 ms in (b), t_{buff}=11.6 ms in (c), and t_{buff}=23.2 ms in (d).

Headphone-based spatial sound system 5

Headphone-based spatial sound system 5

Multichannel audio signal processing has undergone major development in recent years. The incorporation of spatial information into an immersive audiovisual virtual environment or into video games provides a better sense of "presence" to applications. In a binaural system, spatial sound consists of reproducing audio signals with spatial cues (spatial information embedded in the sound) through headphones. This spatial information allows the listener to identify the virtual positions of the sources corresponding to different sounds. Headphone-based spatial sound is obtained by filtering different sound sources through a collection of special filters (whose frequency responses are called *Head-Related Transfer Functions*) prior to rendering them through headphones. These filters belong to a database composed by a limited number of spatial fixed position. A complete audio application that can render multiple sound sources in any position of the space and virtualize movements of sound sources in real time demands high computing needs. This chapter presents the design of a headphone-based multisource spatial audio application whose main feature is that all required processing is carried out on the GPU. To this end, two solutions have been approached in order to synthesize sound sources in spatial positions that are not included in the database, and to virtualize sound sources movements between different spatial positions.

5.1 Introduction

The growing need to incorporate new sound effects and to improve the listening experience have increased the development of multichannel audio applications [3]. A spatial audio system based on headphones allows a listener to perceive the virtual position of a sound source [11]. These effects are obtained by filtering sound samples through a collection of special filters whose coefficients shape the sound with spatial information. In the frequency domain, these filters are known as *Head-Related Transfer Functions* (HRTFs). The response of HRTFs describes how a sound wave is affected by properties of the body shape of the individual (i.e., pinna, head, shoulders, neck, and torso) before the sound reaches the listeners eardrum [9]. Each pair of HRTF filters is related to a specific virtual position. A set of HRTFs of different spatial fixed positions configure a HRTF database. When multiple sound sources in different spatial positions move around the scene, fantastic audio effects that provide more realism to the scene are achieved. These spatial sounds are usually added to video games, video conference systems, movies, music performances, etc. However, if a CPU processor were used to calculate these tasks, the CPU processor would be overloaded and the whole application would slow down. When this happens, spatial sound information is usually avoided and, unfortunately, is not added to the applications. This resource problem can be solved if these computational tasks are carried out by Graphics Processing Units (GPUs).

In order to develop a complete multisource spatial application, it is necessary to be able to render sound sources at any position of the space and virtualize movements of the sound sources. We deal with these two common situations by taking maximum profit of the computational resources of GPUs: GPU capacity for executing multiple convolutions concurrently. To this end, we present interpolation and switching techniques that are based on weighting and combining different convolutions that are computed simultaneously. These techniques allow us to synthesize sounds in virtual positions that do not belong to the collection of the filters, and to virtualize sound sources movements. To support the adopted solutions to these problems, subjective and objective analyses are also presented for both techniques.

5.2 Processing Head-Related Transfer functions

Head-Related Transfer functions represent the frequency response of *head-related impulse response* (HRIR) filters, which are in the time domain. As stated in Section 3.2, there are multiple database of HRTFs or HRIRs on internet such as, [58] or [59]. In our case, for the application described throughout this section, we use the HRIR measures from G. Vandernoot and F. Lienhart [58]. These measures belong to a tall male with short hair. The values of the filters can be found under measures IRC_1007 in the website in [58].

This HRIR database adds spatial information from specific different positions in the space to the audio wave. Two filters specify each virtual position since there is a filter for each ear. Figure 5.1 shows a sound source, for instance a piano that is located in the virtual position (θ, ϕ, r), where ϕ represents the elevation coordinate, ϕ $\in$[-90 °, +90 °], θ represents the azimuth coordinate, θ $\in$[0 °, +360 °], and r is the distance between the virtual position and the user, r $\in$[0, ∞]. The spatial effect is achieved by convolving natural monophonic sounds that are recorded in an anechoic environment with the pair of filters that corresponds to the virtual positions and reproducing them through head-phones.

In a real-time processing system, the audio card provides L audio samples of every sound source with a rate of $\frac{L}{f_s}$ sec (f_s represents the sampling frequency). The convolution is carried out by using blocks of samples of size $2L$ with a 50% overlap. Thus, for the audio processing, we define $\mathbf{x}_{\text{buff}}$ as an input-data buffer that is composed of $l_o = 2L$ audio samples from sound source x.

The impulse responses HRIRs corresponding to position (θ, ϕ, r) in the time domain are denoted as $\mathbf{h}_{\text{r}}(\theta, \phi, r)$ and $\mathbf{h}_{\text{l}}(\theta, \phi, r)$ for the right and left ear, respectively. There are as many input-data buffers as number of sound sources in the application. Thus, considering a system that is composed of M sources, the input-data buffer $\mathbf{x}_{\text{buff}_m}$ represents the buffer of the samples of source m where $m \in [0, M-1]$. The output-data buffers ($\mathbf{y}_{\text{buff}l}$ and $\mathbf{y}_{\text{buff}r}$) are calculated as shown by (5.1) in the time domain, where * denotes the convolution operation.

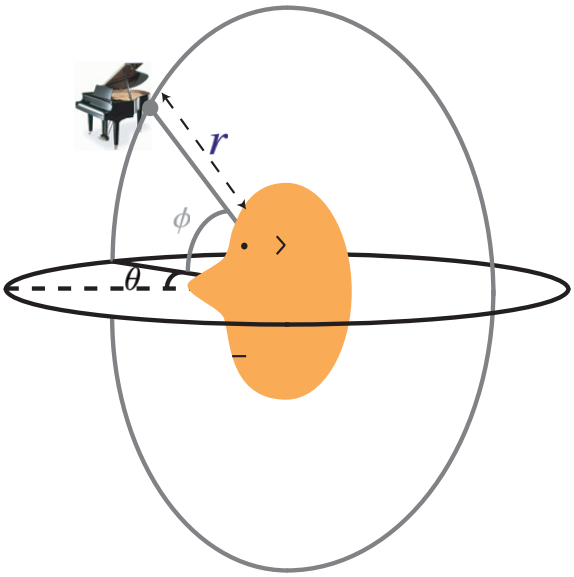


Figure 5.1. The HRIR filtering allows a person to perceive a piano sound as if it were located in a virtual position in the space given by the coordinates (θ, ϕ, r).

$$\mathbf{y}_{\mathrm{buff}l} = \sum_{m=0}^{M-1} (\mathbf{h}_{\mathrm{l}}(\theta_m, \phi_m, r_m) * \mathbf{x}_{\mathrm{buff}m}), \tag{5.1}$$
$$\mathbf{y}_{\mathrm{buff}r} = \sum_{m=0}^{M-1} (\mathbf{h}_{\mathrm{r}}(\theta_m, \phi_m, r_m) * \mathbf{x}_{\mathrm{buff}m}).$$

This HRIR database has an azimuth resolution and an elevation resolution that is denoted by $\Delta\theta$ and $\Delta\phi$. Resolutions $\Delta\theta$ and $\Delta\phi$ represent the minimum separation in degrees between two positions of the database in azimuth and in elevation, respectively. For our HRIR database, the resolution in both the azimuth and the elevation is 15° ($\Delta\theta = 15^\circ$ and $\Delta\phi = 15^\circ$). This database indicates that all HRIR measures were carried out to a distance r_0 of 1.95 m from the center of the head, and that all of the HRIR filters have been windowed to a length of L=512 coefficients.

Thus, as it was described, the number of filters in databases limits the virtual positions to render. One important aspect in this context is the

synthesis of sound in virtual positions that do not belong to the collection of the filters. Another particular situation occurs when the sound moves, which in practice means to filter through another HRIR. If the switch between HRIRs is not properly carried out, multiple audio clipping effects could be generated. To solve both situations, two audio techniques are designed taking into account the GPU capacity of carrying out multiple convolutions concurrently: 1) an interpolation technique that allows us to interpolate any position of the space, and 2) a switching technique that allows us to switch properly between positions in the space.

The following sections describe the two designed techniques that allow us, by using the massive computational GPU resources, to render sound sources in any position of the space and virtualize any movement of them. We also present subjective and objective analyses of both techniques in order to verify that both of them meet the requirements of this headphone-based spatial application.

5.3 Switching technique

This technique is used mainly for the virtualization of source movements, which is carried out in this application by varying smoothly the virtual positions of sound sources over time. For example, let us suppose the sound source x_m moves from one position, which we call the old position $(\theta_{old}, \phi_{old}, r_{old})$, to a new position $(\theta_{new}, \phi_{new}, r_{new})$. In practice, this means to switch the rendering from the old position to the new position. However, this switching could produce multiple audible clipping effects if it is not properly executed. The switching technique that we employ to reduce the possible artifacts is based on [104]. They suggested carrying out a fading, which is a gradual increase in the sound filtered by the new position while the sound filtered by the old position decreases in the same way. To this end, the current input-data buffer $\mathbf{x}_{\mathrm{buff}_m}$ must be convolved with the filters of the old positions, $\mathbf{h}(\theta_{old}, \phi_{old}, r_{old})$, and with the filters of the new position, $\mathbf{h}(\theta_{new}, \phi_{new}, r_{new})$. The fading is carried out by element-wise multiplying the results of the two convolutions by two fading vectors, called $\mathbf{f}$ and $\mathbf{g}$, respectively.

Finally, the output-data buffer $\mathbf{y}_{\mathrm{buff}}$ is obtained by element-wise summing the two previous multiplications. Equation (5.2) shows the fading

operations to execute with the input-data buffer when a switching is produced, where symbol $\otimes$ represents element-wise multiplication.

$$\mathbf{y}_{\text{buff}} = ((\mathbf{h}(\theta_{old}, \phi_{old}, r_{old}) * \mathbf{x}_{\text{buff}m}) \otimes \mathbf{f}) + ((\mathbf{h}(\theta_{new}, \phi_{new}, r_{new}) * \mathbf{x}_{\text{buff}m}) \otimes \mathbf{g}). \tag{5.2}$$

Both $\mathbf{f}$ and $\mathbf{g}$ are complementary fading vectors and therefore must satisfy

$$\mathbf{f}[s] = \mathbf{g}[2L - 1 - s], \tag{5.3}$$

and the following boundary conditions,

$$\begin{aligned} \mathbf{f}[L] = \mathbf{g}[2L - 1] = 1, \\ \mathbf{f}[2L - 1] = \mathbf{g}[L] = 0. \end{aligned} \tag{5.4}$$

where $s \in [L, 2L - 1]$. As there is 50% overlap and fading is applied to current audio buffer, the first L values of both vectors are a constant equal to 0.

$$\mathbf{f}[s] = \mathbf{g}[s] = 0, \tag{5.5}$$

where $s \in [0, L - 1]$.

Table 5.1 presents several possible values for fading vectors $\mathbf{f}[s]$ and $\mathbf{g}[s]$ that fit with the conditions shown in (5.3) and in (5.4). We denote each pair of vectors as indicated in the first column of Table 5.1. It is important to point out that the fifth vector, which we call SIMPLE, represents the option in which a fading does not apply. The change from the old position to the new position for the fading vector SIMPLE consists in changing only the filters. Thus, all of the values of fading vector $\mathbf{f}[s]$ are a constant equal to 0, and all of the values of vector $\mathbf{g}[s]$ are a constant equal to 1.

On the other hand, the fourth vector, FOURIER, comes from a Fourier series expression where not only have the previous boundary conditions been assumed, (5.4), but also, the following sum-square-constant conditions,

$$\begin{aligned} \mathbf{f}[L + \frac{L}{2}] = \mathbf{g}[L + \frac{L}{2}] = \sqrt{2}/2, \\ \mathbf{f}^2[L + \frac{L}{4}] + \mathbf{g}^2[L + \frac{L}{4}] = 1. \end{aligned} \tag{5.6}$$

Thus, the first four coefficients are used, as it is also shown in [104]. Their values are: $a_0 = \frac{1+\sqrt{2}}{4}$, $a_1 = \frac{1}{4}(1 + \sqrt{\frac{5-2\sqrt{2}}{2}})$, $a_2 = \frac{1-\sqrt{2}}{4}$, and $a_3 = \frac{1}{4}(1 - \sqrt{\frac{5-2\sqrt{2}}{2}})$. This is the reason why summation index varies from 0 to 3 in Table 5.1.

Table 5.1. Fading vectors of $\mathbf{f}[s]$ and $\mathbf{g}[s]$ with $s \in [L, 2L-1]$.

VECTORS	$\mathbf{f}[s]$	$\mathbf{g}[s]$
RAMP	$\frac{s-L}{L-1}$	$1 - \frac{s-L}{L-1}$
SQRT	$\sqrt{\frac{s-L}{L-1}}$	$\sqrt{1 - \frac{s-L}{L-1}}$
TRIG	$\sin(\frac{\pi \cdot (s-L)}{2 \cdot (L-1)})$	$\cos(\frac{\pi \cdot (s-L)}{2 \cdot (L-1)})$
FOURIER	$\sum_{r=0}^{3} a_r \cdot \cos(\frac{r \cdot \pi \cdot (s-L)}{L-1})$	$\sum_{r=0}^{3} a_r \cdot \cos(r \cdot \pi \cdot (1 - \frac{(s-L)}{L-1}))$
SIMPLE	0	1

5.3.1 Evaluation of the switching technique

When a switch in the source virtual position is carried out, non-linear artifacts appear in the signal due to the doppler effect and the arising discontinuity when the filter HRIR is changed. This evaluation is intended to assess if the proposed post-processing carried out by (5.2) helps to reduce the non-linear artifacts. To this end, two kinds of analyses have been carried out: objective and subjective.

The objective analysis focuses on measuring the percentage of the energy that is out of band when a switch between virtual positions is carried out. For this purpose, a signal composed of three representative tones (859.65 Hz, 4298 Hz, and 8596 Hz) was used to determine the behavior of the energy for these tones in the switching interval. This signal has three equally spaced tones that are sufficiently separated in the audible spectrum. The sample frequency f_s used in the test was 44.1 kHz.

To measure the percentage of the energy out of the bands, we calculated the FFT of the L "core samples" shown in Fig. 5.2. Note that Fig. 5.2 is composed by three buffers $\mathbf{y}_{\text{buff}}$ composed of L audio samples. This occurs because the first L samples of the $2L$ initial samples have been already

discarded after the processing. The result of FFT was $\mathbf{Y_{core}}$, which is composed of L samples. After that, we obtained the total energy E_{tot} through

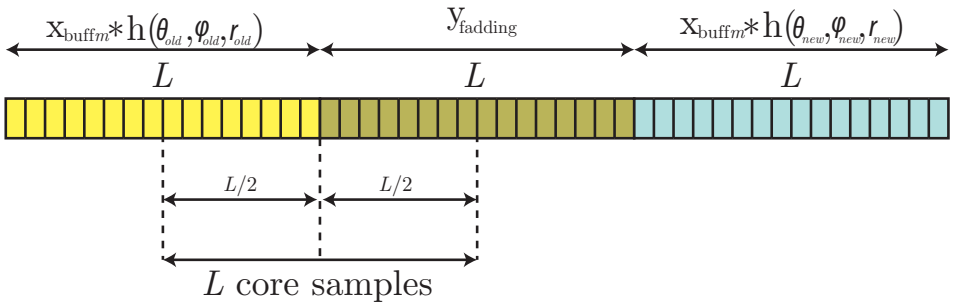


Figure 5.2. L Core samples that are used to obtain the total energy.

$$E_{tot} = \sum_{r=0}^{L} |Y_{core}(r)|^2. \tag{5.7}$$

Then, we calculated the contributions of the energy at the tones mentioned above through

$$E_{tones} = |Y_{core}(f = 859.65)|^2 + |Y_{core}(f = 4298)|^2 + |Y_{core}(f = 8596)|^2. \tag{5.8}$$

Finally, we calculated the percentage of the energy that was out of band as:

$$100 \cdot \frac{E_{tot} - E_{tones}}{E_{tot}}(\%). \tag{5.9}$$

Figure 5.3 represents the percentage of the energy out of the band when the different fading vectors were applied. The represented transitions have an elevation ϕ=0° and go from initial position θ=0° to the position indicated on the horizontal axis. The right side of the figure corresponds to the percentage computed for the right channel, whereas the left side corresponds to the percentage for the left channel. Thus, for example,

when the horizontal axis is 60°, the value in the vertical axis represents the percentage of the energy generated in the transition from 0° to 60° for the left ear, on the left side, and for the right ear, on the right side.

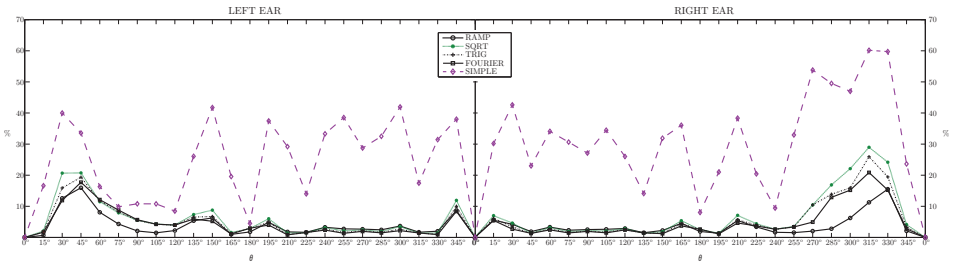


Figure 5.3. Percentage of the energy out of the band when the different fading vectors are applied. The right side of the figure corresponds to the percentage computed for the right channel, whereas the left side corresponds to the percentage for the left channel.

Thus, we can appreciate in Fig. 5.3 that the number of non-linear artifacts decreases significantly when a switching technique is used. The fading vector SIMPLE (absence of fading processing) generates more artifacts in the signal during the switch than any of the other fading vectors, which present very similar results. The SQRT, TRI, and FOURIER fading vectors exhibit a few greater percentage of energy out of the bands than the RAMP fading vector as can be observed more clearly in any transition that is computed from position θ=0° up to position θ=120° for the left channel. The same happens to transitions from position θ=240° up to θ=0° for the right channel. For the rest of the transitions, the difference among SQRT, TRI, FOURIER, and RAMP fading vectors are not meaningful. The performance shown in this analysis indicates that if a switching technique were not employed, the quantity of the artifacts would be remarkably high during the rendering. Thus, we employ this technique for the spatial audio rendering of our application.

The subjective analysis is carried out in order to decide which fading vector is used for the developing application, i.e, which fading vector produces the smoothest switch, taking into account the perception of the users. To verify this, we carried out a test with five sounds. All of the sounds consisted of a musical note. The person used a headphone and the musical note began to be heard at the virtual position (90°, 0°) (the left ear). After a

while, it was moved 15 ° in azimuth and so on until it reached the virtual position (270 °, 0 °) (the right ear). For each sound, the processing applied during the switch corresponded to each fading vector shown in Table 5.1. We used a paired comparison test using a hidden reference paradigm [105] to analyze the most continuous movement among the different configurations. The five sounds were compared in pairs in a test of 10 questions. The audio files used for the tests are available in the section *Melodies used for assessing the different fading vectors of a switching technique between HRIRs* from the web page referenced in [14].

A total of 20 people participated in the listening experiment; their ages were between 23 and 35 years old. The hearing of all of the test subjects was tested using standard audiometry. None of the people had reportable hearing loss that could affect the results. Once the results were collected, the mathematical transformations based on Bradley-Terry model [105] were applied. The result of this post-processing is a percentage of the preference of each sound as Fig. 5.4 shows. Test results indicate that most people felt more natural when the FOURIER fading vector was applied, followed closely by the RAMP fading vector. The TRI fading vector was also very close to the preferences of the RAMP fading vector. The SIMPLE and SQRT fading vectors showed the worst results. Thus, both FOURIER and RAMP are suitable fading vectors for the spatial audio rendering of our application.

5.4 Interpolation technique

The objective of this technique is to synthesize sound sources in virtual positions that do not belong to the used HRIR database. Until now, most of the interpolation techniques used in the literature, such as [106] and [107], reduced the coefficients of HRIR filters using the principal component analysis (PCA) since real-time processing of HRIR filters was computationally expensive. In [106], the authors propose a binaural impulse-response interpolation algorithm based on the solution for the rational minimal state-space interpolation problem and compare it with other interpolation techniques: the bilinear method [108], the DFT method [109], and the spline-function method [110]. The interpolation technique presented herein is based on combining the audio rendering of nearby positions in the HRIR database.

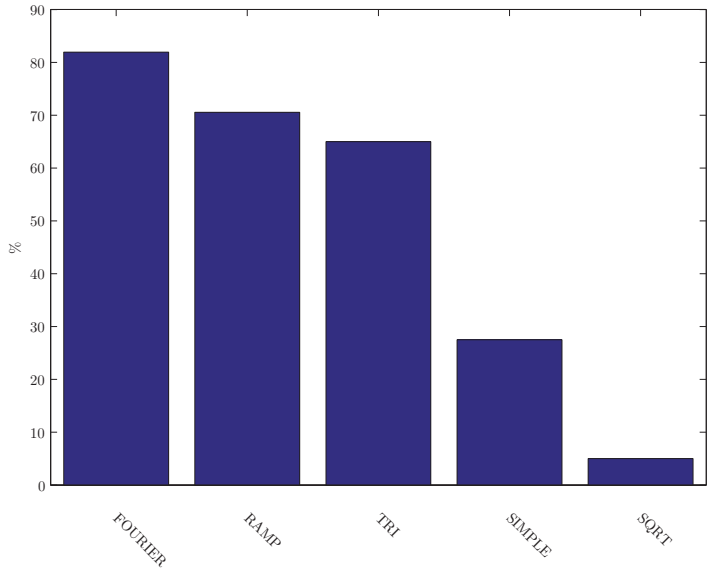


Figure 5.4. Percentage of preference obtained with the paired comparison test when RAMP, SQRT, TRI, FOURIER, and SIMPLE fading vectors were compared.

Unlike [106] and [107], we do not reduce the length of HRIR filters.

A generic position to render is (θ_S, ϕ_S). Focusing on the specific elevation plane ϕ_S, the azimuth position to render θ_S is obtained by combining the rendered sound from the two nearby azimuth positions as shown in Fig. 5.5. Both positions are weighted by w_A and w_B, respectively. These weighted factors are calculated as shown in (5.10):

$$\begin{aligned} w_A &= \frac{\theta_S - \theta_1}{\Delta\theta}, \\ w_B &= \frac{\theta_2 - \theta_S}{\Delta\theta}. \end{aligned} \tag{5.10}$$

Equation (5.11) shows the computation of $\mathbf{y}(\theta_S)$:

$$\mathbf{y}(\theta_S) = w_B \cdot \mathbf{y}(\theta_1) + w_A \cdot \mathbf{y}(\theta_2). \tag{5.11}$$

In the same way, focusing on the specific azimuth plane θ_S, the elevation position to render ϕ_S is also obtained by combining the rendered sound from

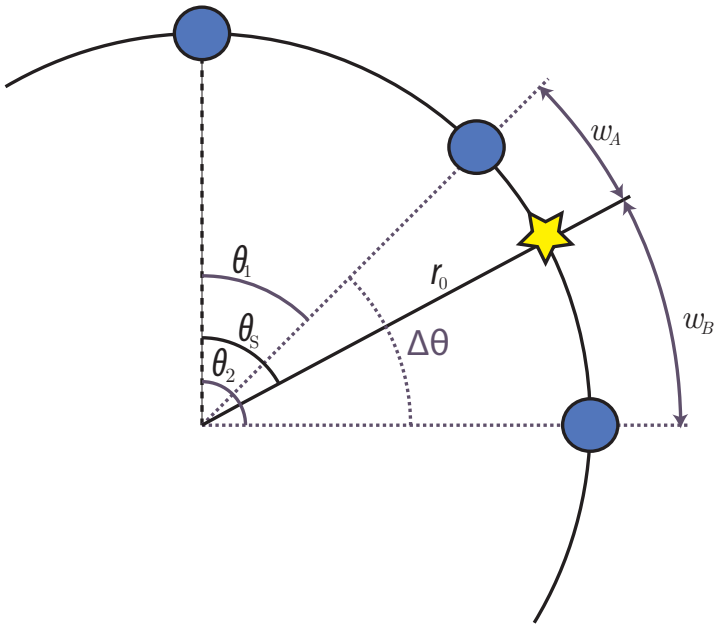


Figure 5.5. The star represents the position to be synthesized in the elevation plane ϕ_S. This position is synthesized by combining the two nearby azimuth positions using the weighted factors w_A and w_B.

the two nearby elevation positions. Equation (5.12) shows the rendered $\mathbf{y}(\phi_S)$, where w_C and w_D are the weighted factors that are computed in (5.13). Figure 5.5 is also available for rendering a position with elevation ϕ_S, by substituting θ for ϕ and the resolution $\Delta\theta$ for $\Delta\phi$.

$$\mathbf{y}(\phi_S) = w_D \cdot \mathbf{y}(\phi_1) + w_C \cdot \mathbf{y}(\phi_2). \tag{5.12}$$

$$\begin{aligned} w_C &= \frac{\phi_S - \phi_1}{\Delta\phi}, \\ w_D &= \frac{\phi_2 - \phi_S}{\Delta\phi}. \end{aligned} \tag{5.13}$$

Combining (5.11) and (5.12), the rendered sound at virtual position

(θ_S, ϕ_S) is as follows,

$$\mathbf{y}(\theta_S, \phi_S) = w_D \cdot (w_B \cdot \mathbf{y}(\theta_1, \phi_1) + w_A \cdot \mathbf{y}(\theta_2, \phi_1)) + w_C \cdot (w_B \cdot \mathbf{y}(\theta_1, \phi_2) + w_A \cdot \mathbf{y}(\theta_2, \phi_2)). \tag{5.14}$$

Another important aspect in spatial sound is related to distance virtualization. This consists in the fact that the rendered sound either gets closer or moves away. The way we carry out this change in distance is to vary the amplitude of the sound and delay it for a number of samples. We perform this effect by element-wise multiplying the output signal in the frequency domain, $\mathbf{Y}(\theta_S, \phi_S, r_0)$, by a complex factor $\mathbf{R}(r)$ as shown in (5.16), where f_s is the sample frequency, v_s is the speed of sound (343.2 m/sec), $l_o = 2L$ is the number of samples of the processing block $\mathbf{x}_{\text{buff}_m}$, and k is the *k-th* sample inside the block.

$$\mathbf{R}(r) = \frac{1}{(1 + \frac{f_s}{v_s} \cdot (r - r_0)^2)} \cdot \exp(-j \cdot 2 \cdot \pi \cdot \frac{f_s}{v_s} \cdot (r - r_0) \cdot \frac{1}{l_o} \cdot [0..k..(l_o - 1)]). \tag{5.15}$$

$$\mathbf{Y}(\theta_S, \phi_S, r_S) = \mathbf{Y}(\theta_S, \phi_S, r_0) \otimes \mathbf{R}(r_S) \tag{5.16}$$

5.4.1 Evaluation of the interpolation technique

In the literature, there are multiple works that look for criteria that objectively measure the HRIR interpolation. Different techniques can be applied depending on the features of the database. In [111], multiple interpolation techniques are enumerated. They indicate that the signal-to-distortion ratio (SDR) can be regarded as a good predictor for estimating the differences in perceptual directions between the stimuli from the measured HRIRs and the stimuli from the approximated HRIRs. The signal-to-distortion ratio (SDR) was computed according to the (5.17), where $y(k)$ comes from the convolution of an input signal with a HRIR filter, and the other output signal $\widehat{y}(k)$ is synthesized by combining the convolutions of the input signal with HRIR filters of the four adjacent positions (two from the azimuth position and two from the elevation position). In order to compare both results, $3 \cdot L$ samples of both configurations were obtained. Variable k denotes the *k-th* sample of signals y and $\widehat{y}$.

$$SDR = 10 \cdot \log \frac{\sum_{k=0}^{3 \cdot L-1} y(k)^2}{\sum_{k=0}^{3 \cdot L-1} (y(k) - \widehat{y}(k))^2}. \tag{5.17}$$

To this end, we used five different input signals. All of them were generated from gaussian white noise of zero mean. Each signal was obtained by low-pass filtering of the generated noise: the first signal up to 250 Hz, the second signal up to 500 Hz, the third signal up to 1000 Hz, the fourth signal up to 2000 Hz and the fifth signal up to 4096 Hz. For instance, an output signal was obtained by convolving an input signal with the HRIR position $(15\,^\circ, 0\,^\circ)$, which is in the HRIR database, and also by using the interpolation technique with the four adjacent positions $(0\,^\circ, +15\,^\circ)$, $(0\,^\circ, -15\,^\circ)$, $(30\,^\circ, +15\,^\circ)$, and $(30\,^\circ, -15\,^\circ)$. As the given example shows, it is important to point out that for synthesizing the signal $\widehat{y}(k)$, the two adjacent positions in the azimuth and the two for elevation are separated at twice the initial resolution ($30\,^\circ$ for this HRIR database. Thus, w_A=w_B=w_C=w_D=0.5).

The positions that are rendered in this application use the adjacent positions that are separated less than $15\,^\circ$ (resolutions of HRIR database). Thus, the results obtained in the comparisons under normal conditions are better than the ones obtained in these experiments. Table 5.2 and Table 5.3 show the SDR values obtained from a selection of different virtual positions for the left and right ear.

Table 5.2. SDR for left/right ear

Left / Right	250 Hz	500 Hz	1000 Hz
(15 °,0 °)	17,35 / 20,17 dB	14,04 / 16,79 dB	11,45 / 12,96 dB
(345 °,0 °)	20,22 / 18,14 dB	16,71 / 14,85 dB	13,27 / 12,51 dB
(0 °,0 °)	18,37 / 18,89 dB	15,04 / 15,63 dB	11,40 / 12,00 dB
(180 °,0 °)	22,14 / 23,01 dB	18,21 / 19,38 dB	15,28 / 16,61 dB

In Table 5.2 and Table 5.3, it is important to note the concordance between the 20.22 dB achieved in the position (345 °, 0 °) for the left ear and the 20.17 dB obtained for the right ear in the position (15 °, 0 °) for

Table 5.3. SDR for left/right ear

Left / Right	2000 Hz	4096 Hz
(15°,0°)	11,30 / 7,01 dB	8,43 / 6,25 dB
(345°,0°)	10,92 / 16,51 dB	8,73 / 7,62 dB
(0°,0°)	10,88 / 9,01 dB	6,91 / 5,63 dB
(180°,0°)	16,63 / 15,97 dB	11,39 / 11,63 dB

the first signal. Moreover, the values obtained in position (0°, 0°) are approximately the same for both the right and the left ear. This ocurrs because of the symmetry of the head. These values are a little different when we compare them with the ones in position (180°, 0°) since the head and the pinnae are not affected in the same way. In any case, it is clear that as the bandwidth of the signal increases, SDR values decrease. Low-frequency signals are better interpolated.

On the other hand, the results shown in Table 5.2 are in agreement with the results shown in [107], in which HRTF interpolation is the hot research topic. Thus, the interpolation technique proposed can be considered to be a suitable interpolation for the spatial audio rendering of our application.

Using the previously described switching technique, we can reduce the non-linear artifacts when a switching in the virtual position is carried out. The question that arises is whether the proposed interpolation technique can be also leveraged to obtain better movement virtualization. The following subjective analysis is intended to assess a sound source trajectory that is rendered in different ways. To this end, we asked people to listen to five different sounds. The person used a headphone and the melody began to be heard at the virtual position $(90°, 0°)$ (the left ear); after a while, it switched to the new virtual position $((90 + j)°, 0°)$, where j takes 1°, 7°, 15°, 30°, and 45° for the first, second, third, fourth, and fifth sound, respectively. The melody switched among the different virtual points until it reached the virtual position $(270°, 0°)$ (the right ear). The switch between positions was performed by using the RAMP fading vector $\mathbf{f}$ and $\mathbf{g}$. The time used by the melody to move from the left ear to the right ear was the same in the five cases. Thus, the time that the melody remained in each virtual position was the maximum for the case that j=45°. The audio files

used for the tests are available in the section *Melodies used for assessing the necesity of a interpolation technique in order to render sound sources movements* from the web page referenced in [14].

As in the evaluation of the switching technique, the same 20 people had to assess by 10 comparisons which configuration most accurately represented the movement of the melody from the left ear to the right ear. Once the results were collected, the mathematical transformations based on Bradley-Terry model [105] were applied again. Figure 5.12 shows the percentage of the preferences. The subjects preferred to listen a melody that changes between close positions. This study demonstrates that not only can the interpolation technique be used for synthesizing virtual positions, but also for virtualizing a sensation of continuous movement of the sound, especially if there is no HRIR database with fine resolutions available.

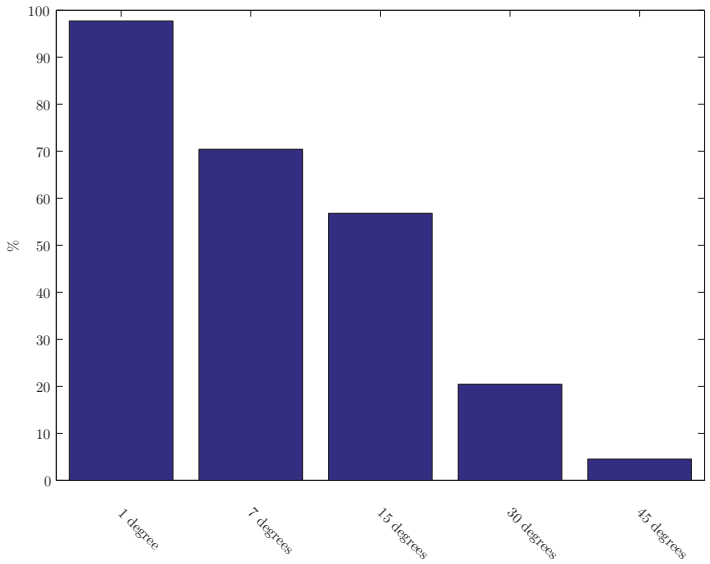


Figure 5.6. The percentage of preference obtained when melodies that switched from virtual positions that were separated by 1, 7, 15, 30, and 45 degrees were compared.

5.5 GPU-based implementation of a head-phone audio application

This section presents the design of a complete head-phone audio application that renders multiple sources simultaneously and is capable of virtualizing their continuous movements. Two main features of this application are: 1) all audio processing is totally carried out on a GPU, and 2) its implementation is totally portable and, thus it adapts to any GPU-device.

One important aspect lies on the data-flow management by the GPUs. Figure 5.7 shows an execution diagram of the head-phone audio application in real time. The application works with three large audio buffers: A-buffer, B-buffer, and C-buffer. A-buffer accumulates the samples from the M sources; B-buffer is on the GPU; and C-buffer is a fixed buffer that accumulates the processed samples that are sent from the GPU to the CPU. A-buffer is composed of all the input-data buffers $\mathbf{x}_{\text{buff}_m}$, whereas C-buffer is composed of the output-data buffers ($\mathbf{y}_{\text{buff}_r}$ and $\mathbf{y}_{\text{buff}_l}$) that are subsequently rendered through the right and left headphone, respectively. Three tasks occur simultaneously in this environment.

1. A-buffer gets filled by the incoming audio-samples.
2. B-buffer is processed on the GPU.
3. Samples from C-buffer are reproduced by the loudspeakers.

When tasks 1) and 2) are over, A-buffer becomes B-buffer and B-buffer becomes A-buffer. In order to achieve the real-time performance of the application, the processing tasks of B-buffer must end before the filling tasks of A-buffer. The time to fill the input buffer (A-buffer or B-buffer) t_{buff} is calculated as $\frac{L}{f_s}$, and it is independent of the number of sources since each source has its own buffer. In contrast, the processing time t_{proc} depends on the number of sources M. Therefore, it is important to develop an efficient implementation on the GPU that achieves maximum M that satisfy $t_{\text{proc}} < t_{\text{buff}}$. As it was analyzed, the processing to be executed on the GPU consists mainly of convolving the input-data buffers with the filters. The technique employed to develop massive convolutions is overlap-save with a 50% overlap in the frequency domain [97]. Before beginning the processing,

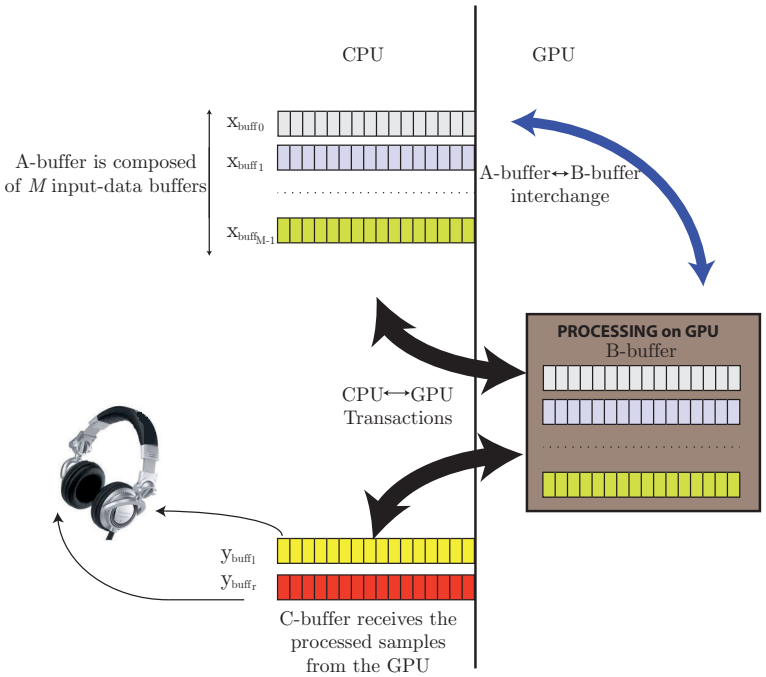


Figure 5.7. GPU diagram of a head-phone audio application.

all the HRIRs are transferred from the CPU to the GPU. All the processing is carried out in the frequency domain. Thus, FFT transformations are carried out on the GPU over all the filters converting HRIRs into HRTFs. In the same way, when A-buffer is received by the GPU, a FFT transformation is applied to each input-data buffer $\mathbf{x}_{\text{buff}_i}$. NVIDIA has its own FFT library [87] which is used for our application, and allows multiple one-dimesional FFTs to be obtained simultaneously. Frequency domain means that convolution is converted into an element-wise multiplication with the corresponding HRTF filters. Executing multiple element-wise multiplications is the key operation between input-data buffers and their corresponding location filters. However, to emulate the movement of the source and the synthesis from any position in space, some post-processing is required. Equation (5.18) represents the equation (5.14) in the frequency domain.

The following implementation presents two levels of parallelism. All sound sources are simultaneously processed, and internally, every sample of the audio buffers is also processed simultaneously.

$$\begin{aligned}\mathbf{Y}(\theta_S, \phi_S, r_S) &= w_D \cdot w_B \cdot \mathbf{Y}(\theta_1, \phi_1, r_S) \\ &+ w_D \cdot w_A \cdot \mathbf{Y}(\theta_2, \phi_1, r_S) \\ &+ w_C \cdot w_B \cdot \mathbf{Y}(\theta_1, \phi_2, r_S) \\ &+ w_A \cdot w_C \cdot \mathbf{Y}(\theta_2, \phi_2, r_S).\end{aligned} \tag{5.18}$$

Equation (5.19) shows the computation of $\mathbf{Y}(\theta_1, \phi_1, r_S)$. The rest of the components of (5.18): $\mathbf{Y}(\theta_2, \phi_1, r_S)$, $\mathbf{Y}(\theta_1, \phi_2, r_S)$, and $\mathbf{Y}(\theta_2, \phi_2, r_S)$ are computed in the same way.

$$\mathbf{Y}(\theta_1, \phi_1, r_S) = \mathbf{X}_{\text{buff}_m} \otimes \mathbf{H}(\theta_1, \phi_1, r_0) \otimes \mathbf{R}(r_S). \tag{5.19}$$

Equation (5.18) exploits the commutative property from multiplication and allows us to efficiently use the architecture of the GPUs. Therefore, for the rendering of each sound in the worst case, eight convolutions are necessary (four convolution per ear). Also, to emulate the movement between two positions, sixteen convolutions (eight convolutions for the old position and eight more for the new position) are required. Thus, the total processing of a multisource application may require high computing capacity. GPU code is written in a kernel function that launches multiple threads that execute the same operation with multiple data. In order to carry out the operations described in (5.18) in all sound sources simultaneously, the following computational kernels are launched by means of 128-size thread blocks [20]:

CUDA Kernel 8

This kernel is dedicated to execute all the element-wise multiplications that require the computation of every summand of (5.18) concurrently for each one of the sound sources. In order to determine the number of threads launched in this kernel, we have to take into account the following aspects:

- the size of the complex buffer of $\mathbf{X}_{\text{buff}_m}$, $l_o = 2L$, since the overlap is 50%.
- the worst case is when the synthesis of the rendered sound is a combination of 4 convolutions,
- the number of concurrently rendered M sources,

- the number of output-buffers (in this case 2, $\mathbf{Y}_{\text{buff}_l}$ and $\mathbf{Y}_{\text{buff}_r}$).

Hence, this kernel launches $8Ml_o$ threads. Each thread takes an element from $\mathbf{X}_{\text{buff}_m}$, multiplies it by the corresponding element of $\mathbf{H}$, then the result multiplies it by the corresponding element of $\mathbf{R}$ and finally that result multiplies it by the two corresponding weighted factors (two scalar values). In total, each thread carries out two complex multiplications and two scalar multiplications. The number of blocks that this kernel launches is $\frac{2l_o}{64} \times M \times 1$, being the block size $64 \times 4 \times 1$. A pseudocode of CUDA kernel 8 is presented below, followed by a detailed description of the variables that are used in the CUDA kernel 8.

CUDA Kernel 8 Element-wise multiplication with HRTFs

Input: $\mathbf{S}$, $\mathbf{H}$,M,N,l_o, $\mathbf{w}$, $\mathbf{P}$, $\mathbf{A}$, $\mathbf{B}$, $\mathbf{d}$

Output: $\mathbf{S}_{res}$

```
1: Complex cRet;  //It computes  R(r_S)
2: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
3: int daux = BlockIdx.y * BlockDim.y; // Source Selection
4: int Row = daux + ThreadIdx.y;
5: float faux = (1.0/(1.0 + (1.0/128.0)*(d[BlockIdx.y])^2));
6: float faux1 = ( (float)ColMod /( (float)l_o)) ;
7: faux1 = 2.0 * M_PI * (1.0/128.0);
8: int ColMod = Col & (l_o-1);
9: // Distance Calculation;
10: cRet.x = faux *(__cosf( d[BlockIdx.y] * faux1);
11: cRet.y = (-1)*faux *(__sinf(d[BlockIdx.y] * faux1);
12: // Computation of output index
13: int outidx = Col + Row*2*l_o + ThreadIdx.z*M*4*l_o*2;
14: // Computation of index H
15: int hidx = Col + p[Row + ThreadIdx.z*4*M]*2*l_o;
16: // Computation of index  S
17: int sidx = ColMod + blockIdx.y*l_o;
18: // Computation of output samples
19: S_res[outidx]= ComplexMult(H[hidx], S[sidx]);
20: // Scaling the output samples
21: S_res[outidx]= ComplexScale(S_res[outidx], 1/l_o);
22: // Index of weight factors A
23: int indxA = daux + a[ThreadIdx.y] + ThreadIdx.z*4*M;
```

```
24: // Index of weight factors B
25: int indxB = daux + b[ThreadIdx.y] + ThreadIdx.z*4*M;
26: // Weighting the convolution
27: S_res[outidx]= ComplexMult(S_res[outidx], w[indxA];
28: S_res[outidx]= ComplexMult(S_res[outidx], w[indxB];
29: S_res[outidx]= ComplexMult(S_res[outidx], cRet);
```

Variables used in CUDA kernel 8

To clarify better the described code, here is the description of the cuda variables that have been used in the CUDA kernel 8:

- Matrix $\mathbf{S}$ composed of M rows and l_o columns. It contains audio samples of the M sound sources. Thus, row m of matrix $\mathbf{S}$ is composed $\mathbf{X}_{\text{buff}_m}$.
- Matrix $\mathbf{H}$ contains the HRTF filters. Each row of matrix $\mathbf{H}$ is assigned to one spatial position. First l_o columns correspond to frequency bins of HRTF filters of the left ear, while the second l_o columns corresponds to frequency bins of HRTF filters of the right ear.
- Vector $\mathbf{w}$ is a column vector composed of $4M$ components. Each group of 4 components is composed of the weighted factors of one sound source $[w_A, w_B, w_C, w_D]^{\mathrm{T}}$. Thus, and in order to clarify the reader, $\mathbf{w}[0]$ corresponds to the weighted factor w_A of sound source $m = 0$, while $\mathbf{w}[5]$ corresponds to the weighted factor w_B of sound source $m = 1$.
- Vector $\mathbf{p}$ is a column vector composed of $4M$ components. Each group of 4 components points out each one of the filters, by the which samples of one sound source must be convolved. Each vector component points out a row of the matrix $\mathbf{H}$. For example, sound source $m = 1$ uses the components $\mathbf{p}[4]$,$\mathbf{p}[5]$,$\mathbf{p}[6]$, and $\mathbf{p}[7]$. Then, if $\mathbf{p}[4]$ is 27, then, the filters that are situated in the row 27 of the matrix $\mathbf{H}$ are used to be convolved by the audio samples of sound source $m = 1$.
- Vector $\mathbf{a}$ and vector $\mathbf{b}$ are used to generate the four combinations among the weighted factors that are shown in (5.18). Thus, $\mathbf{a} = [0, 1, 0, 1]$, while $\mathbf{b} = [2, 2, 3, 3]$.

- Vector $\mathbf{d}$ is a column vector composed of M components. Each component represents the distance in meters that is computed by $(r_S - r_0)$ in (5.15) for each one of the sound sources.

- Matrix $\mathbf{S}_{res}$ is composed of $4M$ rows and $2l_o$ columns. It contains the convolved samples.

- The value 128 comes from the division between $\frac{f_s}{v_s} = \frac{44100}{343}$, while `M_PI` represents number π.

The code presented in CUDA kernel 8 could be considered that uses a large number of registers, however, few of them are used in the real code. Variables such as `faux` or `faux1` are only used in this pseudocode in order to make it more understandable. On the other hand, there are some references to variable `ThreadIdx.z` which will be explained in Section 5.5.1.

CUDA Kernel 9

This kernel completes the computation of (5.18) by element-wise summing all the summands computed in CUDA kernel 8 for each one of the two outputs. The result is two complex vectors of size l_o corresponding to $\mathbf{Y}_{\text{buff}_l}$ and $\mathbf{Y}_{\text{buff}_r}$ (the left and right ear in the frequency domain). In this case, $2l_o$ threads are launched. Each thread performs four complex sums for each source; in total, $4M$ complex sums are performed. The number of blocks that this kernel launches is $\frac{2l_o}{128} \times 1 \times 1$, being the block size $128 \times 1 \times 1$.

Figure 5.8 shows the previously described operations that carry out CUDA kernel 8 and CUDA kernel 9 for the particular case $M = 2$. After them, two iFFTs are computed to obtain the final outputs $\mathbf{y}_{\text{buff}_l}$ and $\mathbf{y}_{\text{buff}_r}$.

CUDA Kernel 9 Element-wise Sum from equation (5.18)

Input: $\mathbf{S}_{res}$,M,N,l_o
Output: $\mathbf{S}_{res}$

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: for k = 1,...,4·M − 1 do
3:    S_res[Col] = ComplexSum(S_res[Col], S_res[Col + l_o*k*2]);
4: end for
```

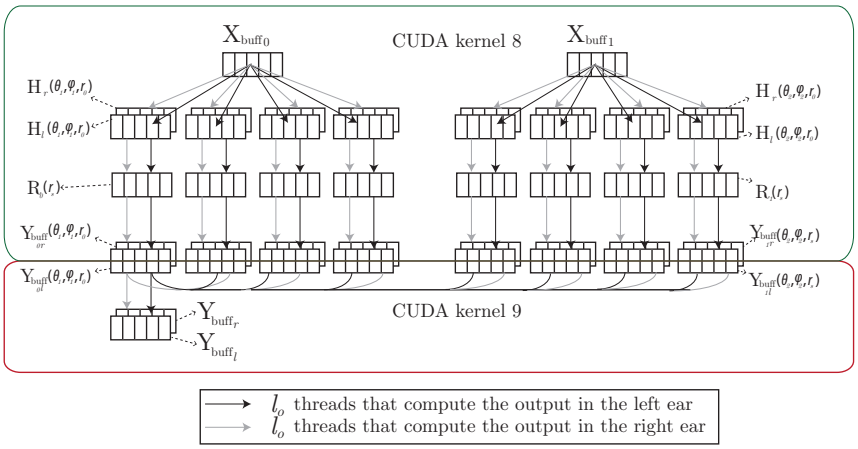


Figure 5.8. Operations carried out by CUDA kernel 8 and CUDA kernel 9. Each thread is responsible for the computation of a sample.

Variables used in CUDA kernel 9

In this code, the first row of matrix $\mathbf{S}_{res}$ contains frequency bins of $\mathbf{Y}_{\text{buff}_l}$ (first l_o columns) and $\mathbf{Y}_{\text{buff}_r}$ (second l_o columns).

5.5.1 Emulating a source movement

Emulating a source movement means carrying out the same operations in two positions (the old position and the new position). The movement is produced between them. The fading vectors ($\mathbf{f}$ and $\mathbf{g}$) are applied to both outputs. The worst case is considered to be when all the sources move at the same time. Thus, for the rest of the subsection, we consider the worst case. When this happens, CUDA kernel 8 launches double the number of threads to execute, $16Ml_o$ threads, as we have the double of filters (see Fig. 5.9). The required processing is equivalent to doubling the number of existing sources. In this case, the number of CUDA blocks launched by CUDA kernel 8 is $\frac{2l_o}{32} \times M \times 1$, being the block size $32 \times 4 \times 2$. Note that `ThreadIdx.z` is now used in CUDA kernel 8. Afterwards, three more kernels are launched. All of them replace the CUDA kernel 9 that is described above.

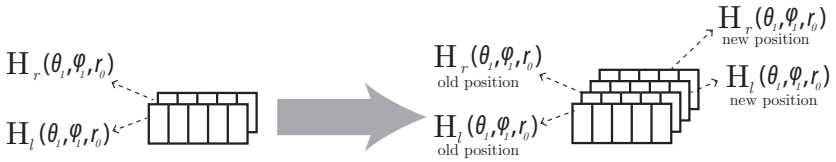


Figure 5.9. The number of filters is double in CUDA kernel 8. The processing is carried out for the old position and for the new position.

CUDA kernel 10

This kernel performs an element-wise sum of the computed summands in CUDA kernel 8 that belong to a position and a source. The result of this kernel is $4M$ complex vectors of size l_o. For this kernel, $4Ml_o$ threads are launched. Each thread carries out four complex sums (see Fig. 5.10). The number of blocks that this kernel launches is $\frac{2l_o}{128} \times 2M \times 1$, being the block size $128 \times 1 \times 1$. In this case, the results of the multiplications are stored at different memory positions, which are denoted as matrix $\hat{\mathbf{S}}_{res}$.

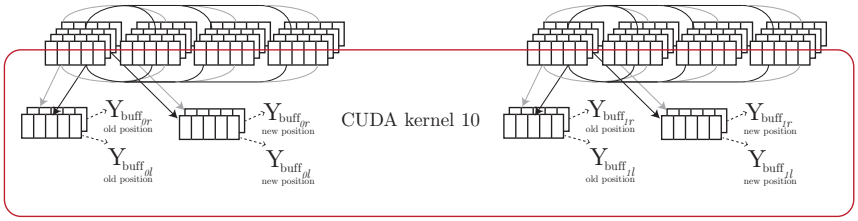


Figure 5.10. CUDA kernel 10 groups the buffers that belong to a position and a source for the particular case $M = 2$.

CUDA Kernel 10 Element-wise sum of the computed summands

Input: $\mathbf{S}_{res}$,M,N,l_o

Output: $\hat{\mathbf{S}}_{res}$

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int Row = BlockIdx.y * BlockDim.y + ThreadIdx.y;
3: int index1 = Col + Row*2*l_o;
4: int index2 = Col + l_o*k*2 + Row*4*2*l_o;
5: for k = 0,...,3 do
6:    Ŝ_res[index1] = ComplexSum(Ŝ_res[index1], S_res[index2]);
7: end for
```

After CUDA kernel 10, $4M$ iFFTs of size l_o are carried out in order to obtain the result vectors $\hat{\mathbf{S}}_{res}$ of CUDA kernel 10 in the time domain.

CUDA kernel 11

This kernel element-wise multiplies the corresponding fading vectors (**f** and **g**) by the result vectors in the time domain obtained after iFFTs. Each thread executes a multiplication. For this kernel, $4Ml_o$ threads are required. The number of blocks that this kernel launches is $\frac{2l_o}{64} \times M \times 1$, being the block size $64 \times 1 \times 2$.

CUDA Kernel 11 Application of the fading vectors

Input: $\hat{\mathbf{S}}_{res}$, **s**, **ch**
Output: $\hat{\mathbf{S}}_{res}$

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int Row = BlockIdx.y * BlockDim.y + ThreadIdx.y;
3: int index1 = Col + Row*2*l_o + ThreadIdx.z*M*2*l_o;
4: int index2 = Col + ThreadIdx.z*2*l_o;
5: if(s[Row])
6:     S_res^[index1] = ComplexMult(S_res^[index1],ch[index2]);
7: end if
```

Variables used in CUDA kernel 11

- Vector **ch** is composed of $2l_o$ elements and contains both fading vectors **f** and **g**. First l_o elements of **ch** correspond to elements of vector **f** while the seconds l_o elements correspond to **g**, i.e **ch** = [**f** **g**].
- Vector **s** is composed of M elements and its values can take only two values: 0 and 1. When **s**[2]=0, it means that sound source labeled as $m = 2$ has not been moved and stays in the same place. Thus, fading must not be applied. Otherwise (**s**[2]=1), fading must be applied. Values of vector **s** are modified by the user from the main program and are transferred asynchronously to the GPU.

CUDA kernel 12

This kernel launches $2l_o$ threads in order to element-wise sum the result vectors obtained after CUDA kernel 11 for each output. The number of blocks that this kernel launches is $\frac{2l_o}{128} \times 1 \times 1$, being the block size $128 \times 1 \times 1$.

CUDA Kernel 12 Element-wise Sum Application of the fading vectors

Input: $\hat{\mathbf{S}_{res}}$, **s**
Output: $\hat{\mathbf{S}_{res}}$

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int index = Col + M*2*l_o;
3: if(s[0])
4:    S_res[Col] = ComplexSum(S_res[Col], S_res[index]);
5: end if
6: for k = 1,...,M-1 do
7:    S_res[Col] = ComplexSum(S_res[Col], S_res[Col + k*2*l_o]);
8:    if(s[k])
9:       S_res[Col] = ComplexSum(S_res[Col], S_res[index + k*2*l_o]);
10:   end if
11: end for
```

Variables used CUDA kernels 11 and 12

- Output-buffers $\mathbf{y}_{\text{buff}_l}$ and $\mathbf{y}_{\text{buff}_r}$ are contained in Matrix $\hat{\mathbf{S}_{res}}$.
- Regarding implementation details, it is important to remark that all data were hosted in the GPU *global-memory* while the constant variables, such as l_o, M, N, etc were hosted in *constant-memory*. The resources of the *shared-memory* were used as a L1 cache memory in order to reduce memory access times to *global-memory*.

5.5.2 Interaction with the user

Figure 5.11 shows the flowchart from this spatial audio application whose processing is executed on the GPU. Firstly, the correct initialization of the application is performed (both audio and computational parameters). Then, the user introduces the *Hearing Position* of each source. This position is given in degrees for the azimuth θ and the elevation ϕ, and in meters for the distance r. Then, the nearby positions (ϕ_1, ϕ_2, θ_1, θ_2) that belong to the database, together with their weighted factors w_A, w_B, w_C, and w_D for each *Hearing position*, are calculated. The filters from the HRIR database, the positions, and the weighted factors are transferred to the GPU. The application starts with the acquisition of the first audio samples from different sources. When all the buffers $\mathbf{x}_{\text{buff}_m}$ are full, they are sent to the GPU. Then, the processing begins by launching first CUDA kernel 8

and then CUDA kernel 9 when the locations of the all the sound sources do not move, or launching CUDA kernel 8, then CUDA kernel 10, then CUDA kernel 11, and then CUDA kernel 12, if any of the sources has changed its position. These four kernels are launched independently of the number of sources that changed their positions since they are designed to give a response to the worst case, which is when all the sources move at the same time. When the processing ends, two output-buffers, $\mathbf{y}_{\text{buff}_r}$ and $\mathbf{y}_{\text{buff}_l}$, are sent back to the CPU. Whenever a sound source moves, the new nearby positions (ϕ_1, ϕ_2, θ_1, θ_2), together with the new weighted factors w_A, w_B, w_C, and w_D of that sound source, are calculated. It is important to note the tridimensionality of application processes that are executed on the GPU, such as FFT, CUDA kernel 8, CUDA kernel 9, etc (see Fig. 5.11). This symbolizes that all of them make use of multiple threads that are launched in parallel.

5.6 Results

The main result of this chapter is a headphone-based multisource spatial audio application whose audio processing is carried out on the GPU. The application is being used on a notebook Intel Core i7 at 1.60 GHz with a GPU Geforce GTS360M with 1.2 capability, see Fig. 5.12. Sound examples recorded with this application are available in the section *Examples of binaural sounds generated by our GPU-based binaural application* from the web page referenced in [14].

We explore also the computational performance of the application for two different cases. On the one hand, we consider that the application managed M sources and all sound sources stay fixes (the path executed by FFTs, CUDA kernel 8 and CUDA kernel 9 in flowchart of Fig. 5.11), and on the other hand, all sound sources were moving at the same time in order to execute the maximum number of operations (CUDA kernels 8, 10, 11, and 12, the other path in flowchart of Fig. 5.11). The computational experiment consisted of launching the application and gradually increasing the number of sources. Each time, the time t_{proc} was measured and compared with the fixed time t_{buff}. The application always works in real time as long as $t_{\text{proc}} < t_{\text{buff}}$. When this condition is no longer valid, the number of sources reaches the maximum value of M. However, this does not mean that the

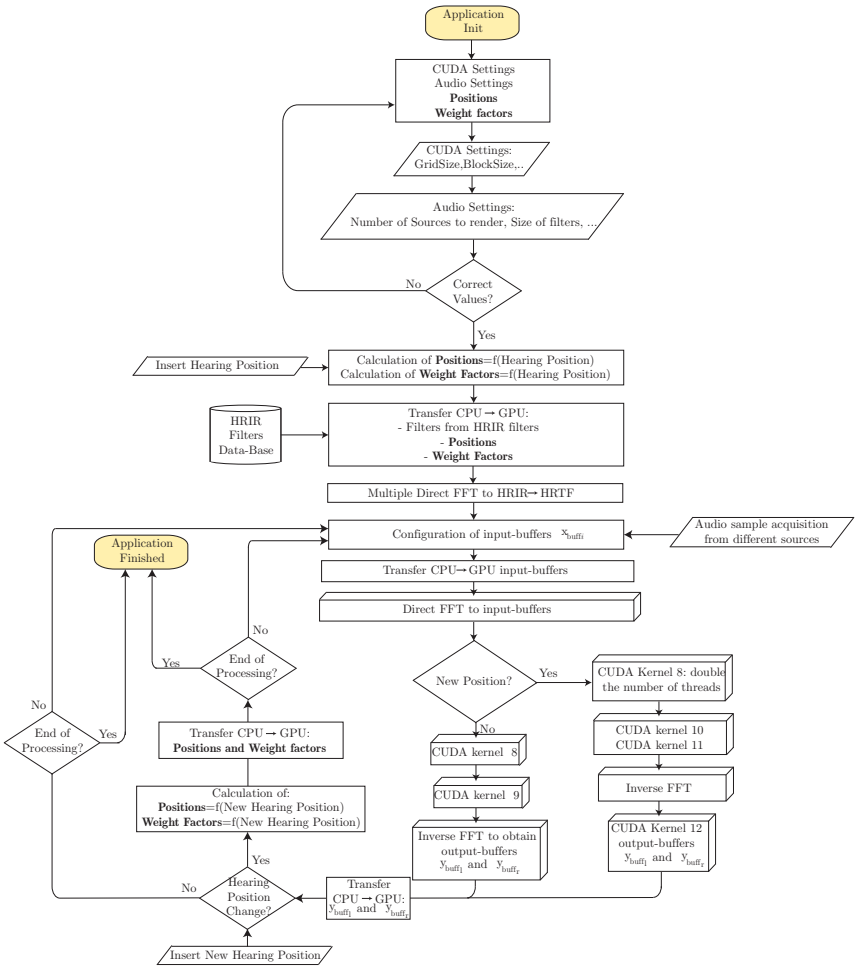


Figure 5.11. Flowchart from a spatial audio application whose audio processing is totally carried out on the GPU. Tridimensional application processes are executed on the GPU and symbolize the use of multiple threads that are launched in parallel.

application stops working, since it could work as an off-line application. Figure 5.13 and Figure 5.14 show the evolution of t_{proc} as a function of the number of sources. The time t_{buff}, which symbolizes the border between off-line applications and real-time applications, is also shown. Obviously, if this application has to share GPU resources with other applications, peak

Figure 5.12. Developed headphone-based spatial application running on a notebook with the GPU GTS360M.

performance would decrease. Besides the performance analysis in GTS-360M, we present also performances from other GPUs that are usually employed in powerful computers. The GPU hardware used in the tests have the characteristics shown in Table 5.4.

Table 5.4. Characteristics of the GPUs.

Cuda Device	GTS-360M	TESLA C2075	GTX-580
Architecture	Tesla	Fermi	Fermi
CUDA Capability	1.2	2.0	2.0
Number of SM	12	14	16
CUDA Cores per SM	8	32	32
Cache Hierarchy L1/L2	No	Yes	Yes
Maximum number threads per block	512	1024	1024
Warp Schedulers per SM	1	2	2

Comparing both figures, it is clear that the path composed by only two kernels computes faster than the path composed by four kernels. The

maximum number of sources that can be rendered in real time is reduced between 1.5 and 2.0 times when all sound sources move, since the added processing is approximately twice (old positions and new positions). Analyzing both figures, it is appreciable as performances of TESLA C2075 and GTX-580 outperform the performance of GTS-360M. This occurs because their capabilities are different. GTS-360M has SMs composed by 8 cuda cores, while the other have SMs composed of 32 cuda cores. This implies that thread blocks can be processed faster, as the SMs has more physical cores. On the other hand, as GTX-580 has two SMs more than TESLA C2075, more thread blocks can be distributed at runtime. This explains why performances of GTX-580 are better than TESLA C2075. Moreover, capability 2.0 has two *warp schedulers* which reduce the latency times by switching among different *warps* during execution, in contrast with GTS-360M that has only one. Other advantage of capability 2.0 is that allows to use *shared-memory* as a L1 cache, and this together to the L2 cache reduce access times to the GPU *global-memory*.

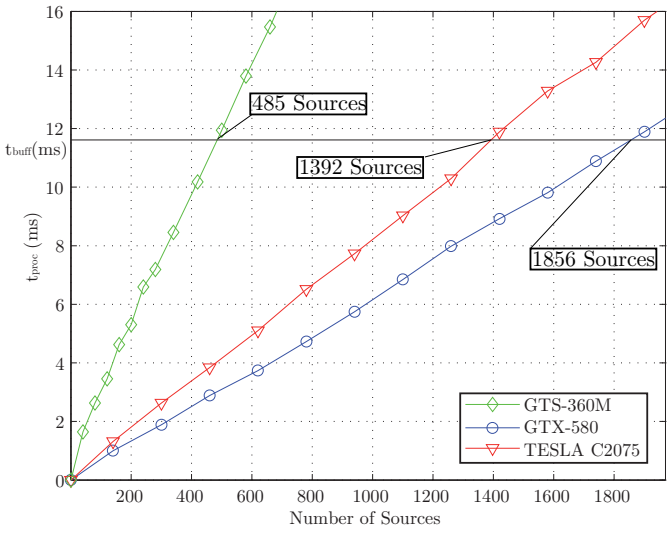


Figure 5.13. Number of sound sources that can be managed by our proposed spatial sound application when all the sound sources stay static in real time.

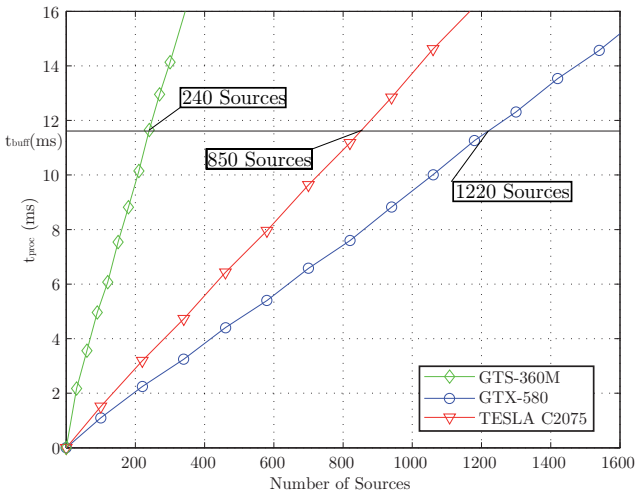


Figure 5.14. Number of sound sources that can be managed by our proposed spatial sound application when all the sound sources are moving in real time.

5.7 Conclusions

This chapter has presented a complete multisource spatial application in a binaural system. Multisource applications require high computing resources. The development of the GPUs has allowed that different engineering problems can be approached. In this work, we have developed a spatial audio application whose massive processing is carried out on the GPU, without overload the CPU, and freeing its resources to be used for other tasks.

To render a sound source in a specific spatial location with a binaural system, it is necessary to convolve audio samples with HRIR filters that provide spatial information. Two common problems have been resolved during the design: synthesizing sound sources positions that are not in the HRIR database, and virtualizing the movement of the sound sources between different positions. Both problems were approached by increasing the number of convolutions which are later weighted and combined in different

ways, taking maximum profit of the GPUs capacity for executing multiple convolutions simultaneously. Both solutions were assessed by performing different audio analyses.

Finally, the implementation of the headphone-based spatial sound application on the GPU is described. The implementation is composed of different computational kernels that are executed depending on the movement of the sound sources, and the characteristics of the database. The development of this application on GPU hardware has allowed us to manage multiple sources without overloading the CPU because of its huge capacity for parallel computation. The results show that this application can manage up to 240 sources simultaneously when all the sources are moving at the same time. It is important to remark that this application is totally portable and can be run on different GPUs. Depending on the GPU resources, such as the number of SMs, *warp schedulers*, etc, the number of sources rendered in real time can vary. Thus, this chapter demonstrates that the use of the GPU hardware provides a suitable solution to build multisource binaural sound applications that demand high computational needs.

More details can be found in [112] and in [113].

6 Wave Field Synthesis system

6 Wave Field Synthesis system

Wave Field Synthesis (WFS) is a spatial audio reproduction system that provides an accurate spatial sound field in a wide area. This sound field is rendered through a high number of loudspeakers to emulate virtual sound sources. WFS systems require high computational capacity since they involve multiple loudspeakers and multiple virtual sources. Furthermore improvements of the spatial audio perception imply even higher processing capacity, mainly to avoid artifacts when the virtual sources move, and compensate the room effects at certain control points within the listening area. In this chapter, we propose a GPU implementation of a WFS system with Room Compensation that yields to render maximum number of sound sources. This GPU implementation seeks maximum parallelism by adapting the required computations to the different GPU architectures (Tesla, Fermi and Kepler).

6.1 Theory of a WFS system

Wave Field Synthesis is a sound reproduction method, based on fundamental acoustic principles [13], [63]. It enables the generation of sound fields with natural temporal and spatial properties within a volume or area

bounded by secondary sources (arrays of loudspeakers, see Fig. 6.3). This method offers a large listening area with uniform and high reproduction quality.

The theoretical basis of WFS is given by the Huygens' principle. According to this, the propagation of a wave front can be described by recursively adding the contribution of a number of secondary point sources distributed along the wave front. This principle can be used to synthesize acoustic wave fronts of an arbitrary shape.

A synthesis operator for each loudspeaker can be derived. The general 3-D solution can be transformed into the 2-D solution, which is sufficient for reconstructing the original sound field in the plane of listening [114], [64], [65]. For that purpose a linear array of loudspeakers is employed to generate the sound field of virtual sources.

Following a model-based rendering in which point sources and plane waves are used [115], the field rendered by a sound source m at point R within the area surrounded by the loudspeakers can be expressed as equation

$$P(\mathbf{x}_R, \omega) = \sum_{n=0}^{N-1} Q_n(\mathbf{x}_m, \omega) \frac{e^{\frac{-j\omega\triangle r}{c}}}{\triangle r}, \tag{6.1}$$

where c is the speed of the sound, $\mathbf{x}_m$ is the position of the virtual sound m, $\mathbf{x}_R$ is the position of the point R, and $\triangle r$ is the distance between the nth loudspeaker and the point R.

The driving signal of the nth loudspeaker in a rendering system composed of N loudspeakers is represented by $Q_n(\mathbf{x}_m, \omega)$, which is computed as

$$Q_n(\mathbf{x}_m, \omega) = S(\omega)\sqrt{\frac{j\omega}{2\pi c}} C \frac{1}{\sqrt{r}} \cos(\theta) e^{\frac{-j\omega r}{c}}. \tag{6.2}$$

where C is a geometry dependent constant, $r = \mid \mathbf{x}_m - \mathbf{x}_n \mid$, and $\mathbf{x}_n$ is the position of the loudspeaker n. Figure 6.1 shows the geometry of the system, where θ is the angle between the line that connects $\mathbf{x}_m$ and $\mathbf{x}_n$, and the normal vector $\mathbf{n}$ of the loudspeaker n. The piano represents the sound source m in Fig. 6.1. The driving signal (6.2) consists of several elements that have different functionalities. The term $S(\omega)$ is the frequency-domain characteristics of the source signal, while the term

$$H(\omega) = \sqrt{\frac{j\omega}{2\pi c}}, \tag{6.3}$$

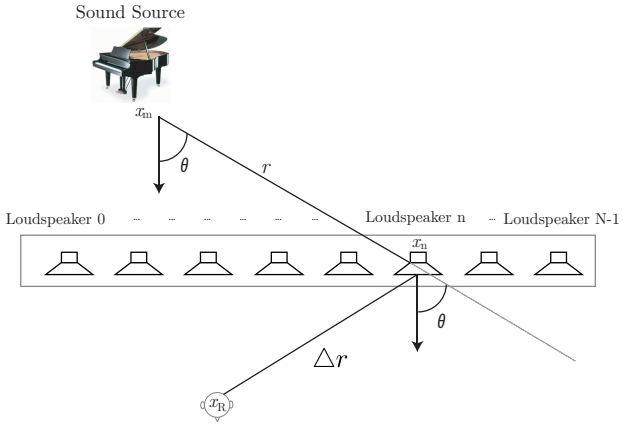


Figure 6.1. Geometry of a WFS system where it is appreciated the sound source m, the N loudspeakers, and the different distances among sound source, loudspeakers and a listener.

represents a filtering operation that is independent of the position of the virtual source. In [116], it is referred as a WFS pre-equalization filter that represents a lowpass filter with a constant slope of 3 dB/octave if loudspeaker is considered a monopole secondary source, while it forms a high-pass with a magnitude increase of 3 dB/octave in case of dipole secondary sources. An important contribution in (6.2) is

$$a_{mn} = \frac{C}{\sqrt{r}} \cos(\theta) \tag{6.4}$$

that denotes an amplitude factor that depends on the positions of the sound source m, and the loudspeaker n. Finally, the $e^{\frac{-j\omega r}{c}}$ represents a time delay that is proportional to the distance between the virtual sound source m and the loudspeaker n, being τ_{mn} defined as:

$$\tau_{mn} = \frac{\mid \mathbf{x}_m - \mathbf{x}_n \mid}{c} \tag{6.5}$$

The driving signal shown in (6.2) is expressed also in time domain as

$$q_n^m = a_{mn} \cdot s_m * h * \delta(t - \tau_{mn}). \tag{6.6}$$

where $*$ denotes the convolution operator, s_m is the signal of sound source m, h is the inverse Fourier transform of $H(\omega)$ in (6.3).

In a multi source system composed of M virtual sound sources, the loudspeaker driving signal of the n loudspeaker is

$$q_n = \sum_{m=0}^{M-1} q_n^m. \tag{6.7}$$

In a discrete-time signal processing system with sampling frequency f_s, expressions (6.6) and (6.7) turn to

$$q_n = \sum_{m=0}^{M-1} a_{mn} \cdot s_m * h * \delta[k - \tau_{mn}]. \tag{6.8}$$

where k is the sample index k.

6.1.1 Room Compensation in a WFS system

As introduced in Chapter 3, the interaction of the driving signals with the listening room can deteriorate the localization properties that a WFS system has. The synthesized sound-field can be altered by new echoes that are introduced by the listening room and that reduce the spatial effect. Lopez et al. designed and validated in [18, 117] a multichannel inverse filter bank that corrects these room effects at selected points within the listening area. In a WFS system composed of N loudspeakers, this implies to add N^2 filters to the system, increasing its computational demand. Equation 6.9 shows the operations that are carried out in a multichannel inverse filter bank with every driving signal. The final signal to be reproduced by the nth loudspeaker y_n is a combination of all the filtered signals, as illustrated in Fig. 6.2, where the filter f_{0n} goes from the driving signal q_0 to the loudspeaker n.

$$y_n = \sum_{j=0}^{N-1} q_j * f_{jn}. \tag{6.9}$$

The calculation of the inverse filters can be carried out in a setup stage, since the main room reflections can be considered invariant for each specific room. Different methods have been proposed to obtain the bank of correction filters. There are methods that compute an approximate solution in frequency domain using FFT [94]. However, we use in this work a method

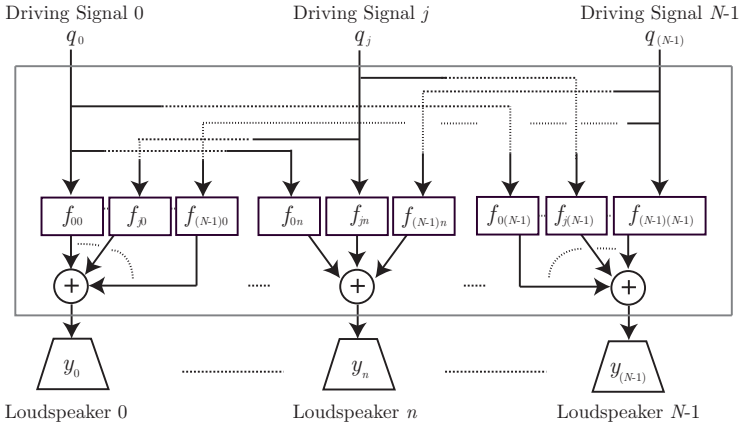


Figure 6.2. Multichannel inverse filter bank, where every driving signal is convolved by N filters. The signal that is reproduced by a loudspeaker is a combination of all the filtered signals.

that guarantees a minimal square error solution in time domain [118] for computing the correction filters f_{jn}. A detailed description of the operation to carry out for the computation of the filters is achieved in [18].

6.1.2 Practical Implementation of a WFS system

To this end, processing audio buffers of size $2L$ with a 50% overlap in the frequency domain were used. Thus, the processing of the audio buffers is composed by the current input-data buffer and the previous one that comes from the audio card. The filters of the RC block are also composed by L coefficients. We denote $\mathbf{x}_{\text{buff}_m}$ to the input-data buffer that is composed of $l_o = 2L$ samples of the sound source x_m, where $m \in [0, M-1]$. The FFT of size l_o of this block of samples is denoted by $\mathbf{X}_{\text{buff}_m}$. Expression (6.10) shows the computation of $\mathbf{Q}^m_{\text{buff}_n}$, which is composed by l_o frequency bins; $\mathbf{Q}^m_{\text{buff}_n}$ is equivalent to the expression (6.2) particularized for the sound source m. We use $\otimes$ for representing an element-wise multiplication between vectors.

$$\mathbf{Q}^m_{\text{buff}_n} = (\mathbf{X}_{\text{buff}_m} \otimes \mathbf{h} \otimes \mathbf{e}^{\frac{-\mathbf{j2}\pi[\mathbf{0}\ldots(\mathbf{l_o}-\mathbf{1})]\tau_{\mathbf{mn}}}{\mathbf{l_o}}}) \cdot a_{mn}. \tag{6.10}$$

Vectors $\mathbf{h}$ and $\mathbf{e}^{-\mathbf{j2}\pi[\mathbf{0}\ldots(\mathbf{l_o}-\mathbf{1})]\tau_{\mathbf{mn}}}$ are also composed by l_o frequency bins. In a multisource system, $\mathbf{Q}_{\text{buff}_n}$ is computed as

$$\mathbf{Q}_{\text{buff}_n} = \sum_{m=0}^{M-1} \mathbf{Q}^m_{\text{buff}_n}. \tag{6.11}$$

Expression (6.11) shows the computation of only a WFS rendering system. The additional processing of a Room Compensation block is computed as

$$\mathbf{Y}_{\text{buff}_n} = \sum_{r=0}^{N-1} \mathbf{Q}_{\text{buff}_r} \otimes \mathbf{F}_{rn}. \tag{6.12}$$

where $\mathbf{F}_{rn}$ is one of the N^2 filters of the multichannel inverse filter bank that composes the RC block. The subscript r indicates the rth input of the block, while the subscript n indicates the nth output that fits in the nth loudspeaker. Thus, the L samples of the output-data buffer of the nth loudspeaker are extracted from the l_o frequency bins of $\mathbf{Y}_{\text{buff}_n}$.

Movement Virtualization

For synthesizing moving sound sources, mathematical formulae were derived by Jansen in [119]. He analyzed the doppler-effect impact and showed that for slow moving sound sources the doppler effect is negligible and one can resort to updating locations for each location and changing those in time. Thus, we choose this approach in this system. However, this solution generated artifacts in the rendered sounds.

To avoid these clicks, we duplicate the rendering of the sound source and apply a fading, i.e a gradual increase in the sound rendered by the new position (*fade-in*) while the sound rendered by the old position decreases (*fade-out*) in the same way. When this occurs, more processing to obtain the driving signal for each loudspeaker is required.

If source m shifts from position $\mathbf{x}_{mA}$ to $\mathbf{x}_{mB}$ (A and B represents two arbitrary positions of the virtual sound source m whose coordinates are $\mathbf{x}_{mA}$ to $\mathbf{x}_{mB}$, respectively), two driving signals for the nth loudspeaker must be computed according to Expression (6.2). The results are: $\mathbf{Q}^{mA}_{\text{buff}_n}$ and $\mathbf{Q}^{mB}_{\text{buff}_n}$. After that, iFFTs of size l_o are applied to both buffers, obtaining two buffers in time domain: $\mathbf{q}^{mA}_{\text{buff}_n}$ and $\mathbf{q}^{mB}_{\text{buff}_n}$. To these two buffers, the fading is applied. To this end, the buffer $\mathbf{q}^{mA}_{\text{buff}_n}$ is element-wise multiplied

by a *fade-in* vector $\mathbf{f}$, and the buffer $\mathbf{q}^{mB}_{\text{buff}_n}$ is element-wise multiplied by a *fade-out* vector $\mathbf{g}$. Finally, the buffer $\mathbf{q}^{m}_{\text{buff}_n}$ is obtained by element-wise summing the two previous multiplications, as shown in

$$\mathbf{q}^{m}_{\text{buff}_n} = (\mathbf{q}^{mA}_{\text{buff}_n} \otimes \mathbf{f}) + (\mathbf{q}^{mB}_{\text{buff}_n} \otimes \mathbf{g}). \tag{6.13}$$

Both $\mathbf{f}$ and $\mathbf{g}$ are complementary vectors of size $l_o = 2L$ that must satisfy

$$\mathbf{f}[s] = \mathbf{g}[2L - 1 - s], \tag{6.14}$$

where $s \in [L, 2L - 1]$. As the fading is applied to current audio buffer, the first L values of both vectors are a constant equal to 0

$$\mathbf{f}[s] = \mathbf{g}[s] = 0, \tag{6.15}$$

where $s \in [0, L - 1]$. Different values of vectors $\mathbf{f}$ and $\mathbf{g}$ are presented in [104].

In a multisource system, all audio buffers are later summed

$$\mathbf{q}_{\text{buff}_n} = \sum_{m=0}^{M-1} \mathbf{q}^{m}_{\text{buff}_n}. \tag{6.16}$$

Discarding the first L samples of the $\mathbf{q}_{\text{buff}_n}$, we have audio samples to be rendered by the nth loudspeaker of the system. However, if a Room Compensation block is applied in the system, more processing is required. In order to apply expression (6.12) to $\mathbf{Q}_{\text{buff}_n}$, a FFT (size of FFT l_o) of $\mathbf{q}_{\text{buff}_n}$ is needed.

Thus, processing a fading means to increase mainly the number of FFT transformations to carry out in the system.

6.2 Test system

All measurements has been carried out at the laboratory of the Audio and Communications Signal Processing Group (GTAC) [14] of the Universitat Politècnica de València (UPV). This laboratory is composed by 96 loudspeakers (N=96) that are positioned using an octogonal geometry (see Fig. 6.3).

We use a standard audio card at the laboratory. The audio card uses the ASIO (Audio Stream Input/Output) driver to communicate with the CPU and provides 512 samples per channel every 11.61 ms (sample frequency f_s=44100 Hz). This time is called for us t_{buff}. Therefore, in our system, L=512 which implies that filters $\mathbf{F}$ in RC block are composed by 512 coefficients. Observing Fig. 6.3, it is appreciated as our laboratory has semi-anechoic characteristics. Thus, for this kind of room, the use of 512 coefficients is enough, [94] [120]. If a room presents more reverberations, the necessary filter coefficients of the RC block will be longer. If this happens, two kind of solutions could be raised: 1) take the size of the buffer L longer, which implies increasing the latency, or 2) carry out a partition filtering [121].

One important aspect lies on the data-flow management by the GPU. Firstly, the M input-data buffers are filled, then they are transferred from the CPU to the GPU. After processing, N output-data buffers are sent back from the GPU to CPU to be rendered by the N loudspeakers. We define t_{proc} as the processing time since the M input-data buffers are filled till the N output-data buffers are received. The spatial audio system (see Fig. 5.4) works in real-time as long as $t_{\text{proc}} < t_{\text{buff}}$.

6.2.1 System Setup

All variables are stored at *global-memory* of the GPU. Before beginning the real-time processing, values of filter $\mathbf{h}$ are transferred to the GPU, together to the coordinates of the virtual sound sources. Then, the delay and amplitude factors per loudspeaker and per source are computed. Each one of them is stored in the memory space reserved for the position A inside a tridimensional matrix whose size depends on: the number of loudspeakers, the number of sources, and the two positions per source (position A and position B), see Fig.6.4(a). Position A and position B are necessary in order to carry out properly the fading. The resulting output-data buffers of (6.10) are stored in the memory space reserved for Position A inside another tridimensional matrix whose size depends on: the number of loudspeakers, the number of sources, the number of samples in the output-data buffer (l_o samples) per loudspeaker, and the two positions per source (Fig.6.4(b)). All output-data buffers are stored in consecutive memory locations in order to achieve the coalescing access to the *global-memory*.

When a user introduces a new position for any of the sound sources, new

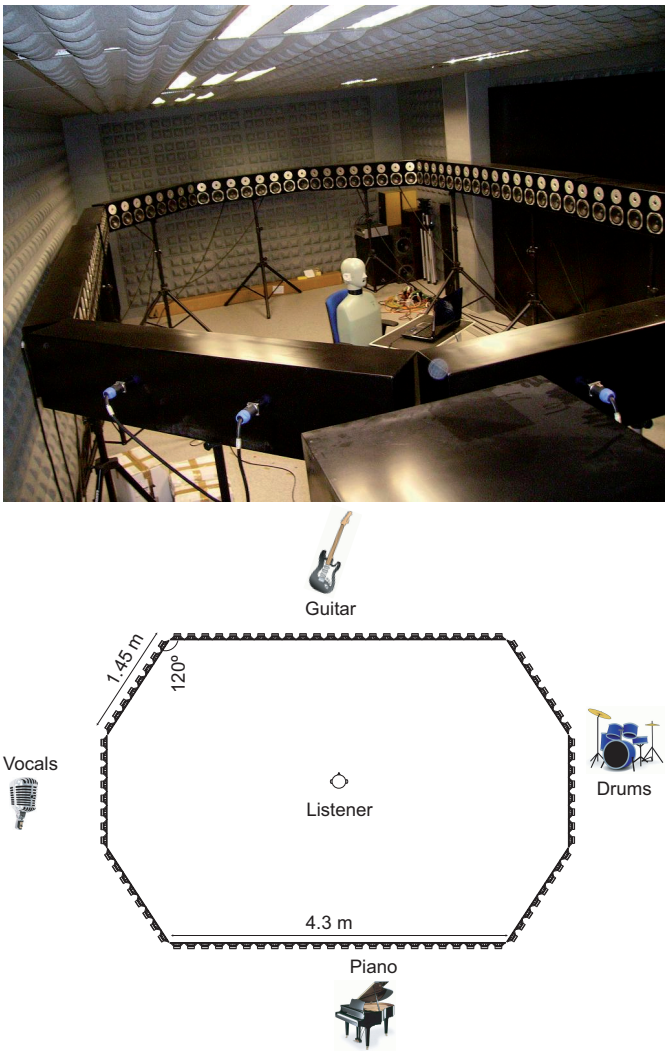


Figure 6.3. Configuration of the array at the laboratory of the GTAC at UPV

delay and amplitude factors are generated for all the sources. In this case, they are stored in the corresponding memory spaces reserved for positions B (Fig.6.4(a)). The next output-data buffers are computed by using delay and amplitude factors of both positions (discontinuous path in Fig.(6.5)). Memory spaces reserved for the position A and the position B are now used

for storing the output-data buffers (Fig.6.4(b)).

Afterwards, the following output-data buffers are computed by using only delay and amplitude factors of position B, since the current delay and amplitude factors are in this moment in position B. Thus, as positions of the virtual sound sources change, storing and processing is swapped between position A and position B. Only when the fading is applied, both positions are used. Fig.(6.5) summaries this process.

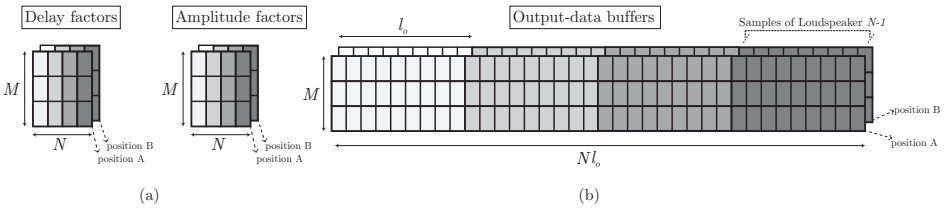


Figure 6.4. Data structures used for storing in the *global-memory* of the GPU: the delay factors and the amplitude factors in (a), and the output-data buffers in (b).

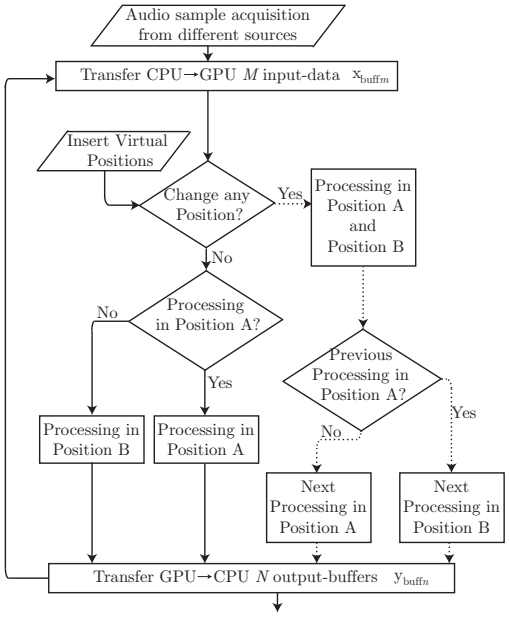


Figure 6.5. Flowchart of the processing executed on the GPU.

The advantage of using two different memory spaces is to reduce the number of accesses to the *global-memory*. Thus, we reduce intern data copies within *global-memory* when a fading is carried out.

6.2.2 Computational kernels implemented on GPU

We employed for the implementations three different NVIDIA GPUs whose characteristics are shown in Table 6.1. In order to map the described operations, we launch the following CUDA kernels:

Table 6.1. Characteristics of the GPUs.

Cuda Device	GTS-360M	GTX-590	GTX-690
Architecture	Tesla	Fermi	Kepler
Number of SM	12	16	8
Number of cores	96	512	1536
Warp Schedulers per SM	1	2	4

CUDA kernel 13

In this kernel, each thread is devoted to compute $\mathbf{x}_n - \mathbf{x}_m$. Both position are composed of two coordinates. Thus, the number of blocks that this kernel launches is $\frac{N}{128} \times 2 \times M$, being the block size $128 \times 1 \times 1$.

CUDA Kernel 13 Difference in coordinates between Loudspeakers and Sound Sources

Input: N, **S**, **L**, S_{elect}
Output: **D**

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int Row = BlockIdx.y * BlockDim.y + ThreadIdx.y;
3: int High = BlockIdx.z * BlockDim.z + ThreadIdx.z;
4: int index = Col + Row*N;
5: int index2 = index + High*N*2 + 2*S_elect*N*M;
6: int index3 = Row + High*2;
7: // Difference in coordinates Sound Sources - Loudspeaker.
8: D[index2] = L[index] - S[index3];
```

Variables used CUDA kernel 13

- Matrix **S** has a tridimensional structure with these dimensions: ($1 \times 2 \times M$). It has one column, two rows that indicate the coordinates of the sound source, and the third dimension is devoted to the number of sound sources in the system.

- Matrix **L** has a bidimensional structure with these dimensions: ($N \times 2 \times 1$). It stores the coordinates of the N loudspeakers in the system.

- Matrix **D** has a 4-D structure with these dimensions: ($N \times 2 \times M \times 2$). First 3-D components ($N \times 2 \times M \times 1$) stores the differences $\mathbf{x}_n - \mathbf{x}_m$ for all the sound sources in the Position A (see Section 6.2.1), while the second 3-D components stores them for the Position B. The selection between Position A and Position B is related to the fading, i.e when a sound source is moving.

- Variable S_{elect} indicates if the differences must be stored in Position A or in Position B. If S_{elect}=0, coordinate differences are stored in Position A. In case S_{elect}=1, coordinate differences are stored in Position B (See Fig. (6.5)). This variable can only take these two kind of values and is manipulated by the main program.

CUDA kernel 14

In this kernel, each thread is devoted to compute different parameters that are necessary for the proper rendering in the system. The number of blocks that this kernel launches is $\frac{N}{64} \times 1 \times \frac{M}{2}$, being the block size $64 \times 1 \times 2$.

CUDA Kernel 14 Computation of Amplitudes and Delays Loudspeakers/Source

Input: **S**, **L**, S_{elect}, **D**, N, G, S_{elect}

Output: **A**, **T**, $\mathbf{m}_{ask}$

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int High = BlockIdx.z * BlockDim.z + ThreadIdx.z;
3: int index = Col + High*N + S_elect*N*M;
4:  // Variable D has one dimension more:
5: int index2 = index + High*N + S_elect*N*M;
6: int index3 = index2 + N;
```

```
 7: //  Angle between positions: sound source and loudspeaker .
 8: float fangle = atan2f(D[index3],D[index2]);
 9: //  The Geometry condition G and the angle: fangle
10: if(fangle <> G)
11:    //  ..determines if loudspeaker per source is ON or OFF
12:    m_ask[index1] = 1.0;
13: end if
14: //  Computation equation (6.4).
15: float faux = D[index3]*D[index3];
16: faux = faux + D[index2]*D[index2];
17: faux = sqrtf(faux);  // Here is computed r
18: A[index] = C*cosf(P[index])/sqrtf(faux);;
19: //  Computation equation (6.5).
20: T[index] = rintf(44100.0*faux/343.0);
```

Variables used in CUDA kernel 14

- Matrix $\mathbf{A}$ and Matrix $\mathbf{T}$ have a tridimensional structure: ($M \times N \times 2$). Both structures contain all the amplitudes and delays values per loudspeaker and sound source: a_{mn} and τ_{mn}. The third dimension is devoted to select the storing of the computed parameters: Position A or Position B. This selection is carried out by the variable S_{elect}, as in CUDA kernel 13.

- Matrix $\mathbf{m}_{ask}$ has the same tridimensional structure as $\mathbf{A}$ and $\mathbf{T}$. Their values depend on the geometry of the WFS system and indicates if a loudspeaker must reproduce a specific sound source in a specific position. Its functionality is related with the correct build of the wavefront within the geometry of the WFS system.

CUDA kernel 15

This kernel computes $\mathbf{Q}^m_{\text{buff}_n}$ in (6.10). The number of blocks that this kernel launches is $\frac{l_o}{128} \times N \times M$, being the block size $128 \times 1 \times 1$.

CUDA Kernel 15 Computation of the Driving signals

Input: $\mathbf{A}$, $\mathbf{T}$, $\mathbf{h}$, $\mathbf{S}$, $\mathbf{m}_{ask}$, l_o, N, M, S_{elect}

Output: $\mathbf{Q}$,

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int Row = BlockIdx.x * BlockDim.x + ThreadIdx.x;
3: int High = BlockIdx.z * BlockDim.z + ThreadIdx.z;
4: int HighMod = High % M;
5: Complex cRet;
6:  // Global Index;
7: int index = Col + Row*l_o;
8: int indexTot = index + High*N*l_o + S_elect*N*M*l_o;
9: int indS = Col + HighMod*l_o;
10: int indMask = Row + High*N + S_elect*N*M;
11:  // Computation Delay in frequency-domain
12: float faux =  2.0*M_PI*T[indMask];
13: faux = faux*( (float)Col /( (float)l_o)) ;
14: cRet.x = A[indMask]*(__cosf(faux));
15: cRet.y = (-1.0)*A[indMask]*(__sinf(faux));
16: Q[indexTot] = ComplexMult(S[indS],H[Col]);
17: Q[indexTot] = ComplexScale(Q[indexTot],1/l_o);
18: Q[indexTot] = ComplexMult(Q[indexTot],cRet);
19: Q[indexTot] = ComplexScale(Q[indexTot],m_ask[indMask]);
```

Variables used in CUDA kernel 15

- Matrix $\mathbf{S}$ is composed of M rows and l_o columns. Elements of the row m are composed by the l_o frequency bins of the buffer $\mathbf{X}_{\text{buff}_m}$.
- Matrix $\mathbf{Q}$ has a 4-D structure with these dimensions: ($N \times l_o \times M \times 2$). First two dimensions correspond to all the driving signals for a sound source. The third dimension is devoted to the number of sound sources M. Thus, the value m in the third dimension within the row n stores the frequency bins of the driving signal $\mathbf{Q}^m_{\text{buff}_n}$. The fourth dimension is devoted to select the storing of the driving signals: Position A or Position B. This selection is carried out by the variable S_{elect}.
- Vector $\mathbf{h}$ corresponds to l_o frequency bins of the filter h, see (6.3).
- Note that variable `HighMod` is not relevant in this code. However, it

will have an important role when the fading is carried out.

CUDA kernel 16

This kernel computes $\mathbf{Q}_{\text{buff}_n}$ in (6.11). The number of blocks that this kernel launches is $\frac{l_o}{128} \times N \times 1$, being the block size $128 \times 1 \times 1$. Each thread carries out M sums of complex numbers.

CUDA Kernel 16 Computation of the Driving signals

Input: $\mathbf{Q}$, l_o, N, M, S_{elect}
Output: $\mathbf{Q}$,

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int Row = BlockIdx.y * BlockDim.y + ThreadIdx.y;
3: int index = Col + Row*l_o + S_elect*N*M*l_o;
4: for k = 1,...,M − 1 do
5:    Q[index] = ComplexSum(Q[index],Q[index + k*N*l_o]);
6: end for
```

CUDA kernel 17

This kernel computes the element-wise multiplications of (6.12). The number of blocks that this kernel launches is $\frac{l_o}{128} \times N \times N$, being the block size $128 \times 1 \times 1$. Each thread carries out a complex multiplication.

CUDA Kernel 17 Element-wise Products using the Room Compensation filters

Input: $\mathbf{Q}$, $\mathbf{F}$ l_o, N, M, S_{elect}
Output: $\mathbf{O}$,

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int Row = BlockIdx.y * BlockDim.y + ThreadIdx.y;
3: int High = BlockIdx.z * BlockDim.z + ThreadIdx.z;
4: HighMod = High % (N); // N can not be power of two
5: int index = Col + Row*l_o;
6: int indexTot = index + High*l_o*N;
7: int index1 = index + HighMod*N*l_o + S_elect*N*M*l_o;
8: O[indexTot] = ComplexMult(Q[index1],F[indexTot]);
```

Variables used in CUDA kernel 17

- Matrix $\mathbf{F}$ has a tridimensional structure whose dimensions are ($N \times l_o \times N$). This matrix stores the frequency responses of the N^2 filters

that compose the inverse filter bank.

- Matrix **O** has the same tridimensional structure as matrix **F** and is devoted to store the element-wise products of (6.12).

CUDA kernel 18

This kernel performs the accumulation of $\mathbf{Y}_{\text{buff}_n}$ in (6.12). The number of blocks that this kernel launches is $\frac{l_o}{128} \times N \times 1$, being the block size $128 \times 1 \times 1$. Each thread carries out N sums of complex numbers. Matrix **O** contains the frequency bins of $\mathbf{Y}_{\text{buff}_n}$ in its first $N \cdot l_o$ elements.

CUDA Kernel 18 Element-wise Sum using Room Compensation filters

Input: **O**, l_o, N, M, S_{elect}

Output: **O**,

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int Row = BlockIdx.y * BlockDim.y + ThreadIdx.y;
3: int index = Col + Row*l_o;
4: O[index] = ComplexSum(O[index],O[index + k*l_o*N]);
```

Movement virtualization in CUDA kernels

In case the fading is carried out, CUDA kernel 15 launches double number of threads, since double of driving signals must be computed. This is achieved by setting variable S_{elect}=0 and by using this CUDA grid: $\frac{l_o}{128} \times N \times 2M$, being the block size $128 \times 1 \times 1$.

Additionally, CUDA kernel 16 is substituted by the following operations

- $2NM$ iFFTs of size l_o are executed to obtain all $\mathbf{q}^{mA}_{\text{buff}_n}$ and $\mathbf{q}^{mB}_{\text{buff}_n}$ of (6.13).
- **CUDA kernel 19** computes the element-wise multiplications of (6.13). To this end, $2NMl_o$ threads are launched. Each thread performs a multiplication.
- **CUDA kernel 20** performs the accumulation of (6.13) and (6.16) by using Nl_o threads.
- N FFTs of size l_o are carried out to obtain all $\mathbf{Q}_{\text{buff}_n}$.

Finally, CUDA kernel 17 and CUDA kernel 18 are launched. It is important to point out that the necessary FFT and iFFT transformations are carried out by the Nvidia FFT library CUFFT [20] on the GPU.

CUDA kernel 19

The number of blocks that this kernel launches is $\frac{Nl_o}{64} \times M \times 1$, being the block size $64 \times 1 \times 2$.

CUDA Kernel 19 Element-wise Sum using Room Compensation filters

Input: **Q**, l_o, N, M, S_{elect}, **ch**
Output: **O**,

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int Row = BlockIdx.y * BlockDim.y + ThreadIdx.y;
3: int ColMod = Col & (l_o-1);
4: int index = Col + Row*l_o*N + ThreadIdx.z*l_o*M*N;
5: int index2 = ColMod + ThreadIdx.z*l_o + 2*l_o*S_elect;
6: if(s[Row])
7:     O[index] = ComplexMult(O[index], ch[index2]);
8: end if
```

Variables used in code of CUDA kernel 19

- Vector **ch** is composed of $4l_o$ elements and contains both fading vectors **f** and **g** twice, i.e **ch** = [**f** **g** **g** **f**]. This kernel applies the fading to the driving signals. Thus, in case the fading changes from Position A to Position B, then S_{elect}=0. In case the fading changes from Position B to Position A, then S_{elect}=1.

- Vector **s** is composed of M elements and its values can take only two values: 0 and 1. It has the same role as in CUDA kernel 11.

CUDA kernel 20

The number of blocks that this kernel launches is $\frac{Nl_o}{128} \times 1 \times 1$, being the block size $128 \times 1 \times 1$.

CUDA Kernel 20 Application of the fading vectors

Input: **O**, **s**, S_{elect}
Output: **O**

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int index1 = Col + S_elect*N*M*l_o;
3: int index2 = Col + N*M*l_o - S_elect*N*M*l_o;
4: if(s[0])
5:    O[index1] = ComplexSum(O[index1],O[index2]);
6: end if
7: for k = 1,...,M-1 do
8:    O[index1] = ComplexSum(O[index1], O[index1 + k*N*l_o]);
9:    if(s[k])
10:      O[index1] = ComplexSum(O[index1], O[index2 + k*N*l_o]);
11:   end if
12: end for
```

6.3 Performance and results

Figure 6.6(a) presents the time t_{proc} as the number of virtual sound sources increases in a WFS system, with and without a RC block, and for three different GPU accelerators. In the sequel, we refer to them by their architectures name. Analyzing Fig. 6.6(a), it can be appreciated the performance difference when a RC block is added. If RC block is not applied, all GPUs achieve configurations that work in real time. The Tesla achieves rendering in real time up to 50 sources, the Fermi up to 80 and the Kepler up to 300. However, The Tesla can not achieve any configuration that works in real time when a RC block is added. It is important to point out that this GPU accelerator belongs to a notebook. For the Fermi and Kepler architecture, the maximum number of rendering sound sources is 40 and 260, respectively. The additional processing with the RC block affects noticeably to Tesla and Fermi since they have less physical CUDA cores and less schedule units, while Kepler performance is hardly affected.

On the other hand, Fig. 6.6(b) shows the achieved performance in case a fading is carried out and all the sound sources move at the same time. Comparing this illustration with Fig. 6.6(a), we can appreciate that the influence of using a fading is more meaningful than simply adding a RC block. If a RC block is not applied, the Tesla can render barely up to 8

sources in real time, the Fermi up to 14 sources and the Kepler up to 30 sources. Again, when a RC block is applied, the Kepler performance is hardly affected and is able to render up to 28 sources in real time, while the Fermi achieves up to 8 sources in real time.

The tasks assigned to each thread in the different kernels allow to carry out a fine-grain parallelism. This kind of assignment benefits from the number of cuda cores of the GPU. Thus, the Kepler achieves best performance. Moreover, this architecture has four warp schedulers per SM, what allows this GPU to execute more than one launched thread in one physical core at the same time. This is because the *warp scheduler* overlaps memory access times of some threads with the execution of others. As the Tesla and Fermi have less *warp schedulers*, they cannot so easily overlap these times.

6.4 Conclusion

The new emerging GPU architectures such as Nvidia Kepler has allowed us to face up different computational problems in the audio processing that had not been approached before. In this chapter, we have developed a WFS rendering system that uses a Room Compensation block and that is able to render up to 300 virtual sound sources in real time. The main feature of this system is that all its audio processing was carried out by a GPU, while the CPU could be being used for other tasks at the same time. Therefore, the use of the Nvidia hardware provides a good solution to build multisource spatial inmersive systems that require high computational capacity. More details can be found in [122]

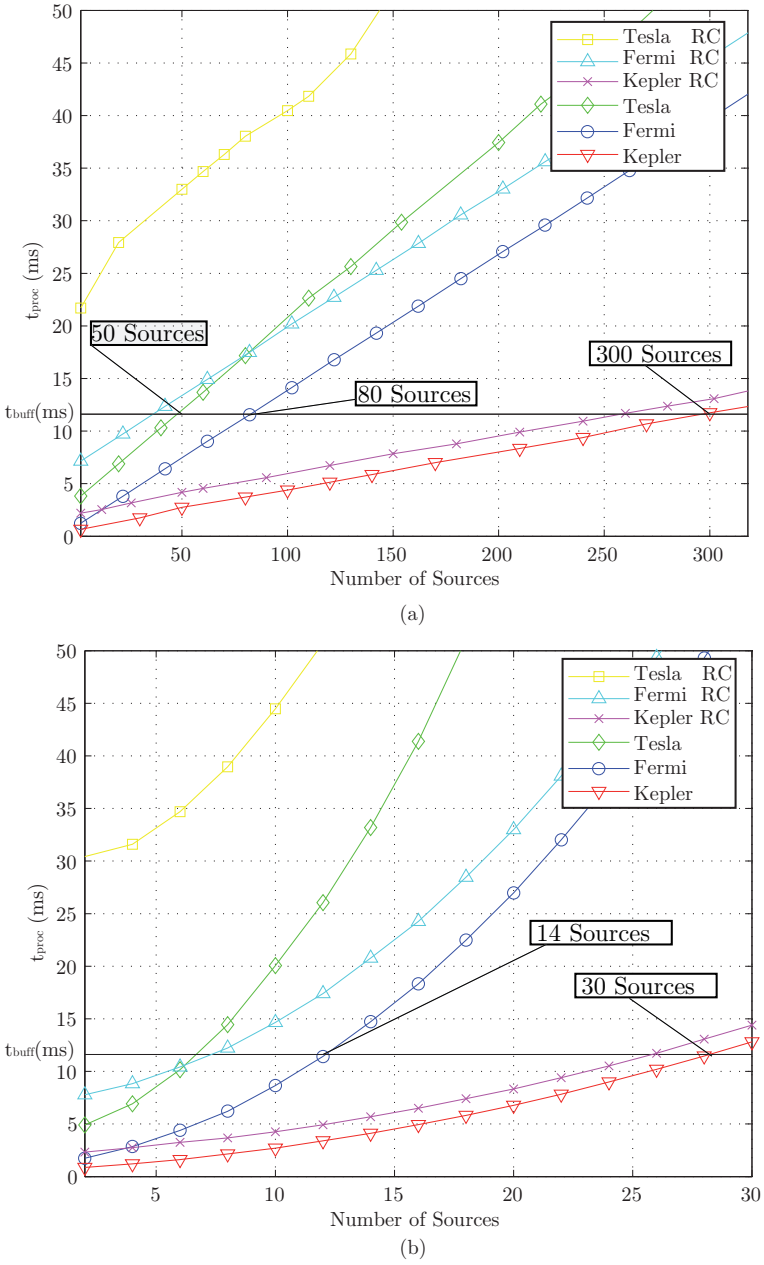


Figure 6.6. Processing time for different number of sound sources that are rendered in a spatial audio system (WFS + RC) without fading processing (a) and with fading processing (b) on different GPUs.

7 Sound Source Localization

7 Sound Source Localization

Sound source localization is an important topic in microphone array signal processing applications, such as automatic camera steering systems, human-machine interaction, video gaming or surveillance systems. The Steered Response Power with Phase Transform (SRP-PHAT) algorithm is a well-known approach for sound source localization due to its robust performance in noisy and reverberant environments. This algorithm analyzes the sound power captured by an acoustic beamformer on a defined spatial grid, estimating the source location as the point that maximizes the output power. Since localization accuracy can be improved by using high-resolution spatial grids and a high number of microphones, accurate acoustic localization systems require high computational power. GPUs offer multiple parallelism levels; however, properly managing their computational resources becomes a very challenging task. In fact, management issues become even more difficult when multiple GPUs are involved, adding one level more of parallelism. In this chapter, the performance of an acoustic source localization system using distributed microphones is analyzed over a massive multichannel processing framework in a multi-GPU system. In this context, we evaluate both localization and computational performance in different acoustic environments, always from a real-time implementation perspective.

7.1 Introduction

Sound source localization plays an important role in a large-scale number of applications such as human-computer interfaces, teleconferencing or robot artificial audition. Localization becomes difficult when sound sources are surronded by high noise and reverberation.

This chapter is aimed at demonstrating how localization systems using a high number of microphones distributed within a room can perform real-time sound source localization in adverse environments by using GPU massive computation resources. We discuss how massive signal processing for sound source localization can be efficiently performed by Multi-GPU systems, analyzing different performance aspects on a set of simulated acoustic environments. Specifically, the well-known SRP-PHAT algorithm is considered here. Coarse-to-fine search strategies have been proposed to overcome many of the processing limitations of SRP-PHAT [123, 124, 125]. However, while these strategies provide more efficient ways to explore the localization search volume, they only provide better performance than the conventional SRP-PHAT when the number of operations is restricted. Thus, the performance of the conventional SRP-PHAT with fine spatial grids is usually considered as an upper bound in these cases.

Relevant parameters that affect the computational cost of the algorithm (number of microphones and spatial resolution) are analyzed, showing their influence on the localization accuracy in different situations. We also discuss the scalability of the algorithm when multi-GPU parallelization issues are considered. The conclusions of this chapter highlights the need for massive computation in order to achieve high-accuracy localization in adverse acoustic environments, taking advantage of GPUs to fulfill the computational demands of the system.

7.2 Sound Source Localization using SRP-PHAT Algorithm

Consider the output from microphone m, $b_m(t)$, in an M microphone system. The Steered Response Power (SRP) at the spatial point $\mathbf{x} = [x, y, z]^T$

for a time frame r of length T_L is defined as

$$P_r(\mathbf{x}) \equiv \int_{rT_L}^{(r+1)T_L} \left| \sum_{m=1}^{M} w_m b_m \left(t - \tau(\mathbf{x}, m)\right) \right|^2 dt, \tag{7.1}$$

where w_m is a weight and $\tau(\mathbf{x}, m)$ is the direct time of travel from location $\mathbf{x}$ to microphone m. DiBiase [126] showed that the SRP can be computed by summing up the Generalized Cross-Correlations (GCCs) for all possible pairs of the set of microphones. The GCC for a microphone pair (r, m) is defined as

$$R_{b_r b_k}(\tau) = \int_{-\infty}^{\infty} \Phi_{rm}(\omega) B_r(\omega) B_m^*(\omega) e^{j\omega\tau} d\omega, \tag{7.2}$$

where τ is the time lag, * denotes complex conjugation, $B_m(\omega)$ is the Fourier transform of the microphone signal $b_m(t)$, and $\Phi_{rml}(\omega)$ is a combined weighting function in the frequency domain. The phase transform (PHAT) [75] has been shown to be a suitable GCC weighting for time delay estimation in reverberant environments. The PHAT weighting is expressed as:

$$\Phi_{rm}(\omega) \equiv \frac{1}{|B_r(\omega) B_m^*(\omega)|}. \tag{7.3}$$

Taking into account the symmetries involved in the computation of (7.1) and removing some fixed energy terms [126], the part of $P_r(\mathbf{x})$ that changes with $\mathbf{x}$ can be isolated as

$$P'_r(\mathbf{x}) = \sum_{r=1}^{M} \sum_{m=r+1}^{M} R_{b_r b_k}\left(\tau_{rm}(\mathbf{x})\right), \tag{7.4}$$

where $\tau_{kl}(\mathbf{x})$ is the *inter-microphone time-delay function* (IMTDF). This function is very important, since it represents the theoretical direct path delay for the microphone pair (r, m) resulting from a point source located at $\mathbf{x}$. The IMTDF is mathematically expressed as [127]

$$\tau_{rm}(\mathbf{x}) = \frac{\|\mathbf{x} - \mathbf{x}_r\| - \|\mathbf{x} - \mathbf{x}_m\|}{c}, \tag{7.5}$$

where c is the speed of sound (≈ 343 m/s), and $\mathbf{x}_r$ and $\mathbf{x}_m$ are the locations of the microphone pair (r, m).

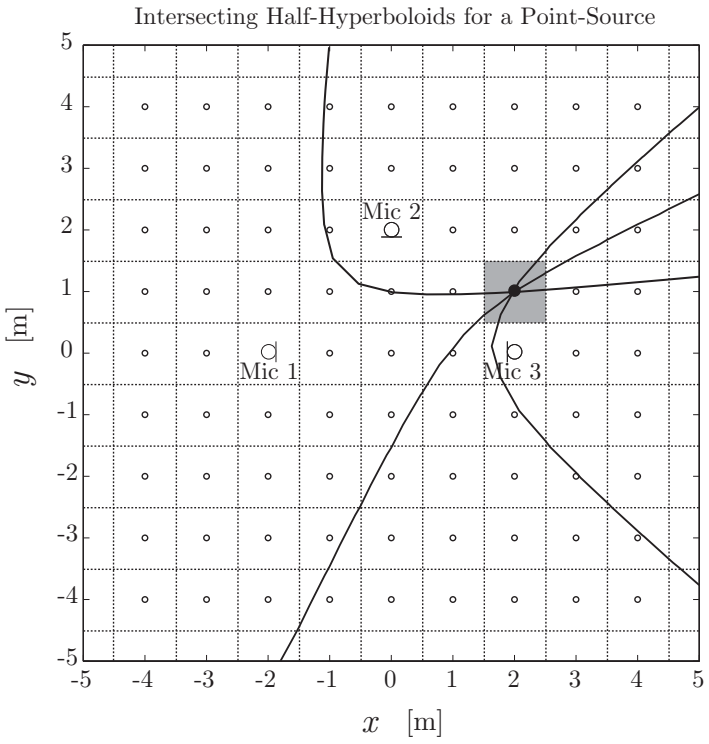


Figure 7.1. Intersecting half-hyperboloids for $M = 3$ microphones. Each half-hyperboloid corresponds to a TDOA peak in the GCC.

The SRP-PHAT algorithm consists in evaluating the functional $P'_r(\mathbf{x})$ on a fine grid $\mathcal{G}$ with the aim of finding the point-source location $\mathbf{x}_s$ that provides the maximum value:

$$\mathbf{x}_s = \arg\max_{\mathbf{x}\in\mathcal{G}} P'_n(\mathbf{x}). \tag{7.6}$$

Figure 7.1 shows schematically the intuition behind SRP-PHAT localization. In this figure, an anechoic environment is assumed so that the GCC for each microphone pair is a delta function located at the real Time Difference of Arrival (TDOA). Each TDOA defines a half-hyperboloid of potential source locations. The intersection resulting from all the half-hyperboloids matches the point of the grid having the greatest accumulated value.

7.2.1 SRP-PHAT Implementation

The SRP-PHAT algorithm is usually implemented on a grid by carrying out the following steps:

1. A spatial grid $\mathcal{G}$ is defined with a given spatial resolution r_{sp}. The theoretical delays from each point of the grid to each microphone pair are pre-computed using (7.5).

2. For each analysis frame, the GCC of each microphone pair is computed as expressed in (7.2).

3. For each position of the grid $\mathbf{x} \in \mathcal{G}$, the contribution of the different cross-correlations are accumulated (using delays pre-computed in 1), as in (7.4).

4. Finally, the position with the maximum score is selected as in (7.6).

The SRP-PHAT localization performance depends on the selected spatial resolution r_{sp}. Figure 7.2 illustrates the algorithm performance when considering different spatial grid resolutions. The accumulated SRP-PHAT values for each spatial grid location are shown for a 2-D plane in a 4×6 m room with $M = 6$ microphones. Note how the location of the source is more easily detected when finer spatial resolutions are used, as in the case of $r_{sp} = 0.01$ m.

7.2.2 Computational Cost

The SRP-PHAT algorithm is usually implemented by performing a frequency-domain processing of the input microphone signals. Given M microphones, the number of microphone pairs to process is $Q = M(M-1)/2$. For a DFT size of L (equal to the time-window size), an FFT takes $5L \log_2 L$ arithmetic operations that result from $\frac{L}{2} \log_2 L$ complex multiplications and $L \log_2 L$ complex additions. Note that one complex multiplication is equivalent to four real multiplications and one real addition, while a complex addition is equivalent to two real additions. As a result, the signal processing cost for computing the GCC is given by:

- **DFT**: Compute M FFTs, then, $M \times 5L \log_2 L$.

- **Cross-Power Spectrum**: A complex multiplication for L points, resulting in $6L$ operations (4 real multiplications and 2 real additions). This is done for Q microphone pairs, resulting in a cost of $6QL$.

- **Phase Transform**: Magnitude of the L points of the GCC, which costs L operations. This is also done for Q pairs, resulting in QL operations.

- **IDFT**: The IDFT for Q pairs must be performed, which requires $Q5L\log_2 L$ operations.

Moreover, for each functional evaluation, the following parameters must be calculated:

- M Euclidean distances, $\|\mathbf{x}_m\|$, requiring 3 multiplications, 5 additions and 1 square root (≈ 12 operations): $20M$ operations

- Q TDOAs, requiring 2 operations (1 subtraction and 1 division by c) per microphone pair: $2Q$ operations.

- The SRP requires truncating the TDOA values to the closest sample according to the system sampling frequency, multiplying the cross-power spectrum to obtain the phase transform for each microphone pair and adding up all the GCC values: $5Q$ operations.

As a result, the cost of the SRP-PHAT (number of operations) is given by:

$$\begin{aligned} Cost &= \left(\frac{M+M^2}{2}\right)5L\log_2 L + \\ \frac{7M(M-1)}{2}L &+ \nu\left(20M+\frac{7M(M-1)}{2}\right), \end{aligned} \tag{7.7}$$

where ν is the total number of functional evaluations. In the conventional full grid-search procedure, ν equals the total number of points of the grid $\mathcal{G}$. Figure 7.3 shows the computational cost of the algorithm for different spatial resolutions and number of microphones, considering a 3D grid search space with a uniform spatial resolution of r_{sp} meters.

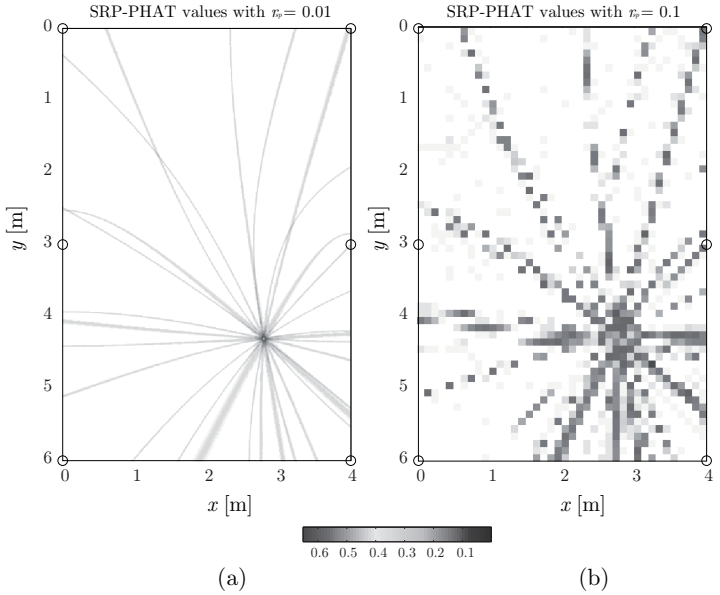


Figure 7.2. Accumulated SRP-PHAT values for a 2-D spatial grid (4×6 m and $M = 6$ microphones) with different spatial resolutions. (a) $r_{sp} = 0.01$ m. (b) $r_{sp} = 0.1$ m.

7.3 Algorithm Parallelization for real-time GPU implementation

The GPU-based implementation of the SRP-PHAT algorithm is applied to Nvidia hardware devices with Kepler architecture GK110 [37].

Since the localization is carried out in three dimensions, three different resolutions r_{sp}^x, r_{sp}^y, and r_{sp}^z define the spatial grid $\mathcal{G}$. Taking a shoe-box-shaped room as a model room with dimensions $l^x \times l^y \times l^z$, the size of the grid is $\nu = P^x \times P^y \times P^z$, where $P^x = \frac{l^x}{r_{sp}^x}$, $P^y = \frac{l^y}{r_{sp}^y}$ and $P^z = \frac{l^z}{r_{sp}^z}$.

The real-time implementation of the SRP-PHAT algorithm uses processing blocks of size $l_o = 2L$ with 50%. Also, the audio card provides audio sample buffers of size L per each microphone each $\frac{L}{f_s}$ s. These ML samples are transferred to the GPU first. A GPU buffer (denoted here as $\mathbf{T}_{GPU}$) stores the audio samples in consecutive memory positions as they arrive to the GPU. One aspect that affects the performance for all audio

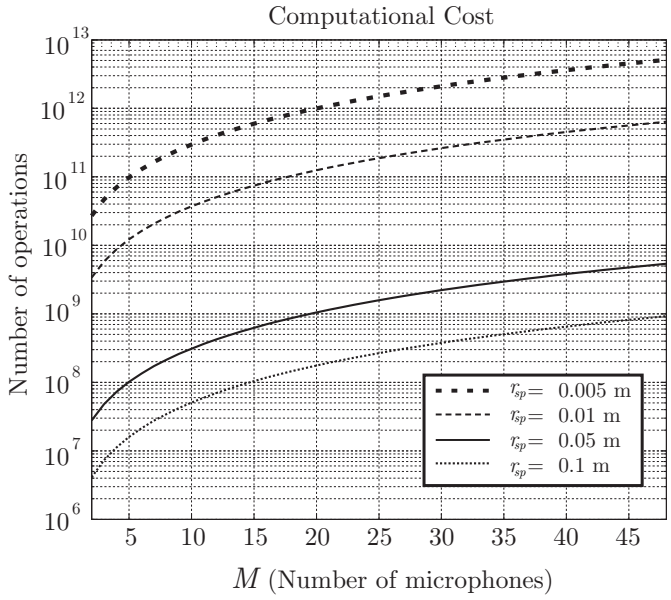


Figure 7.3. Computational cost when for different number of microphones M and spatial resolutions r_{sp}.

signal processing applications on GPU is the transfer of audio samples from CPU to GPU. As mentioned in Section 2.6.1, in Chapter 2, *streams* can be used to parallelize these transfers and overlap them with the computation. The processing is carried out in blocks of size $2L$ which are composed of the current audio-sample buffer and the previous one. Thus, a size of $2LM$ is used for $\mathbf{T}_{GPU}$. Therefore the SRP-PHAT GPU implementation carries out the following steps:

1. M *streams* are created (one stream for each microphone in the system). The *streams* are launched consecutively in an asynchronous way. *Stream* m transfers L samples captured by microphone m to the GPU and stores them in $\mathbf{T}_{GPU}$, with $m = 0, \ldots, M-1$. Then, *stream* m launches CUDA kernel 21, which is responsible for grouping $l_o = 2L$ elements of microphone m (L samples from previous buffers and L samples from current buffers). These l_o elements are also weighted using a Hamming window vector. For this purpose, the *stream* launches a kernel that is composed of 128-size thread blocks in

a CUDA grid of dimensions ($\frac{l_o}{128} \times 1$) (i.e., it is composed of l_o CUDA threads). Each thread computes one element of the l_o elements.

CUDA kernel 21

CUDA Kernel 21 Grouping l_o elements and Multiply by Hamming Windows

Input: $\mathbf{T}_{GPU}$, $\mathbf{w}$, l_o, I

Output: $\mathbf{F}$,

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int ipos = (Col +  I*L ) & (l_o-1);
3:  // Element-wise multiplication by a Hamming Windows
4: F[Col].x = ((float)T_GPU[ipos]*w[Col];
5: F[Col].y = 0.0;
```

The tasks carried out by CUDA kernel 21 are simple. Each thread reads one value from *global-memory*, multiplies it by a `float` number (a value of Hamming window vector) and stores it in a different position of *global-memory*. The accesses to *global-memory* are totally coalesced, since audio samples are stored in consecutive memory positions both when reading and when writing (see Fig. 7.4). Also, L is a power of 2 and is always larger than 1024. Thus, each thread block reads and writes in 128 consecutive memory positions. The selection of 128 for the block size was done experimentally between 64, 128, 256 and 512, with 128 being the one that requires less time.

Vector $\mathbf{w}$ is composed of l_o elements that corresponds to a Hamming window of size l_o. Variable I points the audio samples to use from $\mathbf{T}_{GPU}$. Matrix $\mathbf{F}$ has dimensions ($M \times l_o$) and stores the audio samples of all the microphones once they have been multiplied by the hamming window. Variable I and the access to the rows of matrix $\mathbf{F}$ are manipulated from the main programm.

2. Once CUDA kernel 21 has finished, *stream* m uses the CUFFT library to perform a FFT of size l_o using these l_o elements. As a result of the computation performed by all the *streams*, M vectors that are composed of l_o frequency bins (denoted as $\mathbf{f}_m$, $m = 0, \ldots, M-1$ and stored in matrix $\mathbf{F}$) are obtained. The use of *streams* allows us to overlap data transfers with computation. For example, while *stream* 1 is transferring samples from microphone 1, *stream* 0 can be executing CUDA kernel 21. However, the next steps involve operations

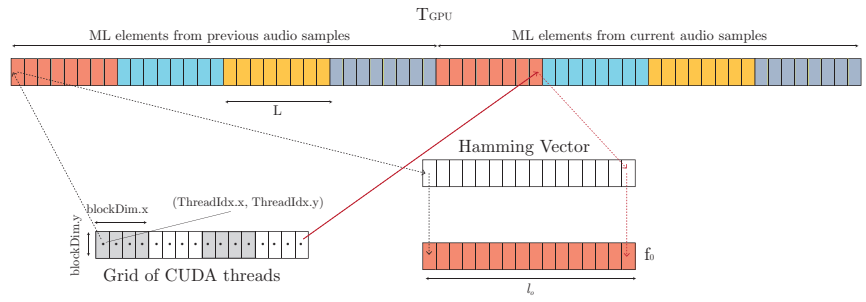


Figure 7.4. Operations that are carried out by CUDA kernel 21 in case M=4.

among different channels. Thus, all the previous operations must finalize before continuing. This implies synchronization among all the *streams*. The following steps are computed by only one *stream*.

3. The **GCC** matrix is computed by means of another kernel (CUDA kernel 22). In this kernel, a GPU thread takes one value from each of two different $\mathbf{f}_m$ buffers that are at the same vector position. It conjugates one of the values and multiplies it by the corresponding value of the other $\mathbf{f}_m$ buffer. The phase of the complex number obtained by the multiplication is stored in the corresponding position in the **GCC** matrix.

CUDA kernel 22

CUDA Kernel 22 Configuration of **GCC** matrix

Input: $\mathbf{F}$, $\mathbf{P}_{airs}$, l_o,
Output: **GCC**,

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int Row = BlockIdx.y * BlockDim.y + ThreadIdx.y;
3: Complex Caux;
4: int index;
5: int ind1;
6: while(Row < Q)
7:    index = Col + Row*BlockDim.x*gridDim.x
8:    ind1 = Col + P_airs[2*Row]*(blockDim.x)*(gridDim.x);
9:    ind2 = Col + P_airs[2*Row + 1]*(blockDim.x)*(gridDim.x);
```

```
10:    Caux = ComplexMult( ind1 ,ind2 );
11:    GCC[index].x = cosf( atan2f( Caux.y, Caux.x) );
12:    GCC[index].y = sinf( atan2f( Caux.y, Caux.x) );
13:    Row = Row + gridDim.y;
14: end while
```

The accesses to the two $\mathbf{f}_m$ buffers by GPU threads are totally coalesced since consecutive threads access consecutive memory positions (see Fig. 7.5). CUDA kernel 22 is limited by the instruction bandwidth since GPU-native functions `cosf`, `sinf`, and `atan2f` are used and all of them require several clock cycles. CUDA kernel 22 computes Ql_o values of the **GCC** matrix, where Q represents the number of microphone pairs. In order to define the size and the number of blocks to launch in CUDA kernel 22, different tests were executed. Less execution time was achieved by using 128-size thread blocks in a CUDA grid with size 32×16. This implies launching 65536 threads, where each thread is responsible for computing $\frac{Ql_o}{65536}$ values of the **GCC** matrix. In this case, increasing the amount of work per thread block in CUDA kernel 22 is more beneficial than launching more blocks with fewer operations per GPU thread. Thus, the grid configuration applied to CUDA kernel 22 achieves maximum occupancy when 512 blocks are launched. This kernel does not require using *shared-memory* but preferably a large number of registers. Thus, we set L1 cache to 48 KB. As described in [37], cache L1 is used for register spills, local memory, and stack, which are all private per-thread variables.

Matrix $\mathbf{P}_{airs}$ has dimensions $(Q \times 2)$. Each row points out the combination of pairs that are used in the operations. This matrix is fixed and is stored at the GPU *constant-memory*.

4. Q inverse FFTs of size l_o are carried out by using again the CUFFT library. The **GCC** matrix is now composed of temporal (time delay) values (i.e., Ql_o real values).

5. The computation of a tridimensional matrix **SRP** storing the accumulated SRP values is carried out by CUDA kernel 23. This kernel also launches thread blocks of size 128 in a tridimensional CUDA grid whose dimension depends on the number of points of the grid $\mathcal{G}$ (ν).

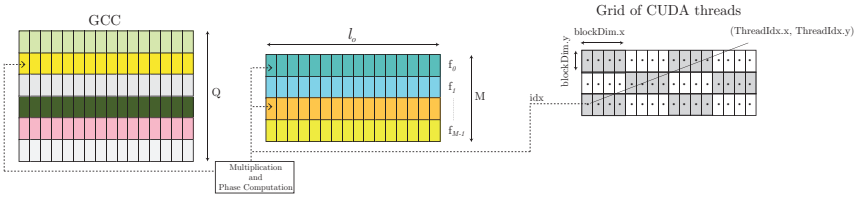


Figure 7.5. Operations that are carried out by CUDA Kernel 22.

In total, ν threads are launched. In this kernel, each GPU thread is devoted to the computation of the total value of the SRP at each point of the grid. To this end, each thread computes and accumulates Q GCC values (it takes a value from each row of the **GCC** matrix and accumulates it). The computation of the SRP requires Q calculations of the IMTDF (7.5) at each point of the grid. The IMTDF of a pair of microphones specifies the column of the **GCC** matrix that should be selected and then accumulated in the SRP. Figure 7.6 illustrates these operations.

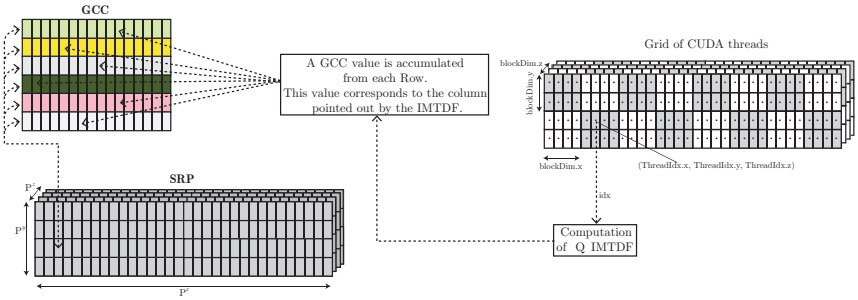


Figure 7.6. Operations that are carried out by CUDA kernel 23.

Since the value of the IMTDF can indicate any position of the column of the **GCC** matrix, coalesced access to the *global-memory* is not guaranteed. In fact, the most probable situation is that the accesses will be quite disordered, so that the kernel employs most of its time in memory accessing. However, this limitation can be reduced if we force the compiler to use the Kepler read-only data cache with the **GCC**

matrix, since this cache does not require aligned accesses. This read-only cache memory has also been used in recent GPU-based audio research such as [81] and [128]. Furthermore, as in CUDA kernel 22, we set L1 cache to 48 KB to favor possible register spills. In the accumulation loop of the SRP values, we have set a `#pragma unroll` to accelerate the computation.

CUDA kernel 23

CUDA Kernel 23 Configuration of **SRP** matrix

Input: GCC, P^x, P^y

Output: SRP,

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int Row = BlockIdx.y * BlockDim.y + ThreadIdx.y;
3: int High = BlockIdx.z * BlockDim.z + ThreadIdx.z;
4: float faux = 0.0;
5: int index = Col + (Row)*P^x + High*P^x*P^y;
6: if(Col < (P^x*P^y)) do
7:    index = Col + Row*BlockDim.x*gridDim.x
8:    #pragma unroll
9:    for k = 0,...,Q-1
10:      faux += GCC[TimeDelay(Col, Row, High, k) + k*l_o].x;
11:   end for
12:   SRP[index] = faux;
13: end if
///---Code of GPU FUNCTION : TimeDelay---/////
```

Input: `Col`, `Row`, `High`, `k`, r_{sp}^x, r_{sp}^y, $\mathbf{P}_{os}$, $\mathbf{P}_{airs}$, $\mathbf{D}_z$

Output: `out`,

```
1: float fgx = r_sp^x/2 + Col*r_sp^x;
2: float fgy = r_sp^y/2 + Row*r_sp^y;
3: float fgz = D_z[High];
4: float x1 = P_os[3*P_airs[2*k]];
5: float y1 = P_os[3*P_airs[2*k] + 1];
6: float z1 = P_os[3*P_airs[2*k] + 2];
7: float x2 = P_os[3*P_airs[2*k + 1]];
8: float y2 = P_os[3*P_airs[2*k + 1] + 1];
9: float z2 = P_os[3*P_airs[2*k + 1] + 2];
10: faux1 = sqrtf( (fgx - x1)^2 + (fgy - y1)^2 + (fgz - z1)^2 );
```

```
11: faux2 = sqrtf( (fgx - x2)^2 + (fgy - y2)^2 + (fgz - z2)^2 );
12: out = rintf(f_s/c * (faux1 - faux2) );
```

6. The grid position corresponding to the maximum SRP value has to be searched. To this end, we launch CUDA kernel Reduction. This kernel exactly follows the reduction example in Harris' implementation [129] that comes with the Nvidia GPU Computing SDK (Software development kit), but it changes the sum operation for a maximum operation. However, even though this code is optimized for finding the maximum value, it does not indicate its position. Thus, after obtaining the maximum, we launch another kernel (CUDA kernel 24). This kernel launches as many threads as elements of the **SRP** matrix and only performs a comparison operation with the maximum. If the comparison matches, the thread writes the value of its index in a variable. We confirmed experimentally that the combination of these two kernels is faster than storing the positions of different intermediate SRP maximums.

   CUDA kernel 24

CUDA Kernel 24 Localization of Maximum SRP value

Input: **SRP**, m_{ax}, P^x, P^y, P^z

Output: P_{osMax},

```
1: int Col = BlockIdx.x * BlockDim.x + ThreadIdx.x;
2: int p = P^x*P^y*P^z;
3: if(SRP[Col] == m_ax) && (Col < p) do
4:     P_osMax = Col;
5: end if
```

   The code of CUDA kernel Reduction can be found at the SDK in the webpage [20]. Part of this code will be used in the future code CUDA kernel 25 which is shown in Chapter 8.

7.3.1 Considerations in code of CUDA kernels 23 and 24

The computation of the IMTDF could be carried out off-line since the grid resolutions and the microphone locations are static. However, this would imply storing a 4-dimensional data structure composed of $\nu \cdot Q$ elements.

If we use a standard room size (such as 6.0 × 4.0 × 3.0 m), a resolution of $r_{sp} = r_{sp}^x = r_{sp}^y = r_{sp}^z = 0.01$ m, and $M = 48$ microphones, this data structure would require using more than eight gigabytes of *global-memory*. This exceeds the *global-memory* size of most available GPU devices at the time of writing this dissertation. Thus, every IMTDF value is computed for each group of processed buffers. These are the variables used in CUDA kernel 23:

- Matrix $\mathbf{P}_{os}$ has dimensions $(Q \times 3)$. Each row points out the position of each microphone inside the room. This matrix is fixed and is stored at the GPU *constant-memory*, as Matrix $\mathbf{P}_{airs}$
- Vector $\mathbf{D}_z$ has P^z . We store the positions of the z-axis because the resolution r_{sp}^z presents different values in comparison with r_{sp}^x and r_{sp}^y in the tests that will be described in Section 7.4.
- Variables f_s and c represents the samples frequency and the speed of the sound, respectively.
- Variables m_{ax} and P_{osMax} represents the maximum SRP value and its position inside matrix **SRP**.

There are also other variables that are used to compute the values of the **GCC** and **SRP** matrices, such as the room dimensions, the number of microphones and their position. Since all of these read-only variables must be available for all of the threads, they are stored in the *constant memory* (with size 64 KB).

7.3.2 Multi-GPU Parallelization

Distributing the above processing tasks among different GPUs is not straightforward. The greatest computational load relies on Step 6, which consists in computing the maximum value of the **SRP** matrix. Table 7.1 shows the elapsed time corresponding to each step for M =48 microphones and a spatial grid resolution of r_{sp} =0.01 m.

The tasks from CUDA kernels 23, Reduction and 24 can be easily distributed among N_{GPU} GPUs (the number of GPUs present in the system): each GPU computes $\frac{\nu}{N_{GPU}}$ elements of the **SRP** matrix and locates the maximum among its computed elements. To this end, N_{GPU} CPU threads

Table 7.1. Elapsed Time in each kernel with M=48 and spatial grid resolution r=0.01.

Steps of the algorithm	**Time** [ms]
Transfer + CUDA Kernel 21 + FFT (steps 1 and 2 in Section III)	1.416
CUDA Kernel 22 (step 3: Computation of GCC)	0.015
IFFT of GCC (step 4)	0.006
CUDA Kernel 23 (Computation of SRP matrix)	0.007
CUDA Kernel Reduction (Computation of Maximum SRP value)	121.267
CUDA Kernel 24 (Localization of the Maximum)	0.009
Total elapsed time	122.720

are created at the beginning of a parallel region by means of *openMP* (see Section 2.6.2 in chapter 2 to know how *openMP* can deal with multiple GPUs). This strategy is only focused on multi-GPU parallelization of the **SRP** matrix.

In Appendix A.1, there is a description of an alternative strategy that aims at parallelizing both the computation of the **SRP** matrix and of the **GCC** matrix. This strategy uses also the UVA feature (Unified Virtual Addressing, more details in Section 2.6.2, Chapter 2) for inter-GPU communication. This strategy requires different synchronization points that significantly penalize their performances, specially when compared to the recently described parallelization.

7.3.3 Basic Implementation using two GPUs

As shown in Section 7.4, the performance of the SRP-PHAT algorithm is assessed in a system that is composed of two GPUs. Using all the parallelization techniques previously presented, the SRP-PHAT algorithm is implemented on two GPUs as follows:

1. A parallel region is created with two CPU threads. Each CPU thread is bound with a GPU.

2. Since different audio buffers are received in the system, each CPU thread independently and asynchronously sends all audio buffers to its GPU by using *stream* parallelization. The CUDA kernels 21 and the FFTs are computed for each channel inside the *streams*.

3. As in step 2 of Section 7.3, *stream* synchronization is addressed. Only one *stream* is used to compute the rows of the **GCC** matrix.

4. Since both of them have computed the **GCC** matrix, each GPU computes $\nu/2$ elements of the **SRP** matrix and locates a maximum value among the computed elements.

5. Each GPU transfers back to the CPU its maximum value and its location inside the **SRP** matrix. Then, a synchronization barrier for both CPU threads is set followed by an *openMP* section that is only executed by the master thread. This thread compares the two maximum values and chooses the greatest one, getting its location. This location indicates the sound source position. Figure 7.7 illustrates the computation of the SRP-PHAT when $M = 12$.

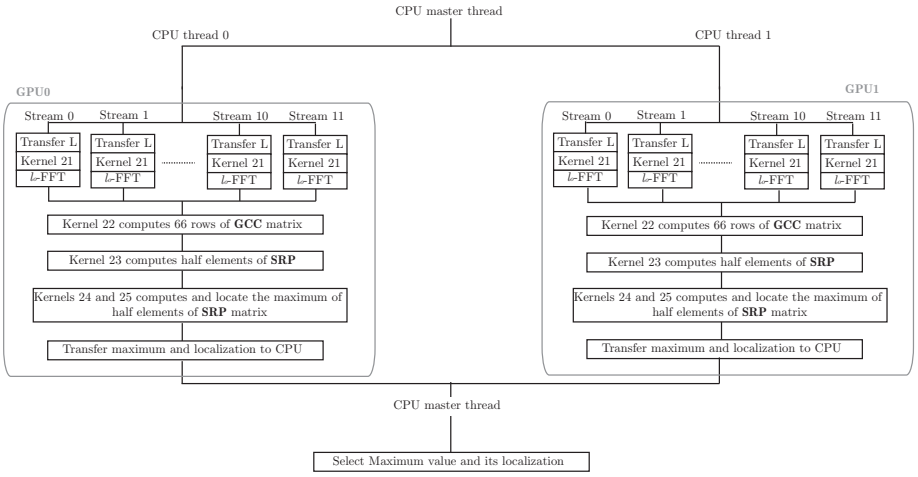


Figure 7.7. Steps of the GPU-based SRP-PHAT implementation using two GPUs and *openMP*.

7.4 Experiments and Performance

To analyze both the computational and localization performance of the above GPU implementations, a set of acoustic simulations using the image-source method [130] have been considered. A shoe-box-shaped room with

dimensions 4×6×3 m and wall reflection factor ρ [120] was simulated using different numbers of microphones ($M \in \{6, 12, 24, 48\}$). The microphone set-up for the considered systems are shown in Fig. 7.8. Note that the microphones are located on the walls of the room and are placed on eight different planes ($z = \{0.33, 0.66, 1.00, 1.33, 1.66, 2.00, 2.33, 2.66\}$) following hexagon-like shapes. Moreover, different reflection factors ($\rho \in \{0, 0.5, 0.9\}$) were used to take into account different reverberation degrees. In all cases, independent white Gaussian noise was added to each microphone signal in order to simulate different *Signal to Noise Ratios* (SNR $\in \{0, 5, 10, 20\}$) (in dB).

The audio card used in the real-time prototype uses an ASIO (Audio Stream Input/Output) driver to communicate with the CPU and provides 2048 samples per microphone (L=2048) every 46.43 ms (sample frequency of 44.1 kHz). This time is denoted by t_{buff}. The time employed for the computation is denoted by t_{proc}. This time takes into account all transfer times and measures the time from the first audio sample transferred to the GPU until the final source location is estimated (at each time frame). The localization system works in real time as long as $t_{\text{proc}} < t_{\text{buff}}$. Otherwise, microphone samples would be lost and the localization would not be correctly performed. The simulations were carried out in the Nvidia GPU K20c [37], which has the characteristics shown in Table 7.2.

Table 7.2. Characteristics of the GPU K20c.

Cuda Device	Tesla K20c
Architecture	Kepler
Capability	3.5
Number of SM	13
Total number of cores	2496
Max. dimension of a block	1024 x 1024 x 64
Max. dimension of a grid	2^{31}-1 x 65535 x 65535
Total amount of global memory	4 GB

Both computational and localization performances have been assessed taking into account three spatial grid resolutions ($r_{sp} \in \{0.1, 0.05, 0.01\}$) in the XY plane (resolutions r_{sp}^x and r_{sp}^y are equal). The resolution r_{sp}^z is

0.33 m (resulting from dividing the height of the room into eight slots).

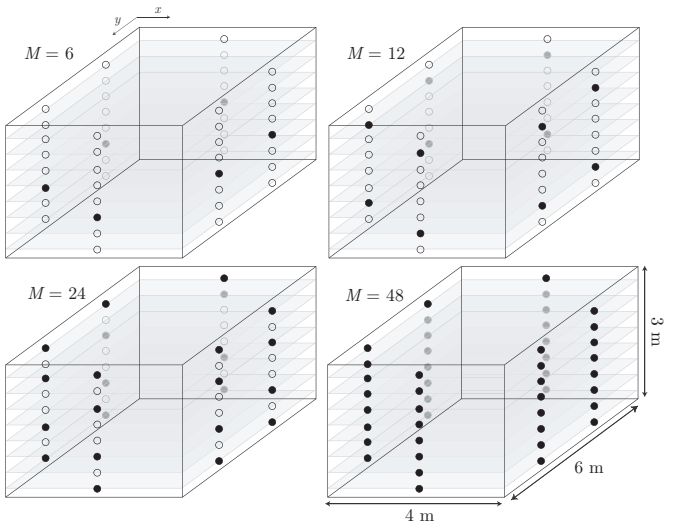


Figure 7.8. Microphone set-ups for $M = 6$, $M = 12$, $M = 24$ and $M = 48$. The black dots denote the actual active microphones in each configuration.

7.4.1 Localization Performance

The source signal used in this study was a 5-second male speech signal with no speech pauses. Pauses were manually suppressed to evaluate localization performance only over frames where speech was present. The processing was carried out by using 50% overlap in time windows of length 4096 samples (size $l_o = 2L$), with sampling frequency $f_s = 44.1$ kHz. For each frame, a source location $\hat{\mathbf{x}} = [\hat{x}, \hat{y}, \hat{z}]^T$ was estimated. A total number of 107 frames (N_f=107) per 40 different positions (N_p=40) that were uniformly distributed over the room space were performed. Localization accuracy was computed by means of the Mean Absolute Error, which is given by:

$$\text{MAE} = \frac{1}{N_f}\frac{1}{N_p}\sum_{i=1}^{N_f}\sum_{j=1}^{N_p}|\mathbf{e}_{ij}|_2, \tag{7.8}$$

where $\mathbf{e}_{ij}$=$\mathbf{x}_{ij} - \hat{\mathbf{x}}_{ij}$, with $\mathbf{x}_{ij}$ and $\hat{\mathbf{x}}_{ij}$ being the true and estimated source locations at a given time frame i and source position j. Note that the above

MAE was computed for each environmental condition (reflection factor and signal to noise ratio), microphone setup and spatial grid resolution. Figure 7.9 shows the results for different values of wall reflection factor ρ taking into account different spatial resolutions and number of microphones.

It is important to point out that using a high number of microphones helps to substantially improve localization accuracy under high noise and reverberation. The error decreases as the SNR increases and/or reverberation decreases (lower ρ). It is important to see how the spatial resolution has an impact when the number of microphones is small. In this case, a coarse spatial grid is not sufficient to correctly find the minimum of the SRP search space, which is more easily detected when the SRP is enhanced by the contributions of additional microphone pairs. In fact, when the number of microphones is 12 or higher, the performance difference between $r_{sp} = 0.01$ and $r_{sp} = 0.1$ is almost negligible. Accuracy differences among different values of ρ are noticiable. It should be emphasized that, under favorable acoustic conditions (high SNR and low ρ), the experimental error is always below the maximum expectable error independently of the number of microphones. Note that the maximum error in anechoic conditions is given by the largest diagonal of the cuboids forming the 3D grid (≈ 0.179 m for $r_{sp} = 0.1$ and ≈ 0.165 m for $r_{sp} = 0.01$). In all cases, the use of a higher number of microphones helps significantly in reducing this error.

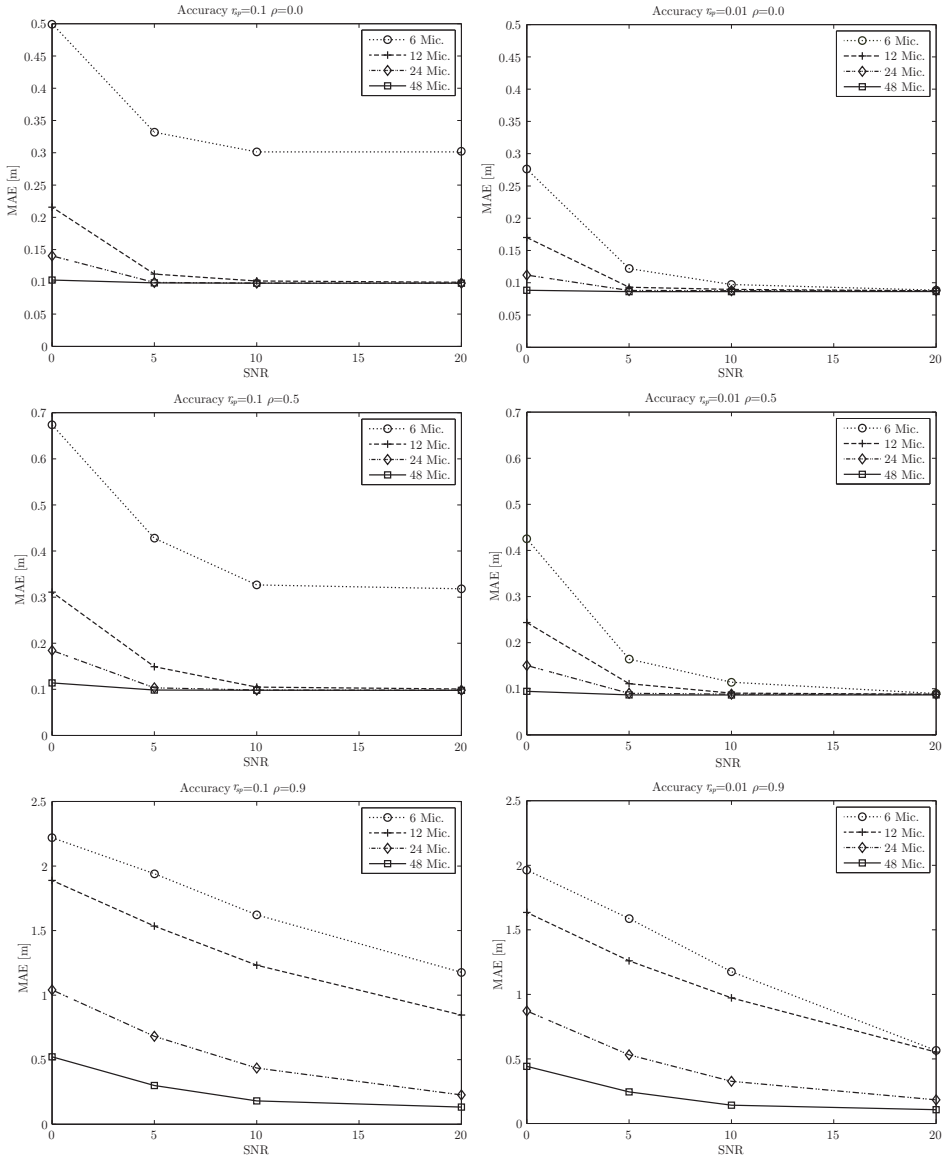


Figure 7.9. Localization accuracy for different wall reflection factors ($\rho \in \{0, 0.5, 0.9\}$) as a function of the SNR and the number of microphones M. Each row presents results for different spatial resolutions ($r_{sp} = 0.01$ and $r_{sp} = 0.1$ m).

7.4.2 Computational Performance

The spatial resolutions considered in this work result in large-scale SRP matrices. While previous works had evaluated up to 360000 points [76], we achieve 1920000 points. Table 7.3 shows the processing times t_{proc} for different combinations of r_{sp} and M when using two GPUs. It can be observed that the only that does not obtain a t_{proc} lower than 46.43 ms (t_{buff}) is the configuration composed of $M = 48$ and $r_{sp} = 0.01$. Thus, real-time processing is not possible in this case. However, by looking at the results shown in Table 7.4, it is possible to observe that the influence of the second GPU becomes relevant. In the case of $M = 48$, the processing time is halved for any resolution. Real-time processing would be easily achieved for $M = 48$ and $r_{sp} = 0.01$ by adding an additional GPU. Figure 7.10 shows more clearly the time differences among all the configurations by varying the number of microphones and the grid resolutions $r_{sp} \in \{0.1, 0.05, 0.01\}$. Note that the time t_{buff} is marked by a solid black line.

Table 7.3. Processing time t_{proc} using two GPUs.

r_{sp}	$M = 6$	$M = 12$	$M = 24$	$M = 48$
0.01	1.031 ms	3.578 ms	15,564 ms	60.108 ms
0.05	0.381 ms	0.758 ms	2.238 ms	6.433 ms
0.1	0.371 ms	0.650 ms	1.588 ms	4.588 ms

Table 7.4. Processing time t_{proc} using one GPU.

r_{sp}	M=6	M=12	M=24	M=48
0.01	1.894 ms	6.731 ms	30.145 ms	122.720 ms
0.05	0.564 ms	1.132 ms	3.484 ms	11.203 ms
0.1	0.546 ms	0.926 ms	2.336 ms	7.493 ms

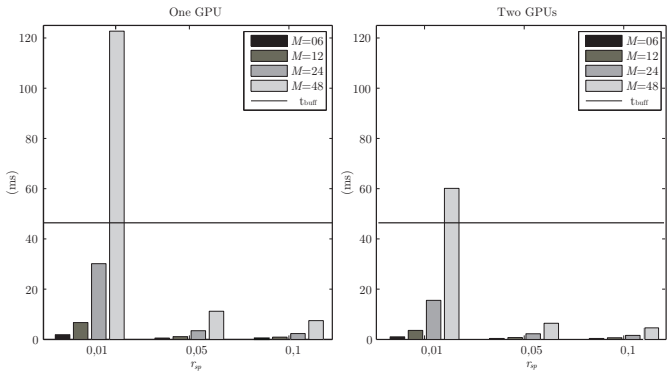


Figure 7.10. Time t_{proc} for different resolutions and number of microphones.

7.5 Conclusion

New emerging GPU architectures help to overcome different computational problems in sound source localization, such as having very fine spatial resolutions or dealing with a high number of microphones. In this chapter, we have proposed an scalable multi-GPU implementation of the well-known SRP-PHAT algorithm and we have assessed its performance on a real-time localization system. The results show that employing a high number of microphones has a direct impact on localization performance in highly reverberant environments. However, systems with a high number of microphones also require high computing capacity. The chapter highlights the important role that GPU architectures and massive computation have in acoustic localization tasks in adverse environments. In this context, we have described in detail the most important implementation issues. Moreover, the SRP-PHAT algorithm has been analyzed considering localization in three dimensions, which is a task that requires high computational resources. Another important aspect to point out is that, since the massive processing is carried out by the GPU, the CPU resources could be used for other tasks. The presented work demonstrates that the use of the GPU hardware provides a good solution for building real-time sound source localization systems. More details of this work can be found in [131] and in [132].

8 Multichannel IIR Filtering

8 Multichannel IIR Filtering

In the audio signal processing field, multiple IIR filters are required in many applications. Up to now, the use of the GPUs for implementing IIR filters has not been clearly addressed in audio processing because of its feedback loop that prevents its total parallelization. However, using the Parallel form of IIR filters, this feedback is reduced, since every single sample is computed in a parallel way. This chapter analyzes the performance of multiple IIR filters using GPUs and compares it with a powerful multi-core computer.

8.1 Definition of the problem

Modeling or equalizing an acoustic or electro-acoustic transfer function by digital filters is a typical task in audio signal processing. By taking into account the properties of the human hearing, significant savings can be achieved in the required computational power at a given sound quality. As an example, the frequency resolution of the human auditory system has led to the development of special filter design methodologies with a logarithmic frequency resolution, as opposed to the linear frequency resolution of traditional FIR and IIR filters. These techniques include frequency warping [133], Kautz filters [134], or fixed-pole parallel filters [135]. Typically, the

required filter order is reduced by a factor of 5 compared to traditional IIR filters (e.g., designed by the Steiglitz–McBride method [136]) with all the above techniques. This advantage is slightly reduced in the case of warped and Kautz filters, because they are implemented by special filter structures, but not for fixed-pole parallel filters, since they are simply implemented as a set of second-order sections.

One application that is specially important in the context of multichannel acoustic signal processing using IIR filters is the equalization of a Wave Field Synthesis (WFS) system. WFS systems require high computational capacity since they involve multiple loudspeakers, such as the WFS system at the Universitat Politècnica de València (UPV) (shown in [14]) that has 96 loudspeakers, or the IOSONO WFS system (shown in [15]) that has 120 loudspeakers. Equalizing a WFS system requires such a massive amount of filtering that even when using parallel filters, a significant amount of CPU time is taken for filtering, and in some cases, real-time operation is not possible even in modern multi-core computers.

8.1.1 Fixed-pole parallel filters

Traditionally, the parallel second-order form of digital filters has been used because of its lower sensitivity to coefficient quantization and better quantization noise performance compared to direct form IIR filters [137]. In these applications, first a direct form IIR filter is designed and then factored to a parallel form using the partial fraction expansion.

In fixed-pole parallel filters, the filter is designed directly in the second-order form by first setting the poles to predetermined positions. The advantage of fixing the poles is that now the filter design reduces to a linear-in-parameter problem which has a unique solution. Nevertheless, the most important property of parallel filters is that the pole frequencies allow a direct control of the frequency resolution: the more poles are placed in a specific frequency range, the higher resolution is obtained. For example, placing the poles according to a logarithmic frequency scale results in a logarithmic frequency resolution, and the modeled response resembles the fractional-octave smoothed version of the target [138]. For a thorough comparison of pole positioning methods see [139].

The general form of the parallel filter consists of a parallel set of second-order sections and an optional FIR filter path [140]. In this implementation,

the FIR part is reduced to a single signal path, and the following form is used:

$$H(z^{-1}) = \sum_{r=1}^{K} \frac{b_{r,0} + b_{r,1} z^{-1}}{1 + a_{r,1} z^{-1} + a_{r,2} z^{-2}} + d_0, \tag{8.1}$$

where K is the number of second-order sections. The filter structure and the second-order section are depicted in Fig. 8.1 and Fig. 8.2, respectively.

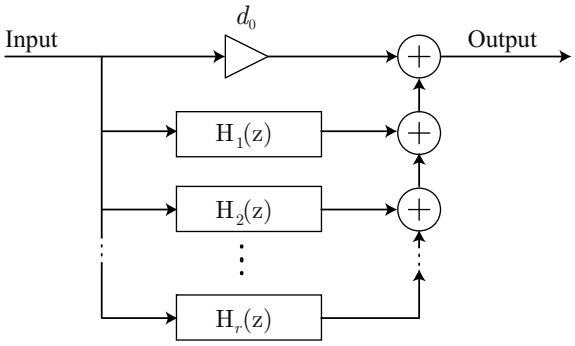


Figure 8.1. Structure of the parallel second-order filter.

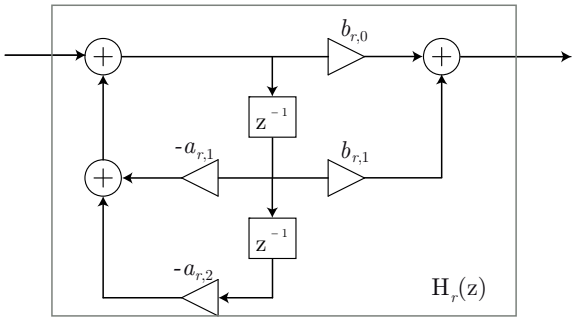


Figure 8.2. Structure of the second-order section used in Fig. 8.1

8.1.2 Filter design

We can assume that the poles of the parallel filter p_r are known (e.g., set to a logarithmic scale). Then the denominator coefficients are determined

by the poles ($a_{r,1} = p_r + \overline{p}_r$ and $a_{r,2} = |p_r|^2$), and the filter design problem becomes linear in its free parameters (weights) $b_{r,0}$, $b_{r,1}$ and d_0.

Using the substitution $z^{-1} = e^{-j\vartheta_n}$ in (8.1) and writing it in matrix form for a finite set of ϑ_n angular frequencies yields [140]

$$\mathbf{h} = \mathbf{M}\mathbf{p}, \tag{8.2}$$

where $\mathbf{p} = [b_{1,0}, b_{1,1}, \dots b_{r,0}, b_{r,1}, d_0]^T$ is a column vector composed of the free parameters. The first column of the modeling matrix $\mathbf{M}$ contains the all-pole transfer function of the first section $1/(1 + a_{1,1}e^{-j\vartheta_n} + a_{1,2}e^{-j2\vartheta_n})$ for the ϑ_n angular frequencies, and the second column contains its delayed version version $e^{-j\vartheta_n}/(1 + a_{1,1}e^{-j\vartheta_n} + a_{1,2}e^{-j2\vartheta_n})$ for all ϑ_n. The third and fourth columns are the all-pole transfer functions for the second section $1/(1+a_{2,1}e^{-j\vartheta_n}+a_{2,2}e^{-j2\vartheta_n})$ and its delayed version $e^{-j\vartheta_n}/(1+a_{2,1}e^{-j\vartheta_n}+a_{2,2}e^{-j2\vartheta_n})$ for all ϑ_n. The remaining part of matrix $\mathbf{M}$ is constructed similarly, except the last column, which belongs to the constant gain path, and it is 1 for all ϑ_n. Finally, $\mathbf{h} = [H(\vartheta_1) \dots H(\vartheta_N)]^T$ is a column vector composed of the resulting frequency response.

The optimal parameters $\mathbf{p}_{\text{opt}}$ in the mean squares sense are found by the well-known least-squares (LS) solution

$$\mathbf{p}_{\text{opt}} = (\mathbf{M}^H\mathbf{M})^{-1}\mathbf{M}^H\mathbf{h}_{\text{t}}, \tag{8.3}$$

where $\mathbf{M}^H$ is the conjugate transpose of $\mathbf{M}$, and $\mathbf{h}_{\text{t}}$ is the target frequency response. Note that (8.3) assumes a filter specification $H_{\text{t}}(\vartheta_n)$ given for the full frequency range $\vartheta_n \in [-\pi, \pi]$. Matlab code for parallel filter design can be downloaded from `http://www.mit.bme.hu/~bank/parfilt`.

8.2 Implementations on Many-core architectures (GPU and multi-cores)

This section presents a multichannel GPU-based implementation of parallel IIR filters that can be used for equalizing a WFS system and compares its computational performance with the performance of a powerful multi-core computer.

8.2.1 GPU-based parallel implementation

If we want to equalize a WFS system composed of N loudspeakers, we need to carry out N IIR filter processes concurrently. Thus, we launch N thread blocks to run the CUDA kernel. Each thread block of U threads ($U \in \{128, 256, 512\}$) corresponds to one IIR filter that has K second-order sections. A thread inside a thread block computes $\frac{K}{U}$ sections, and stores its result in the *shared-memory*. Then, a synchronization barrier is set in order to wait that all the threads have finished. After that, the reduction algorithm described by Harris [129] is implemented. It consists in summing up in a parallel way all the values of a vector that is stored in the *shared-memory*. Finally, the FIR coefficient d_0 is executed at the end by only one of the threads of the thread block. Figure 8.3 illustrates the described operations for one parallel IIR filter process. Subsection CUDA kernel 25 shows the code used in this implementation.

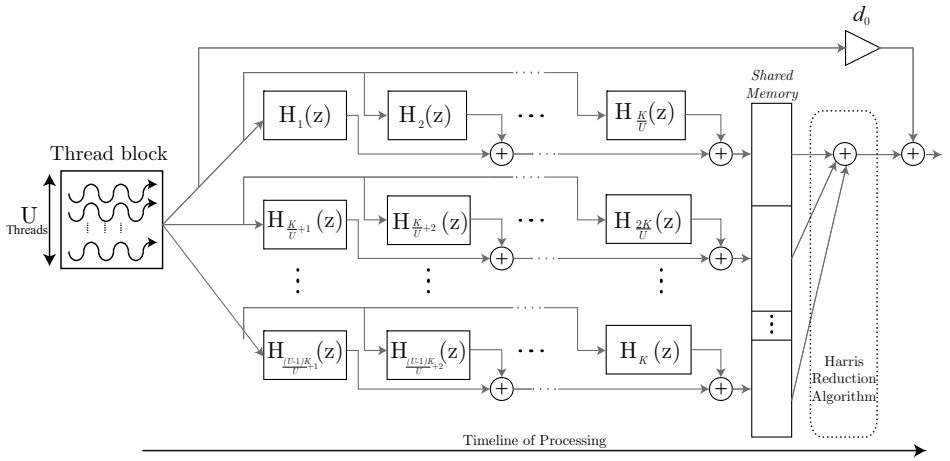


Figure 8.3. GPU-based Parallel Implementation of one IIR filter processing.

8.2.2 Multicore-based parallel implementation

In order to assess the computational performance achieved by the GPU implementation, a comparison with a powerful multicore computer is required. To this end, we implement our algorithm using *openMP* [40]. This programming framework allows us to parallelize our algorithm using all the

cores that a computer owns. The multicore computers are more suited to task-based parallelism, instead of the fine-grain parallelism, which can be easily managed by a GPU. The advantage of a multicore implementation is that data transferring from/to GPU to/from CPU is not needed.

The serial algorithm to carry out N IIR filters is composed by three nested loops: number of filters, number of samples per filter, and number of second-order sections. By using *openMP*, we implement our algorithm by distributing the iterations of the most external loop among all the cores, since each filter process is independently performed. In case that our multicore computer is composed of S_c processors, each processor is responsible for carrying out $\frac{N}{S_c}$ filters.

Comparing to the GPU-based implementation, one CPU processor computes at least the same operations as a thread block. However, one thread block only computes one filter process, while one CPU processor could compute more than one. GPU-based parallelization presents two levels: concurrency in computing multiple filter processes, and concurrency in computing multiple second-order sections inside one filter. In contrast, the CPU-based parallelization presents only concurrency in multiple filter processes.

CUDA kernel 25

CUDA Kernel 25 Multichannel IIR filtering

Input: $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{B}_1$, $\mathbf{B}_2$, $\mathbf{V}_1$, $\mathbf{V}_2$, $\mathbf{F}_{ir}$, $\mathbf{S}$, K, U, L
Output: $\mathbf{O}$,

```
1: __shared__ float sd[U];
2: int blq = K/U;
3: int iAux = 0;
4: float v1[blq]
5: float v2[blq];
6: float v0;
7: int i=0;
8: int index = BlockIdx.x*K + ThreadIdx.x;
9:  // Initialization of the shared-memory
10: for j = 0,...,blq−1 do
11:    i = index + j*U;
12:    v1[j] = V1[i];
```

```
13:    v2[j] = V2[i];
14: end for
15: for t = 0,...,L-1 do
16:    sd[ThreadIdx.x] = 0.0;
17:    for j = 0,...,blq-1 do
18:       i = index + j*U;
19:       vo = (float)S[t + BlockIdx.x*L] - A1[i]*v1[j] - A2[i]*v2[j];
20:       sd[ThreadIdx.x] = sd[ThreadIdx.x] + B1[i]*v0 + B2[i]*v1[j]
21:       v2[j] = v1[j];
22:       v1[j] = v0;
23:    end for
24:     __syncthreads();
25:    if(U >=512)
26:       if(ThreadIdx.x < 256)
27:          sd[ThreadIdx.x] += sdsdata[ThreadIdx.x + 256];
28:       end if
29:        __syncthreads();
30:    end if
31:    if(U >=256)
32:       if(ThreadIdx.x < 128)
33:          sd[ThreadIdx.x] += sdsdata[ThreadIdx.x + 256];
34:       end if
35:        __syncthreads();
36:    end if
37:    if(U >=128)
38:       if(ThreadIdx.x < 64)
39:          sd[ThreadIdx.x] += sdsdata[ThreadIdx.x + 256];
40:       end if
41:        __syncthreads();
42:    end if
43: // now that we are using warp-synchronous programming (below)
44: // we need to declare our shared memory volatile so that
45: // the compiler doesn't reorder stores to it and
46: // induce incorrect behavior.
47:    if(ThreadIdx.x <32)
48:       volatile float *smem = sd;
49:       if(U >=64)
50:          smem[ThreadIdx.x] += smem[ThreadIdx.x + 32];
51:       end if
```

```
52:       if(U ≥32)
53:          smem[ThreadIdx.x] += smem[ThreadIdx.x + 16];
54:       end if
55:       if(U ≥16)
56:          smem[ThreadIdx.x] += smem[ThreadIdx.x + 8];
57:       end if
58:       if(U ≥8)
59:          smem[ThreadIdx.x] += smem[ThreadIdx.x + 4];
60:       end if
61:       if(U ≥4)
62:          smem[ThreadIdx.x] += smem[ThreadIdx.x + 2];
63:       end if
64:       if(U ≥2)
65:          smem[ThreadIdx.x] += smem[ThreadIdx.x + 1];
66:       end if
67:    end if
68:    if(ThreadIdx.x == 0)
69:       iAux = t + blockIdx.x*L;
70:       O[iAux] = sd[0] + F_ir[BlockIdx.x]*((float)S[iAux]);
71:    end if
72: end for
73:  // Storing values for the next block of samples
74: for j = 0,...,blq−1 do
75:    i = index + j*U;
76:    V_1[i] = v1[j];
77:    V_2[i] = v2[j];
78: end for
```

Variables used in CUDA kernel 25

- Vectors $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{B}_1$ and $\mathbf{B}_2$ have K elements and store the coefficients $a_{r,1}$, $a_{r,2}$, $b_{r,1}$ and $b_{r,2}$ of every second-order, respectively with $r \in [0, K-1]$.
- Vectors $\mathbf{V}_1$ and $\mathbf{V}_2$ have K elements and store the intermediate values in order to carry out the delay of the block z^{-1} in Fig. 8.2. Vector $\mathbf{F}_{ir}$ has also K values and stores the values d_0 of every second-order structure.
- Vectors $\mathbf{S}$ and $\mathbf{O}$ have the dimension $(N \times L)$, and stores the input

samples and the output samples, respectively.

- Variable L is the size of the audio sample buffers

8.3 Results

We test our GPU-based implementation on an Nvidia Tesla K20c that is based on the Kepler architecture and is composed of 13 SMXs, and our multicore-based implementation on a computer composed of two SMPs (*Symmetric Multi-Processing*) Intel Xeon CPU X5680 at 3.33 GHz, which is a hexacore. Thus, our multicore computer is composed of 12 cores (S_c=12).

We use a standard audio card at the laboratory. The audio card uses the ASIO (Audio Stream Input/Output) driver to communicate with the CPU and provides ($L \in \{32, 64, 128\}$) samples per channel every 0.72 ms, 1.45 ms, and 2.90 ms, respectively (sample frequency f_s=44100 Hz), which we call buffer times t_{buff}. Assuming that our WFS system requires to equalize N loudspeakers, we define t_{proc} as the processing time since the N input-data buffers are available till the N output-data buffers are totally processed. Data transfer times in the case of GPU-based implementation are included in t_{proc}. The equalization of a WFS system works in real-time as long as $t_{\text{proc}} < t_{\text{buff}}$.

Figure 8.4 shows the results when a system is executed using a buffer size of L=32 samples. Computational performance has been assessed by assuming that all filters are composed of 128 second-order sections for the first example, and with filters composed of 1024 second-order sections for the second example. We execute the system by increasing N gradually and by measuring each time t_{proc}. Note that the maximum number of filters that can be executed in real time are marked with a circle $\circ$ in Fig. 8.4, and their values are shown in the legend of the figure for all cases. The proposed GPU-based implementation can run 1256 filters in real time with 0.72 ms latency when the filters are composed of 128 second-order sections. In case of 1024 second-order sections, 272 equalization filters can be executed.

Very similar results are obtained if we increase the buffer size to L=64 and L=128 samples, as can be appreciated in Fig. 8.5. The number of filter processes that can be achieved in real time increases slightly as the buffer size does, in spite that transfer times between GPU and CPU in-

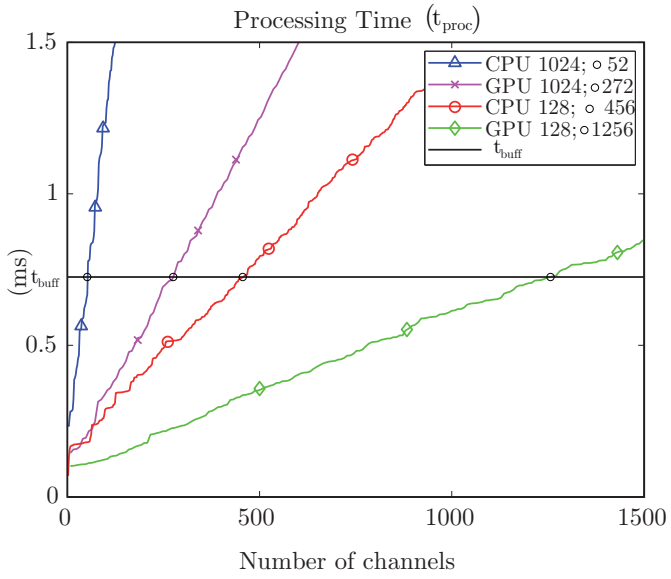


Figure 8.4. Performance comparison between multi-core CPU and GPU implementations for parallel filters composed of 1024 and 128 second-order sections with a buffer size of 32 samples.

crease. This occurs because transfer time is still not significant compared to the parallelization resources that GPUs offer. For filters composed of 128 second-order sections, GPU outperforms in 2.75, 2.6, and 2.55 times the CPU, for buffer sizes of L=32, L=64 and L=128 samples, respectively. Otherwise, for filters composed of 1024 second-order sections, GPU outperforms in 5.23, 4.8 and 4.66 times the CPU for the same previous buffer sizes. Thus, computing filters composed of 1024 second-order is more efficient on GPU, since more computational resources of the GPU are utilized. Note that the speed-up GPU/CPU decreases slightly as long as the buffer size increases.

The CPU-based implementation runs on a powerful computer composed of 12 cores, which is a fair comparison with the GPU. In all cases, the GPU-based implementation outperforms the multicore-based implementation, which indicates that GPUs are well suited for performing massive IIR filter processes, even more when the sample buffer sizes are very short.

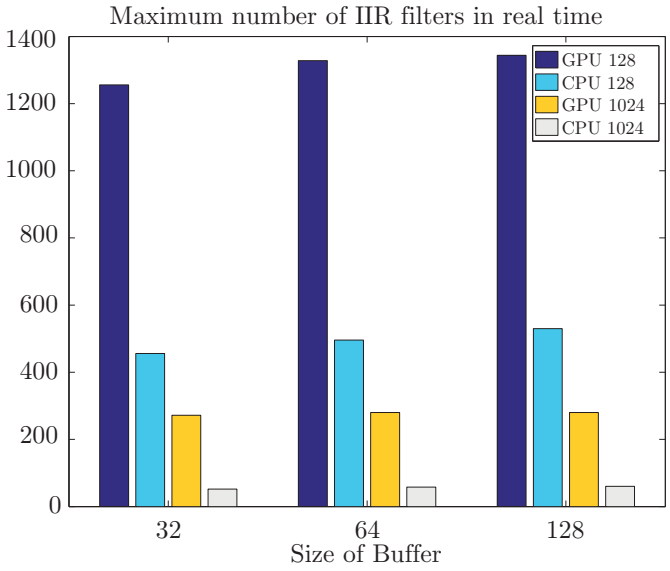


Figure 8.5. Maximum number of IIR filters that can be realized in real time for the multi-core and GPU implementation for filters composed of 1024 and 128 second-order sections.

Regarding GPU-based computational aspects, filters composed of 128 second-order sections require to launch a CUDA *kernel* composed of thread blocks with U=128, where each thread performs one section. However, filters composed of 1024 second-order sections first require to test different values for U. Table 8.1 shows the maximum number of equalization filters that can be executed in real time for different values of U. It can be noticed that using the thread block size of 256 threads gives the best performance.

Table 8.1. Maximum number of real-time IIR filters for the GPU implementation using different thread block sizes U and buffer sizes of 32 and 64 samples. The best results are bolded.

SIZE	$U = 128$	$U = 256$	$U = 512$	$U = 1024$
32	192	**272**	168	120
64	208	**280**	176	128

As another performance measure, taking into the number of loudspeakers of the WFS systems referenced previously, we have tested the maximum number of second-order sections that an equalizer filter of these systems can have by using the GPU. Regarding the WFS system at the Universitat Politècnica de València (UPV), we have achieved to equalize the 96 loudspeakers using filters composed of up to 2048 second-order sections under real-time conditions. Otherwise, as IOSONO WFS system is composed of 120 loudspeakers, the massive equalization can be carried out also under real-time conditions if we use up to 1536 second-order sections. Thus, GPUs allow us to use high filter orders for equalizing large-scale WFS systems.

An additional advantage of the GPU-based implementation is that the GPU can be used as a co-processor where all audio processing is being carried out, while the CPU could be used for other tasks at the same time.

8.4 Conclusion

This work has demonstrated the power of the parallel form of IIR filters in parallel computing. This form allows us not only the robust design of high-order filters with logarithmic frequency resolution, but also a very efficient implementation on GPUs. The proposed implementation can carry out up to 1256 equalizers with a filter order of 256 in real time, which means 321536 total filter order, for a buffer size of 32 samples. Many applications can be favored from this result, including a total equalization of a WFS system. In addition, we have compared the proposed GPU-based implementation with a multicore-based implementation in a powerful computer. Results show that GPU outperforms the powerful multicore computer in all example cases.

This work was conducted in fall 2013 when I was visiting the Aalto University Department of Signal Processing and Acoustics. More details of this work can be found in [141].

9 Massive Multiple Allpass filtering

9 Massive Multiple Allpass filtering

Limiting dynamic range whilst maintaining sound quality has become an important issue in audio signal processing. Previous works indicate that an allpass filter chain can reduce the peak amplitude of an audio signal, without introducing the distortion associated with traditional non-linear techniques. However, selecting proper coefficients and delay-line lengths for the allpass filters is a major challenge when the signal shape is not predictable, since a large number of possibilities exists. Previously, the selection of delay-line lengths was random and the filter coefficient values were fixed, which does not necessarily ensure the best reduction. We propose a GPU-based implementation of multiple allpass filter chains that is able to cover all relevant delay-line lengths and perform a wide search on possible coefficient values in order to get closer to the optimal choice.

9.1 Definition of the problem

Dynamic range reduction in audio signals is important. This process consists of restricting the dynamic range of an audio signal to a smaller space [142], hence allowing maximization of loudness. Useful dynamic range reduction is achieved if the peak amplitude of a signal decreases with respect

to its RMS amplitude.

Up to now, dynamic range reduction was achieved by using non-linear techniques [143, 144]. Allpass filters were presented previously in [1] as a method for reducing the peak to RMS amplitude ratio, since they present a flat frequency response, but a non-linear phase response. By modifying the phase of the signal, maximum peak level can be reduced without introducing new frequency content. Listening tests suggest that if the impulse response length of the allpass filters is below 4 ms, the change is inaudible [145].

The work presented in [1] analyzes in detail the behavior of a first-order allpass filter. This analysis concludes that peak amplitude of the impulse response is minimised when the allpass coefficient is equal to $\pm\Phi$, where Φ is the inverse of the *golden ratio*. To three decimal places, the value of the *golden ratio* is 1.618 and its inverse Φ is 0.618. This property was also referenced in [146], without proof.

In addition, the work in [1] suggests that the unit delay in the first-order allpass is replaced with a longer delay-line length whose maximum value d_{max} is set to 30. This maximum is chosen in order to restrict the length of the impulse response to the range in which the spreading of the signal is inaudible. Similar allpass filters with a long delay line have previously been used for artificial reverberation [147, 148] and for spectral delay in audio processing [149, 150]. The authors in [1] propose the structure shown in Fig. 9.1. This structure is composed of M_A filters in parallel, where each filter is made of a cascade of three allpass filters with three different embedded delay-line lengths $\{d_1, d_2, d_3\}$, and three coefficients $\{a, b, c\}$ whose values are set to $a = -\Phi$, $b = +\Phi$, and $c = -\Phi$. Their test consists of processing an input signal through 100 filters in parallel (i.e. in their tests, $M_A = 100$). Each one of the filters has its delay-line lengths $\{d_1, d_2, d_3\}$ determined randomly, between 1 and the state d_{max} of 30. The output of every filter is examined, and the one that produces the lowest peak amplitude is selected as the one that offers best linear dynamic range reduction. Note that the structure also contains a path that bypasses the processing, since in rare cases the lowest peak amplitude produced by the allpass filters could be higher than the input. In that case, the input itself is selected as the output.

Their results show that even with such a small random selection of delay line lengths, the dynamic range is generally reduced [1]. However, the

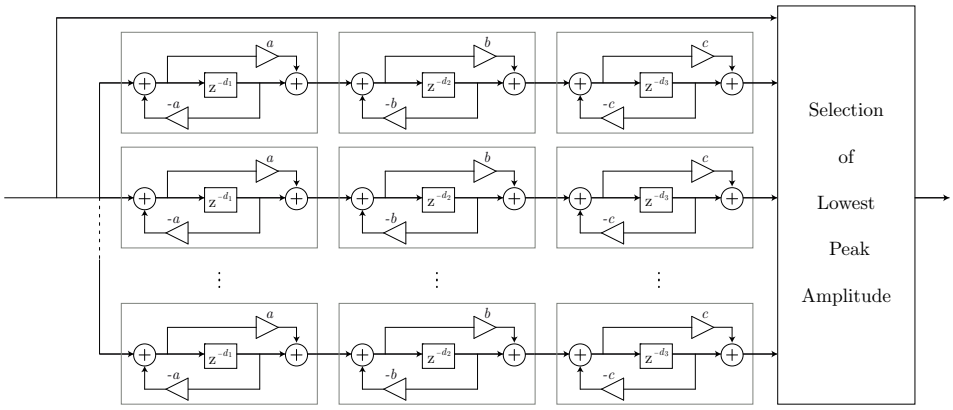


Figure 9.1. Block diagram of the M parallel allpass filter chains.

theoretical maximum dynamic range reduction is not achieved. Targeting more delay-line length combinations in order to maximise the reduction requires the use of more computational resources. This resource problem can be vastly reduced by noticing that the process is intrinsically highly parallel, and hence suitable for computation with a Graphics Processing Unit (GPU). As commented in Section 9.1, a GPU can be considered to be a Single Instruction Multiple Data machine (SIMD), i.e., a computer in which a single set of instructions is executed on a large number of data sets simultaneously. The analogy with the proposed parallel allpass structure should be clear.

This work presents a GPU-based implementation of the structure that aims to seek the maximum dynamic range reduction of a signal. To this end, not only are all combinations of delay-line lengths examined, but also different values of coefficients, in order to validate or reject the use of the *golden ratio* as a unique coefficient.

9.2 Test Setup

Taking into account that the maximum delay-line length d_{max} of the allpass filter is chosen to be 30, it would be easy to think that we are dealing with 27,000 possibilities. However, as an allpass filter cascade

is a linear system and can be re-ordered freely, many of the possibilities are redundant. Thus, starting with only three possible values for the delay-line lengths $\{d_1, d_2, d_3\}$, the ten possible combinations without redundancy are:$\{d_1, d_1, d_1\}$, $\{d_1, d_1, d_2\}$, $\{d_1, d_1, d_3\}$, $\{d_1, d_2, d_2\}$, $\{d_1, d_2, d_3\}$, $\{d_1, d_3, d_3\}$, $\{d_2, d_2, d_2\}$, $\{d_2, d_2, d_3\}$, $\{d_2, d_3, d_3\}$, and $\{d_3, d_3, d_3\}$. These combinations are obtained if we apply *Multiset theory* [151]. This theory indicates that given p elements, the number of multisets of cardinality q is:

$$\frac{(p+q-1)!}{q!(p-1)!}. \tag{9.1}$$

In our case, p=30 and q=3. Thus, combining all possible delay-line lengths implies 4,960 combinations of interest. Additionally, there are also the three different coefficients $\{a, b, c\}$ that previously were fixed to Φ. However, as was mentioned in Sec.9.1, it was not proved that Φ offers maximum reduction for general signals, only for impulses. Therefore, it may be advantageous to explore a larger coefficient space. If we vary the coefficients between 0.3 and 0.7 in steps of 0.05, we have nine possibilities for each coefficient. As we have three coefficients, we have 729 possibilities more plus one. We must also add the combination of the work in [1] (all coefficients are equal to Φ). It is clear that a single allpass filter can have a coefficient of $\pm a$ and maintain a stable impulse response. Therefore, we must also consider the different combinations of the sign in the coefficients: $\pm a$ and $\mp a$, $\pm b$ and $\mp b$, and $\pm c$ and $\mp c$. This implies 8 sign-based combinations for each $\{a, b, c\}$ coefficient combination. In total, to tackle all the described combinations, we must compute 28,966,400 allpass filter chains. Thus, it makes sense to implement and perform the process on a GPU.

9.3 GPU-based Implementation

The hardware we use is a Nvidia Tesla K20c that is based on the Kepler architecture and is composed of 13 SMXs. The implementation we propose aims to launch as many threads as combinations we have. To this end, we can divide the combinations in two: coefficients combinations and delay-line lengths combinations. These combinations are stored in two matrices at the GPU *global-memory*. The CUDA grid we launch is two-dimensional and is composed of blocks of 256 threads. In this case, the identification of

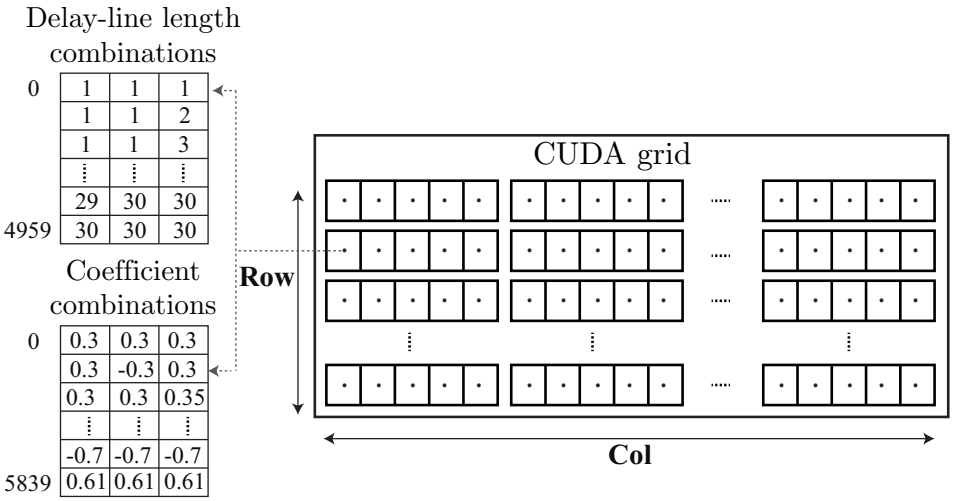


Figure 9.2. Two-dimensional CUDA grid configuration. One thread performs an allpass filter using a delay line combination with a coefficient combination. **Col** defines the delay-line lengths and the **Row** determines the lookup in the coefficient table.

a thread is given by two variables `Col` and `Row`.

Figure 9.2 shows how thread (`Col`, `Row`) performs the allpass filter chain with coefficient combination `Row` and delay-line lengths combination `Col`. Each thread has three vectors of size d_{max} whose role is to simulate the delay lines. These $3d_{max}$ elements per thread are stored at the GPU registers. All the samples of the input are processed by the thread, which only stores in the *shared-memory* the maximum absolute value of the signal. Afterwards, a synchronization barrier is set in order to wait for all the threads to finish. Subsection CUDA kernel 26 shows the code used for implementing the described operations. After that, the reduction algorithm described by Harris [129] is implemented. It consists in looking at the minimum of all stored values and identifying its combination `Col` and `Row`. The code described by Harris can be found at the SDK in the webpage [20]. Part of this code is used in CUDA kernel 25 which is shown in Chapter 8.

CUDA kernel 26

CUDA Kernel 26 Multiple allpass filter chain concurrently

Input: **s**, **C**, **D**, C_{cf}, C_{dy}, d_{max}, l_s

Output: **o**,

```
1: int Col = BlockDim.x*BlockIdx.x + ThreadIdx.x;
2: int Row = BlockDim.y*BlockIdx.y + ThreadIdx.y;
3: if(Col  < C_dy)
4:    int idx = Row*C_dy + Col;
5:    int Col1 = C[3*Col];
6:    int Col2 = C[3*Col + 1];
7:    int Col3 = C[3*Col + 2];
8:    float Row1 = D[3*Row];
9:    float Row2 = D[3*Row + 1];
10:   float Row3 = D[3*Row + 2];
11:    // Three vectors of size d_max
12:   float vec1[d_max];
13:   float vec2[d_max];
14:   float vec3[d_max];
15:   for k = 0,...,d_max - 1 do
16:      vec1[k] = 0.0;
17:      vec2[k] = 0.0;
18:      vec3[k] = 0.0;
19:   end for
20:   float out=0.0;
21:   float MaxOut=0.0;
22:   int ini1=0;
23:   int fin1=1;
24:   int ini2=0;
25:   int fin2=1;
26:   int ini3=0;
27:   int fin3=1;
28:   for j = 0,...,l_s - 1 do
29:      //------First Chain-------------------------%
30:      vec1[ini1] = (float)s[j] - Row1*vec1[fin1];
31:      out = vec1[fin1] + Row1*vec1[ini1];
32:      //------Second Chain------------------------%
33:      vec2[ini2] = out - Row2*vec2[fin2];
34:      out = vec2[fin2] + Row2*vec2[ini2];
```

```
35:      //------Third Chain-------------------------%
36:      vec3[ini3] = out - Row3*vec3[fin3];
37:      out = vec3[fin3] + Row3*vec3[ini3];
38:      out = abs(out);
39:      //out to Global
40:      if(out  > MaxOut)
41:         MaxOut = out;
42:      end if
43:      // New values for pointers ini.
44:      ini1 = fin1;
45:      ini2 = fin2;
46:      ini3 = fin3;
47:      //New values for pointers fin.
48:      fin1=(fin1+1)%Col1;
49:      fin2=(fin2+1)%Col2;
50:      fin3=(fin3+1)%Col3;
51:    end for
52:    o[idx] = MaxOut;
53: end if
```

Variables used in code of CUDA kernel 26

- Matrix $\mathbf{C}$ has dimensions $(C_{cf} \times 3)$ where variable C_{cf} is the number of combinations that are achieved varying the coefficients values.

- Matrix $\mathbf{D}$ has dimensions $(C_{dy} \times 3)$ where variable C_{dy} is the number of combinations that are achieved varying the delay lengths.

- Vector $\mathbf{s}$ contains the audio signal and is composed of l_s audio samples.

- Vector $\mathbf{o}$ has as many elements as the number of combinations that have been launched. Each component stores the maximum value that has been achieved in the audio signal after the filtering for each combination.

- The number of blocks that this kernel launches is $\frac{C_{dy}}{128} \times C_{cf} \times 1$, being the block size $128 \times 1 \times 1$.

9.4 Results

The described implementation was applied to five isolated clean musical sounds, in the same way as in the work [1]. The sounds consist of single drum hits recorded from a Roland TR-808, a single piano tone played at C3 and a single synthesised mallet-like sound. All input signals have a sample rate of 44.1 kHz, and are normalized so that their peak amplitude is 1.

The work in [1] evaluated only 100 random possibilities. The first test we have carried out consisted in evaluating all delay lines length combinations using the same coefficients that were used previously ($a = -\Phi$, $b = +\Phi$, $c = -\Phi$), which means 4960 combinations. The second test introduces three different variations in the sign of the coefficients but maintains the same value, i.e 14880 combinations. Finally, third test launches the 28,966,400 combinations. Table 9.1 shows the results of the three tests and includes: the maximum peak value of all the signals with its corresponding delay-line lengths, and $\{a, b, c\}$ coefficients for each test. It is noticeable how the maximum reduction in every signal is improving as more combinations are attempted. One important result to point out is that maximum reduction is not necessarily achieved by using Φ as a coefficient.

Figure 9.3 presents in an increasing way the maximum peak value obtained for the 28,966,400 combinations for all the signals. On the left are the best results providing compressions whereas most of the random combinations would increase the dynamic range. Although it depends on the signal, few combinations achieve to reduce the dynamic range meaningfully. Thus, in case of using allpass filters for reducing dynamic range, many combinations must be tackled. Figure 9.4 compares the waveforms of the input signal, the signal obtained after processing from work [1], and the signal obtained after processing the third test: 28,966,400 combinations. It is noticed that the improvement is remarkable.

9.4.1 Computational Performance

A real-time scenario could be given in studio situation where every signal frame would have to be processed in a time of around 0.5 s, which implies the need to process 22050 samples (sample frequency f_s=44100 Hz) in less than 0.5 s. In order to assess the computational performance achieved by the GPU implementation, we have also performed the three tests in a pow-

Table 9.1. Delay lines lengths, coefficient values and maximum peak value that offer maximum dynamic range reduction obtained after the three tests for the five sounds. Maximum reduction is bolded. PW, 1st, 2nd, and 3rd correspond to results from the work in [1], first test, second test, and third test, respectively.

Sound	d_1	d_2	d_3	a	b	c	**Peak(out)**
Bass PW	24	22	28	-Φ	Φ	-Φ	0.94
Bass 1st	29	29	29	-Φ	Φ	-Φ	0.89
Bass 2nd	13	28	30	-Φ	-Φ	-Φ	0.76
Bass 3rd	19	23	27	-0.4	-0.65	-0.7	**0.74**
Snare PW	21	14	26	-Φ	Φ	-Φ	0.77
Snare 1st	23	28	29	-Φ	Φ	-Φ	0.72
Snare 2nd	17	20	23	-Φ	-Φ	-Φ	0.71
Snare 3rd	20	21	30	-0.70	-0.65	0.60	**0.69**
Hi-hat PW	1	19	11	-Φ	Φ	-Φ	0.85
Hi-hat 1st	2	20	20	-Φ	Φ	-Φ	0.82
Hi-hat 2nd	1	8	18	Φ	-Φ	Φ	0.80
Hi-hat 3rd	11	24	26	0.55	-0.40	0.55	**0.75**
Piano PW	20	28	5	-Φ	Φ	-Φ	0.86
Piano 1st	16	16	26	-Φ	Φ	-Φ	0.85
Piano 2nd	14	28	30	Φ	-Φ	Φ	0.80
Piano 3rd	20	25	30	0.40	-0.70	0.55	**0.77**
Mallet PW	11	14	29	-Φ	Φ	-Φ	0.87
Mallet 1st	11	16	28	-Φ	Φ	-Φ	0.79
Mallet 2nd	3	30	30	-Φ	-Φ	-Φ	0.76
Mallet 3rd	5	30	30	-0.45	-0.70	-0.70	**0.73**

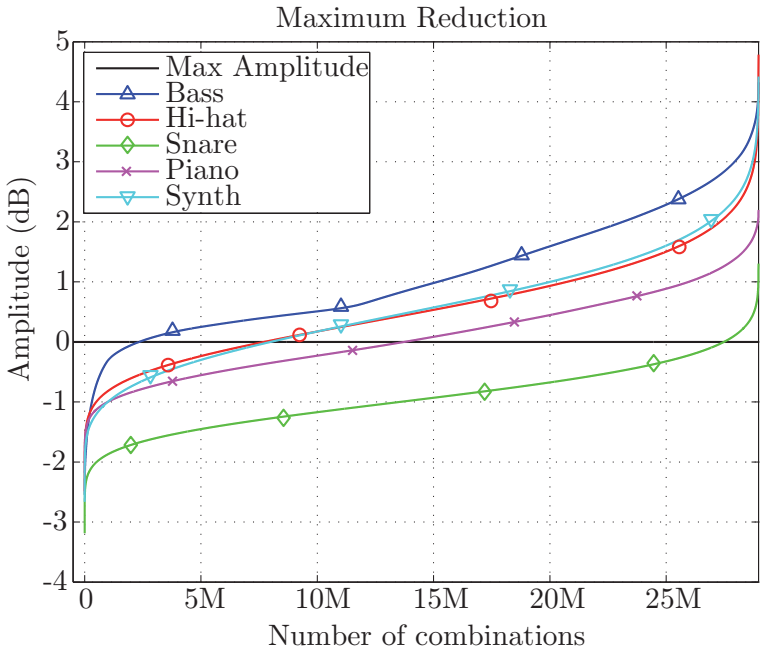


Figure 9.3. Maximum peak value obtained for the 28,966,400 combinations for all the signals.

erful multi-core computer that has one SMPs (*Symmetric Multi-Processing*) Intel Xeon CPU X5680 at 3.33 GHz, which is a hexacore. Thus, our multi-core computer is composed of six cores. We have tested all the combinations in a sequential way (one core carries out all combinations), and in a parallel way (distributing among the six cores all combinations) by using the programming framework *openMP* [40]. Table 9.2 shows the required time in processing 22050 samples for the three tests. As can be appreciated, only the third test can not be performed in real time by the GPU implementation. In all cases, the GPU implementation outperforms the CPU-based multicore implementation. Thus, employing a GPU as a computational accelerator for reducing dynamic range has sense.

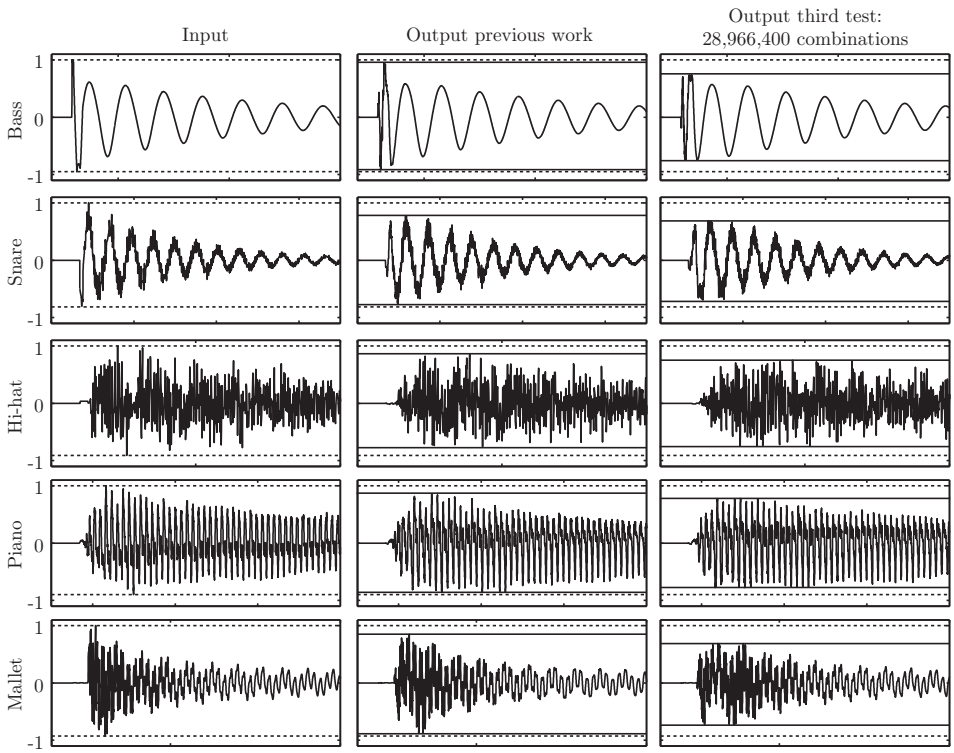


Figure 9.4. Waveforms of the five isolated musical sound, before and after being processed (the work in [1] and third test: 28,966,400 combinations). The horizontal dashed lines show the positive and negative peaks of the original waveform whilst the solid horizontal lines show the positive and negative peaks after processing.

9.5 Conclusion

The use of the GPU in large-scale audio processing is getting more widespread. In this case, we have used its computational capacity for evaluating dynamic range reduction in musical audio. The results show that the use of the inverse of the *golden ratio* as a coefficient in the allpass filter chains does not necessarily give the maximum reduction. Moreover, we have verified that the more combinations are tackled, the better reduction can be achieved, but a reduction of more than 3 dB is achieved by few combinations. We

Table 9.2. Processing time employed by the CPU and GPU to process the three described tests.

Test	1st	2nd	3rd
Combinations	4960	14800	28,966,400
One core - CPU Time	1.35 s	4.08 s	7914 s
Six cores - CPU Time	0.23 s	0.71 s	1374 s
GPU Time	**0.07** s	**0.32** s	**760** s

have proposed a GPU implementation based on a two-dimensional CUDA grid that combines different coefficient values and delay-line lengths. In total, 28,966,400 combinations have been computed in 760 s using a GPU-based implementation, which is two times faster than performing the same number of combinations in a six-core powerful computer.

This work was conducted in fall 2013 when I was visiting the Aalto University Department of Signal Processing and Acoustics. More details can be found in [132].

10 Conclusion

Conclusion 10

The overall aim of this research is to deepen into the Audio Signal Processing algorithms that deal with immersive audio schemes, and evaluate their potential when they are implemented on a Graphics Processing Units (GPUs). The motivation of this research comes from the necessity of developing and accelerating immersive audio applications that require high computational resources.

This chapter summarizes the findings of this research work, revisiting the research objectives given in the introductory chapter. First, Section 10.1 reviews the contents of this study, outlining the main conclusions that were extracted from each chapter. Recommendations for future research are discussed in Section 10.2. Additionally, the final sections contain a list of works published during the course of candidature for the P.h.D. degree, and the projects and stipends that have funded the presented work.

10.1 Main Contributions

The first part of this dissertation presents the fundamental operations that are carried out in audio signal processing, and also summarizes the main features of the GPU architecture together with the tools that have been

used in this dissertation.

First implementations on GPU have been devoted to carry out multiple convolutions on GPU concurrently. As a result, it was developed an application that requires to execute and combine multiple convolutions concurrently: a Generalized Crosstalk Cancellation and Equalization (GCCE). The selection of the correct placement of data in the different GPU memories is crucial to achieving good performance. It is described an efficient way to do it by exploiting parallelism and taking advantage of *shared-memory*. The evaluated tests for the developed application show that, with only an input-data buffer of 128 samples, it is possible to achieve up to real-time multichannel applications with 1408 filters of 2048 coefficients. This number gets larger as the input-data buffer increases.

Concurrent convolutions were used to develop a complete multisource spatial application in a binaural system. To render a sound source in a specific spatial location with a binaural system, it is necessary to convolve audio samples with HRIR filters that provide spatial information. Two common problems have been resolved during the design: synthesizing sound sources positions that are not in the HRIR database, and virtualizing the movement of the sound sources between different positions. Both problems were approached by increasing the number of convolutions which are later weighted and combined in different ways, taking maximum profit of the GPUs capacity for executing multiple convolutions simultaneously. Both solutions were assessed by performing different audio analyses. As a main contribution of this dissertation, it must be highlighted the development of a real headphone-based multisource spatial audio application whose audio processing is carried out on the GPU.

Other spatial audio application that has been developed is a Wave Field Synthesis system. This system uses an inverse filter bank in order to reduce the room effects and thus, to facilitate the virtual sound source localization within this spatial system. A realist demo is developed at the WFS system, that is composed of 96 loudspeakers, at the Universitat Politècnica de València. Thus, more than 9216 filters are involved in the system. This demo allows to render smoothly movements of virtual sound sources by executing multiple convolutions concurrently.

An scalable multi-GPU implementation of the well-known SRP-PHAT algorithm is also covered in this dissertation. The results show that employing a high number of microphones has a direct impact on localization per-

formance in highly reverberant environments. The SRP-PHAT algorithm was analyzed considering localization in three dimensions. It is important to point out the role that GPU architectures and massive computation have in acoustic localization tasks in adverse environments.

In case massive filtering is carried out by using IIR filter structures, we have proposed also a GPU-based implementation that highlights the power of the parallel form of IIR filters in parallel computing. The proposed implementation can carry out up to 1256 equalizers with a filter order of 256 in real time, which means 321536 total filter order, for a buffer size of 32 samples. A large number of applications can be favored from this result, including a total equalization of a WFS system. In addition, we have compared the proposed GPU-based implementation with a multicore-based implementation in a powerful computer. Results show that GPU outperforms the powerful multicore computer in all example cases.

Finally, the GPUs are also used for accelerating the search of specific parameters whose purpose is to reduce the dynamic range in musical audio. To this end, massive filtering based on the allpass filter chains is implemented and performed in GPU, which is able to compute 28,966,400 of these filters in 760 s. The results show that the more combinations are tackled, the better reduction can be achieved, but a reduction of more than 3 dB is achieved by few specific parameters.

The important conclusion to point out in this dissertation is that GPUs can be used as coprocessor that carries out the massive audio processing tasks. All the proposed GPU implementations offer excellent performances regarding the audio resources they can manage. Moreover, the fact of using GPUs for audio processing allows the CPU resources can be used for other tasks. Thus, this dissertation demonstrates that the use of the GPUs provides a good solution to build applications that require massive audio processing.

10.2 Further Work

There are still a large number of audio applications that can be accelerated by using GPUs, such as room acoustics, speech deconvolution in reverberant environments, among others. Most of them use computational operations that have been addressed through this dissertation and that can be used

as a model by software developers in the field of audio processing.

However, there are other challenges that require the use of even more powerful many-core processors such as Intel Xeon Phi. The Xeon Phi 3100-series was designed to deliver over 1TFLOPS peak double precision performance. One feature that differences this processor from GPUs architectures is task parallelism, which could be beneficial to develop ambitious and costly applications.

Future research can be mainly focused on the multichannel audio signal processing with feedback: To render signals through a large number of loudspeakers, and to pick up signals through a large number of microphones. Applications that use microphones and loudspeakers are related with the cancellation of selected noise. To achieve this objective requires to study a special case in the FIR filtering: Multichannel Adaptive Filtering, as shown in Fig.10.1

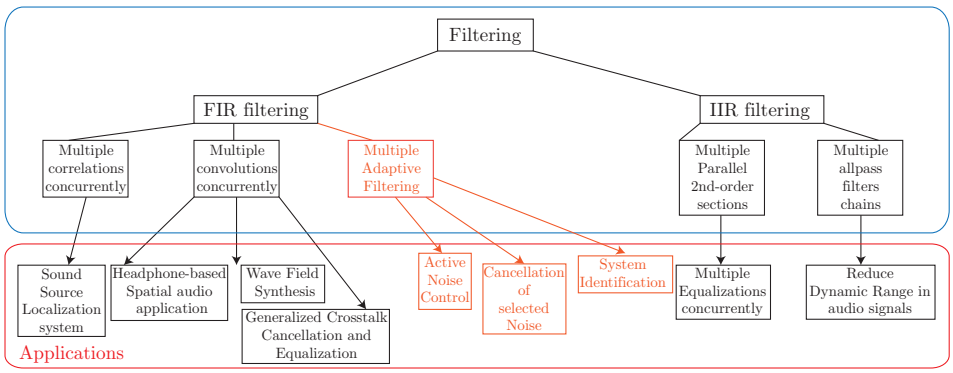


Figure 10.1. Developed and future applications that require massive multichannel signal processing.

Adaptive filtering is applied to modelling or system identification, and active noise control. The target of these applications consists of filtering some input signal to match a desired response. The filter parameters are updated by making some measurements and applying them to the adaptive filtering algorithm such that the difference between the filter output and the desired response is minimized. One common scenario occurs at home when the windows are opened in order to leave fresh air to enter home. In some countries, this is a large problem since the traffic noise is extremly loud (Asian countries mainly). A large system composed of loudspeakers

and microphones could reduce this noise pollution.

However, computing needs increase as the number of speakers and microphones increase. Thus, developing systems that require adaptive filtering in future many-core architectures is presented as a great challenge to pursue in the field of immersive and friendly scenarios.

On the other hand, as a consequence of the developed work throughout this dissertation, we have now a large number of computational kernels. As a future work, all these kernels could be packed in specific libraries for audio applications using GPUs. Libraries are valuable tools for specialists of a particular field, since it facilitates the development of scientific codes without knowing GPUs characteristics. These future libraries will also consider the new advances in the GPU architectures. Thus, it is expected that the performances of these libraries improve meaningfully the performances that are collected in this manuscript.

10.3 List of Publications

A list of published work produced during the course of candidature for the degree is presented in what follows.

Publications as First Author

Journal Papers indexed in JCR

- J. A. Belloch, M. Ferrer, A. Gonzalez, F. J. Martinez-Zaldivar, A. M. Vidal, "Headphone-based virtual spatialization of sound with a GPU accelerator", Journal of the Audio Engineering Society, vol. 61, No. 7/8, July/August 2013. *Impact Factor*: 0.831.

- J. A. Belloch, A. Gonzalez, F. J. Martinez-Zaldivar, A. M. Vidal, "Multichannel Massive Audio Processing for a Generalized Crosstalk Cancellation and Equalization application using GPUs", Integrated Computer-Aided Engineering - An International Journal, vol. 20, no.2 pp 169-182, April 2013. *Impact Factor*: 3.451.

- J. A. Belloch, A. Gonzalez, F. J. Martinez-Zaldivar, A. M. Vidal, "Real-time massive convolution for audio applications on GPU", Jour-

nal of Supercomputing, vol. 58, no. 3, pp. 449-457, December 2011. *Impact Factor*: 0.578.
Note that this paper is shown in NVIDIA SHOW CASES webpage: http://www.nvidia.co.uk/object/cuda-showcase-uk.html#

- J. A. Belloch, A. Gonzalez, A. M. Vidal, M. Cobos "On the Performance of Real-Time Massive Microphone Systems for Sound Source Localization Using Multiple GPUs", Submitted for publication.

International Conference Papers

- J. A. Belloch, J. Parker, L. Savioja, A. Gonzalez, V. Välimäki, "Dynamic Range Reduction of Audio Signals Using Multiple Allpass filters on a GPU accelerator", Accepted for publication in EUSIPCO 2014. Lisbon, Portugal, September 2014.

- J. A. Belloch, B. Bank, L. Savioja, A. Gonzalez, V. Välimäki, "Multichannel IIR Filtering of Audio Signals Using a GPU", Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2014). ISBN: 978-1-4799-2893-4. Florence, Italy, May 2014.

- J. A. Belloch, M. Ferrer, A. Gonzalez, J. Lorente, A. M. Vidal, "GPU-based WFS systems with mobile Virtual Sound Sources and Room Compensation", Proceedings of the 52nd Conference on Sound Field Control - Audio Engineering Society, Guildford, England, September 2013.

- J. A. Belloch, A. Gonzalez, A. M. Vidal, M. Cobos "Real-Time Sound Source Localization on Graphics Processing Units", International Conference on Computational Science (ICCs), pp. 2549 2552, Barcelona, Spain, June 2013.

- J. A. Belloch, M. Ferrer, A. Gonzalez, F. J. Martinez-Zaldivar, A. M. Vidal, "Headphone-based spatial sound with a GPU accelerator", International Conference on Computational Science (ICCs), pp. 116-125 Omaha, Nebraska, USA, June 2012.
doi: 10.1016/j.procs.2012.04.013

- J. A. Belloch, A. Gonzalez, F. J. Martinez-Zaldivar, A. M. Vidal, "A real-time crosstalk canceller on a notebook GPU", IEEE International Conference on Multimedia and Expo (ICME), I.S.B.N. 978-1-61284-348-3, pp. 1-4, Barcelona, Spain, July 2011.
  Note that this paper is shown in NVIDIA SHOW CASES webpage: http://www.nvidia.co.uk/object/cuda-showcase-uk.html#

- J. A. Belloch, F. J. Martinez-Zaldivar, A. M. Vidal, A. Gonzalez, "Analysis of GPU thread structure in a multichannel audio application", Proceedings of the 11th International Conference on Computational and Mathematical Methods in Science and Engineering, I.S.B.N. 978-84-614-6167-7 vol. 1, pp. 156-163, Benidorm, Spain, Junio 2011.

- J. A. Belloch, A. M. Vidal, F. J. Martinez-Zaldivar, A. Gonzalez, "Real-time Multichannel Audio Convolution", GPU Technology Conference 2010,
  http://nvidia.fullviewmedia.com/gtc2010/0923-n-2116.html,
  San José, California, USA, September 2010.

- J. A. Belloch, A. M. Vidal, F. J. Martinez-Zaldivar, A. Gonzalez, "Multichannel acoustic signal processing on GPU", Proceedings of the 10th International Conference on Computational and Mathematical Methods in Science and Engineering, I.S.B.N. 978-84-613-5510-5 vol. 1, pp. 181-187, Almería, Spain, June 2010.

Other coauthored publications related with this thesis

International Conference Papers

- J. Lorente, M. Ferrer, M. de Diego, J. A. Belloch, A. Gonzalez, "GPU Implementation of Frequency-domain Modified Filtered-x LMS algorithm for Multichannel Local Noise Control", Proceedings of the 52nd Conference on Sound Field Control - Audio Engineering Society, Guildford, England, September 2013.

- J. Lorente, J. A. Belloch, M. Ferrer, A. Gonzalez, "Multichannel Active Noise Control System using a GPU accelerator", Internoise, I.S.B.N. 0736-2935, pp. 13-24, New York, United States of America, August 2012.

- J. Lorente, A. Gonzalez, M. Ferrer, J. A. Belloch, M. de Diego, G. Piñero, A. M. Vidal, "Active Noise Control Using Graphics Processing Units", 19th International Congress on Sound and Vibration, 978-609-459-079-5, pp. 1-8, Vilnius, Lithuania, July 2012.

- J. Lorente, G. Piñero, A. M. Vidal, J. A. Belloch, A. Gonzalez, "Parallel implementations of beamforming design and filtering for microphone array applications", 19th European Signal Processing Conference, I.S.S.N. 2076-1465, pp 501-505 Barcelona, Spain, August 2011.

Peer-reviewed non-ISI Journal Papers

- J. Lorente, M. Ferrer, J. A. Belloch, G. Piñero, M. de Diego, A. Gonzalez, A. M. Vidal, "Real-time adaptive algorithms using GPUs", Waves, vol. 4, pp. 59-68, September 2012.

- A. Gonzalez, J. A. Belloch, F. J. Martinez-Zaldivar, P. Alonso, V. M. Garcia, E. S. Quintana-Ort, A. Remon, A. M. Vidal, "The Impact of the Multi-core Revolution on Signal Processing", Waves, vol. 2, pp. 64-75, September 2010.

- A. Gonzalez, J. A. Belloch, G. Piñero, J. Lorente, M. Ferrer, S. Roger , C. Roig, F. J. Martinez, M. de Diego, P. Alonso, V. M. Garcia, E. S. Quintana-Ort, A. Remon and A. M. Vidal, "Application of Multi-core and GPU Architectures on Signal Processing: Case Studies", Waves, vol. 2, pp. 86-96, September 2010.

Spanish Conference Papers

- P. Alonso, J. A. Belloch, A. Gonzalez, E. S. Quintana-Ort, A. Remon, A. M. Vidal, "Evaluación de bibliotecas de altas prestaciones para el clculo de la FFT en procesadores multincleo y GPUs", II Workshop en Aplicaciones de Nuevas Arquitecturas de Consumo y Altas Prestaciones, I.S.B.N. 978-84-692-7320-3, pp. 1-9 Mostoles, Spain, November 2009.

10.4 Institutional Acknowledgements

This work has received financial support of the following projects and stipends:

- Project TEC2009-13741: Spatial audio systems based on massive parallel processing of multichannel acoustic signals with general purpose-graphics processing units (GP-GPU) and multicores. (Spanish Ministry of Science and Innovation)
- Project TEC2012-38142-C04-01: Distributed and Collaborative Sound Signal Processing: algorithms, tools and applications. (Spanish Ministry of Science and Innovation)
- Project PROMETEO 2009/2013: Computación de altas prestaciones sobre arquitecturas actuales en problemas de procesado de múltiples señales. (Generalitat Valenciana).
- Projects PAID-05-11 and PAID-05-10: Programa de Apoyo a la Investigacin y Desarrollo (Universitat Politècnica de València).
- Stipend BES-2010-037793: FPI program (Spanish Ministry of Science and Innovation).
- Stipend EEBB-I-13-06059: Research Internship (Spanish Ministry of Science and Innovation).

Bibliography

[1] J. Parker and V. Välimäki, "Linear dynamic range reduction of musical audio using an allpass filter chain," *IEEE Signal Processing Letters*, vol. 20, no. 7, pp. 669–672, 2013.

[2] Y. A. Huang, J. Chen, and J. Benesty, "Immersive audio schemes," *IEEE Signal Processing Magazine*, vol. 28, no. 1, pp. 20–32, 2011.

[3] E. Torick, "Highlights in the history of multichannel sound," *J. Audio. Eng. Soc.*, vol. 46, no. 5, pp. 27–31, 1998.

[4] R. Rabenstein, S. Spors, and P. Steiffen, "Wave Field Synthesis techniques for spatial sound reproduction," *Topics in Acoustic Echo and Noise Control*, vol. 5, pp. 517–545, 2006.

[5] S. Spors, R. Rabenstein, and W. Herbordt, "Active listening room compensation for massive multichannel sound reproduction system using wave-domain adaptive filtering," *J. Acoustic. Soc. Am.*, vol. 122, pp. 354–369, 2007.

[6] Y. Huang, J. Benesty, and J. Chen, "Generalized crosstalk cancellation and equalization using multple loudspeakers for 3d sound reproduction at the ears of multiple listeners," in *IEEE Int. Conference on Acoustics, Speech and Signal Processing*, Las Vegas, USA, October 2008, pp. 405–408.

[7] N. Madhu and R. Martin, *Advances in Digital Speech Transmission*. New York, NY, USA: Wiley, 2008, ch. Acoustic Source Localization with Microphone Arrays, pp. 135–166.

[8] F. Rumsey, *Spatial Audio*. Elsevier, 2001.

[9] J. Blauert, *Spatial Hearing - Revised Edition: The Psychophysics of Human Sound Localization*. The MIT Press, 1996.

[10] R. Rabenstein, S. Spors, and J. Ahrens, *Spatial Sound Synthesis*. vol 4. Chapter 32. Academic Press Library in Signal Processing Oxford UK, 2014.

[11] V. Algazi and R. Duda, "Headphone-based spatial sound," *IEEE Signal Processing Magazine*, vol. 28, no. 1, pp. 33–42, 2011.

[12] D. Schnstein and B. Katz, "Variability in perceptual evaluation of HRTFs," *J. Audio Eng. Soc*, vol. 60, no. 10, pp. 783–793, 2012.

[13] A. Berkhout, D. de Vries, and P. Vogel, "Acoustic control by Wave Field Synthesis," *J. Acoustic. Soc. Amer*, vol. 93, pp. 2764–2778, May 1993.

[14] "Audio and Communications Signal Processing Group at Universitat Politecnica de Valencia," *http://www.gtac.upv.es/enlaces.asp*.

[15] "IOSONO Wave Field Synthesis System," *http://www.timelab-hhi.com/en/system-description/iosono-wave-field-synthesis-system.html*.

[16] E. Hulsebos, D. de Vries, and E. Bourdillat, "Improved microphone array configurations for auralization of sound fields by Wave-Field Synthesis," *J. Audio Eng. Soc*, vol. 50, no. 10, pp. 779–790, 2002.

[17] S. Spors, H. Buchner, and R. Rabenstein, "Efficient active listening room compensation for Wave Field Synthesis," in *Proceedings of the 116th AES Convention*, Berlin,Germany, May 2004.

[18] J. Lopez, A. Gonzalez, and L. Fuster, "Room compensation in Wave Field Synthesis by means of multichannel inversion," in *Applications of Signal Processing to Audio and Acoustics, 2005. IEEE Workshop on*, oct. 2005, pp. 146 – 149.

[19] J. H. DiBiase, H. F. Silverman, and M. S. Brandstein, "Robust localization in reverberant rooms," in *Microphone Arrays: Signal Processing Techniques and Applications*, M. S. Brandstein and D. Ward, Eds. Berlin, Germany: Springer-Verlag, 2001, ch. 8, pp. 157–180.

[20] "Nvidia CUDA Developer Zone," https://developer.nvidia.com/cuda-downloads, (accessed 2014 March 10).

[21] "CUDA ZONE: CUDA Community Showcase," http://www.nvidia.co.uk/object/cuda_apps_flash_new_uk.html#, (accessed 2014 January 28).

[22] J. Moura, "What is Signal Processing?" *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 6–6, 2009.

[23] A. Kwasinski, M. Kaveh, and L. Deng, "The Discipline of Signal Processing: Part 2," *IEEE Signal Processing Magazine*, vol. 31, no. 1, pp. 157–159, 2014.

[24] A. Oppenheim, A. Willsky, and S. Nawab, *Signals and Systems*, 1996.

[25] T. Apostol, *Mathematical Analysis*, 1960.

[26] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, pp. 297 – 301, 1965.

[27] S. S. Soliman and M. D. Srinath, *Continuous and Discrete Signals and Systems*, Prentice Hall, 1997.

[28] V. Välimäki, J. D. Parker, L. Savioja, J. O. Smith, and J. Abel, "Fifty Years of Artificial Reverberation," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 5, pp. 1421–1448, 2012.

[29] A. Torger and A. Farina, "Real-time partitioned convolution for Ambiophonics surround sound," in *IEEE Workshop on the Applications of Signal Processing to Audio and Acoustics*, 2001, pp. 195–198.

[30] E. C. Ifeachor and B. W. Jervis, *Digital signal processing: a practical approach*. Prentice-Hall, 2002.

[31] "Xilinx,"
http://www.xilinx.com/, (accessed 2014 March 03).

[32] N. Tsingos, W. Jiang, and I. Williams, "Using programmable graphics hardware for acoustics and audio rendering," *J. Audio Eng. Soc*, vol. 59, no. 9, pp. 628–646, 2011.

[33] G. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, pp. 114–117, 1965.

[34] G. Blake, R. Dreslinski, and T. Mudge, "A Survey of Multicore Processors," *IEEE Signal Process. Mag.*, vol. 26, no. 6, pp. 26–37, 2009.

[35] E. Lindholm, M. Kilgard, and H. Moreton, "A User-Programmable Vertex Engine." in *Proceeding of the 28th Ann. Conf. on Computer Graphics and Interactive Techniques*, Los Angeles, August 2001.

[36] M. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. 21, pp. 948–960, 1972.

[37] "NVIDIA Kepler Architecture,"
http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, (accessed 2013 May 28).

[38] "Features of the Nvidia CUDA capabilities,"
https://developer.nvidia.com/cuda-gpus, (accessed 2014 April 16).

[39] S. Cook, *A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann, 2013.

[40] "openMP API Specifications,"
http://www.openmp.org, (accessed 2013 June 05).

[41] "Steinberg Media Technologies Gmbh,"
http://www.steinberg.net/en/company/developer.html, (accessed 2014 April 18).

[42] "Generic ASIO drivers asio4all,"
http://www.asio4all.com/, (accessed 2014 April 18).

[43] B. S. Atal and M. R. Schroeder, "Apparent sound source translator," U.S. Patent 3,236,949, Tech. Rep., 1966.

[44] B. B. Bauer, "Stereophonic earphones and binaural loudspeakers," *J. Audio Eng. Soc*, vol. 9, no. 1, pp. 148–151, 1961.

[45] S. M. Kuo and G. H. Canfield, "Dual-channel audio equalization and cross-talk cancellation for 3-D sound reproduction," *IEEE Transactions Consum. Electron.*, vol. 43, no. 4, p. 11891196, 1997.

[46] S. Miyabe, M. Shimada, T. Takatani, H. Saruwatari, and K. Shikano, "Multi-channel inverse filtering with selection and enhancement of a loudspeaker for robust sound field reproduction," in *Proc. of IWAENC 2006*, Paris, France, September 2006.

[47] K. Matsui, "Binaural reproduction of 22.2 multichannel sound over frontal loudspeakers," in *Proc. of 3DSA 2013*, Osaka, Japan, June 2013.

[48] J. J. Lopez, A. Gonzalez, and F. Ordua-Bustamante, "Measurement of cross-talk cancellation and equalization zones in 3-D sound reproduction under real listening conditions," in *Proc. of the 16th AES Conference*, Rovaniemi, Finland, April 1999.

[49] O. Kirkeby, P. Rubak, L. G. Johansen, and P. A. Nelson, "Implementation of Cross-talk Cancellation Networks Using Warped FIR Filters," in *Proc. of the 16th AES Conference*, Rovaniemi, Finland, April 1999.

[50] "openGL," *online at: http://www.opengl.org/.*

[51] B. Cowan and B. Kapralos, "GPU-Based One-Dimensional Convolution for Real-Time Spatial Sound Generation," *Loading...: The Journal of the Canadian Game Studies Association*, vol. 3, no. 5, pp. 1–14, 2009.

[52] F. Wefers and J. Berg, "High-Performance real-time FIR-filtering using fast convolution on graphics hardware," in *Proc. of the 13th Conference on Digital Audio Effects*, Graz, Austria, September 2010.

[53] L. Savioja, V. Välimäki, and J. O. Smith, "Audio Signal Processing using Graphics Processing Units," *J. Audio Eng. Soc*, vol. 59, no. 1-2, pp. 3–19, 2011.

[54] J. Blauert and all, *The technology of binaural listening*, 2013.

[55] S. Spors and J. Ahrens, "Efficient range extrapolation of head-related impulse responses by Wave Field Synthesis techniques," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, Prague, Czech Republic, May 2011.

[56] Y. Kahana and P. A. Nelson, "Numerical modelling of the spatial acoustic response of the human pinna," *Journal of Sound and Vibration*, vol. 292, no. 1-2, pp. 148 – 178, 2006.

[57] G. Enzner, M. Krawczyk, F.-M. Hoffmann, and M. Weinert, "3d reconstruction of hrtf-fields from 1d continuous measurements," in *Applications of Signal Processing to Audio and Acoustics (WASPAA), 2011 IEEE Workshop on*, oct. 2011, pp. 157 –160.

[58] "Room Acoustics Team, IRCAM Database," *online at: http://recherche.ircam.fr/equipes/salles/listen/index.html*.

[59] "The CIPIC HRTF Database," *online at: http://interface.cipic.ucdavis.edu/sound/hrtf.html*.

[60] E. Gallo and N. Tsingo, "Efficient 3D Audio Processing with the GPU," in *GP2: ACM Workshop on General Purpose Computing on Graphics Processors*, Los Angeles, USA, August 2004.

[61] S. Siltaten, T. Lokki, and L. Savioja, "Frequency domain acoustic radiance transfer for real-time auralization," *Acta Acustica/Acustica*, vol. 95, pp. 106–117, 2009.

[62] B. Cowan and B. Kapralos, "Spatial sound for video games and virtual environments utilizing real-time GPU-Based Convolution," in *Proc. 2008 Conf. on Future Play: Research, Play, Share*, Ontario, Canada, November 2008.

[63] A. Berkhout, "A holographic approach to acoustic control," *J. of the Audio Engineering Society*, vol. 36, pp. 2764–2778, May 1988.

[64] P. Vogel, "Application of Wave Field Synthesis in room acoustics," Ph.D. dissertation, Delft University of Technology, 1993.

[65] E. Start, "Direct sound enhancement by Wave Field Synthesis," Ph.D. dissertation, Delft University of Technology, 1997.

[66] E. Verheijen, "Sound reproduction by Wave Field Synthesis," Ph.D. dissertation, Delft University of Technology, 1997.

[67] J.-J. Sonke, "Variable acoustics by Wave Field Synthesis," Ph.D. dissertation, Delft University of Technology, 2000.

[68] E. Hulsebos, "Auralization using Wave Field Synthesis," Ph.D. dissertation, Delft University of Technology, 2004.

[69] L. Romoli, P. Peretti, S. Cecchi, L. Palestini, and F. Piazza, "Real-time implementation of Wave Field Synthesis for sound reproduction systems," in *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, 30 2008-dec. 3 2008, pp. 430 –433.

[70] D. Theodoropoulos, G. Kuzmanov, and G. Gaydadjiev, "A minimalistic architecture for reconfigurable wfs-based immersive-audio," in *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, dec. 2010, pp. 1 –6.

[71] ——, "Multi-core platforms for beamforming and Wave Field Synthesis," *IEEE Transactions on multimedia*, vol. 3, no. 2, pp. 235–245, April 2011.

[72] M. Brandstein and D. Ward, *Microphone arrays*, B. Verlag, Ed. Springer, 2001.

[73] J. Chen, J. Benesty, and Y. Huang, "Time delay estimation in room acoustic environments: an overview," *EURASIP Journal on Applied Signal Processing*, vol. 2006, pp. 1–19, 2006.

[74] B. Xu, G. Sun, R. Yu, and Z. Yang, "High-Accuracy TDOA-Based Localization without Time Synchronization," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, no. 8, pp. 1567–1576, 2013.

[75] C. H. Knapp and G. C. Carter, "The generalized correlation method for estimation of time delay," *Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-24, pp. 320–327, 1976.

[76] V. Peruffo Minotto, C. Rosito Jung, L. Gonzaga da Silveira, and B. Lee, "GPU-based approaches for real-time sound source localization using the SRP-PHAT algorithm," *International Journal of High Performance Computing Applications*, 2012.

[77] Y. Liang, Z. Cui, S. Zhao, K. Rupnow, Y. Zhang, D. L. Jones, and D. Chen, "Real-time implementation and performance optimization of 3D sound localization on GPUs," in *DATE'12*, 2012, pp. 832–835.

[78] C. J. Webb and S. Bilbao, "Virtual room acoustics: A comparison of techniques for computing 3D-FDTD schemes using CUDA," in *Proceedings of the 130th AES Convention*, London, U.K., May 2011.

[79] L. Savioja, "Real-time 3D finite-difference time-domain simulation of low- and mid-frequency room acoustics," in *Proc. of the Int. Conf. Digital Audio Effects*, Graz, Austria, September 2010.

[80] A. Southern, D. Murphy, G. Campos, and P. Dias, "Finite difference room acoustic modelling on a General Purpose Graphics Processing Unit," in *Proc. of the 128th AES Convention*, London, United Kingdom, May 2010.

[81] B. Hamilton and C. J. Webb, "Room acoustics modelling using GPU-accelerated finite difference and finite volume methods on a face-centered cubic grid," in *Proc. Conference on Digital Audio Effects (DAFx-13)*, Maynooth, Ireland, September 2013.

[82] M. Jedrzejewski and K. Marasek, "Computation of room acoustics using programmable video hardware," *Computational Imaging and Vision*, vol. 32, pp. 587–592, September 2006.

[83] N. Rober, U. Kaminski, and M. Masuch, "Ray acoustics using computer graphics technology," in *Conference on Digital Audio Effects (DAFx-07) proceedings*, Bourdeaux, France, June 2007.

[84] L. Savioja, V. Välimäki, and J. O. Smith, "Real-time additive synthesis with one million sinusoids using a GPU," in *Proc. of the 128th AES Convention*, London, United Kingdom, May 2010.

[85] R. Bradford, J. Ffitch, and R. Dobson, "Real-time sliding phase vocoder using a commodity GPU," in *Proc. of ICMC 2011*, University of Huddersfield, United Kingdom, August 2011.

[86] J. Lorente, G. Piñero, A. Vidal, J. Belloch, and A. Gonzalez, "Parallel implementations of beamforming design and filtering for microphone array applications," in *Proc. of EUSIPCO 2011*, Barcelona, Spain, August 2011.

[87] "NVIDIA Library CUFFT, howpublished = http://docs.nvidia.com/cuda/pdf/cufft_library.pdf, note = (accessed 2014 July 23)."

[88] J. A. Belloch, A. M. Vidal, F. J. Martínez-Zaldívar, and A. Gonzalez, "Multichannel acoustic signal processing on GPU," in *Proceedings of the 10th International Conference on Computational and Mathematical Methods in Science and Engineering*, vol. 1, Almeria, Spain, June 2010, pp. 181–187.

[89] "Nvidia CUDA toolkit 2.3," https://developer.nvidia.com/cuda-toolkit-23-downloads, (accessed 2014 April 15).

[90] J. A. Belloch, A. M. Vidal, F. J. Martínez-Zaldívar, and A. Gonzalez, "Real-time Multichannel Audio Convolution," in *GPU Technology Conference 2010*, San Jose, California, USA, September 2010, http://www.gputechconf.com/gtcnew/on-demand-gtc.php.

[91] 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*, vol. 58, no. 3, pp. 449–457, December 2011.

[92] M. Gardner, "Historical background of the haas and/or precedence effect," *J. Acoust. Soc. Am.*, vol. 43, no. 6, pp. 1243–1248, 1968.

[93] B. S. Xie and S. Q. Guan, "Some Recent Works on Head-Related Transfer Functions and Virtual Auditory Display in China," in *Proc. of the 40th AES Conference*, Tokyo, Japan, October 2010.

[94] O. Kirkeby, P. Nelson, H. Hamada, and F. Orduna-Bustamante, "Fast deconvolution of multichannel systems using regularization," *Speech and Audio Processing, IEEE Transactions on*, vol. 6, no. 2, pp. 189 –194, mar 1998.

[95] W. Chu, "Impulse response and reverberation-decay measurements made by using a periodic pseudorandom sequence," *Applied Acoustics*, vol. 29, pp. 193–205, 1990.

[96] J. A. Belloch, A. Gonzalez, F. J. Martinez-Zaldivar, and A. M. Vidal, "A real-time crosstalk canceller on a notebook GPU," in *2011*

IEEE International Conference on Multimedia and Expo (ICME), Barcelona, Spain, july 2011, pp. 1 –4.

[97] A. V. Oppenheim, A. S. Willsky, and S. Hamid, "Signals and systems," ser. Processing series. Prentice Hall, 1997.

[98] G. Garcia, "Optimal filter partition for efficient convolution with short input/output delay," in *Proceedings of the 113th AES Convention*, Los Angeles,U.S.A., October 2002.

[99] W. G. Gardner, "Efficient convolution without input-output delay," *Journal of the Audio Engineering Society*, vol. 43, pp. 127–136, 1995.

[100] P. M. Gerald and P. C. W. Sommen, "A new method for efficient convolution in frequency domain by nonuniform partitioning for adaptive filtering," *IEEE Transactions on signal processing*, vol. 44, pp. 127–136, 1996.

[101] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju, "Auto-tuning of fast Fourier transform on graphics processors," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11, 2011, pp. 257–266.

[102] J. A. Belloch, F. J. Martínez-Zaldívar, A. M. Vidal, and A. Gonzalez, "Analysis of GPU thread structure in a multichannel audio application," in *Proceedings of the 11th International Conference on Computational and Mathematical Methods in Science and Engineering*, vol. 1, Benidorm, Spain, June 2011, pp. 156–163.

[103] J. A. Belloch, A. Gonzalez, F. Martinez-Zaldivar, and A. M. Vidal, "Multichannel massive audio processing for a generalized crosstalk cancellation and equalization application using GPUs," *Integrated Computer-Aided Engineering*, vol. 20, no. 2, pp. 169–182, 2013.

[104] A. Kudo, H. Hokari, and S. Shimada, "A study on switching of the transfer functions focusing on sound quality," *Acoustical Science and Technology*, vol. 26, no. 3, pp. 267–278, 2005.

[105] H. David, *The method of paired comparisons*, ser. Griffin's statistical monographs & courses. Griffin, 1963.

[106] F. Keyrouz and K. Diepold, "A rational hrtf interpolation approach for fast synthesis of moving sound," in *Digital Signal Processing Workshop, 12th - Signal Processing Education Workshop, 4th*, sept. 2006, pp. 222 –226.

[107] ——, "A new HRTF interpolation approach for fast synthesis of dynamic environmental interaction," *J. Audio Eng. Soc*, vol. 56, no. 1/2, pp. 28–35, 2008. [Online]. Available: http://www.aes.org/e-lib/browse.cfm?elib=14373

[108] D. R. Begault, *3-D sound for virtual reality and multimedia*. San Diego, CA, USA: Academic Press Professional, Inc., 1994.

[109] M. Matsumoto, S. Yamanaka, M. Toyama, and H. Nomura, "Effect of Arrival Time Correction on the Accuracy of Binaural Impulse Response Interpolation–Interpolation Methods of Binaural Response," *J. Audio Eng. Soc*, vol. 52, no. 1/2, pp. 56–61, 2004.

[110] S. M. Robeson, "Spherical methods for spatial interpolation: review and evaluation," *Cartography and Geographic Information Systems*, vol. 24, no. 1, pp. 3–20, 1997.

[111] K.-S. Lee and S.-P. Lee, "A relevant distance criterion for interpolation of head-related transfer functions," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 19, no. 6, pp. 1780 –1790, aug. 2011.

[112] J. A. Belloch, M. Ferrer, A. Gonzalez, F. Martinez-Zaldivar, and A. M. Vidal, "Headphone-based spatial sound with a gpu accelerator," *Procedia Computer Science*, vol. 9, no. 0, pp. 116 – 125, 2012, proceedings of the International Conference on Computational Science, ICCS 2012.

[113] ——, "Headphone-based virtual spatialization of sound with a GPU accelerator," *J. Audio Eng. Soc*, vol. 61, no. 7/8, pp. 546–561, 2013.

[114] M. M. Boone, E. N. G. Verheijen, and P. F. Van Tol, "Spatial Sound-Field Reproduction by Wave-Field Synthesis," *J. Audio Eng. Soc*, vol. 43, no. 12, pp. 1003–1012, 1995.

[115] S. Spors, A. Kuntz, and R. Rabenstein, "An Approach to Listening Room Compensation with Wave Field Synthesis," in *Proc. of the 24th AES Conference*, Banff, Canada, May 2003.

[116] S. Spors and J. Ahrens, "Analysis and Improvement of Pre-equalization in 2.5-Dimensional Wave Field Synthesis," in *Proceedings of the 128th AES Convention*, London, UK, May 2010.

[117] L. Fuster, J. J. Lopez, A. Gonzalez, and P. Faus, "Time and frequency domain room compensation applied to Wave Field Synthesis," in *Proc. Conference on Digital Audio Effects (DAFx-05)*, Madrid, Spain, September 2005.

[118] M. Miyoshi and Y. Kaneda, "Inverse filtering of room acoustics," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 36, no. 2, pp. 145 –152, feb 1988.

[119] G. Jansen, "Focused wavefields and moving virtual sources by wavefield synthesis," Master's thesis, Delft University of Technology, 1997.

[120] H. Kuttruff, *Room acoustics*, S. Press, Ed. Abingdon, Oxford, UK: Taylor & Francis, October 2000, 368 pages.

[121] F. Wefers and M. Vorländer, "Optimal filter partitions for real-time FIR filtering using uniformly-partitioned FFT-based convolution in the frequency-domain," in *Proc. of the 14th Conference on Digital Audio Effects*, Paris, France, September 2010.

[122] J. A. Belloch, M. Ferrer, A. Gonzalez, J. Lorente, and A. M. Vidal, "GPU-based WFS Systems with Mobile Virtual Sound Sources and Room Compensation," in *Proc. of the 52nd AES Conference*, Guildford, United Kingdom, September 2013.

[123] H. Do and H. F. Silverman, "A fast microphone array SRP-PHAT source location implementation using coarse-to-fine region contraction (CFRC)," in *Proc. of the IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, New Paltz, USA, September 2007.

[124] A. Said, B. Lee, and T. Kalker, "Fast steered response power computation in 3D spatial regions," HP Labs, Palo Alto, USA, Tech. Rep. HPL-2013-40, April 2013.

[125] A. Marti, M. Cobos, and J. J. Lopez, "A steered response power iterative method for high-accuracy acoustic source location," *Journal of the Acoustical Society of America*, vol. 134, no. 4, 2013.

[126] J. H. DiBiase, "A high accuracy, low-latency technique for talker localization in reverberant environments using microphone arrays," Ph.D. dissertation, Brown University, Providence, RI, May 2000.

[127] M. Cobos, A. Marti, and J. J. Lopez, "A modified SRP-PHAT functional for robust real-time sound source localization with scalable spatial sampling," *IEEE Signal Processing Letters*, vol. 18, no. 1, pp. 71–74, January 2011.

[128] S. Bilbao and C. J. Webb, "Physical modeling of timpani drums in 3D on GPGPUs," *J. Audio Eng. Soc*, vol. 61, no. 10, pp. 737–748, 2013.

[129] "Optimizing Parallel Reduction in CUDA NVIDIA," http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf, (accessed 2013 June 10).

[130] J. B. Allen and D. A. Berkley, "Image method for efficiently simulating small-room acoustics," *J. Acoust. Soc. Am.*, vol. 65, no. 4, pp. 943–950, 1979.

[131] J. A. Belloch, A. Gonzalez, A. M. Vidal, and M. Cobos, "Real-Time Sound Source Localization on Graphics Processing Units," in *Proc. of the International Conference on Computational Science, ICCS 2013*, Barcelona, Spain, June 2013.

[132] J. A. Belloch, J. Parker, L. Savioja, A. Gonzalez, and V. Välimäki, "Dynamic Range Reduction of Audio Signals Using Multiple Allpass Filters on a GPU Accelerator," in *Accepted for publication in EUSIPCO*, Lisbon, Portugal, September 2014.

[133] A. Härmä, M. Karjalainen, L. Savioja, V. Välimäki, and J. Huopaniemi, "Frequency-Warped Signal Processing for Audio Applications," *J. Audio Eng. Soc*, vol. 48, no. 11, pp. 1011–1031, 2000.

[134] M. Karjalainen and T. Paatero, "Equalization of loudspeaker and room responses using Kautz filters: Direct least squares design," *EURASIP J. on Advances in Sign. Proc., Spec. Iss. on Spatial Sound and Virtual Acoustics*, vol. 2007, p. 13, 2007.

[135] B. Bank, "Perceptually motivated audio equalization using fixed-pole parallel second-order filters," *IEEE Signal Processing Letters*, vol. 15, pp. 477–480, 2008.

[136] K. Steiglitz and L. E. McBride, "A technique for the indentification of linear systems," *IEEE Trans. Autom. Control*, vol. AC-10, pp. 461–464, Oct. 1965.

[137] L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*. Englewood Cliffs, New Jersey, USA: Prentice-Hall, 1975.

[138] B. Bank, "Audio Equalization with Fixed-Pole Parallel Filters: An Efficient Alternative to Complex Smoothing," *J. Audio Eng. Soc*, vol. 61, no. 1/2, pp. 39–49, 2013.

[139] ——, "Loudspeaker and room response equalization using parallel filters: Comparison of pole positioning strategies," in *Proc. 51st AES Conf.*, Helsinki, Finland, Aug. 2013.

[140] ——, "Logarithmic Frequency Scale Parallel Filter Design with Complex and Magnitude-Only Specifications," *IEEE Signal Processing Letters*, vol. 18, no. 2, pp. 138–141, Feb. 2011.

[141] J. A. Belloch, B. Bank, L. Savioja, A. Gonzalez, and V. Välimäki, "Multi-channel IIR Filtering of Audio Signals Using a GPU," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Florence, Italy, May 2014.

[142] U. Zölzer, "Dafx - digital audio effects (second edition)," Edited by Udo Zölzer, 2011.

[143] E. Vickers, "The loudness war: Do louder, hypercompressed recordings sell better?" *J. Audio Eng. Soc*, vol. 59, no. 5, pp. 346–351, 2011.

[144] G. Giannoulis, M. Massberg, and J. Reiss, "Digital dynamic range compressor design: A tutorial and analysis," *J. Audio Eng. Soc*, vol. 60, no. 6, pp. 399–408, 2012.

[145] J. Kates and K. Arehart, "Multichannel dynamic-range compression using digital frequency warping," *EURASIP J. on Applied Signal Process.*, vol. 18, pp. 3003–3014, 2005.

[146] D. Griesinger, "Impulse response measurements using all-pass deconvolution," in *Proceedings of the 11th AES Conference*, Portland, May 1992.

[147] M. Schroeder and B. Logan, "Colorless artificial reverberation," *J. Audio Eng. Soc*, vol. 9, no. 3, pp. 192–197, 1961.

[148] V. Välimäki, J. Parker, L. Savioja, J. Smith, and J. Abel, "Fifty years of artificial reverberation," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 5, pp. 1421–1448, 2012.

[149] V. Välimäki, J. Abel, and J. Smith, "Spectral delay filters," *J. Audio Eng. Soc*, vol. 57, no. 7-8, pp. 521–531, 2009.

[150] V. Välimäki, J. Parker, and J. Abel, "Parametric spring reverberation effect," *J. Audio Eng. Soc*, vol. 58, no. 7-8, pp. 547–562, 2010.

[151] W. Blizard, "Multiset theory," *J. Formal Logic Notre Dame*, vol. 30, no. 1, pp. 36–66, 1989.

Appendix A

Appendix A

A.1 Alternative Multi-GPU Parallelization strategy

The challenge of this strategy consists in parallelizing the computation of the **GCC** matrix. Initially, all the GPUs must have access to this matrix since each point of the **SRP** matrix requires a contribution from each pair of microphones (each row of the **GCC** matrix).

The strategy that we present aims at achieving a good trade-off between the total operations carried out in each GPU and the number of transferred audio buffers. For example, if the number of microphones is $M = 12$, the number of pairs to compute in **GCC** matrix is $Q = 66$. These pairs are distributed among the N_{GPU} in a pseudo-triangular way. Figure A.1 shows the distribution of the computation and audio buffers among 2, 3 and 4 GPUs. The notation 01 x 05, indicates the element-wise multiplication of vector 1 and vector 5 of all computed vectors $\mathbf{f}_m$, $m = 0, \ldots, M-1$ (see step 2 of section 7.3). Note that the GPU that performs more multiplications deals with less audio buffers, minimizing the data transfers between CPU and GPU. This triangular structure can be considered independently of the number of microphones.

Finally, after the distributed computation of the **GCC** matrix, all GPUs need all of the rows of the **GCC** matrix in order to compute their corresponding ν/N_{GPU} elements of the **SRP** matrix. The use of UVA (see Appendix 2.6.2) allows each GPU to access other GPU via peer-to-peer over the PCI-E bus rather than copying data back to the host and then to another GPU. Thus, each GPU transparently accesses the memories of other GPUs by just referencing a memory location.

A.1.1 Basic Implementation using two GPUs

Using all the parallelization techniques presented in Chapter 2, the SRP-PHAT algorithm is implemented on two GPUs as follows:

1. A parallel region is created with two CPU threads. Each CPU thread is bound with a GPU.
2. Since different audio buffers are received in the system, each CPU thread independently and asynchronously sends its corresponding audio buffers to its GPU by using stream parallelization. The Kernels A and the FFTs are computed for each channel inside the streams.
3. As in step 2 of Section 7.3, stream synchronization is addressed. Only one stream is used to compute the rows of the **GCC** matrix. According to Figure A.1, in the case of $M = 12$, one GPU would compute 35 vectors and the other one would compute 31 vectors.
4. By using UVA, each GPU has access to the whole **GCC** matrix in order to compute $\nu/2$ elements of the **SRP** matrix and locates a maximum value among the computed elements.
5. Each GPU transfers back to the CPU its maximum value and its location inside the **SRP** matrix. Then, a synchronization barrier for both CPU threads is set followed by an *openMP* section that is only executed by the master thread. This thread compares the two maximum values and chooses the greatest one, getting its location. This location indicates the sound source position.

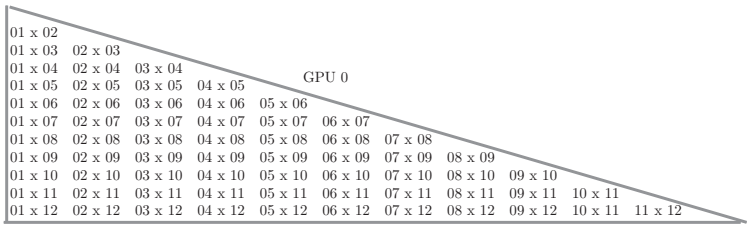
A.1.2 Comparison between strategies

Table A.1 shows the speed up that the implementation strategy presented in section 7.3.3 achieves with respect to the strategy presented in this ap-

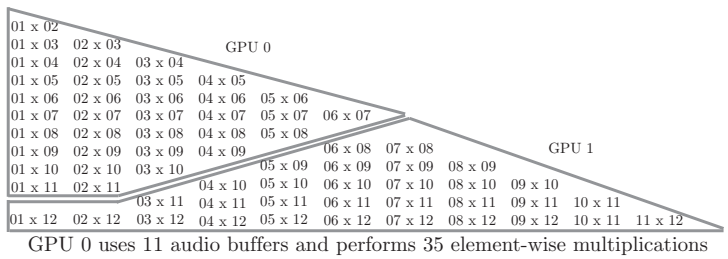
pendix. Two important aspects significantly penalize the performance of this strategy in comparison with the strategy in section 7.3.3. First, since each GPU does not contain the whole **GCC** matrix, each GPU must access the *global-memory* of the other GPU in order to compute the **SRP** matrix; second, after computing the corresponding elements of the **GCC** matrix, both GPUs must be synchronized.

Table A.1. Speed up between strategies.

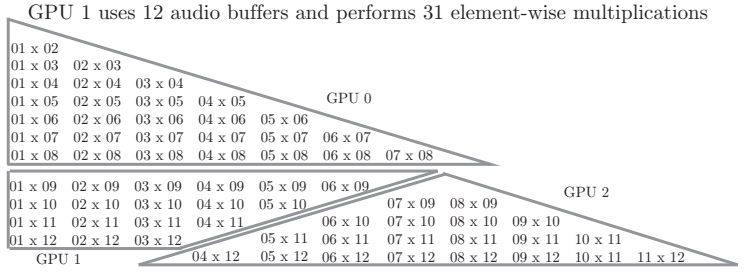
r_{sp}	$M = 6$	$M = 12$	$M = 24$	$M = 48$
0.01	30.097	35.443	36.968	31.649
0.05	12.259	24.043	31.291	43.313
0.1	4.815	9.861	15.310	21.249



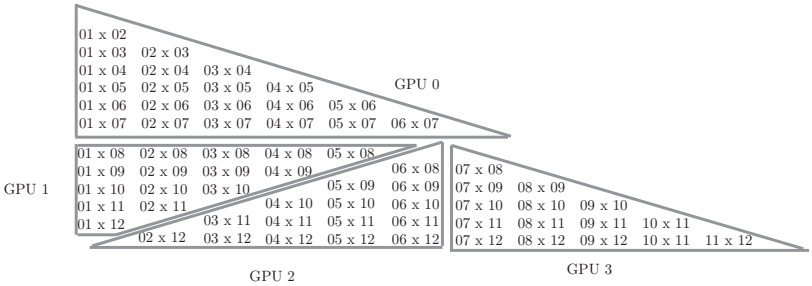
GPU 0 uses 12 audio buffers and performs 66 element-wise multiplications



GPU 0 uses 11 audio buffers and performs 35 element-wise multiplications
GPU 1 uses 12 audio buffers and performs 31 element-wise multiplications



GPU 0 uses 08 audio buffers and performs 28 element-wise multiplications
GPU 1 uses 10 audio buffers and performs 18 element-wise multiplications
GPU 2 uses 09 audio buffers and performs 20 element-wise multiplications



GPU 0 uses 07 audio buffers and performs 21 element-wise multiplications
GPU 1 uses 09 audio buffers and performs 15 element-wise multiplications
GPU 2 uses 10 audio buffers and performs 15 element-wise multiplications
GPU 3 uses 06 audio buffers and performs 15 element-wise multiplications

Figure A.1. Distribution of the audio buffers in order to compute the rows of the **GCC** matrix when N_{GPU} is 1,2,3 and 4.