

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

FERNANDO TREBIEN

**An Efficient GPU-based Implementation of
Recursive Linear Filters and Its Application
to Realistic Real-Time Re-Synthesis for
Interactive Virtual Worlds**

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Prof. Dr. Manuel Menezes de Oliveira Neto
Advisor

Porto Alegre, August 2009

CIP – CATALOGING-IN-PUBLICATION

Trebien, Fernando

An Efficient GPU-based Implementation of Recursive Linear Filters and Its Application to Realistic Real-Time Re-Synthesis for Interactive Virtual Worlds / Fernando Trebien. – Porto Alegre: PPGC da UFRGS, 2009.

75 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2009. Advisor: Manuel Menezes de Oliveira Neto.

1. Digital filters. 2. Linear filters. 3. Recursive filters. 4. Signal processing. 5. Sound synthesis. 6. Sound effects. 7. GPU. 8. GPGPU. 9. Realtime systems. I. Neto, Manuel Menezes de Oliveira. II. Título.

Reitor: Prof. Carlos Alexandre Netto

Vice-Reitor: Prof. Rui Vicente Oppermann

Pró-Reitor de Pós-Graduação: Prof. Aldo Bolten Lucion

Diretor do Instituto de Informática: Prof. Flávio Rech Wagner

Coordenador do PPGC: Prof. Álvaro Freitas Moreira

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

ACKNOWLEDGMENTS

Firstly, I would like to thank my research advisor, Prof. Manuel Menezes de Oliveira Neto, for his long-lasting support and incentive which made this work possible, and also for teaching me the correct ways of scientific thinking. I also thank my previous teachers for their dedication in transferring me invaluable knowledge. In special, I would like to thank Prof. Marcelo de Oliveira Johann for our discussions on computer music which enlightened my choice of topic to work on, and Prof. Luigi Carro for his great introductory lessons on signal processing and for all the answers he provided me in the course of this research. Also, I thank Alfredo Barcellos for his implementation of the graphics application that we use to demonstrate this work. I must yet thank all the staff working at our Instituto de Informática, responsible for operating crucial facilities such as the library and the network and for keeping laboratories clean and ready for use.

Additionally, I must acknowledge the research organization CNPq for sponsoring this research work¹. It is thanks to this organization that Brazilian people can continue to produce high-quality scientific work. NVIDIA also kindly donated the GeForce 9800 GTX graphics card used in this research.

Finally, I thank my parents for supporting and aiding me in this achievement. My friends deserve many thanks as well, since they listened to me with excitement and collaborated with a few ideas themselves, embraced me in difficult times, and, most importantly, inspired me continuously to pursue this dream, along with many others.

¹ Fernando Trebien's fellowship number: 130732/2007-9

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS	6
LIST OF FIGURES	8
LIST OF TABLES	9
LIST OF PROCEDURES	10
ABSTRACT	11
RESUMO	12
1 INTRODUCTION	13
1.1 Structure of This Thesis	15
2 RELATED WORK	16
2.1 Theoretical Signal Processing	16
2.2 GPU-Based Audio Processing	17
2.3 Summary	19
3 FUNDAMENTALS OF SIGNAL PROCESSING	20
3.1 Basic Concepts of Sound and Acoustics	20
3.2 Audio Processing Systems	22
3.2.1 Representations of Digital Audio	22
3.2.2 Block Processing	23
3.3 Summary	24
4 FILTER THEORY	25
4.1 The Linear Filter Equation	26
4.2 Filter Response to Input	27
4.3 Convolution	27
4.4 Fourier Representations of Signals	27
4.4.1 Fast Fourier Transform Algorithms	28
4.5 Pole-Zero Representations and the z-Plane	29
4.6 Filter Implementation and Stability	30
4.7 Summary	31

5	COMPUTER PLATFORMS FOR SIGNAL PROCESSING	33
5.1	Digital Signal Processors	33
5.2	The CPU	33
5.3	The GPU	34
5.4	Suitability for Audio Processing	35
5.5	Summary	37
6	EFFICIENT EVALUATION OF RECURSIVE LINEAR FILTERS ON THE GPU	39
6.1	Dependences in Linear Filtering	39
6.2	Unrolling the Filter Equation	40
6.3	Computing the Convolutions	42
6.4	LPC Extraction and Re-Synthesis	45
6.4.1	Stability Concerns	45
6.5	Summary	47
7	VALIDATION OF THE PROPOSED TECHNIQUE	50
7.1	Interactive Graphics Interface Process	50
7.2	Inter-Process Communication Protocol	50
7.3	Audio Generation Process	51
7.4	Performance Comparison	55
7.5	Summary	56
8	CONCLUSION	57
8.1	Future Work	58
	REFERENCES	60
	APPENDIX A SOURCE CODE LISTINGS	65
	APPENDIX B UMA IMPLEMENTAÇÃO EFICIENTE BASEADA EM GPUS DE FILTROS LINEARES RECURSIVOS E SUA APLICAÇÃO A RE-SÍNTESE REALÍSTICA EM TEMPO-REAL PARA MUNDOS VIRTUAIS INTERATIVOS	70
B.1	Trabalhos Relacionados	71
B.2	Avaliação Eficiente de Filtros Lineares Recursivos na GPU	72
B.2.1	Extração LPC e Re-Síntese	72
B.3	Validação da Técnica Proposta	73
B.4	Conclusões	74

LIST OF ABBREVIATIONS AND ACRONYMS

1D	One dimension, one dimensional (mathematical concept)
2D	Two dimensions, two dimensional (mathematical concept)
3D	Three dimensions, three dimensional (mathematical concept)
A/D	Analog-to-digital (electronics concept)
AGP	Accelerated Graphics Port (bus interface)
AMD	Advanced Micro Devices (company)
API	Application programming interface (design concept)
ASIO	Audio Stream Input Output ² (software interface)
ATI	Array Technologies Incorporated (former company)
CD	Compact disc (recording media)
CPU	Central Processing Unit (hardware component)
CTM	Close to Metal ³ (software interface)
CUBLAS	CUDA Basic Linear Algebra Subprograms ⁴ (software interface)
CUDA	Compute Unified Device Architecture ⁴ (software interface)
CUFFT	CUDA FFT Library ⁴ (software interface)
Cg	C for Graphics ⁴ (software interface)
D/A	Digital-to-analog (electronics concept)
dB	Decibel (mathematical concept)
DFT	Discrete Fourier transform (mathematical concept)
DSP	Digital signal processing
DVD	Digital video disc (recording media)
FFT	Fast Fourier transform (mathematical concept)
FFTW	FFT in the West ⁵ (software interface)
FIR	Finite impulse response (digital filter type)
FLOPS	Floating-point operations per second (unit of processor performance)

² Steinberg Media Technologies GmbH. ³ ATI Technologies. ⁴ Nvidia Corporation. ⁵ MIT.

FT	Fourier transform (mathematical concept)
GmbH	Gesellschaft mit beschränkter Haftung (from German, meaning “company with limited liability”)
GLSL	OpenGL Shading Language ⁶ (software interface)
GPGPU	General-purpose GPU programming (design concept)
GPU	Graphics processing unit (hardware component)
HLSL	High Level Shader Language ⁷ (software interface)
HRTF	Head-related transfer function (mathematical concept)
Hz	Hertz (unit of frequency)
IDFT	Inverse discrete Fourier transform (mathematical concept)
IEEE	Institute of Electrical and Electronics Engineers (research institute)
IFT	Inverse Fourier transform (mathematical concept)
IIR	Infinite impulse response (digital filter type)
LPC	Linear predictive coding (mathematical concept)
MATLAB	Matrix Laboratory ⁸ (software interface)
MIT	Massachusetts Institute of Technology (research institute)
NPOT	Non-power-of-two (mathematical concept)
OpenCL	Open Computing Language ⁹ (software interface)
OpenGL	Open Graphics Library ⁶ (software interface)
PCI	Peripheral Component Interconnect (bus interface)
POT	Power-of-two (mathematical concept)
Q	Quality factor of band-pass filter (mathematical concept)
RMS	Root mean square (mathematical concept)
SDK	Software development kit (design concept)
SIMD	Single instruction multiple data (design concept)
SNR	Signal-to-noise ratio (mathematical concept)
SPL	Sound pressure level (mathematical concept)
STFT	Short-time Fourier transform (mathematical concept)

⁶ OpenGL Architecture Review Board. ⁷ Microsoft Corporation. ⁸ MathWorks Inc. ⁹ Apple Inc.

LIST OF FIGURES

4.1	Magnitude response and pole-zero plot	29
4.2	Implementations of a second order filter	30
4.3	Implementation of filters in cascade and parallel forms	31
6.1	Linear convolution implemented as padded cyclic convolution for fast operation	43
6.2	Fast convolution in block processing	44
6.3	Observed behavior of filter with varying stability using the proposed technique	47
6.4	Spectral analysis of an unstable filter's output using the proposed technique	48
7.1	Conceptual diagram of demo application	51
7.2	Snapshots of the implemented scene requiring audio generation	51
7.3	The setup for recording samples for the glass material	52
7.4	Summary of tasks in the demo	54
A.1	Class diagram for the implementation	69

LIST OF TABLES

5.1	Comparison of precision (in ulps) of floating-point implementations. .	36
5.2	Comparison of speed and accuracy across CPU and GPU implemen- tations.	37
7.1	Comparison of maximum number of coefficients supported for real- time operation of the proposed method	56

LIST OF PROCEDURES

A.1	Basic kernels in CUDA and corresponding calls.	65
A.2	Generating blocks of audio using CUDA.	66
A.3	Kernel calls for loading and computing the filter using CUDA.	67
A.4	Kernel calls for generating sinc pulses using CUDA.	68

ABSTRACT

Many researchers have been interested in exploring the vast computational power of recent graphics processing units (GPUs) in applications outside the graphics domain. This trend towards *General-Purpose GPU* (GPGPU) development has been intensified with the release of non-graphics APIs for GPU programming, such as NVIDIA's Compute Unified Device Architecture (CUDA). With them, the GPU has been widely studied for solving many 2D and 3D signal processing problems involving linear algebra and partial differential equations, but little attention has been given to 1D signal processing, which may demand significant computational resources likewise.

It has been previously demonstrated that the GPU can be used for real-time signal processing, but several processes did not fit the GPU architecture well. In this work, a new technique for implementing a *digital recursive linear filter* using the GPU is presented. To the best of my knowledge, the solution presented here is the first in the literature. A comparison between this approach and an equivalent CPU-based implementation demonstrates that, when used in a real-time audio processing system, this technique supports processing of two to four times more coefficients than it was possible previously. The technique also eliminates the necessity of processing the filter on the CPU — avoiding additional memory transfers between CPU and GPU — when one wishes to use the filter in conjunction with other processes, such as sound synthesis.

The recursivity established by the filter equation makes it difficult to obtain an efficient implementation on a *parallel architecture* like the GPU. Since every output sample is computed in parallel, the necessary values of previous output samples are unavailable at the time the computation takes place. One could force the GPU to execute the filter sequentially using synchronization, but this would be a very inefficient use of GPU resources. This problem is solved by unrolling the equation and “trading” dependences on samples close to the current output by other preceding ones, thus requiring only the storage of a limited number of previous output samples. The resulting equation contains convolutions which are then efficiently computed using the FFT.

The proposed technique's implementation is general and works for any time-invariant recursive linear filter. To demonstrate its relevance, an LPC filter is designed to synthesize in real-time *realistic sounds of collisions between objects* made of different materials, such as glass, plastic, and wood. The synthesized sounds can be parameterized by the objects' materials, velocities and collision angles. Despite its flexibility, this approach uses very little memory, requiring only a few coefficients to represent the impulse response for the filter of each material. This turns this approach into an attractive alternative to traditional CPU-based techniques that use playback of pre-recorded sounds.

Keywords: Digital filters, linear filters, recursive filters, signal processing, sound synthesis, sound effects, GPU, GPGPU, realtime systems.

Uma Implementação Eficiente de Filtros Lineares Recursivos e Sua Aplicação a Re-Síntese Realística em Tempo Real para Mundos Virtuais Interativos

RESUMO

Muitos pesquisadores têm se interessado em explorar o vasto poder computacional das recentes unidades de processamento gráfico (GPUs) em aplicações fora do domínio gráfico. Essa tendência ao desenvolvimento de *propósitos gerais com a GPU* (GPGPU) foi intensificada com a produção de APIs não-gráficas, tais como a Compute Unified Device Architecture (CUDA), da NVIDIA. Com elas, estudou-se a solução na GPU de muitos problemas de processamento de sinal 2D e 3D envolvendo álgebra linear e equações diferenciais parciais, mas pouca atenção tem sido dada ao processamento de sinais 1D, que também podem exigir recursos computacionais significativos.

Já havia sido demonstrado que a GPU pode ser usada para processamento de sinais em tempo-real, mas alguns processos não se adequavam bem à arquitetura da GPU. Neste trabalho, apresento uma nova técnica para implementar um *filtro digital linear recursivo* usando a GPU. Até onde eu sei, a solução aqui apresentada é a primeira na literatura. Uma comparação entre esta abordagem e uma implementação equivalente baseada na CPU demonstra que, quando usada em um sistema de processamento de áudio em tempo-real, esta técnica permite o processamento de duas a quatro vezes mais coeficientes do que era possível anteriormente. A técnica também elimina a necessidade de processar o filtro na CPU — evitando transferências de memória adicionais entre CPU e GPU — quando se deseja usar o filtro junto a outros processos, tais como síntese de som.

A recursividade estabelecida pela equação do filtro torna difícil obter uma implementação eficiente em uma *arquitetura paralela* como a da GPU. Já que cada amostra de saída é computada em paralelo, os valores necessários de amostras de saída anteriores não estão disponíveis no momento do cômputo. Poder-se-ia forçar a GPU a executar o filtro sequencialmente usando sincronização, mas isso seria um uso ineficiente da GPU. Este problema foi resolvido desdobrando-se a equação e “trocando-se” as dependências de amostras próximas à saída atual por outras precedentes, assim exigindo apenas o armazenamento de um certo número de amostras de saída. A equação resultante contém convoluções que então são eficientemente computadas usando a FFT.

A implementação da técnica é geral e funciona para qualquer filtro recursivo linear invariante no tempo. Para demonstrar sua relevância, construímos um filtro LPC para sintetizar em tempo-real *sons realísticos de colisões de objetos* feitos de diferentes materiais, tais como vidro, plástico e madeira. Os sons podem ser parametrizados por material dos objetos, velocidade e ângulo das colisões. Apesar de flexível, esta abordagem usa pouca memória, exigindo apenas alguns coeficientes para representar a resposta ao impulso do filtro para cada material. Isso torna esta abordagem uma alternativa atraente frente às técnicas tradicionais baseadas em CPU que apenas realizam a reprodução de sons gravados.

Palavras-chave: Filtros digitais, filtros lineares, filtros recursivos, processamento de sinais, síntese de som, efeitos sonoros, GPU, GPGPU, sistemas de tempo-real.

1 INTRODUCTION

Sound is indispensable to the multimedia experience, conveying both information and emotion. For this reason, many of today's multimedia applications (from mobile phones to video games to the recording studio) make use of some sort of audio processing. Over time, audio and circuit technologies have developed alongside, with the first electronic audio applications consisting of analog circuits, then hard-wired on-chip programs, then software running on modern programmable chips. Currently, most audio software runs on *digital signal processors* (DSPs) — such as those found in sound cards — and on CPUs. Technology has evolved in such a way as to conveniently allow a skilled electronic musician to undergo all steps of industrial-quality recording production using only a laptop and music software running on the CPU (and, perhaps, a microphone and an external audio card, for acoustic recordings).

However, audio processing on personal computers faces some obstacles repeatedly. For instance, most DSPs are not programmable, containing only a few audio algorithms (whichever are considered useful and adequate by each particular vendor) and offering only a few adjustable parameters to the user application. For the DSPs that are programmable, there are no widely accepted standard software interfaces, and this hinders large-scale development for these platforms. CPUs, on the other hand, can compute any given algorithm and allow the development of new ones, but are often unable to handle high audio processing loads. This is particularly problematic when sound needs to be immediately produced in response to some event (*i.e.*, in *real-time*), such as when generating sounds for events such as collisions in a game, or when generating the sound for a musician's keyboard performance. In this respect, DSPs are better because they are designed for real-time operation. For these reasons, most non-professional audio applications (such as in games) make use of simple, low-quality audio algorithms, and musicians usually invest in dedicated equipment able to handle specific processes in real-time.

Professional audio applications (such as those used in recording studios) often make use of considerably more complex methods in order to produce a wide array of sound textures with different auditory attributes and, often, superior quality (both in the sense of signal quality — *i.e.*, less noise and/or distortion — and artistic, subjective quality). Such processes include *reverberation*, used in virtual scenes as an auditory clue that provides the perception of a wide complex space, such as a chamber or a hall. This is often computed as a *convolution* with another signal (the impulse response that models environment resonances) (GARCÍA, 2002), and thus, this process requires more operations per sample for larger impulse responses (some consisting of as much as 200.000 samples and sometimes even more than a million). Another common process that may demand considerable resources is synthesis via *sample playback* (RUSS, 1996), common in videogames and some basic music software. This method, when applied to multiple simultaneous sound

events (*e.g.*, generated by many simultaneous collisions in a scene), easily overloads any current CPU on the market. Simple and common solutions include halting some of the currently playing events or simply not beginning the playback of new events, but both approaches break the auditory expectation of the user. The situation becomes critical when one decides to combine this synthesis method with a reverberation effect to compose a full soundstage for the scene, as is often desired in real-time audio applications. The computational power of CPUs, then, often limits the number and types of processes that can be combined to build such a soundstage.

On the other hand, recent graphics processing units (GPUs) have presented theoretical throughput capabilities that far exceed those of CPUs. For example, the Nvidia GeForce GTX 280 has 240 stream processors working in parallel, providing a theoretical maximum throughput of 933 GFLOPS in single precision mode (NICKOLLS et al., 2008; NVIDIA CORP, 2008a). Also, graphics hardware companies have recently developed technologies such as CUDA and CTM (HOUSTON; GOVINDARAJU, 2007) which are oriented toward *general-purpose processing on the GPU* (GPGPU). More recently, some GPU vendors have announced future support to OpenCL, a public standard for development for parallel architectures. This reflects a demand for offloading compute-intensive processes to the GPU, which is the case of some of the audio applications mentioned previously. If explored properly, the GPU should have the potential to compute larger amounts of audio data. It would also constitute another alternative for load balancing, leaving the CPU free for other processes, such as physics simulations and artificial intelligence.

Although 2D and 3D signal processing have been well explored on the GPU (ANSARI, 2003; JARGSTORFF, 2004; MORELAND; ANGEL, 2003; SPITZER, 2003; SUMANAWEEERA; LIU, 2005), and many mathematical procedures are readily available on libraries such as CUBLAS and CUFFT (NVIDIA CORP, 2008b,c), very little work has been done in the area of GPU-based 1D signal processing, probably due to the interdisciplinary nature of these processes. For example, excluding the technique presented in this thesis, no general solution for linear recursive filters on GPUs has been published, except for evaluating a massive number of recursive filters in parallel (ZHANG; YE; PAN, 2005), which is reasonably straightforward. Moreover, the lack of an efficient parallel algorithm for recursive filters (among other less common processes) has prevented GPUs from being used for professional audio processing, even though there is evidence that the GPU can be used for real-time audio processing (TREBIEN; OLIVEIRA, 2008) and sometimes achieve as much as 20 \times speedups (TREBIEN, 2006). Finally, such work could naturally extend to other applications, such as signal coding and decoding in wireless network routers (YEO et al., 2002).

Therefore, in order to fulfill this missing link, I presented a new technique (TREBIEN; OLIVEIRA, 2009) that allows efficient implementation of linear, time-invariant recursive 1D filters on GPUs. To eliminate dependences, the method consists of unrolling the filter equation until all data dependences refer to available sample values. After unrolling, the equation can be expressed as a pair of convolutions, which in turn can be computed efficiently using the FFT algorithm. Compared to an equivalent state-of-the-art CPU-based technique, this approach supports processing filters with two to four times as many coefficients in real-time.

The effectiveness of this approach and its relevance to computer graphics are demonstrated by re-synthesizing realistic sounds of colliding objects made of different materials (*e.g.*, glass, plastic, and wood) in real time. The sounds can be customized to dynamically

reflect object properties, such as velocity and collision angle. Since the entire process is done through filtering, it essentially requires a set of coefficients describing the material properties, thus having a small memory footprint. This thesis also describes how the coefficients required for re-synthesis can be automatically computed from recorded sounds. Given its flexible and general nature, this approach replaces with advantages, although not entirely, the traditional CPU-based techniques that perform playback of pre-recorded sounds.

1.1 Structure of This Thesis

The remaining of this thesis is organized as follows.

Chapter 2 discusses some related work regarding the use of GPUs for audio processing and some related theoretical works on digital signal processing. Chapter 3 presents the fundamental concepts of audio processing in digital systems. Chapter 4 introduces digital filters and the basic theory for building digital audio systems using digital linear filters, defining concepts such as impulse response, non-recursive and recursive filters, convolution, the Fourier transform, pole-zero plot and filter stability. Chapter 5 introduces digital platforms widely available in the consumer market, with which one can build audio systems and applications.

After all introductory concepts and contextualization are given, Chapter 6 proposes and investigates a solution for the problem of efficient implementation of recursive filters on GPUs, which includes both FIR and IIR filters. The technique can be used efficiently even with very long FIR filters, which are useful for reverb effects. The last section in this chapter presents a known technique called Linear Predictive Coding (LPC), which will be used later to provide coefficients for recursive filters.

In Chapter 7, validation of the proposed technique is achieved by coupling the filter implementation with a graphics process that performs physically-based collision detection. The process uses the proposed implementation of recursive filters to synthesize in real-time realistic sounds for the collision of objects made of different materials. The chapter concludes with results, a performance comparison, some analysis of stability of the implemented filters and implementation details for reproducing and verifying the results.

Finally, in Chapter 8, this work is summarized and directions for future research are provided.

2 RELATED WORK

This chapter presents a critical description of some related works, establishing the major differences between them and my work. Due to the interdisciplinary nature of this research, this chapter is divided in two parts. Section 2.1 discusses some work on signal processing related in special to Chapter 7. Section 2.2 discusses some related work on GPU-based audio processing.

2.1 Theoretical Signal Processing

In this section, most works link to basic signal processing techniques. As such, even though it may come as a surprise, many of them were published several decades ago. Additionally, rather than reporting previous attempts at solving this problem, they present problems associated with CPU-based recursive filtering, what turns them into a valuable source of commonly agreed requirements and methods to evaluate the quality of such implementations.

Liu (1971) discussed the accuracy problems in digital linear filter implementations using both finite word length and floating-point representations. He analyzed the effect of round-off errors on the input signal, filter coefficient quantization and accumulation errors on recursive filters. He also established error bounds for each of these elements and defines conditions that ensure filter stability. By treating the signal in an abstract way, no implications to audible effects of different implementations were drawn.

Kaiser (1965) reviewed some filter design methods and derives an expression that determines the required coefficient accuracy for a given sampling rate and filter complexity. The effects of limited precision on filter zeros and poles were studied, and he also determined which of the canonical implementations provide the best accuracy.

Rader and Gold (1967) provided and discussed an alternative implementation of first and second order recursive filters with significantly reduced errors in poles and zeros positions. Avenhaus and Schüssler (1970) described a method for determining the ideal word length. One of the notable conclusions was that, in some cases, a larger word length may lead to larger filter errors.

Jackson (1969) analyzed and verified experimentally several rules for predicting the occurrence in frequency of limit cycles (*i.e.*, critically stable output) resulting from round-off errors in first and second order filters. He found that in such conditions, the output remains “stable” (*i.e.*, periodic and with similar amplitude) and does not suffer any considerable effect from different input signals. He then estimated the expected behavior of a filter implemented in the Parallel Form (output of sections are summed, therefore the output is a sum of all limit cycles) but only made unverified assertions about the Cascade Form implementation.

In another paper, Jackson (1970) provided an extensive analysis on the effect of quantization in filter implementation. He derived expressions in the frequency domain that bound such errors relative to the filter's transfer function. He did so for multiple filter implementations, including Direct Form I and II and the Transpose Form. These conclusions allow the reader to draw similar conclusions for the Cascade and Parallel Form implementations as well.

Weinstein and Oppenheim (1969) verified experimentally a statistical model for noise introduced in a filtered signal as a result of arithmetic quantization. They applied their experiment using fixed and floating-point arithmetic, and compared the results of the two. One of the important conclusions is that a floating-point-based filter realization desirably provides greater signal-to-noise-ratios (SNR) than an equivalent realization using fixed point with the same word length as the floating-point's mantissa.

Oppenheim (1970) proposed an alternative filter implementation using block-floating-point arithmetic, in which the filter is implemented using fixed point arithmetic with the maximum filter gain normalized to 1 to avoid overflow. The inverse of the normalization factor is then applied to the output. The theoretical characterization of noise in such an implementation was then experimentally verified and compared to the other existing implementations.

Welch (1969) established upper and lower bounds on the RMS spectral error of the FFT algorithm implemented using fixed-point arithmetic. The accuracy was measured as the spectral RMS of the error introduced in the computation of the FFT. Then, he proceeded to an experimental verification on a non two's-complement machine.

Weinstein (1969) also studied the accuracy of the FFT algorithm but using floating-point arithmetic. He proposed and verified experimentally a statistical model that estimates the SNR for a given FFT configuration. His model showed that using truncation instead of rounding on floating-point operations greatly reduces SNR, which is not desirable. His model also showed that rounding down instead of rounding up also increases SNR, demonstrating the possible introduction of correlation between round-off noise and signal, an assumption that is ignored in most studies, including the one described in this thesis.

Parhi and Messerschmitt (1989) developed an incremental block-state structure for hardware realization of recursive filters. Their approach combines incremental output computation and clustered look-ahead and provides minimal use of hardware components (*e.g.*, transistors). Though their technique has linear complexity respective to block size, the implementation is given in pipeline form, thus containing recursive dependences that prevent direct implementation (without modification) on SIMD machines such as the GPU.

2.2 GPU-Based Audio Processing

Several works presented adaptations of the FFT algorithm for GPUs (ANSARI, 2003; MORELAND; ANGEL, 2003; SPITZER, 2003; SUMANAWEEERA; LIU, 2005). Implementations of the FFT are already available in software libraries for both the CPU (FRIGO; JOHNSON, 2005) and GPUs (GOVINDARAJU; MANOCHA, 2007).

Gallo and Drettakis (2003) presented a method for sound spatialization in a 3D virtual world that reduces requirements on audio hardware. In their method, a set of 3D sound sources are clustered according to their relative position to the listener in polar coordinates. Signals from sources of the same cluster are mixed on the GPU, yielding a smaller

number of voices for playback by the sound card. Signals consist of a single channel with 3 sub-bands at 44.1 kHz, and blocks contain 1,024 samples each. The sub-band format allows a simple form of equalization, and the mixing algorithm also performs Doppler shifting by linear resampling of the input signals through texture access. Gain controls (filter parameters, panning and distance attenuation) appear to be computed on the CPU once per frame. No synthesis is performed, so all sound samples are loaded to texture memory before execution. The authors implemented an application with graphics and sound running concurrently. Running the application on a 1.8 GHz Pentium 4 Compaq laptop with audio only, processing the signals on the GPU required about 38% of available CPU time. The audio consisted of 8-bit samples due to graphics hardware limitations. With an ATI Radeon 5700 graphics card, they claim a greater frame rate was achieved with graphics rendered on (rendering scene with 70,000 polygons), but no performance metric is given.

Gallo and Tsingos (2004) presented a report on a similar GPU-accelerated audio processing application, in which Doppler shifting is simulated through a variable delay line. The sound is filtered by head-related transfer functions (HRTFs) (BEGAULT, 1994) to produce the sensation of source position. The signal contains a single channel with 4 sub-bands at 44.1 kHz and audio blocks contain 1,024 32-bit floating-point samples. When running their application on a 3.0 GHz Pentium 4 with an AGP 8x Nvidia GeForce FX 5950, they report that the GPU implementation was 20% slower than a CPU-based implementation.

Jedrzejewski and Marasek (2004) used the GPU for a ray-tracing method that computes an impulse response (ANTONIOU, 1980) from source to listener on occluded virtual environments to produce a realistic room reverberation effect. The impulse response would then be convolved with the source signal, but the authors apparently did not implement that feature. As such, no signal processing takes place in the GPU.

Robelly *et al.* (2004) presented a mathematical formulation based on state variables for computing time-invariant recursive filters on parallel DSP architectures. Their implementation revealed that high speedups can be achieved when both filter order and number of parallel processors are high (around $40\times$ for a 60th-order filter on a 128-way parallel processor). However, when the filter order is low, the achieved speedup is small (not more than $2\times$ for a 1st-order filter in their test conditions regardless of the number of parallel paths). The platform for running the implementation was left unspecified, although the method is sufficiently general for implementation in any parallel architecture.

Whalen (2005) evaluated the performance of the GPU by comparing the speedup achieved in the implementation of a set of simple non-recursive non-real-time audio processes. His tests were run on a 3.0 GHz Pentium 4 with an AGP Nvidia GeForce FX 5200, the signal contained 16-bit samples and a single channel, and the process operated on blocks of 105,000 samples (around 2.2 seconds at 48 kHz). He found up $4\times$ speedups for certain algorithms, notably those requiring only sum and multiplication (delay, low-pass, and high-pass filters), but this is likely a limitation of previous generations of GPUs. For use in applications, these processes would require modifications, introducing some overhead.

Zhang *et al.* (2005) have used GPUs for *modal synthesis*, a physically-motivated synthesis model that represents a vibrating object as a bank of damped harmonic oscillators. The inherent recursivity of synthesizing a single mode was eliminated in favor of the goal of computing a massive number of individual modes. In their approach, for each mode, one sample is generated at each step of the process, and all these samples are summed to obtain the output sample. Their method was not applied to any implementation of

an interactive virtual world, and it did not use masking or culling algorithms to reduce computational load. They reported having computed 32,768 modes in real-time using a GeForce 6800 GT card, but achieving only 5,000 modes with a Pentium 4 2.8 GHz.

Treblen and Oliveira (2008) proposed a real-time audio processing system based on GPUs capable of performing some basic operations and synthesizing basic waveforms. This was achieved by mapping audio concepts to graphics concepts and using OpenGL to activate shaders responsible for doing the processing. Textures were used to store input and output signals, and the signal stream was transferred to main memory and then to the audio device for playback.

Bonneel *et al.* (2008) developed a technique for efficiently generating sounds for collisions in virtual worlds using modal synthesis in the frequency domain, with the use of the Short-Time Fourier Transform (STFT). Their method models well sounds composed of exponentially decaying narrowband modes, and achieves speedups of 5 to 8 \times compared to a time-domain solution. They present a framework allowing the use of pre-recorded sounds, HRTFs, auditory masking and culling, and they also discuss the impact of different windows for overlapping consecutive STFT frames in the time domain. They also conducted a blind user test to validate the quality of their method. Collision sounds were generated on the CPU, even though their technique could be adapted to run on the GPU.

2.3 Summary

This chapter discussed some related work on signal processing and on GPU-based audio processing. The works related to signal processing mostly serve as a theoretical background for the discussion that follows. Most of the works on GPU-based audio processing do not present a real-time application, and most also report little advantage of GPU-based audio processing over a corresponding CPU-based one. Two of these works present some GPU-based implementation of modal synthesis, a subclass of recursive linear filtering.

In the next chapter, the fundamental concepts necessary for audio processing are presented. In particular, the representation of the signal on a computer and the processing of a continuous stream of audio with low delay are discussed.

3 FUNDAMENTALS OF SIGNAL PROCESSING

In this chapter, fundamental concepts of acoustics are introduced sound. First, sound is characterized as an oscillatory perturbation of a medium. Then, measures of sound wave amplitude and frequency are given, along with the capabilities of the human ear to detect sound waves of different types. This is important when choosing operational characteristics of an audio application. The term “audio” is defined as well. A distinction between analog and digital audio is drawn and the representation of audio in a digital machine is discussed. Finally, block processing, an important mechanism in real-time audio applications, is presented.

3.1 Basic Concepts of Sound and Acoustics

Sound can be characterized as a mechanical perturbation which propagates over time in a physical medium (typically the air). It is both produced and propagated through a cascade of adjacent particle displacements, creating regions of compression and rarefaction and, thereby, altering the local *pressure* level of the medium. Because pressure varies continuously and tends to return to an equilibrium point, sound is essentially an *oscillatory* effect. Because of that, instantaneous pressure at a point in space is also labeled *amplitude* of the traveling wave sound. The state of a *sound field* is defined by the instantaneous amplitude levels inside a well-defined region of physical space. When such waves reach the human ear, they induce nervous stimulation inside a complex biological apparatus called *cochlea*. The resulting nervous signals are carried into the brain, which is responsible for auditory *perception* and *cognition*, both complex processes still under scientific investigation (GUMMER, 2003). As such, sound can also be understood as a *signal* (see Chapter 4) containing *auditory information*.

When sound waves interact with particles of another medium, they undergo well-known wave phenomena such as absorption, reflection, refraction, diffraction, interference, among others. The conversion of energy between two different forms, called *transduction*, is responsible for many processes of sound generation and absorption. For instance, when an object strikes another, part of the kinetic energy is transformed into tension as particles near the contact point are displaced to accommodate the impact; as these particles move toward tension equilibrium, they displace air particles, releasing energy in the form of sound (MORSE; INGARD, 1986). The set of processes that generate and transform sound waves are studied in the field of theoretical *acoustics*. The term “acoustic” usually refers to sound waves in air (*e.g.*, acoustic sound, acoustic signal, acoustic audio).

Particularly, transduction can be exploited to *convert* a sound signal to/from an *analog* electrical form where the instantaneous level of either current or voltage in a circuit repre-

sents the instantaneous level of sound amplitude at a specific point in the sound field. This is the principle under which microphones and loudspeakers operate (EARGLE, 2001). The term “analog”, therefore, usually refers to sound signals in analog electrical form (*e.g.*, analog signal, analog audio). Analog signals can be transduced into other forms as well, such as magnetic impressions in magnetic tapes for permanent recording (*i.e.*, a type of analog recording), and also transformed by other electrical components and transduced back to air, producing new sounds. Notably, the human ear is also a transducer because it converts sound waves into electrical signals in the nervous system. The study of processes by which sound can be transduced and transformed in electrical form is called *electroacoustics*.

Observing sound waves, one frequently finds instances of *harmonic motion*, in which amplitude levels appear to vary in an almost perfect sinusoidal movement. In fact, as presented in Section 4.4, sound waves can be expressed as a combination of pure sinusoids (CARSLAW, 1952). A sinusoid is defined by

$$y(t) = A \sin(\omega t + \phi) \quad (3.1)$$

where $y(t)$ represents the actual amplitude of the sinusoid at instant t , A is the peak (*i.e.*, maximum) absolute (*i.e.*, ignoring the sign) amplitude, ω is the frequency and ϕ is the phase of the sinusoid. Notice that, if $\omega = 2\pi f$, then f represents the number of sinusoidal cycles per time unit. The human ear is sensitive to different levels of amplitude and frequency of components of sound and it may be sensitive, to some degree, to their phase as well (LIPSHITZ; POCOCK; VANDERKOOY, 1982).

As sinusoids are composed of positive and negative *cycles*, frequency defines the number of cycles per unit of time. For sound, frequency is usually measured in *Hertz* (Hz), which represents the number of *cycles per second*. The human ear is sensitive to sinusoidal sound waves of frequencies between 20 Hz and 20 kHz (ISO/IEC, 2008). Sounds of frequency below 20 Hz are named *infrasound*, and those above 20 kHz are named *ultrasound*.

At any time, amplitude can only be expressed in respect (as a ratio) to a reference level. The ratio itself is a dimensionless quantity, but it is usually expressed in logarithmic scale as *decibels* (dB), meaning one-tenth of a *bel*. The dB scale is defined as

$$\frac{A_1}{A_0} = 20 \log_{10} \left(\frac{A_1}{A_0} \right) \text{ dB} \quad (3.2)$$

where A_1 and A_0 are the amplitudes being compared, with A_0 representing the reference level. For example, a sinusoidal wave with twice the amplitude of another is rated as $20 \log_{10}(2) \text{ dB} \approx 6.02 \text{ dB}$.

When establishing the amplitude of acoustic waves, one can use the dB SPL (sound pressure level) scale, where the level of 0 dB SPL represents the quietest (*i.e.*, of lowest amplitude) detectable by the human ear. This level is also called the *threshold of hearing*. There is no maximum limit for amplitudes that the ear can detect, but there are thresholds establishing ear damage, pain and ultimately death of the individual. The level of 120 dB SPL is considered the human *threshold of pain*. This level also establishes the relationship between the loudest and the quietest sound an individual is likely to listen in normal conditions, thereby defining the human ear’s *dynamic range*.

Finally, the term *audio* can be defined as the *audible* component of any sound signal. However, in general, the word “audible” is more often used in reference to the detectable frequency range and not to amplitude thresholds.

3.2 Audio Processing Systems

Analog electrical signals can be transformed in a variety of ways in electrical circuits. In particular, they can be *sampled at regular intervals* by an *analog-digital (A/D) converter*. Each sample, in turn, is a *finite (i.e., discrete) digital number* that only approximates the original continuous signal. Such a discrete approximation is called *quantization*. In digital machines, these samples are usually stored as arrays in memory for processing. The resulting numbers can also be converted to continuous signals using a *digital-analog D/A converter*.

As such, sound can be produced and transformed in all the three forms — acoustic, analog and digital — and converted between each other. The digital medium, as opposed to the other two forms, allows for durable storage and for repeated reproduction without loss of quality. It is, then, interesting to use the digital medium both for new audio processes — where developers are free to create whatever transformation they want — and also to simulate common transformations occurring in mechanical and/or analog media, such as *reverberation* in air (when sound reflects on surfaces and other objects) and nonlinear distortion in circuits (when amplitude is distorted by the components the signal goes through).

3.2.1 Representations of Digital Audio

In digital machines, *digital audio* consists of discrete sequences of discrete samples, uniformly spaced in time, each sequence representing an underlying continuous signal. In Equation 3.1 the notation $y(t)$ represents the continuous value of a continuous sinusoidal signal at time t , which is also a continuous domain. A sampled signal is usually represented by $y[n]$ where y is the name of the signal and n is the n -th sample of the signal's sequence corresponding to time $t = nT$, where $T = \frac{1}{R}$ is the sampling interval and R is the sampling rate. For compactness, $y[n]$ is represented as y_n from now on.

According to the Nyquist–Shannon sampling theorem (JERRI, 1979), a signal sampled at rate R can only represent components of frequency between 0 and $\frac{R}{2}$. Higher frequency components are actually “aliased-back” into lower frequency ones. After sampling, these components can no longer be distinguished from the original, non-aliased ones, and so the original continuous signal is unrecoverable. As such, sampling may cause *loss* of sound information. However, if humans have a limited frequency perception range, ultrasound can be discarded, meaning that it is sufficient to sample *audio* at twice the limit of perception, *i.e.*, at 40 kHz or more (*e.g.*, typical sampling rates include 44.1 kHz for CD audio and 48 kHz for DVD audio (BLECH; YANG, 2004)). Sampling at a rate higher than 40 kHz only has the effect of attenuating certain distortions introduced in the signal by the sampling devices (however, this is beyond the scope of this thesis). Finally, it is undesirable to have ultrasonic components aliasing back into the hearing range of frequencies, and so A/D converters usually also include an analog *filter*¹ that removes high frequency components.

Quantization introduces an error in the original signal due to *round-off*. The difference between the discrete and the continuous signal can be seen as an “error signal”. The significance of this error is often expressed in terms of the *signal-to-noise ratio* (SNR) between the amplitude of the signal being represented and the maximum amplitude of the error. Given this and the knowledge of the human threshold of hearing, one may infer that

¹ An “analog filter” is an analog electronic device — such as a resistor, inductor, etc. or any combination of these — that transforms analog signals in some way.

an SNR ratio of 120 dB is needed to ensure that round-off noise remains imperceptible. In fact, due to *auditory masking* (NELSON; BILGER, 1974) — a process taking place inside the cochlea that makes low amplitude components imperceptible in the presence of higher amplitude signals — and to the fact that recordings are usually listened to in much lower levels — 60 to 80 dB in average (BENJAMIN, 2004) —, SNR can be lower for most audio applications. For instance, CD audio is represented using 16-bit integers, and its effective SNR is given by comparing the maximum round-off error (0.5) with the highest amplitude sound that can be represented without saturation (2^{15}), *i.e.*, $20 \log_{10} \left(\frac{2^{15}}{0.5} \right)$ dB ≈ 96.33 dB.

During processing, audio is usually stored in an uncompressed form using arrays in memory. For storage in files, audio will often be compressed using either a *lossy* or a *lossless* codec. The discussion of audio compression, however, is beyond the scope of this thesis.

Sound can be captured from microphones at multiple points in a sound field. Each microphone transduces a different signal, and each signal in this case is called a *channel*. The choice of number of channels is completely arbitrary, with a higher number of channels enhancing human perception of 3D sound position. Channels are also usually treated independently in audio processes. Audio is usually captured in two channels (stereo), such as in CD audio, and less often in a single channel (mono). Recently, there has been a trend toward an inclusion of more channels, used in *surround-sound* systems where loudspeakers are placed around the listener.

3.2.2 Block Processing

Sound cards, like most digital audio equipment, provide analog audio inputs coupled with A/D converters and analog outputs coupled with D/A converters. Once sampled, audio is usually processed in *blocks* of samples. Though it would be theoretically possible to operate on an input sample and convert the resulting output sample to analog almost immediately, doing so in a PC would introduce a significant communication overhead between the audio device and the CPU since the communication between the two is generally much slower than the CPU itself. Processing in blocks, though, requires significantly less system messages to synchronize processing and playback. As discussed in Section 4.4.1, processing audio in blocks may also improve the efficiency of certain algorithms such as FFTs. Providing blocks of data to any device can also be referred to as *streaming*.

With block processing, however, a signal arriving at the sound card's input would take longer to produce an effect in the outputs. Such a delay can be long enough to be perceptible, which is an undesirable effect in many situations. For example, for a musician playing an instrument and using the computer to produce any desired effect on the instruments' captured output audio, a long delay may confuse the player, disturbing rhythmic abilities and, at last, disrupting the player's performance. For this reason, it is important to choose adequately small block sizes for audio applications running in real-time (*i.e.*, where the results are expected in a fixed limited amount of time). This contrasts with non-real-time audio applications such as waveform editors, where processes are expected to take many seconds and even minutes to provide a result to the user.

It is generally agreed that a delay of 10–20 ms is imperceptible to humans and, therefore, tolerable for real-time audio applications. At the typical sampling rate of 44.1 kHz, this would translate to 441 to 882 samples of delay from input to output. In fact, sound cards regularly provide a block of samples to the CPU and receive a block of samples for playback (this process will be referred to as block “exchanging” in this thesis). For any

input sound to produce an effect in the output, it takes at least two block exchanges, since a block needs to be fully captured for processing before the result is available for output. As such, block sizes should range between 220 to 441 samples in order to keep delay imperceptible. They should be even smaller if when admitting the possibility of greater delays introduced by multiple block processing steps, as is often the case. Typical block sizes include 128, 256, 512, 1,024 and 2,048 samples. As discussed in Section 4.4.1, block sizes of powers of two are usually selected to allow for processes that use the FFT algorithm to achieve peak performance.

3.3 Summary

At this point, the reader should now know the basics of sound, audio and audio processing concepts. Keep in mind that sound is a continuous wave and audio refers to the audible part of sound. Audio can be transformed by processes in both acoustic, analog and digital forms, and it can be converted (though imperfectly) between the three forms. Processing in digital form presents some particular advantages in terms of permanent storage and playback quality, but care must be taken to choose audio format characteristics. A reference industry-quality format is given by CD audio, with a sampling rate of 44 kHz, 16-bit integer samples and two signal channels (stereo). Block

4 FILTER THEORY

A *signal* is a measurable varying quantity that carries information. *Signal processing* refers to theoretical methods for extracting or just modifying such information. Signals can be *continuous* (such as sound waves in the physical medium) or *discrete* (such as digital audio after sampling), *deterministic* or *random* (whether the signal can be predicted or not). A signal f may also be classified as *periodic* when the equation

$$f(x + P) = f(x) \quad (4.1)$$

holds for every x given a constant P . If the equation does not hold for any value of P , the signal is called *aperiodic*.

A *filter* is any device that operates on a signal, producing another (usually different) signal. Most physical objects are acoustic filters, absorbing and reflecting differently each component of the sound waves that reach them. Analog components such as resistors, inductors and capacitors are analog filters operating on analog signals. Any algorithm operating on the values of a digital signal and producing a new one in a digital machine is a *digital filter*.

As such, filters operate on *input* signals and produce *output signals*, which also constitute the *response* of the filter to the input signals. In this text, the letter x is used to denote an input signal and y to denote an output signal. The n -th sample of a digital signal x is also represented as x_n , and the instantaneous amplitude value at time t of a continuous signal as $x(t)$.

Unless specified, a filter operates on a single input signal. An interesting exception is the *mixing filter*, defined by

$$y_n = \sum_{s=1}^N g^{(s)} x_n^{(s)} \quad (4.2)$$

where s is the index of the input signal $x^{(s)}$, N is the number of input signals and $g^{(s)}$ is the *gain* applied to signal $x^{(s)}$. For $N = 1$, this filter is simply a *gain filter*, scaling the input signal at a fixed proportion for every sample. The mixing filter is very common in most digital audio processing applications and it simulates very closely the acoustic process of *interference*, *i.e.*, as if the (scaled) input sound signals were crossing the same point in space at the time of capture.

All audio processes actually constitute filters and can be classified in certain categories. A filter may be either *linear* if it performs any sort of linear combination (*i.e.*, if it can be expressed as an order-1 polynomial) involving input samples, or *nonlinear* otherwise. A filter may also be *memoryless* if the value of an output sample y_n is computed using only the value of the corresponding input sample x_n , thus requiring no memory to store previous input samples (*i.e.*, x_{n-1} , x_{n-2} , ...). For instance, the mixing filter in

Equation 4.2 is both linear and memoryless. The vast majority of filters used in audio applications are either linear or memoryless nonlinear filters (MOORE, 1990). For instance, the majority of synthesis techniques are also linear processes. Common memoryless nonlinear filters include waveshaping and amplitude modulation. As discussed in Section 4.1, filters can also be classified in *recursive* and *non-recursive* filters.

4.1 The Linear Filter Equation

A filter is *causal* when its output depends only on input samples that precede or coincide with the time of the output sample. In real-time filtering applications, all filters must be causal by definition, since the future samples of the input signal are not available. The general equation for a time-invariant causal digital linear filter is given by

$$y_n = \sum_{i=0}^P b_i x_{n-i} - \sum_{j=1}^Q a_j y_{n-j} \quad (4.3)$$

where b_i and a_j are respectively feed-forward and feedback coefficients, and P and Q are the feed-forward and the feedback filter orders, respectively. This equation can alternatively be written as

$$w_n = \sum_{i=0}^P b_i x_{n-i} \quad (4.3a)$$

$$y_n = w_n - \sum_{j=1}^Q a_j y_{n-j} \quad (4.3b)$$

where w simply constitutes a theoretical intermediary signal. Equation 4.3a represents the *non-recursive* part of the filter since it establishes dependences with input samples only, whereas Equation 4.3b constitutes the *recursive* part. When all a_j coefficients are null, Equation 4.3b yields the identity $y_n = w_n$ and the filter is called a non-recursive filter; otherwise, it is a recursive filter. Non-recursive filters are also called *finite impulse response* (FIR) filters because any input sample affects the value of a limited number (only P) output samples. A recursive filter, however, establishes a recursion, affecting the values of all succeeding output samples, and, for this reason, they are also called *infinite impulse response* (IIR) filters. If all coefficients b_i and a_j remain constant (*i.e.*, do not vary with n), the filter is called *time-invariant*; otherwise, it is called *time-varying*.

In filter design, sometimes the filter equation is expressed as the difference equation

$$\sum_{i=0}^P b_i x_{n-i} = \sum_{j=0}^Q a_j y_{n-j}$$

where y_n , as opposed to Equation 4.3, appears scaled by coefficient a_0 . This means that, if $a_0 \neq 1$ as a result of the chosen design process, all a coefficients have to be normalized by a_0 (*i.e.*, replaced with $\frac{a_j}{a_0}$) in order to obtain the desired filter implementation. In the remaining of the text, however, it is assumed that $a_0 = 1$, yielding the definition given by Equation 4.3.

Filters can be seen as *operators* over their respective input signals. If the letter H is used to denote a particular filter from Equation 4.3, then the filter may be denoted by

$$y_n = H\{x_n\}$$

4.2 Filter Response to Input

A discrete time-invariant filter can also be completely described by its *impulse response*, which is defined as the output of the filter to the discrete *impulse* signal

$$\delta_n = \begin{cases} 1, n = 0 \\ 0, n \neq 0 \end{cases} \quad (4.4)$$

4.3 Convolution

The discrete convolution of two discrete signals f and g represented by the convolution operator $*$ is given by

$$(f * g)_n = \sum_{m=-\infty}^{\infty} f_m g_{n-m} \quad (4.5)$$

If either signal is defined only for a range of samples (*e.g.*, a recording with beginning and end), the values of samples outside the defined range can be safely assumed to be null. Under this definition, the result has a finite number of non-null samples if both signals are finite too. For instance, if f is defined for M samples and g is defined for N samples, $f * g$ will have $M + N - 1$ defined samples.

Note that the Equation 4.3a expresses a convolution operation. In fact, the whole filter in Equation 4.3 could be expressed as a convolution between the input signal and the filter's impulse response. For instance, let h the discrete signal representing a time-invariant linear filter's impulse response. Then Equation 4.3 can be written as

$$y_n = (h * x)_n \quad (4.6)$$

4.4 Fourier Representations of Signals

Notably, all sinusoids from Equation 3.1 can also be defined using the Euler's formula given by

$$e^{jt} = \cos(t) + j\sin(t) \quad (4.7)$$

where $j = \sqrt{-1}$ represents the imaginary number. With this equation, the sinusoids from Equation 3.1 can be represented using

$$z(t) = Ae^{j(\omega t + \phi)} = Ae^{j\phi} e^{j\omega t} \quad (4.8)$$

which is related to Equation 3.1 by $y(t) = \text{Im}(z(t))$. Writing sinusoids in this form allows certain formulas to be simplified by using operations with complex numbers to compactly express trigonometric relationships involving sines and cosines.

Particularly, Euler's notation can be used to express a *Fourier series*, a formalism proposed by Fourier with which arbitrary *periodic* signals can be approximated as sums of sinusoids. A periodic signal is mapped to its Fourier series through a function called *Fourier transform* (FT). The inverse function is called *inverse Fourier transform* (IFT). The series resulting from the FT is sometimes referred to as the Fourier transform of the signal as well¹ (HAYKIN; VEEN, 1998).

¹ Haykin and Veen actually present a distinction between Fourier series and Fourier transform which may not be in common use. The term "series" refers to a summation and is used only for periodic signals. The term "transform" refers to an integral and is used for aperiodic signals.

Given a discrete periodic signal f with period N , it can be represented by

$$f_n = \sum_{k=0}^{N-1} F_k e^{jk\Omega_0 n} \quad (4.9)$$

where F_k expresses the k -th sinusoid's amplitude and $\Omega_0 = \frac{2\pi}{N}$ represents the distance between the discrete adjacent *bins* over the Fourier domain. The *Fourier domain* consists of all complex numbers of unitary magnitude defined by e^{jt} , each representing a sinusoid of unitary amplitude. The sequence formed by F_k coefficients is called the *Fourier domain representation* of signal f . The Fourier domain is also called *frequency domain* when f is a signal in the *time domain*, such as in analog signals.

The value of each F_k coefficient can be obtained using

$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n e^{-jk\Omega_0 n} \quad (4.10)$$

which, you should note, is a convolution between f and a sinusoid of frequency $-k\Omega_0$. This equation is widely said to define the *discrete Fourier transform* (DFT) operation (HAYKIN; VEEN, 1998, p. 157). Equation 4.9 represents the *inverse* discrete Fourier (IDFT) transform since it converts from Fourier domain back to the original domain.

The quantity F_k is a complex number in Equation 4.10 and, thus, contains real and imaginary components. From Equation 4.9, it can be observed that F_k is representing attributes of a sinusoid. By defining $F_k = a + bi = Ae^{j\phi}$, it follows that

$$A = |F_k| = \sqrt{a^2 + b^2}$$

$$\phi = \arg(F_k) = \pm \arctan\left(\frac{b}{a}\right)$$

where A is the magnitude of the k -th component (also its amplitude) and ϕ is the phase. This representation, called *polar form*, may help clarify the kind of information encoded by the Fourier series.

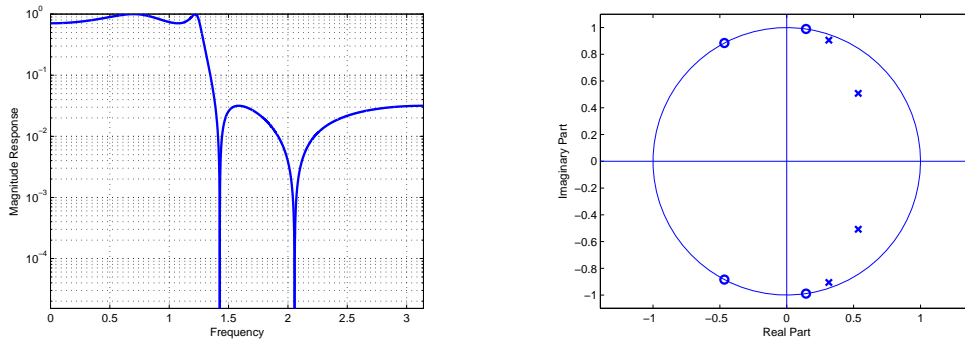
One of the most important properties of the Fourier representations is an equivalence between convolution in time domain, as in Equation 4.5, with pointwise multiplication in the frequency domain. Using the previous notation, this can be stated for discrete signals as

$$y_n = (h * x)_n \longleftrightarrow Y_k = X_k H_k \quad (4.11)$$

and this also holds for both periodic and aperiodic signals. Note that $h * x$ takes $O(n^2)$ to compute n output samples for input signals containing n samples, whereas $X_k H_k$ takes only $O(n)$ for k input samples. For signals in time domain represented with real numbers, it can be shown that $k = \frac{n}{2}$, and with complex numbers, $k = n$; in either case, the reduction in algorithmic complexity is clear. For this reason, multiplication in the frequency domain is often applied to optimize convolutions. It only requires, though, computing the coefficients of the Fourier series of x and h followed by computing the time-domain representation of Y .

4.4.1 Fast Fourier Transform Algorithms

Computing Equation 4.10 should be sufficient for obtaining the Fourier series of a signal. However, this process has complexity $O(n^3)$ if Equation 4.5 is used as the implementation for convolution. It turns out that there is a class of fast methods for computing



(a) Magnitude response. Poles generate the local maxima near frequencies 0.7 and 1.2, while zeros generate the minima near frequencies 1.4 and 2.1.

(b) Pole-zero plot. Zeros are depicted as circles and poles, as crosses. All poles are inside the unitary circle, therefore this filter is stable.

Figure 4.1: Two important ways of characterizing a digital filter. The magnitude response is actually the magnitude of the transfer function over the unitary circle in the pole-zero plot, with its frequency parameter being exactly the angle of the points over this circle.

the DFT called *fast* Fourier transforms (FFTs) which can compute it in $O(n \log(n))$. The most famous algorithm of this class is the Cooley-Tukey algorithm (COOLEY; LEWIS; WELCH, 1969).

4.5 Pole-Zero Representations and the z-Plane

In filter theory, a filter is sometimes described in terms of its *transfer* function (HAYKIN; VEEN, 1998; OPPENHEIM; SCHAFER, 1975), which for Equation 4.3 should be

$$H(z) = \frac{N(z)}{D(z)} = \frac{\sum_{s=0}^S b_s z^{-i}}{\sum_{r=0}^R a_r z^{-j}} \quad (4.12)$$

with $a_0 = 1$ and $z \in \mathbb{Z}$. The numerator $N(z)$ and the denominator $D(z)$ are very important in characterizing the filter's response in the frequency domain. At the roots of $N(z)$, the value of $H(z)$ is zero, so the values of z for which the numerator is null are called the *zeros* of the filter. Similarly, at the roots of $D(z)$, the value of $H(z)$ approaches infinity, and so the roots of $D(z)$ are called the *poles* of the filter. Zeros establish which frequencies the filter attenuates most, while poles determine which frequencies the filter accentuates most.

To illustrate these concepts, consider Figures 4.1(a) and 4.1(b), which respectively present the magnitude response of the transfer function and the pole-zero plot of the following filter:

$$y_n = 0.0863x_n + 0.0557x_{n-1} + 0.1494x_{n-2} + 0.0557x_{n-3} + 0.0863x_{n-4} \\ + 1.6992y_{n-1} - 2.1371y_{n-2} + 1.3257y_{n-3} - 0.5001y_{n-4}$$

The filter's roots are also important in determining the filter's *stability*. If any given pole z_k has a magnitude beyond unity (*i.e.*, $|z_k| > 1$), the filter is *unstable*. This means that if the input signal contains a sinusoid of frequency $\arg(z_k)$, the output will contain an exponentially increasing sinusoid of same frequency and an amplitude envelope that

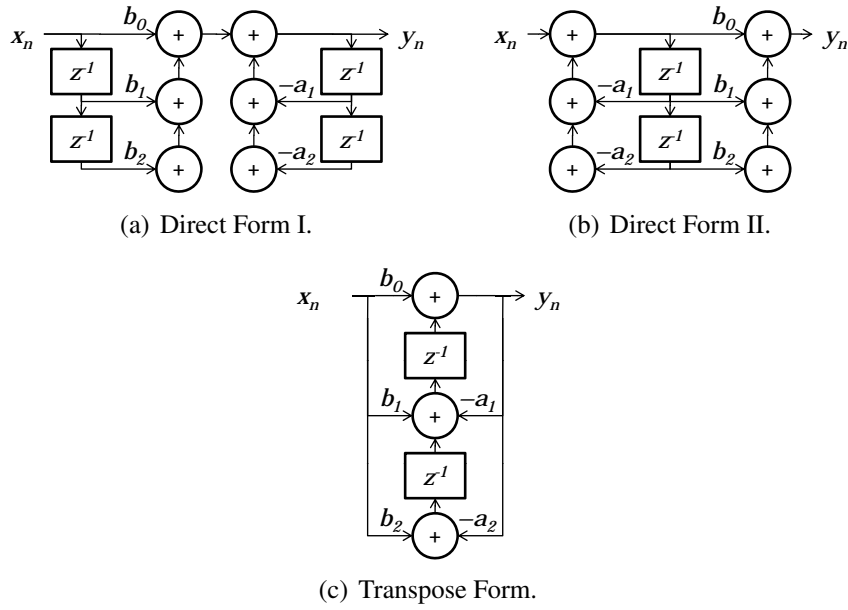


Figure 4.2: Three common forms of implementing a second order filter.

increases exponentially at a rate of $|z_k|^t$, where $t \in \mathbb{R}$ represents time. Similarly, if any pole z_k has a unit magnitude (*i.e.*, $|z_k| = 1$), the filter is *critically stable*, with the output at the pole's frequency resulting in a sinusoid of constant envelope of amplitude 1. In any other case, the filter is *stable*.

4.6 Filter Implementation and Stability

Filter realization on the CPU is usually straightforward. One may choose to directly evaluate Equation 4.3, called the Direct Form I implementation. However, this form is subject to quantization errors in both filter coefficients and the computed output signal, resulting in a z-transform with poles and zeros in slightly different positions. In the worst case, a pole may become unstable due to these numeric errors.

Another choice is to factor the complex numerator and denominator polynomials (PINKERT, 1976) and group the factors into second order (*i.e.*, order 2) *partial fractions*. One then arrives at two other implementations of the filter, the *cascade* and the *parallel* forms. Though all forms are equivalent in a continuous domain, the cascade and parallel forms produce significantly less numerical errors in a digital, discrete computer. The cascade form is given by

$$\begin{aligned}
 H(z) &= H_1(z)H_2(z)H_3(z)\dots \\
 &= \frac{b_0 \prod_{s=1}^S (z - q_s)}{z^{S-R} \prod_{r=1}^R (z - p_r)}
 \end{aligned} \tag{4.13}$$

where p_s are the poles of the filter and q_r are the zeros, with $p_s, q_r \in \mathbb{Z}$. This form of implementation is illustrated in Figure 4.3(a). The second order sections of the form $\frac{z - q_s}{z - p_r}$ are usually implemented either by directly evaluating Equation 4.3 (called Direct Form I), by the Direct Form II or by the Transpose Form (ANTONIOU, 1980). Figure 4.2 shows a flow graph of these forms for second order filters.

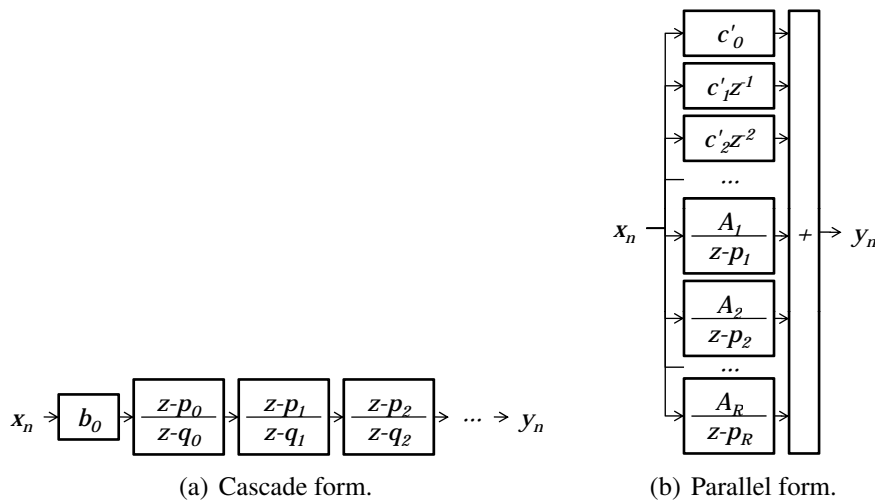


Figure 4.3: Two common forms for breaking down filters with order higher than 2. Each box represents a first or second order filter.

The parallel form is given by

$$H(z) = c'_0 + c'_1 z^{-1} + c'_2 z^{-2} + \dots + \frac{A_1}{z-p_1} + \frac{A_2}{z-p_2} + \dots + \frac{A_R}{z-p_R}$$

where the values of c_s and A_r are obtained by equating the definition of $H(z)$ in Equations 4.13 and 4.14 and solving the resulting system. This form is illustrated in Figure 4.3(b).

The main advantages of computing a recursive filter with the cascade and parallel forms (LIU, 1971) are that the second order sections have a small size and are much less prone to numeric errors when compared with higher-order filters. This fact considerably reduces numeric error in the pole-zero filter structure. Second order filters also have more stable time-varying implementations (LAROCHE, 2007; WULICH; PLOTKIN; SWAMY, 1990). When decimated, a filter's poles can be individually modified as well, which facilitates controlling a time-varying filter.

One may desire to know how the position of poles and zeros will be moved as a result of quantization of filter coefficients. Liu (LIU, 1971) provides the expression

$$\Delta z_k = \sum_{r=1}^R \frac{z_k^{r+1}}{\prod_{\substack{m=1 \\ m \neq r}}^R \left(1 - \frac{z_r}{z_m}\right)} \Delta a_k \quad (4.14)$$

where Δz_k represents by how much the pole was moved and Δa_k represents the quantization error introduced on coefficient a_k , which, for floating-point, is bounded in magnitude by $a_k 2^{-t}$ with t representing the number of bits in the mantissa. This equation applies similarly to zeros by analogy, and thus, allows the study of pole and zero movement in the implementation of the filter.

4.7 Summary

This chapter presented the concept of convolution, digital linear filters and their most important properties and a classification of digital filters. The Fourier series representa-

tion of a signal was given the property of convolution as multiplication on the Fourier domain, established. Filter stability was also discussed, establishing that filters with poles inside the unit circle are stable. This property, however, assumes the operation with real numbers, not with discrete numbers with finite precision such as in a computer. As will be seen in the following chapters, stability may change depending on how the filter is implemented using finite precision.

The next chapter presents digital platforms in which filters can be implemented. Some properties of those platforms, in special their implementation of floating-point operations, are discussed.

5 COMPUTER PLATFORMS FOR SIGNAL PROCESSING

Audio can be processed in a variety of hardware and software platforms. In this thesis, the discussion of analog platforms processing (JACKSON, 1970) is excluded. First the characteristics of digital hardware (CPUs and GPUs specifically) for audio processing are discussed, and then some software libraries that may be used to access this hardware are briefly presented.

There are a number of software platforms and libraries for developing systems with the CPU and with the GPU. One also often needs an implementation of the Fast Fourier Transform (FFT) to perform fast convolution (see Section 4.4), and there are many libraries that compute this.

5.1 Digital Signal Processors

Among the multiple digital platforms, CPUs and GPUs are particularly interesting because of market availability and low cost to the consumer. I have also focused in major brands and products, such as Intel and AMD CPUs and Nvidia and AMD/ATI GPUs.

For digital signal processing (and, in special, digital audio processing), the most important criteria for choosing a platform is signal quality. Signals may be represented in a variety of ways, but currently, most sound processing is performed as floating point operations, so the implementation of floating point in a platform is central to the discussion. In this aspect, CPUs and GPUs share some similarities, specially in recent GPU models which support double precision, 64-bit floating-point. Most GPUs in the market, however, support only 32-bit floating-point numbers, and even though 64-bit floating point can be emulated in such platforms, benchmarks reveal that doing this hurts the performance benefits of GPU-based processing (GÖDDEKE et al., 2005).

5.2 The CPU

The CPUs by Intel and AMD (which dominate the consumer market) implement the IEEE-754 floating-point specification (IEEE, 2008), whereas current GPU implementations deviate in several key aspects (NVIDIA CORP, 2008a), the most significant of which are:

- Division is achieved by multiplication by the reciprocal, resulting in a bigger arithmetic error;
- Denormalized numbers are not supported and are rounded to zero whenever used; and

- Arithmetic in most arithmetic and transcendental functions are subject to bigger error compared to their respective CPU implementations (NVIDIA CORP, 2008a; INTEL CORP, 2004).

In an audio system running on a CPU, performance is primarily determined by the underlying hardware for most tasks, but as in many systems, a choice of language and compiler (PRECHELT, 2000) determines how well a compiled program can access the hardware.

One interesting and freely available FFT library running on CPUs is FFTW3 (FRIGO; JOHNSON, 2005). This library holds a number of FFT algorithms and it can automatically choose one to run by benchmarking the system it is running on. The full source code is available, so, if necessary, one may verify the programs and look for potential errors and bottlenecks. The library also contains benchmarking and accuracy checking routines, allowing users to submit results for their systems to the project's website. Most importantly for market success, the library can be compiled for any of the major commercial operating system (Windows, Linux, Mac).

A significant competitor are the FFTs of the Intel Math Kernel Library (INTEL CORP, 2006). For the reason of not being free, this library could not be tested directly, so the information here comes directly from the library's specification. This library offers a set of FFTW3 wrappers but, unlike FFTW3, does not require system benchmarking¹. The benchmarks reported by Intel, however, disagree with those found in FFTW3's project website, such that both claim to have the best performance in many cases.

5.3 The GPU

The algorithms best suited for parallel architectures, such as GPUs, are those that present few or no data dependences (EYRE; BIER, 2000; HARRIS, 2005), allowing individual parallel processors to work independently on data elements. Because of that, audio processes are classified according to type of data dependences and discuss the impact of dependences on the performance of parallel implementations.

Developing systems that run on GPUs is remarkably different due to their graphics-oriented history. One still needs compilers and libraries that run on the CPU, which in turn becomes more like a process coordinator than an actual computing device, offloading part or all the computation to the GPU.

For programming a general-purpose GPU-based program (GPGPU), one may use a graphics API like OpenGL or Direct3D to access the graphics hardware, and combine them with shaders written in shading languages like HLSL (for Direct3D only), GLSL (OpenGL only) or Nvidia Cg (Direct3D and OpenGL). The shader will perform the actual computation, so one must write, compile and debug it (often a complex task for big programs), load it in the CPU-based program that accesses the graphics APIs and then specify the rendering of graphics primitives (such as lines and squares) in order to get the graphics hardware to run the shader. Many concepts in the problem being solved need to be mapped into respective graphics rendering concepts, which makes the development confusing for developers without a background in computer graphics.

At least one FFT library that runs on such a software platform is known, GPUFFT. Despite the name, the project is not associated with FFTW3, but only inherits the same

¹ FFTW3 does not require benchmarking either actually. When planning a FFT, the program may choose to estimate the best algorithm to use, though this does not guarantee optimal performance.

design goals. Their developers report an at least $4\times$ greater throughput with this library than using FFTW3 with a dual Intel Xeon 3.6 GHz CPU and a Nvidia 7900 GTX GPU. The GPUFFT library, however, is written in Cg and is based on low-level graphics hardware-specific assembly language, requiring users to own a recent Nvidia graphics device².

Recently, graphics device manufacturers have invested in the GPGPU trend and developed new software platforms (and sometimes, corresponding hardware platforms) to solve the issue of concept mapping. Some of these include Nvidia's Compute Unified Device Architecture (CUDA) platform, AMD/ATI's Close to Metal (CTM), PeakStream and OpenCL. PeakStream has been acquired by Google recently and is no longer available. Only recently has OpenCL become available to developers, which took place after developing this work.

Nvidia CUDA presents a 6-level memory organization to the developer and a kernel programming language that is a simple extension of the C language. The platform integrates with Microsoft Visual Studio 2005, allowing developers to easily compile and even debug (in emulation mode) their GPU programs. Emulation mode is not a perfect simulation, as it uses the native CPU implementation of floating-point operations and runs the parallel instances as CPU-based threads, which may run in a different order than that on a GPU. Despite these characteristics, the effects of using CUDA to perform audio processing are similar to those obtained with graphics APIs, with only the "graphical concepts" replaced with those of a typical "parallel machine". In CUDA, however, kernels cannot receive interpolated parameter values as shaders can. This suggests that this interpolation is implemented as shader instructions in recent Nvidia hardware, but this has yet to be confirmed.

Most important regarding CUDA may be CUFFT (NVIDIA CORP, 2008c), which is the FFT library accompanying CUDA. Not unlike the Intel Math Kernel Library and the GPUFFT library, CUFFT has an API inspired by FFTW3, requiring planning before execution. The algorithms run by CUFFT are not specified in the manual and, therefore, there are no published error bounds. The manual also states that transforming power-of-two vectors yields the best performance and accuracy (NVIDIA CORP, 2008c, p. 10), which is on par with most of the other FFT libraries. Unlike FFTW3, CUFFT will transform real vectors into complex vectors prior to processing. When designing a system where multiple FFTs must be computed one after the other, this means that it may be more useful to keep data in the complex domain instead of projecting it into the real domain between each FFT.

Finally, the close CUDA competitor, AMD/ATI's CTM, does not include an FFT library, and works around low-level routine calls, which makes development much more complex.

5.4 Suitability for Audio Processing

In non-recursive processes, numeric error is often treated as a signal that was introduced in the non-quantized, analog signal. The error signal is often assumed to be white noise. Consider the case of mixing (*i.e.*, adding) two signals, where error is generated due to rounding. The introduced error signal would have a maximum amplitude of 0.5×2^{-m} , where m is the number of bits in the mantissa. When processing audio, the relationship between the amplitude of a full-range signal (with amplitude assumed to be 1) and the

² Go to <http://gamma.cs.unc.edu/GPUFFT/> for a list of requirements

Table 5.1: Comparison of precision (in ulps) of floating-point implementations.

Operation	32-bit CPU	32-bit GPU	64-bit CPU	64-bit GPU
$x + y$	0.5	0.5 ^a	0.5	0.5 ^a
$x \times y$	0.5	0.5 ^a	0.5	0.5 ^a
x/y	0.5 ^b	2.5	0.5 ^b	0.5
$1/x$	0.5	1.5	0.5	0.5
\sqrt{x}	0.5	3.5	0.5	0.5
e^x	0.501	2.5	0.502	1.5
$\log(x)$	0.501	1.5	0.501	1.5
$\log_2(x)$	0.501	3.5	0.501	1.5
$\sin(x), \cos(x)$ ^c	0.502	2.5	0.503	2.5
$\tan(x)$	0.501	4.5	0.507	3.5
$\arctan(x, y)$	0.507	3.5	0.501	3.5
$\operatorname{erf}(z)$	0.501	8.5	0.502	6.5

^a For forced intrinsic operations only. Using FMAD instruction (default compiler behavior) is less precise due to truncation of intermediary multiplication result.

^b Without performance optimizations.

^c Also valid for combined SINCOS instruction.

peak amplitude of the noise signal (resulting from rounding), also called *signal-to-noise ratio* (SNR), would be $20\log_{10}\left(\frac{1}{0.5 \cdot 2^{-24}}\right) \approx 150\text{dB}$. Now consider that the human auditory dynamic range is of 120dB. The noise in the resulting signal, reproduced at a comfortable level of amplification (let's suppose the worst case, at 120 dB SPL, would still be 30 dB below the human audibility threshold. Often, people listen to music at 60 to 85 dB SPL, so, using 32-bit floating point, there's even greater room for error in this process. In Table 5.1, I have compiled the precision, given in *units in the last place* (ulp), of several arithmetic operations with floating-point values in current Intel CPUs and Nvidia GPUs.

When synthesizing a sinusoidal wave on the GPU, one gets an additional 2 ulp error. This essentially means that the two last bits of the results contains an error signal, which again, can be assumed to be white noise. This would result in a $20\log_{10}\left(\frac{1}{0.5 \cdot 2^2 \cdot 2^{-24}}\right) \approx 138\text{dB}$ SNR in the resulting signal, showing that noise would still be inaudible. Therefore, the GPU would be adequate to this type of synthesis. Notably, the future Intel Larrabee GPUs will be fully IEEE-754 compliant (SEILER et al., 2008) and, therefore, the expected SNR should be the same as on current CPUs.

Numeric error, however, can and often does become catastrophic in recursive processes. Thus, running a linear recursive filter on a GPU will produce a significantly different waveform when compared to an equivalent implementation (*i.e.*, using the same sequence of instructions) on the CPU. The spectral characteristics of such a waveform are, however, likely to be similar, with the errors possibly affecting (LIU, 1971):

- The position in frequency of the filter's zeros and poles; and
- The amplitude of the poles – and, as a result, the filter's stability.

Table 5.2: Comparison of speed and accuracy across CPU and GPU implementations.

GPU \ CPU	single precision	double precision
	single precision	GPU is faster with slightly more numerical error
native double precision	no clear fast winner but the GPU should present less numerical error	GPU should be faster with slightly more numerical error
simulated double precision	CPU should be faster with considerably more numerical error given GPU's additional precision	CPU should be faster with slightly less numerical error because of a larger mantissa

Actually both errors occur in CPU-based filter designs, even in commercially-available software, though there are strategies to minimize it, as presented in Chapter 4. Table 5.2 compares the expected characteristics of implementing filters on each available platform discussed so far.

Greater inaccuracy in operations may effectively only increase those effects mildly. Slight changes to the filter's poles are usually tolerable for audio processing, and are usually only noticed for poles with a high Q (*i.e.*, narrow bandwidth). The exact amplitude of the poles does affect the perception of *resonance* extracted by the human ear, but often small changes will pass unnoticed or, if noticed, will not be relevant. The worst case is that of a filter becoming unstable. In this case, some modes that would naturally *decay* over time will begin to grow in amplitude, opposing the expected behavior and quickly saturating the signal's dynamic range, therefore, destroying the resulting waveform.

Lastly, communications between the CPU and the GPU have a considerable time delay. This can be observed both when launching GPU programs or when moving memory between video memory and main memory. When developing a complex audio processing application, one will likely desire to do the entire audio processing either on the CPU or the GPU in order to minimize the number of memory transfers between CPU and GPU. Since the GPU is considerably faster for synthesis (TREBIEN; OLIVEIRA, 2008), one is likely to desire to implement the remaining processes on the GPU also, which is challenging in some cases (TREBIEN; OLIVEIRA, 2009). Since issuing the GPU is a slow process, one must also implement strategies to deal with delays in the system (TREBIEN; OLIVEIRA, 2008) without stopping streaming to the audio device.

5.5 Summary

This chapter introduced the three main platforms in the consumer market with which digital filters can be implemented: digital signal processors (DSPs), often found in sound cards; the CPU, found in desktop computers and laptops; and the GPU, found in most but not all of these computers as well. DSPs lack flexibility, since most cannot be programmed or do not adhere to widely accepted programming standards. CPUs are highly

flexible and efficient but often lack computational power for real-time operation. GPUs often provide more computational power but require audio processes to be easily implemented in a parallel machine. GPUs also may cause more stability problems in recursive processes due to lower precision in several floating-point operations.

It has been previously demonstrated that GPUs can perform audio processing in real-time and that implementation of memoryless and non-recursive linear filters on GPUs is straightforward. In the next chapter, an efficient implementation of recursive linear filters for GPUs is developed and discussed in detail.

6 EFFICIENT EVALUATION OF RECURSIVE LINEAR FILTERS ON THE GPU

In the previous chapter, many different types of audio filters were introduced. In particular, non-recursive linear and memoryless nonlinear filters have a trivial implementation in any platform due to the absence of data dependences that would prevent parallel processing. For non-recursive filters, the implementation consists of simply evaluating Equation 4.3a directly, with some additional programming for storing samples from previous blocks in memory and retrieving them when necessary. The mixing filter is also implemented by directly evaluating Equation 4.2, and the implementation for all memoryless processes is analogous.

This chapter presents a method for computing time-invariant recursive linear filters on a GPU in real-time. The method is based on the FFT algorithm and is demonstrated in an application by combining an audio process running the filter with a graphics process implementing a virtual scene where collisions happen and trigger audio events.

With this implementation, it was found that maximum throughput was increased by $2\times$ to $4\times$ for several selected transform sizes. The implementation was based on the Nvidia CUDA platform and the modest speedup, in comparison with previous studies (TREBIEN, 2006), is probably due to a GPU memory bottleneck affecting the computation of the FFT, not to a computational power limit of the GPU. This can be inferred from two facts. First, global memory access (not cached) is used instead of texture memory (cached) in the current implementation. Using texture memory would certainly add complexity to the implementation, but it is likely to increase the speedup factor. And second, the bus bandwidth connecting video memory to the GPU is similar to the one connecting main memory and CPU. Thus, the bus bandwidth is likely to limit memory-intensive algorithms such as the FFT.

6.1 Dependences in Linear Filtering

A non-recursive filter as in Equation 4.3a has no data dependences other than on the input values. Thus, its GPU implementation consists of simply evaluating the filter equation. The resulting output involves only operations with B , which represents the *impulse response* of the filter. Non-recursive filters are often used to simulate *reverberation*, and in this case, the impulse response can be recorded in a real-world environment and then applied as B in the filter to produce a version of the input that sounds as if originated in the same reverberant space where the impulse response was captured. However, reverberation impulse responses are often several seconds in length, ranging from thousands to hundreds of thousands of filter coefficients, vastly affecting computational performance.

Many algorithms have been proposed to handle this problem (HAYKIN; VEEN, 1998; BIERENS; DEPRETTERE, 1998; GARCÍA, 2002), most of which see the filtering operation as a convolution and then perform its computation in the frequency domain using a single FFT.

In a recursive filter, however, the value of an output sample y_n depends on the values of previous ones (*e.g.*, y_{n-1} , y_{n-2} , \dots). For implementation, one could think of directly evaluating Equation 4.3. However, the values of immediately past output samples are usually not available during parallel computation, as they are also being computed simultaneously. Forcing serialization (*i.e.*, waiting for each output sample to be available before computing the next) severely impacts GPU performance, making efficient GPU implementations of recursive filters a non-trivial problem. To the best of my knowledge, the technique described in this thesis is the first one to efficiently implement recursive filters on GPUs. The following sections describe the solution to this problem.

6.2 Unrolling the Filter Equation

Equation 4.3b establishes recursive dependences on y_n with y_{n-1} , y_{n-2} , \dots , y_{n-Q} . In other words, these dependences are established on the output signal y over the range of indices $[n-Q, n-1]$. In the following discussion, for simplicity, the sets of b_i and a_j coefficients may be referred to as

$$\begin{aligned} a &= [1 \quad a_1 \quad \dots \quad a_Q]^T \\ b &= [b_0 \quad b_1 \quad \dots \quad b_P]^T \end{aligned}$$

To avoid the need for synchronization, the recursion can be propagated until all necessary output samples are available from the computation of preceding blocks. This begins with bringing the first term out of the summation

$$y_n = w_n - \sum_{j=1}^Q a_j y_{n-j} = w_n - a_1 y_{n-1} - \sum_{j=2}^Q a_j y_{n-j}$$

If n is replaced with $n-1$ in Equation 4.3b, an expression for y_{n-1} is obtained and the term can be substituted

$$y_n = w_n - a_1 \left(w_{n-1} - \sum_{j=1}^Q a_j y_{n-1-j} \right) - \sum_{j=2}^Q a_j y_{n-j}$$

For clarity, in order to obtain the term y_{n-j} in both summations, j can be replaced with $j-1$ in the first summation

$$y_n = w_n - a_1 w_{n-1} + a_1 \sum_{j=2}^{Q+1} a_{j-1} y_{n-j} - \sum_{j=2}^Q a_j y_{n-j}$$

This an expression for y_n in terms of w_n , w_{n-1} and y_{n-2} , y_{n-3} , \dots , y_{n-Q-1} . The recursive dependences are now established on y over the range $[n-Q-1, n-2]$. Notice that now there are also dependences, though not recursive, on w over the range $[n-1, n]$.

If this unrolling process is repeated one step further, by isolating the first element of every summation

$$y_n = w_n - a_1 w_{n-1} + a_1 a_1 y_{n-2} + a_1 \sum_{j=3}^{Q+1} a_{j-1} y_{n-j} - a_2 y_{n-2} - \sum_{j=3}^Q a_j y_{n-j}$$

then replacing y_{n-2} with Equation 4.3b

$$y_n = w_n - a_1 w_{n-1} + (a_1^2 - a_2) \left(w_{n-2} - \sum_{j=1}^Q a_j y_{n-2-j} \right) + a_1 \sum_{j=3}^{Q+1} a_{j-1} y_{n-j} - \sum_{j=3}^Q a_j y_{n-j}$$

then replacing j with $j-2$ in the new summation and rearranging the terms, the equation becomes

$$y_n = w_n - a_1 w_{n-1} + (a_1^2 - a_2) w_{n-2} - (a_1^2 - a_2) \sum_{j=3}^{Q+2} a_{j-2} y_{n-j} + a_1 \sum_{j=3}^{Q+1} a_{j-1} y_{n-j} - \sum_{j=3}^Q a_j y_{n-j}$$

which depends on y over the range $[n-Q-2, n-3]$ and on w over the range $[n-2, n]$.

This establishes a pattern that can be stated in matrix form. At the m -th step, a dependence on y_{n-m} is replaced by a non-recursive dependence on w_{n-m} and a recursive dependence on $y_{n-m-Q+1}$. For instance, if $Q=4$, at the first step, a dependence on y_{n-1} is traded by others in w_{n-1} and y_{n-5} . At the next step, a dependence on y_{n-2} is then traded by others in w_{n-2} and y_{n-6} , and so on. At any point, one can say they have an expression of the form

$$y_n = \sum_{k=0}^{m-1} c_k w_{n-k} + \sum_{j=0}^{Q-1} d_j y_{n-m-j} \quad (6.1)$$

where

$$C = [c_0 \ c_1 \ \dots \ c_{m-1}]^T$$

$$D^{(m)} = [d_0 \ d_1 \ \dots \ d_{Q-1}]^T$$

The expression $D^{(m)}$ represents the vector D at the m -th step of unrolling. By defining

$$A' = [-a_1 \ -a_2 \ \dots \ -a_Q]^T$$

the vector D can be obtained starting with

$$D^{(0)} = [1 \ 0 \ \dots \ 0]^T$$

and computing the next step iteratively as

$$D^{(k+1)} = D^{(k)} [0] A' + S D^{(k)} \quad (6.2)$$

where $D^{(k)} [i]$ is the i -th coefficient of vector $D^{(k)}$ and

$$S = \begin{bmatrix} 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 \\ 0 & 0 & 0 & 0 & \dots & 0 \end{bmatrix}$$

which represents a positional right shift on vector $D^{(k)}$ and, therefore, multiplication with S can be easily optimized during implementation. The equations obtained during unrolling also establish that

$$C = [D^{(0)} [0] \ D^{(1)} [0] \ \dots \ D^{(m-1)} [0]]^T \quad (6.3)$$

Actually, Equations 6.2 and 6.3 are equivalent to the previous algebraic manipulation where y_{n-m} has been replaced by the definition of the filter and rearranged the terms and summations subsequently.

Due to block processing, one can assume that there is in fact a value for m so that all samples up to y_{n-m} are available as a result of processing previous blocks. Therefore, Equation 6.1 can be expressed in terms of convolutions using Equation 4.5. In other words, the signal c can be defined such that $c_n = C[n]$ for all defined values of $C[n]$ and $c_n = 0$ otherwise. The signal d can also be established as $d = D^{(m)}$ for defined values, otherwise $d_n = 0$. By defining the available part of the output signal as

$$\bar{y}_k = \begin{cases} y_{k-m} & \text{if } k \leq n \\ 0 & \text{if } k > n \end{cases}$$

it follows that Equation 6.1 may be rewritten as

$$y_n = (c * w)_n + (d * \bar{y})_n \quad (6.4)$$

Similarly, Equation 4.3a can be expressed as $w_n = (b * x)_n$. Substituting this into the previous equation finally yields

$$y_n = (c * b * x)_n + (d * \bar{y})_n \quad (6.5)$$

Note that since c and d only depend on the a_j coefficients, $c * b$ and d can be precomputed. Note also that samples from x are obtained by accessing the current and previous input blocks, and samples from \bar{y} are obtained from previous output blocks. This means that the filter can be implemented using Equation 6.5 and the previous definitions of c and d along with the provided definitions of a and b .

6.3 Computing the Convolutions

As discussed in Section 4.4, the DFT can be used to compute a convolution between two discrete signals. Moreover, an FFT algorithm can be used for efficiency. However, special care must be taken when using FFTs for fast convolution since FFT algorithms assume that the input is one period of a periodic signal. Without modifying the signals to be convolved, the result will be the *circular* convolution of the two, *i.e.*, assuming they both are infinite and repeat themselves, which does not apply to the method being developed. In this case, vectors b , c and d contain an aperiodic signal and the undefined regions are assumed to contain null samples. Signals x and \bar{y} are assumed to be aperiodic as well. A convolution between aperiodic signals is called *linear* and it assumes that samples beyond the defined ones are all null, which is the case in the method being proposed. Additionally, note that the two convolutions in Equation 6.5 cannot be merged generally, so the method we are developing will require 2 FFTs.

Interestingly, a circular convolution can be used to express part of the result of a linear convolution, allowing the use of an FFT to optimize the operation. This can be done by *padding*, which consists of null samples added to the beginning or the end of a signal. There are several known ways to implement fast convolution with this method (HAYKIN; VEEN, 1998), and here I provide a solution using a method called *overlap-save* (OPENHEIM; SCHAFER, 1975). Another common method with the same computational requirements is *overlap-add*.

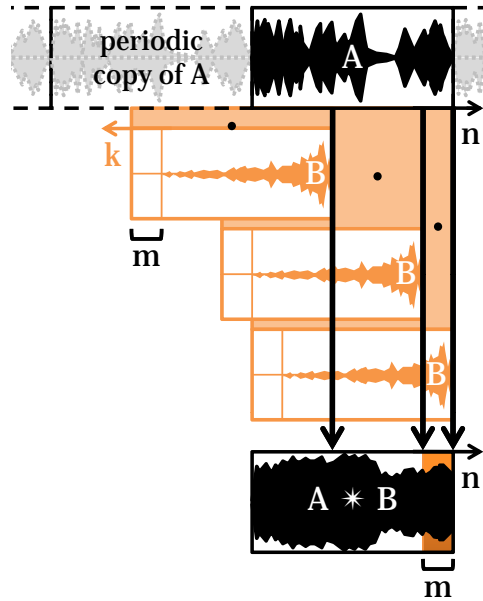


Figure 6.1: Illustration of one step of a fast convolution algorithm. In a cyclic convolution (efficiently computed with the FFT), each n -th output sample (vertical arrows) receives the dot product of $\text{rev}\{B\}$ (B in reverse order) and a periodically-shifted version of A . The illustration shows that, by adding m null samples to the end of B , the last m samples of the output will correspond to m samples of a linear convolution, since only null samples of B multiply the wrapped-around part of A .

A circular convolution between two periodic signals f and g , such as that computed through the Fourier domain property from Equation 4.11, is given by

$$(f * g)_n = \sum_{m=0}^{R-1} g_m f_{n-m+kR}$$

with $k \in \mathbb{Z}$ and R the period of both f and g . The resulting signal $(f * g)$ is a periodic signal as well. Since both signals have the same period, the sample indices for g always wrap around one period, aligning with all samples of f . This process is illustrated in Figure 6.1.

However, if one of the input signals contains r null samples at its end, then r output samples at the end of $(f * g)$ yield the exact same result of a linear convolution. In other words, to obtain the desired linear convolution samples, let $g_n = 0$ for $(R - r - 1) \leq n \leq (R - 1)$ and then evaluate for n in the same range

$$(f * g)_n = \sum_{m=0}^{R-r-1} g_m f_{n-m}$$

This property is applied by padding one of the signals with a block of null samples to efficiently perform linear convolution in the frequency domain. This strategy is used here to independently evaluate the two convolution terms in Equation 6.5, after which their results are added to obtain the output vector y . For simplicity, the process is described in time domain initially.

For the first term of Equation 6.5, the convolution $(c * b * x)$ needs to be computed. Since c has m coefficients (from Equation 6.1) and b has $P + 1$ coefficients (from Equation 4.3a), the result of the convolution $(c * b)$ has $m + P$ coefficients. Thus, let k be the

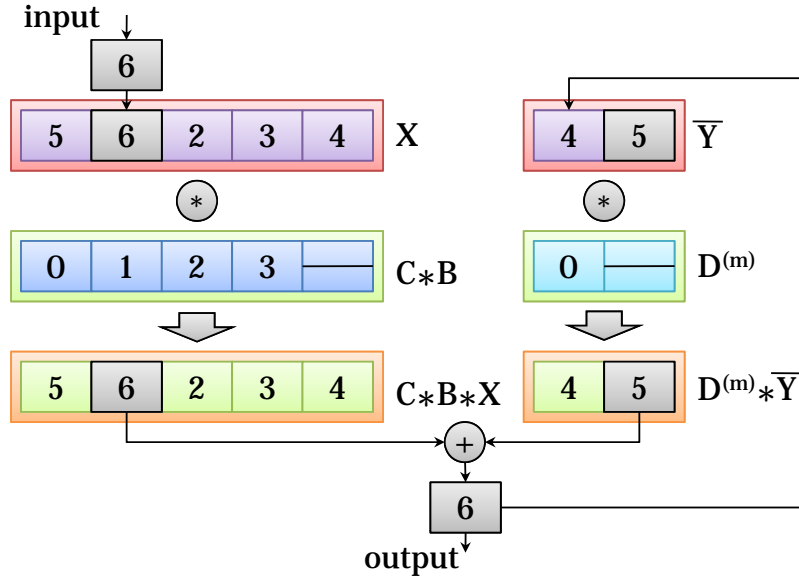


Figure 6.2: Summary of operations for processing an input block of size r from the input circular buffer X . The FFTs of padded versions of $(c * b)$ and d (with at least r null samples each) are precomputed. The FFTs of X and \bar{Y} are computed and multiplied (element-wise) by the spectra of $(c * b)$ and d , respectively. After applying an inverse Fourier transform to these results, the corresponding blocks of r samples are added and update the circular buffer Y . For the next step, the samples in Y update a block in \bar{Y} .

smallest integer such that $kr \geq (m + P + r)$. These $m + P$ coefficients are stored on the left part of a block CB with size $R = kr$ and pad its remaining elements with zeros. This guarantees that CB has at least r null samples at its end. Then r samples of $(c * b * x)$ are computed by storing R input samples on an input block X and convolving it with CB . These partial results are combined with the r samples resulting from the convolution between d and \bar{y} . These are stored in blocks D and \bar{Y} , respectively, with length equal to the smallest integer $lr \geq (Q + R)$. Similarly, the remainder of block D is padded with zeros. The convolution $d * \bar{y}$ produces the r samples from the second part of Equation 6.5. An output block Y is filled in with the sum of these partial results. Figure 6.2 illustrates this process.

Note that the indices for Y are the same as those for X where the r input samples are stored. Since the FFT assumes that all signals are periodic, a circular shift of s samples in the input vector yields the same output vector with an equivalent circular shift of s samples. Therefore, instead of actually shifting X by r samples to accommodate new input samples at the end, this process can be optimized by using circular buffers for both X and Y and by writing new samples to consecutive positions of X , thus reducing the memory overhead of accessing the vector.

To perform this computation in the frequency domain, the FFTs $\mathfrak{Z}(CB)$ and $\mathfrak{Z}(D)$ of CB and D , respectively, are precomputed. During the actual processing, the FFTs $\mathfrak{Z}(X)$ and $\mathfrak{Z}(\bar{Y})$ of X and \bar{Y} are computed. The block with r samples for Y is obtained as:

$$Y_n = (\mathfrak{Z}^{-1}(\mathfrak{Z}(CB) \odot \mathfrak{Z}(X)) + \mathfrak{Z}^{-1}(\mathfrak{Z}(D) \odot \mathfrak{Z}(\bar{Y})))_n \quad (6.6)$$

where \odot is the element-wise multiplication and \mathfrak{Z}^{-1} is the IFFT.

6.4 LPC Extraction and Re-Synthesis

One of the most useful methods for computing the required filter coefficients is re-synthesis through *linear predictive coding* (LPC) (HARRINGTON; CASSIDY, 1999). With LPC, a filter that predicts the output from part of the input is designed to minimize the error ε in

$$x_n = \sum_{j=1}^Q a_j x_{n-j} + \varepsilon_n \quad (6.7)$$

Note ε is a signal, and that this equation is actually a recursive linear filter where x is the *output* and ε is the *input*. That means that the original signal can be exactly re-synthesized given the filter coefficients a_j and the error signal ε (also called *residual* signal). The quantity to be minimized is the quadratic error

$$E = \frac{1}{N} \sum_{k=1}^N \varepsilon_k^2$$

where N is the number of samples in the recorded input signal. With $a_0 = 1$, the optimal solution for the set of coefficients a_j , $1 \geq j \leq Q$, is given by (RABINER; JUANG, 1993):

$$\begin{bmatrix} a_1 & a_2 & \dots & a_Q \end{bmatrix}^T = T^{-1}r \quad (6.8)$$

where

$$T = \begin{bmatrix} r_{x,x}(1) & r_{x,x}(2) & \dots & r_{x,x}(Q) \\ r_{x,x}(0) & r_{x,x}(1) & \dots & r_{x,x}(Q-1) \\ r_{x,x}(1) & r_{x,x}(0) & \dots & r_{x,x}(Q-2) \\ \vdots & \vdots & \dots & \vdots \\ r_{x,x}(Q-1) & r_{x,x}(Q-2) & \dots & r_{x,x}(0) \end{bmatrix}$$

$r_{x,x}(k)$ represents the correlation of the signal x with a shifted version of itself by k samples, defined as

$$r_{x,x}(k) = \frac{1}{N} \sum_{n=0}^{N-1} x_n x_{n-k}$$

Since T is a Toeplitz matrix, its inverse can be computed with the split Levinson-Durbin algorithm (DELSARTE; GENIN, 1986).

The residual signal ε is usually sufficiently simple to be replaced by some primitive waveform that approximates it — usually a pulse, a pulse train or white noise with some envelope (COOK, 2002, p. 48–49) —. In that case, perfect re-synthesis is not achieved, but the resulting signal sounds similar to the original one because the filter models the resonances of the sound source. This method not only saves memory but it also allows the generation of several similar sounds by synthesizing slightly different residuals.

6.4.1 Stability Concerns

A filter designed with LPC is not guaranteed to be stable. Generally, increasing the number of coefficients generates poles in the filter's transfer function that are inside but very close to the unit circle $e^{j\omega}$ on the z domain, causing some numeric error to be fed back into the filter at some frequencies. This can usually be circumvented by performing

the LPC extraction with a smaller number of coefficients. In the application, I have found that 32 coefficients produced a stable response for the recorded sounds, while modeling the sources' resonances acceptably.

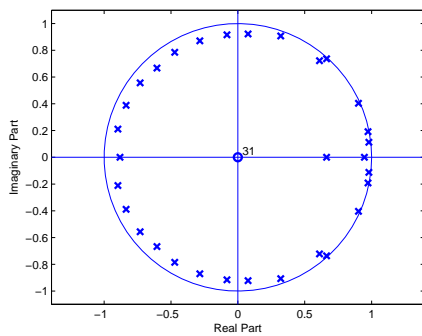
Implementations using the GPU of non-recursive linear filters are stable by definition. On CUDA-based implementations (TREBIEN; OLIVEIRA, 2009), however, the recursive filter is sensitive to arithmetic errors: in some cases, filters designed to have poles near the unit circle on the z -plane often become unstable when realized on the GPU. This is particularly due to the use of an FFT to compute the filter. The main source of error on an FFT is the implementation of floating point operations, which, as mentioned previously, is not IEEE-754 compliant on current GPUs. Another source of error is the type of operations involved, which correctly suggests that different FFT algorithms on the same platform provide different error bounds (MEYER, 1989). In CUDA, FFTs are implemented by the CUFFT (NVIDIA CORP, 2008c), which does not specify which algorithms it uses or their error bounds. An equivalent CPU-based implementation using the FFTW library (FRIGO; JOHNSON, 2005) appears to be less subject to this problem, since I have found no instabilities in the test cases so far.

While research on stability for these filters is ongoing, a simple heuristic can be devised to determine if an implementation is stable. The heuristic consists of running an *impulse signal* through the filter, recording the output, and analyzing it using the STFT in search of decaying and increasing modes. All frequencies are expected to be decaying, though a direct frame-by-frame comparison may sometimes falsely indicate an increase in energy for a decaying mode. This is a result of both numeric error and of interference caused by "leakage" due to the FFT's windowing process. Therefore, one should preferably compare all FFT frames with the first output frame. If a number n of frames (determined by the developer) contains less energy in all frequencies than the first frame, the filter is highly likely to be stable. The number n can be increased as desired to provide greater accuracy for the test.

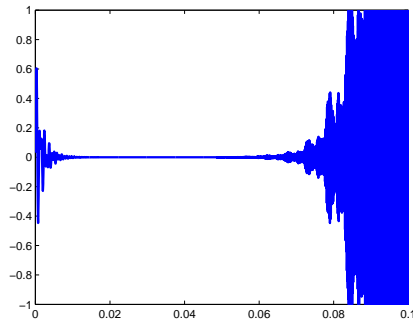
For efficiency, it is recommended to disable the non-recursive section of the filter. This does not affect a stability test, since the non-recursive section is stable by definition. The non-recursive section also often is a long signal which, convolved with the input, produces a large number of interacting modes on the output, causing constructive and destructive interferences and, thus, requiring a much larger analysis window. With a small number of recursive coefficients (as mentioned previously, almost never more than 32), the number of decaying modes is small (not more than 31 in this case) and the amount of interference between them is much less significant.

From my own experimentation, I have chosen a particular audio recording that yields an unstable implementation when running with 256 samples per block, but yields a stable implementation when running with 512 samples per block. Notice, however, that this is not the typical situation. Figure 6.3(a) shows the pole-zero plot of the filter designed with LPC. With all poles inside the unit circle, the filter is theoretically stable, with infinite precision. Many of the poles are near the unit circle, however, making this a filter that can easily become unstable when realized with finite precision. Figures 6.3(b) and 6.3(c) show the impulse response of the filter, demonstrating that one of the implementations is stable, while the other is not. Figure 6.3(d) shows a detailed point-by-point difference in the output signal of the both filters for approximately the first 5 output blocks of 256 samples. This difference is the instability introduced and accumulated by the unstable filter.

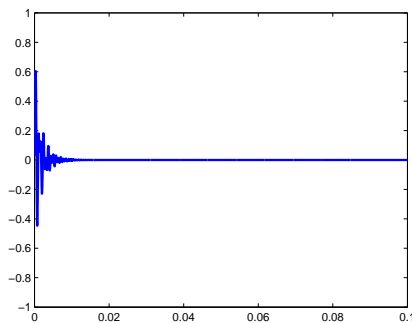
Then I proceeded to compare the frequency content of the output signal at many points



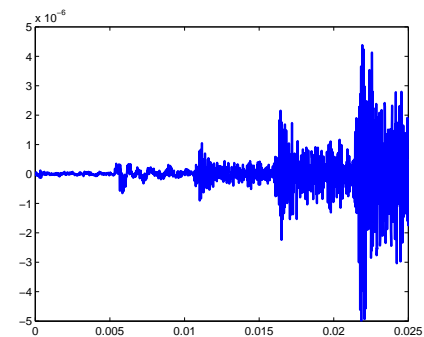
(a) Pole-zero plot for the filter. Many of the poles lie close to the circle of unity, suggesting that an implementation of this filter is not likely to be stable.



(b) Impulse response to the filter implemented using a block size of 256 samples. The accumulation of error becomes evident around $t = 0.06$.



(c) Impulse response to the filter implemented using a block size of 512 samples. The signal decays to zero.



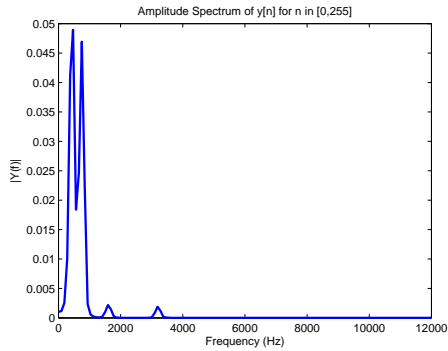
(d) Difference between the two signals, showing the unstable component increasing over time.

Figure 6.3: General observations of a particular filter designed with LPC that becomes unstable when implemented on the GPU.

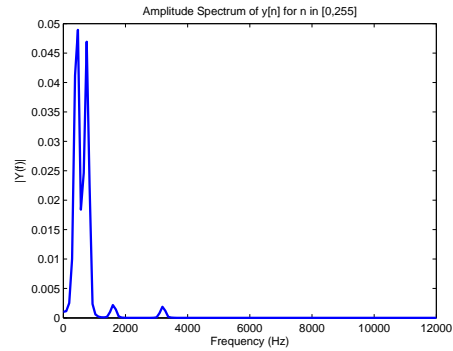
in time; for this, the output signal was divided in “frames” defined as blocks of 256 samples windowed by a Hanning window. Figures 6.4(a) and 6.4(b) show that the frequency content of the output signal is essentially the same for the initial frame (frame 0). The results continue similar until, on frame 5 (see Figures 6.4(c) and 6.4(d)), the unstable increasing modes reach higher amplitude levels than the stable decreasing modes. This remains the behavior of the system for all following frames, as can be observed on frame 20 (see Figures 6.4(e) and 6.4(f)).

6.5 Summary

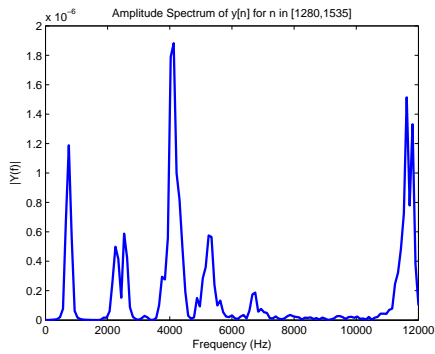
This chapter introduced an efficient technique for implementing recursive filters in parallel platforms, with special interest in GPU-based implementation. The equation defining time-invariant linear filters was unrolled in order to eliminate recursive dependences that prevent a trivial GPU-based implementation. Due to block processing, the number of times the equation needs to be unrolled is limited. When all necessary dependences are eliminated, the resulting equation can be expressed as two linear convolutions, which are then computed efficiently through multiplication on the Fourier domain.



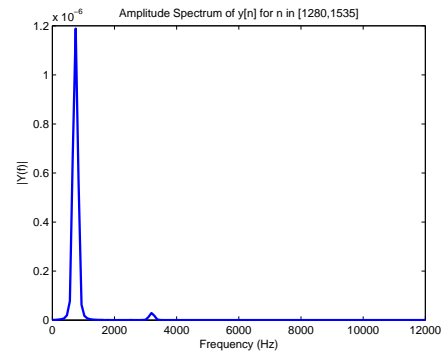
(a) DFT of frame 0 of the implementation using a block size of 256 samples.



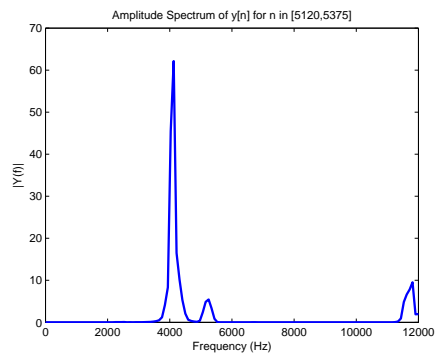
(b) DFT of frame 0 of the implementation using a block size of 512 samples. The output of both implementations is essentially equal across the spectrum.



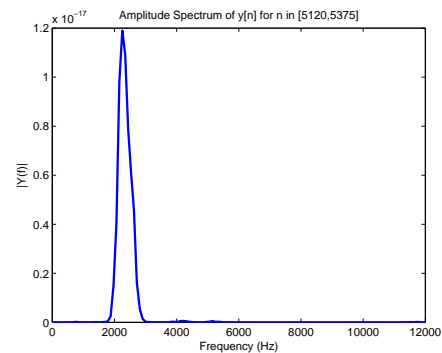
(c) DFT of frame 5 of the implementation using a block size of 256 samples. At this frame, increasing unstable components begin to dominate the spectrum of the signal.



(d) DFT of frame 5 of the implementation using a block size of 512 samples. The strong low-frequency mode is present in both implementations with the same energy.



(e) DFT of frame 20 of the implementation using a block size of 256 samples. Notice the amplitude level; only high-energy unstable components can be observed.



(f) DFT of frame 20 of the implementation using a block size of 512 samples. Only a sub-sonic decaying mode can be found.

Figure 6.4: Discrete Fourier analysis of certain output “frames”, where a frame is part of the output signal multiplied by a window function. Each frame corresponds to 256 samples of the output signal.

The technique of linear predictive coding (LPC) was also presented, which provides coefficients for a recursive filter from recordings. By experimenting with the filter, it was found that this implementation may face stability issues when the designed filter presents poles near the unit circle. Also, the effect of instability was analyzed for samples that, after LPC extraction, produced unstable filters with the proposed method.

In the next chapter, we present an application that used the proposed filter implementation and LPC to generate sound for a virtual scene. The application effectively demonstrates the usefulness of the technique for the graphics community. Additionally, technical details of the system and a performance comparison with a CPU-based implementation are provided.

7 VALIDATION OF THE PROPOSED TECHNIQUE

In order to validate the technique, I have developed an application consisting of two independent processes to demonstrate the approach. The two processes communicate with each other to maintain synchronization of sound events. One of the processes, which is presented in Section 7.1, is a graphics only program which renders a scene where sounds are necessary. The other process, presented in Section 7.3, generates the sound using the proposed approach. The two processes communicate with each other using an inter-process communication API provided by the operating system and described in Section 7.2. Finally, the performance of the method is evaluated in comparison with the same method implemented using the CPU.

7.1 Interactive Graphics Interface Process

To demonstrate the relevance of the proposed approach to real-time graphics applications, a simple application (BARCELLOS, 2008) was created¹ in C# that simulates the collisions among a set of spheres and a piecewise planar surface. The concept is illustrated in Figure 7.1. The physical simulation is based on Newton's laws, and the collision detection, on single-point contacts. The application also simulates different materials by assigning different restitution coefficients to the scene objects. The piecewise planar surface was modeled as made of wood, and the spheres, as either glass, wood or plastic. In the current implementation, all spheres have the same material, but one could modify the sound generating process to accommodate a greater number of materials simultaneously. Figure 7.2 shows two screenshots of the actual application.

7.2 Inter-Process Communication Protocol

As objects collide, the graphics process sends messages to the audio process that implements the method, which then synthesizes the appropriate sound for each event. The messages contain information such as the energy lost in each impact and the bodies involved in the contact.

The sound generating process creates a shared memory block to hold a vector with information about each sphere, and a message queue for collisions. When a sphere in the physically-based simulation is rolling over a surface, the speed of the sphere is updated in its corresponding entry in the shared memory vector. The sound generating process reads this information once per input block and updates the pulse-train generation parameters.

¹ A more reliable physics library such as PhysX could have been used instead, but this implementation was readily available and it fulfilled the demonstration requirements well.

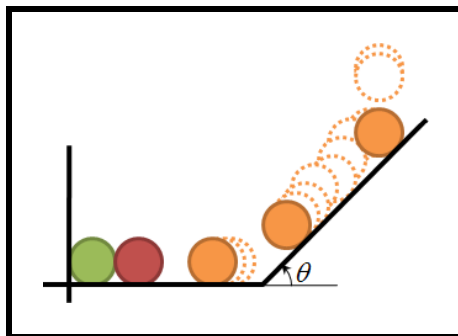


Figure 7.1: A conceptual diagram of the 3D graphics application used to test the recursive filters. A set of spheres of different materials collide among themselves and against a piecewise planar wooden surface.

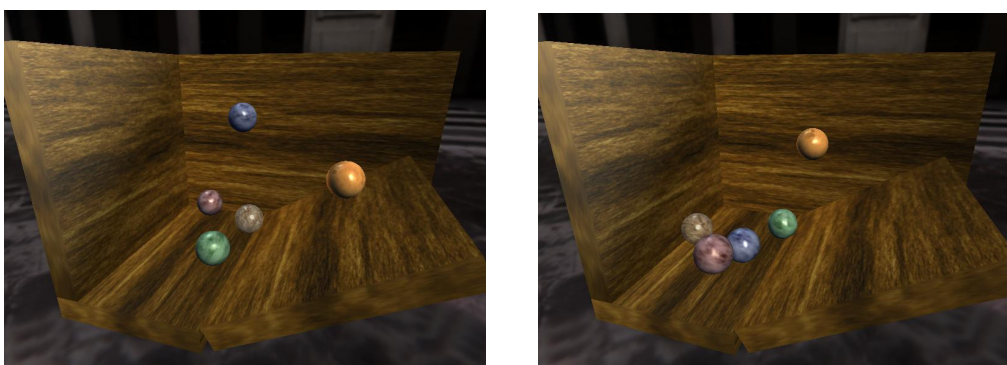


Figure 7.2: Two snapshots of the scene used in the real-time 3D graphics process. A set of spheres collide against a piecewise planar surface, as well as against themselves. The application performs physically-based simulation of collisions.

When a collision is detected on the simulation, the event is recorded in a queue with 1,024 positions and a shared semaphore is released, allowing the sound generating process to read it and create the windowed *sinc* event on the polyphonic synthesizer list of events.

This method of communication represents a very small overhead, allowing both the physically-based simulation and the sound generating processes to run in real time with no perceptible delays between collision detection and collision sound output. A video showing the execution of the application and recorded in real time can be downloaded from http://www.inf.ufrgs.br/Audio_on_GPU/en/media/.

7.3 Audio Generation Process

In order to generate realistic sounds using filtering instead of sample playback, one filter must be designed for each type of material. This can be obtained by computing the filter coefficients from the actual sound of the material. Such a filtering-based approach has advantages over sample playback since by changing a few parameters, one can re-synthesize variations of the original sound. For instance, the sound of impacts between objects at different speeds and contact angles can be simulated. Then, the computation of filter coefficients from the actual material sounds is described.

For the audio process, coefficients were precomputed for a number of sounds representing the strike between two spheres and the strike with a wooden surface. I could have



Figure 7.3: The setup environment for recording samples for the glass material. Two glass surfaces (the small glass jars) were suspended with threads, which were connected to a broomstick (at the top), which in turn was suspended on two small wooden stools. A microphone was placed directly below the jars, and a blanket surrounded the whole setup to help dampening some ambient noise. An additional thread was connected to one of the jars' thread (the right one) so that the jar could be swung from outside the recording area, allowing the blanket (on the background) to entirely cover these elements. Nonetheless, the sounds were recorded at a time of day where little noise was present, and several samples were recorded and later the best sample was selected. The same procedure was repeated for the sounds of plastic and wood materials, but then using plastic jars and wooden spherical objects instead.

used as source some recordings from a sample library, but as I did not have any satisfactory sound, I have recorded the source waveforms for each material in a loosely controlled environment (without considerable noise dampening, for instance), which was found to produce acceptable results. Pairs of hollow surfaces made of glass, wood and plastic were suspended with threads and the sound of the impact between surfaces of the pair was recorded, as shown in Figure 7.3. The sound for impact with the table was obtained in the same environment by recording the sound of a finger striking a wooden table. This process was repeated a number of times and I selected one representative sample for each material.

Recording the input sounds in a controlled environment (such as in a recording studio) would certainly improve the quality of the output, increasing the correspondence between the designed filters and the pure, noise-free source sounds. As the recordings contained ambient noise, the sound was then adaptively-filtered using the software Adobe Audition, which improved the perceived output sound quality. Without this, the results would contain more noise and might even contain additional resonant modes belonging to other

ambient sound sources. The impact of this procedure is that, since the source recordings contain more environmental noise, the resulting filter allows more noise to pass to the output. Nonetheless, the practical experimentation seems to show that most of this noise is masked out by strong resonant modes that the filter models. Additionally, the goal is not to achieve exact physical accuracy but only to generate convincing results, and in most cases, careful modulation of the output signal is sufficient to provide that sensation.

By inspection, I noticed that the residuals I obtained from each recording seemed to approximate a waveform with a short (somewhat low-pass filtered) pulse at the beginning, followed by a long decaying filtered white noise tail. This suggests that a residual composed of a filtered pulse with a filtered white noise tail should produce an output that sounds like the input. Part of this noise tail is likely due to ambient noise at the time of capture, but when I attempted to re-synthesize the sounds, this noise tail seemed important to convey the same resonant qualities captured in the recording of the source. It is possible that other types of synthesized residuals could create even more realistic output sounds.

To obtain such a residual, a granule of a Hann-windowed *sinc* function was generated for each sound event (*i.e.*, collision). The expression that computes a granule of N samples of amplitude α and a bandwidth parameter β is given by

$$x_n = \frac{\alpha}{2} \left(\cos \left(2\pi \frac{n}{N} \right) + 1 \right) \text{sinc}(\beta n), -\frac{N}{2} \leq n \leq \frac{N}{2}$$

The resulting signal was then convolved (by filtering) with another signal B consisting of unfiltered exponentially-decaying white noise:

$$b(t) = \kappa \text{rand}_u(t) \left(1 - e^{-b_a t} \right) \left(e^{-b_d t} \right) \quad (7.1)$$

where $\kappa = \frac{1}{128}$ is defined as an energy compensation coefficient (to avoid saturated output), the attack rate $b_a = 512$ and the decay rate $b_d = 16$. These values were defined empirically. The convolution with a windowed *sinc* pulse is a form of low-pass filtering. This simulates what is observed in most real sounds, where high frequencies are most notably attenuated when the originating event takes place with less intensity.

The filters that the application executes to re-synthesize collision sounds are formed by the non-recursive coefficients from B (Equation 7.1) and the recursive coefficients A (Equation 6.8) obtained by the LPC method. Figure 7.4 summarizes the tasks involved both in preprocessing and runtime of this implementation. The application uses two filters, one (*SS*) for sphere-on-sphere collision sounds, and another (*SP*) for sphere-on-plane collision sounds. A polyphonic synthesizer generates a *sinc* pulse for every collision. For sphere-on-sphere collisions, the pulses are fed to *SS* only, while for a sphere-on-plane collision, the pulse is fed to both filters. The amplitude parameter α and the bandwidth parameter β are proportional to the energy dissipated in the impact.

For every sphere that is rolling over a surface, my implementation generates a quasiperiodic train of pulses with amplitude α and bandwidth β proportional to the sphere's speed v . The interval Δt between each pulse is a sum of a periodic component, which is inversely proportional to v , with a random Poisson variable with rate parameter λ proportional to v .

Finally, the LPC implementation for computing the filter coefficients from the sounds of the materials (see Section 6.4) was written in MATLAB and the filters have been implemented using C++ and CUDA with the CUFFT library. A class diagram of the filter's implementation can be found in Figure A.1. Source code for the implementation of the

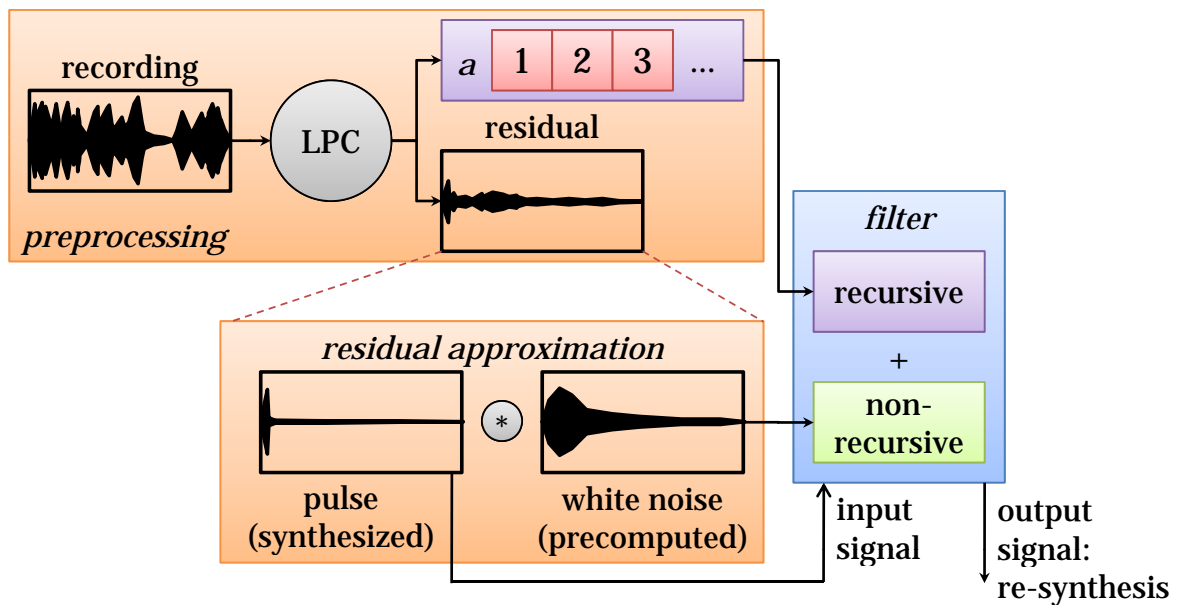


Figure 7.4: A summary of all tasks and data produced at each stage in the current demo. Before runtime, every recording is passed to the LPC algorithm, which extracts corresponding recursive coefficients a for the filter and residual signal. By inspection, I found that the residual can be approximated during runtime as a convolution of a *sinc* pulse and windowed white noise. Since a convolution is performed by the non-recursive part of the filter, this part of the process can be implemented by simply setting the non-recursive coefficients as the white noise signal. During runtime, *sinc* pulses are synthesized and passed as the input signal to the filter, which then computes a re-synthesized version of the original recording for each input pulse. Parameters of the *sinc* pulse are synthesized for each sound event, producing variations of the sound as the output.

filter is given in Listings A.1, A.2 and A.3, and for generation of the “sinc” events, in Listing A.4.

It should be noted that, for complete realism, other auditory cues such as distance attenuation and ambient reverberation should be added in a more serious application. This was not included in this work because the implementation was only intended to demonstrate the recursive filter in operation.

7.4 Performance Comparison

In order to compare the performance of the GPU approach against a CPU implementation, I devised a simple method that verifies when the process is running in real-time under a specific configuration (defined by the choice of block size, filter length, sampling rate, etc.). For this, a low overhead modification was added to the callback called by the low-latency audio interface Steinberg ASIO (ASIO, 2006). This callback receives and provides blocks to the audio device and with some modification additionally checks if a block is available for playback. If this condition is sustained for a number of consecutive calls (equivalent to at least 10 seconds of the audio stream in the chosen sampling rate/block size configuration), the system signals that the real-time test has passed. This method is then used to compare an implementation in C++ running on the CPU with the CUDA implementation running most of its computation on the GPU. It should be noted that only the graphics process was running during the benchmarks.

The tests were run on a PC with an Intel Core 2 Duo E6300 processor at 1.86 GHz and 2 GB of DDR2 RAM at 333 MHz with a Gigabyte 945GM-S2 motherboard and an Nvidia GeForce 9800 GTX graphics card with 768 MB of memory (x16 PCI-Express). The implementation used version 2.0 of the Nvidia CUDA driver, toolkit and compiler with the optimization settings provided in the CUDA template projects. The method also relies heavily on the performance of the FFT algorithm. For this comparison, I used the FFTW library for the CPU implementation and CUFFT, which is provided with the CUDA SDK, for the GPU implementation. Both FFT libraries are considered highly optimized for their particular platforms.

The experiment should have been run across a different set of platforms to better validate the results rather than just one. Unfortunately, I did not have these resources at the time this work was developed. However, since the method is strongly based on FFTs, the actual performance may be inferred by already available benchmarks of the FFT libraries in use.

For the non-recursive section of the filter, the implementation was tested with the following vector sizes for the convolution: 8,192, 16,384, 32,768, 65,536, 131,072, 262,144, 524,288, 1,048,576, 12,288, 24,576, 49,152, 98,304, 196,608, 393,216, 786,432 and 1,572,864. Note that only multiples of 2^{11} were chosen, since this is the largest block size available for this comparison. For the recursive section, only a few (*i.e.*, less than 128) coefficients were used. This was done for two reasons: using more than 16 coefficients in the recursive section is rare in practice; also the cost of using more coefficients is similar to the case of non-recursive filters.

For these experiments, the signal stream consisted of two channels of 32-bit floating-point samples at a sampling rate of 44.1 kHz. The test was repeated for block sizes of 128, 256, 512, 1,024 and 2,048. Table 7.1 presents the maximum filter sizes that could be processed in real-time in each configuration. The GPU was capable of handling filters with 2 to 4 times more coefficients than the CPU. However, the main practical advantage

Block Size	CPU Filter POT	CPU Filter NPOT	GPU Filter POT	GPU Filter NPOT
128	32,768	24,576	-	-
256	65,536	49,152	131,072	98,304
512	131,072	98,304	262,144	196,608
1,024	131,072	196,608	524,288	786,432
2,048	262,144	393,216	1,048,576	786,432

Table 7.1: Maximum number of coefficients achieved in real-time processing for the CPU-based and the GPU-based implementations, for power-of-two (POT) and non-power-of-two (NPOT) convolution sizes for the non-recursive section. For block size of 128 elements, the GPU implementation did not achieve real-time performance due to the overhead of CUDA kernel launch calls on the CPU side.

of using the proposed method is that a greater number of audio processes can be performed using only the GPU, thus avoiding unnecessary transfers for processing on the CPU.

7.5 Summary

This chapter presented an interactive application consisting of a graphics rendering process and an audio generation process. It was described how the graphics application provides collision and state events to the audio process using inter-process communication. It was also described how the recursive and non-recursive coefficients of the filters in use are obtained, and how the input signal is generated. Finally, a comparison of the proposed technique with a CPU-based implementation was presented. It revealed that the GPU implementation is capable of real-time execution of filters with 2 to 4 times as many as the equivalent implementation on the CPU.

8 CONCLUSION

It is clear that the GPU can perform real-time audio processing with no noticeable delays by humans. It is also clear that all non-recursive audio processes can be implemented on the GPU straightforwardly with performance benefits, whereas recursive processes require new approaches to achieve fast computation. For these reasons, in this thesis, I have presented a new technique that allows efficient implementation of recursive filters on GPUs. To the best of my knowledge, the proposed solution is the first presented in the literature. Compared to the equivalent CPU-based technique using the same FFT operations, this approach supports real-time processing of 2 to 4 \times more coefficients than it was previously possible, allowing 1D signal processing applications that require real-time filtering with a large number of non-recursive coefficients.

Since larger speedups were expected, I have concluded that FFT algorithms, which are frequently used for signal processing, benefit little of the GPU computing power, being restricted by memory bandwidth and access patterns. It still makes sense, however, to implement FFT-based processes on the GPU if they are combined with other processes that do explore the GPU computational power, such as mixing, synthesis and non-recursive filtering. The only exception to this, however, should be the case of a recursive filter placed at the end of a signal flow chain, in which case it could be implemented on the CPU and operate on the audio before being sent to the audio device for playback.

Using the FFT makes the method scale optimally for large recursive filters. However, this kind of filter is very uncommon (the order of the recursive section is almost never larger than 32 in typical audio applications). At such sizes, performing this filter on a GPU is often slower than on a CPU, given the complexity of the FFT algorithm and the fact that it is a memory-intensive operation instead of a simple sequential memory access (as recursive filtering running on the CPU is). However, the method still has advantages when used as a part of a larger signal processing system consisting of interconnected audio processes. This results from the fact that moving data from the GPU to the CPU for efficient processing and then back to the GPU is extremely slow due to large delays in memory transfers between the devices. Therefore, the technique has a major impact in systems where the signal processing stream involves both highly GPU-friendly processes (such as synthesis and non-recursive filtering) and recursive filtering.

The GPU hardware, in special its floating point implementation, is adequate for most non-recursive processes, but more numeric error may accumulate on recursive processing in comparison with CPU-based processing. Such errors may, in the case of linear recursive filters, actually turn a stable filter (by design) into an unstable one. It should be noted that recursive filters may suffer the same problems when implemented directly from the filter equation using the CPU. We expect some of these problems to be, at least, reduced if the filter is implemented using Larrabee, which will be IEEE-754 compliant. However,

this remains to be tested since Larrabee is still under development.

Among the applications that can benefit from the proposed method, I have implemented a demo using linear prediction coding (LPC) to produce audio for another process that generates 3D graphics and physically-based motion and collision detection. Both audio and graphics processes run in real-time and concurrently share both CPU and GPU resources. When a collision event takes place in the graphics process, the audio application re-synthesizes realistic sounds for the colliding objects based on their materials, speed, and collision angles. The re-synthesis process uses a set of coefficients specifying a recursive filter that describes the materials' acoustic properties. I have also shown how such coefficients can be automatically computed from recordings of the objects' actual sounds. Together, these techniques provide a more flexible way of using realistic sounds in interactive applications, and constitute an interesting alternative to the traditional sample playback approach. With the appropriate parameters, the described techniques can re-synthesize a variety of sounds from a reference one, greatly benefiting games and other interactive applications that require immediate auditory feedback. Such a procedural approach is also in accordance with the observed intention in computer games of replacing conventional 3D modeling with procedural one, in order to reduce the amount of data that needs to be downloaded from remote servers.

Other audio applications that may benefit from the technique include parametric equalization (which makes use of Q-filters, modeled as second order recursive filters), multi-band compression (where recursive filtering may be used for splitting audio in individual bands), vocoder (where a modified LPC-based method may be used to extract formants and reapplied, with a different residual, to generate new sounds), modal synthesis (where modes can be modeled by 2-pole filters) and subtractive synthesis (where IIR filters, specially low-pass, high-pass and band-pass, are applied directly to other synthesized signals to modify their spectra).

8.1 Future Work

To deal with the problems of filter instability described in Section 6.4.1, one could rewrite the GPU-based FFT algorithms to emulate double-precision floating points – which would further reduce performance on the GPU – or write a diverse set of GPU-based FFT algorithms and find one with sufficient precision. Though this should have some impact in stabilizing the filter, it is generally agreed that increasing the number of bits in the representation is not a good solution for problems of error accumulation.

A more general solution would be implementing the filter to run on the GPU (TREBIEN; OLIVEIRA, 2009), then compute the impulse response of the filter running on the GPU, compare the response with the provided filters specification, and then attempt to compensate for the errors by decimating the filter, moving individual poles, and recombining the individual filters. One could devise a method to do such using some type of metaheuristic optimization algorithms (HASEYAMA; MATSUURA, 2006).

To obtain higher speedups, a new implementation of the FFT could be used. The algorithm used by CUFFT has not been revealed by Nvidia, but I suppose it should be a typical method, such as the Cooley-Tukey algorithm. Several radix-8 algorithms in current use can, in theory, perform better for power-of-two vector sizes, but an implementation on the GPU would be necessary to verify this. Ultimately, memory access is the major limitation for good FFT performance. In fact, my findings highlight the fact that, while the GPU is powerful for parallel *computation*, it may not be as powerful for *memory-intensive*

parallel applications, even those designed with optimal memory access patterns in mind.

Most of the filter theory and of filter applications are based on time-invariant filters, as defined by Equation 4.3. In some cases, however, filter coefficients may vary for each instance of n , such as in

$$y_n = \sum_{i=0}^P b_i(n) x_{n-i} - \sum_{j=1}^Q a_j(n) y_{n-j} \quad (8.1)$$

In this case, the filter is said to be *time-varying*. The methods presented in Chapter 6 can not be easily extended to handle time-varying filters in general. Additionally, the methods cannot, at this time, be generalized for *nonlinear* filters so as to obtain an efficient implementation for parallel architectures. This is specially troublesome with nonlinear recursive filters. One goal would be producing an efficient and stable implementation for a second-order time-varying filter. Using existing techniques (see Section 4.6), such filter could then be used to implement any filter.

REFERENCES

ANSARI, M. **Video Image Processing Using Shaders. Presentation at Game Developers Conference.** 2003.

ANTONIOU, A. **Digital Filters: analysis and design.** T M H.ed. [S.l.]: Tata McGraw-Hill, 1980.

ASIO. **All You Need To Know About ASIO.** 2006. Available at: <http://www.soundblaster.com/resources/read.asp?articleid=53937&cat=2>. Last access: May 2009.

AVENHAUS, E.; SCHÜSSLER, H. On the Approximation Problem in the Design of Digital Filters with Limited Wordlength. **Archiv Elektronik Übertragungstechnik Electronics**, [S.l.], v.24, p.571–572, 1970.

BARCELLOS, A. M. **Desenvolvimento de um Motor de Física para o Console de Jogos Xbox 360™.** 2008. Graduation Thesis — UFRGS, Porto Alegre, Brazil.

BEGAULT, D. R. **3-D sound for virtual reality and multimedia.** San Diego, CA, USA: Academic Press Professional, Inc., 1994.

BENJAMIN, E. Preferred Listening Levels and Acceptance Windows for Dialog Reproduction in the Domestic Environment. In: JOURNAL OF THE AUDIO ENGINEERING SOCIETY, 2004. **Anais...** [S.l.: s.n.], 2004. v.117, n.6233.

BIERENS, L.; DEPRETTERE, E. Efficient Partitioning of Algorithms for Long Convolutions and their Mapping onto Architectures. **Journal of VLSI Signal Processing Systems**, Hingham, MA, USA, v.18, n.1, p.51–64, 1998.

BLECH, D.; YANG, M.-C. DVD-Audio versus SACD: perceptual discrimination of digital audio coding formats. In: AUDIO ENGINEERING SOCIETY, 2004. **Proceedings...** [S.l.: s.n.], 2004. v.116, n.6086.

BONNEEL, N.; DRETTAKIS, G.; TSINGOS, N.; VIAUD-DELMON, I.; JAMES, D. Fast modal sounds with scalable frequency-domain synthesis. In: SIGGRAPH '08: ACM SIGGRAPH 2008 PAPERS, 2008, New York, NY, USA. **Anais...** ACM, 2008. v.27, n.3, p.1–9.

CARSLAW, H. S. **Introduction to the Theory of Fourier's Series and Integrals.** 3rd.ed. New York, NY: Dover Publications, 1952.

- COOK, P. R. **Real Sound Synthesis for Interactive Applications**. 1st.ed. Wellesley, Massachussets: A K Peters, 2002.
- COOLEY, J. W.; LEWIS, P. A. W.; WELCH, P. D. The Fast Fourier Transform and Its Applications. **IEEE Transactions on Education**, [S.l.], v.12, n.1, p.27–34, Mar 1969.
- DELSARTE, P.; GENIN, Y. The Split Levinson Algorithm. In: IEEE TRANSACTIONS ON ACOUSTICS., SPEECH AND SIGNAL PROCESSING, 1986. **Anais...** [S.l.: s.n.], 1986. p.470–478.
- EARGLE, J. **The Microphone Book**. [S.l.]: Focal Press, 2001.
- EYRE, J.; BIER, J. **The evolution of DSP processors**. **IEEE Signal Processing Magazine**, vol. 17, n. 2, Mar. 2000, pp. 43-51 18. 2000.
- FRIGO, M.; JOHNSON, S. G. The Design and Implementation of FFTW3. **Proceedings of the IEEE**, [S.l.], v.93, n.2, p.216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".
- GALLO, E.; DRETTAKIS, G. **Breaking the 64 Spatialized Sources Barrier**. 2003.
- GALLO, E.; TSINGOS, N. **Efficient 3D Audio Processing with the GPU**. **Electronic Proceedings of the ACM Workshop on General Purpose Computing on Graphics Processors**, pp. C-42. 2004.
- GARCÍA, G. Optimal Filter Partition for Efficient Convolution with Short Input/Output Delay. In: AUDIO ENGINEERING SOCIETY, 2002. **Proceedings...** [S.l.: s.n.], 2002. n.113.
- GÖDDEKE, D.; STRZODKA, R.; ; TUREK, S. Accelerating Double Precision FEM Simulations with GPUs. **Proceedings of the 18th Symposium on Simulation Techniques (ASIM)**, [S.l.], Sep 2005.
- GOVINDARAJU, N. K.; MANOCHA, D. Cache-efficient numerical algorithms using graphics hardware. **Parallel Computing**, Amsterdam, The Netherlands, The Netherlands, v.33, n.10-11, p.663–684, 2007.
- GUMMER, A. W. (Ed.). **Biophysics of the Cochlea: from molecules to models**. Singapore: World Scientific, 2003.
- HARRINGTON, J.; CASSIDY, S. **Techniques in Speech Acoustics**. [S.l.]: Kluwer Academic Publishers, 1999. p.211–238.
- HARRIS, M. **Mapping Computaional Concepts to GPUs**. In **GPU Gems 2**, Matt Pharr (editor). [S.l.]: Addison Wesley, 2005. p.493–508.
- HASEYAMA, M.; MATSUURA, D. A Filter Coefficient Quantization Method with Genetic Algorithm, Including Simulated Annealing. **IEEE Signal Processing Letters**, [S.l.], v.13, n.4, p.189–192, April 2006.
- HAYKIN, S.; VEEN, B. V. **Signals and Systems**. New York, NY, USA: John Wiley & Sons, 1998.
- HOUSTON, M.; GOVINDARAJU, N. **GPGPU COURSE**. 2007.

IEEE. IEEE Standard for Floating-Point Arithmetic. **IEEE Std 754-2008**, [S.l.], p.1–58, 29 2008.

INTEL CORP. **Highly Optimized Mathematical Functions for the Intel® Itanium® Architecture**. 2004.

INTEL CORP. **Intel® Math Kernel Library Reference Manual**. Version 23.ed. [S.l.]: Intel Corporation, 2006.

ISO/IEC. **Normal Equal-Loudness-Level Contours**. Geneva, Switzerland: International Organization for Standardization, 2008. Published. (226:2003).

JACKSON, L. B. Analysis of limit cycles due to multiplication rounding in recursive digital filters. **Proceedings of the 7th Annual Allerton Conference on Circuits Systems Theory**, [S.l.], p.69–78, Oct 1969.

JACKSON, L. B. On the interaction of round-off noise and dynamic range in digital filters. **Bell System Technical Journal**, [S.l.], v.49, p.159–184, Feb 1970.

JARGSTORFF, F. **A Framework for Image Processing**. In **GPU Gems, Randima Fernando (editor)**. [S.l.]: Addison Wesley, 2004. p.445–467.

JEDRZEJEWSKI, M.; MARASEK, K. Computation of Room Acoustics Using Programmable Video Hardware. In: INTERNATIONAL CONFERENCE ON COMPUTER VISION AND GRAPHICS ICCVG'2004, 2004. **Anais...** [S.l.: s.n.], 2004.

JERRI, A. J. Correction to "The Shannon sampling theorem—Its various extensions and applications: a tutorial review". **Proceedings of the IEEE**, [S.l.], v.67, n.4, p.695–695, Apr 1979.

KAISER, J. F. Some Practical Considerations in the Realization of Linear Digital Filter. **Proceedings of the 3rd Allerton Conference on Circuit and System Theory**, [S.l.], 1965.

LAROCHE, J. On the Stability of Time-Varying Recursive Filters. **Journal of the Audio Engineering Society**, [S.l.], v.55, n.6, p.460–471, June 2007.

LIPSHITZ, S. P.; POCOCK, M.; VANDERKOOY, J. On the Audibility of Midrange Phase Distortion in Audio Systems. **Journal of the Audio Engineering Society**, [S.l.], v.30, n.9, p.580–595, Sep 1982.

LIU, B. Effect of finite word length on the accuracy of digital filters—a review. **IEEE Transactions on Circuit Theory**, [S.l.], v.18, n.6, p.670–677, Nov 1971.

MEYER, R. Error Analysis and Comparison of FFT Implementation Structures. **Acoustics, Speech, and Signal Processing, 1989. ICASSP-89., 1989 International Conference on**, [S.l.], p.888–891 vol.2, May 1989.

ROTTINO, M. P. (Ed.). **Elements of Computer Music**. Upper Saddle River, New Jersey, United States of America: Prentice Hall PTR, 1990.

MORELAND, K.; ANGEL, E. The FFT on a GPU. In: HWWS '03: PROCEEDINGS OF THE ACM SIGGRAPH/EUROGRAPHICS CONFERENCE ON GRAPHICS HARDWARE, 2003. **Anais...** Eurographics Association, 2003. p.112–119.

MORSE, P. M.; INGARD, K. U. **Theoretical Acoustics**. [S.l.]: Princeton University Press, 1986.

NELSON, D. A.; BILGER, R. C. Pure-Tone Octave Masking in Normal-Hearing Listeners. **Journal of Speech and Hearing Research**, [S.l.], v.17, n.2, p.223–251, 1974.

NICKOLLS, J.; BUCK, I.; GARLAND, M.; SKADRON, K. Scalable parallel programming with CUDA. **ACM Queue**, New York, NY, USA, v.6, n.2, p.40–53, 2008.

NVIDIA CORP. **NVIDIA CUDA Programming Guide 2.0**. Santa Clara, CA, USA: NVIDIA Corporation, 2008. Available for free as a part of the NVIDIA CUDA Toolkit 2.0.

NVIDIA CORP. **NVIDIA CUBLAS Library 2.0**. Santa Clara, CA, USA: NVIDIA Corporation, 2008. Available for free as a part of the NVIDIA CUDA Toolkit 2.0.

NVIDIA CORP. **NVIDIA CUFFT Library 2.0**. Santa Clara, CA, USA: NVIDIA Corporation, 2008. Available for free as a part of the NVIDIA CUDA Toolkit 2.0.

OPPENHEIM, A. Realization of digital filters using block-floating-point arithmetic. **IEEE Transactions on Audio and Electroacoustics**, [S.l.], v.18, n.2, p.130–136, Jun 1970.

OPPENHEIM, A. V.; SCHAFER, R. W. **Digital Signal Processing**. 1st.ed. Englewood Cliffs, NJ, USA: Prentice-Hall, 1975.

PARHI, K. K.; MESSERSCHMITT, D. G. Pipeline interleaving and parallelism in recursive digital filters. II. Pipelined incremental block filtering. **IEEE Transactions on Acoustics, Speech and Signal Processing**, [S.l.], v.37, n.7, p.1118–1134, Jul 1989.

PINKERT, J. R. An Exact Method for Finding the Roots of a Complex Polynomial. **ACM Transactions on Mathematical Software**, New York, NY, USA, v.2, n.4, p.351–363, 1976.

PRECHELT, L. **An Empirical Comparison of C, C++, Java, Perl, Python, REXX, and Tcl for a Search/String-Processing Program**. Karlsruhe, Germany: Universität Karlsruhe, 2000.

RABINER, L.; JUANG, B.-H. **Fundamentals of Speech Recognition**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.

RADER, C. M.; GOLD, B. Effects of parameter quantization on the poles of a digital filter. **Proceedings of the IEEE**, [S.l.], v.55, n.5, p.688–689, May 1967.

ROBELLY, J.; CICHON, G.; SEIDEL, H.; FETTWEIS, G. Implementation of recursive digital filters into vector SIMD DSP architectures. **IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)**, [S.l.], v.5, p.165–168, May 2004.

RUMSEY, F. (Ed.). **Sound Synthesis and Sampling**. Oxford, England: Focal Press, 1996.

SEILER, L.; CARMEAN, D.; SPRANGLE, E.; FORSYTH, T.; ABRASH, M.; DUBEY, P.; JUNKINS, S.; LAKE, A.; SUGERMAN, J.; CAVIN, R.; ESPASA, R.; GROCHOWSKI, E.; JUAN, T.; HANRAHAN, P. Larrabee: a many-core x86 architecture for visual computing. **ACM Transactions on Graphics**, New York, NY, USA, v.27, n.3, p.1–15, 2008.

SPITZER, J. **Implementing a GPU-Efficient FFT. NVIDIA course presentation at SIGGRAPH 2003.** 2003.

SUMANAWEEERA, T.; LIU, D. **Medical Image Reconstruction with the FFT. In GPU Gems 2, Matt Pharr (editor).** [S.l.]: Addison Wesley, 2005. p.765–784.

TREBIEN, F. **A GPU-Based Real-Time Modular Audio Processing System.** 2006. Graduation Thesis — UFRGS, Porto Alegre, Brazil.

ENGEL, W. (Ed.). **ShaderX6: advanced rendering techniques.** 1.ed. Hingham, MA, USA: Charles River Media, Inc., 2008. p.583–604.

TREBIEN, F.; OLIVEIRA, M. M. de. Realistic Real-Time Sound Re-Synthesis and Processing for Interactive Virtual Worlds. **The Visual Computer**, [S.l.], v.25, n.5–7, p.469–477, May 2009.

WEINSTEIN, C. Round-off noise in floating point fast Fourier transform computation. **IEEE Transactions on Audio and Electroacoustics**, [S.l.], v.17, n.3, p.209–215, Sep 1969.

WEINSTEIN, C.; OPPENHEIM, A. V. A comparison of round-off noise in floating point and fixed point digital filter realizations. **Proceedings of the IEEE**, [S.l.], v.57, n.6, p.1181–1183, June 1969.

WELCH, P. A fixed-point fast Fourier transform error analysis. **IEEE Transactions on Audio and Electroacoustics**, [S.l.], v.17, n.2, p.151–157, Jun 1969.

WHALEN, S. **Audio and the Graphics Processing Unit. Unpublished manuscript.** 2005.

WULICH, D.; PLOTKIN, E. I.; SWAMY, M. N. S. Stability of Second Order Time-Varying Constrained Notch Filters. **IEEE Electronics Letters**, [S.l.], v.26, n.18, p.1508–1509, Aug. 1990.

YEO, D. L. G.; WANG, Z.; ZHAO, B.; HE, Y. Low power implementation of FFT/IFFT processor for IEEE 802.11a wireless LAN transceiver. In: THE 8TH INTERNATIONAL CONFERENCE ON COMMUNICATION SYSTEMS, 2002, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2002. p.250–254.

ZHANG, Q.; YE, L.; PAN, Z. Physically-Based Sound Synthesis on GPUs. In: ICEC, 2005. **Anais...** Springer, 2005. p.328–333. (Lecture Notes in Computer Science, v.3711).

APPENDIX A SOURCE CODE LISTINGS

Listing A.1: Basic kernels in CUDA and corresponding calls.

```

1 // kernel for setting all elements of an array to a specific value
2 void __global__ kMemsset(float* buffer, unsigned long NumSamples, float val) {
3     const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
4     buffer[i] = val;
5 }
6
7 // kernel for producing a linear combination of two arrays
8 void __global__ kCombine(float* a, float A, float* b, float B, float* out, unsigned long NumSamples) {
9     const unsigned long i = blockIdx.x * blockDim.x + threadIdx.x;
10    if (i < NumSamples)
11        out[i] = A * a[i] + B * b[i];
12 }
13
14 // kernel for producing a linear combination of two arrays
15 void __global__ kLimiter(float* buffer, unsigned long NumSamples) {
16     const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
17     float x = exp(-buffer[i]);
18     buffer[i] = (1.0f - x) / (1.0f + x);
19 }
20
21 // kernel for multiplying corresponding elements of two arrays of complex numbers
22 void __global__ kComplexMul(cuFloatComplex* x, cuFloatComplex* y, cuFloatComplex* out, unsigned long NumPoints) {
23     const unsigned long i = blockIdx.x * blockDim.x + threadIdx.x;
24     if (i < NumPoints)
25         out[i] = cuCmulf(x[i], y[i]);
26 }
27
28 // function for computing the sinc function on the GPU
29 float __device__ sinc(float x) {
30     float t = 3.1415926535897932384626433832795f * x;
31     if (x == 0.0)
32         return 1.0f;
33     else
34         return sin(t) / t;
35 }
36
37 // kernel for producing and mixing a windowed sinc in one or two output vectors
38 void __global__ kSinc(float* BufferA, float* BufferB, unsigned long NumSamples, long StartSample, long EndSample,
39     float SamplingPeriod, float Amplitude, float Bandwidth) {
40     const unsigned long i = blockIdx.x * blockDim.x + threadIdx.x;
41     float n = (i - 0.5f * (StartSample + EndSample - 1));
42     float w = 0.5f + 0.5f * cos(6.28318530717958647692f * n / (float)(EndSample - StartSample));
43     float y = Amplitude * sinc(Bandwidth * n) * w;
44     if (i < NumSamples) {
45         BufferA[i] += y;
46         if (BufferB != 0)
47             BufferB[i] += y;
48     }
49 }
50
51 // exported C functions for calling the kernels
52 extern "C" void cudaSetValue(float* buffer, float val, unsigned long NumSamples) {
53     kMemsset<<<(NumSamples+255)/256, 256>>>(buffer, NumSamples, val);
54 }
55
56 extern "C" void cudaCombine(float* a, float A, float* b, float B, float* out, unsigned long NumSamples) {
57     kCombine<<<(NumSamples+255)/256, 256>>>(a, A, b, B, out, NumSamples);
58 }
59
60 extern "C" void cudaLimiter(float* Buffer, unsigned long NumSamples) {
61     kLimiter<<<(NumSamples+255)/256, 256>>>(Buffer, NumSamples);
62 }
63
64 extern "C" void cudaComplexMul(cuFloatComplex* x, cuFloatComplex* y, cuFloatComplex* out, unsigned long NumPoints) {
65     kComplexMul<<<(NumPoints+255)/256, 256>>>(x, y, out, NumPoints);
66 }
67
68 extern "C" void cudaSinc(float* BufferA, float* BufferB, unsigned long NumSamples, long StartSample, long EndSample,
69     float SamplingPeriod, float Amplitude, float Bandwidth) {
70     kSinc<<<(NumSamples+255)/256, 256>>>(BufferA, BufferB, NumSamples, StartSample, EndSample, SamplingPeriod,
71     Amplitude, Bandwidth);
72 }

```

Listing A.2: Generating blocks of audio using CUDA.

```

1  float NextGranule[16];
2
3  MyEventMap Events;
4  wxMutex MapGuard;
5
6  void OnCollision(bool WithSurface, float Intensity, float Time) {
7      MapGuard.Lock();
8      Events[IDCounter] = new MyEventCUDA(WithSurface, Intensity, 0);
9      IDCounter++;
10     MapGuard.Unlock();
11 }
12
13 float PoissonInterval(float Speed) {
14     float lambda = 2.0f * Speed / 44100.0f;
15     float uniform = (rand() + 1) / 32768.0f;
16     float exp = (-1.0f / lambda) * log(uniform);
17     return exp;
18 }
19
20 float MyInterval(float Speed) {
21     return 44100.0f / Speed + 0.2f * PoissonInterval(Speed);
22 }
23
24 void GenerateWhiteNoiseTail(float *b, unsigned long P) {
25     for (unsigned long i = 0; i < P; i++) {
26         float t = (float)i / SamplingRate;
27         b[i] = (2.0f * ((float)rand() / 32768.0f - 0.5f))
28             * (1.0f - expf(-512.0f * t))
29             * (expf(-16.0f * t))
30             * (1.0f - i/P) / 128.0f;
31     }
32 }
33
34 void MyProducerCUDA::ComputeOutput(void) {
35     float *pChannel = &mApplicationBuffer[mNumInputs
36         * mBufferLayout.SamplesPerChannel];
37
38     cudaSetValue(mpBufferA, 0, mBufferLayout.SamplesPerChannel);
39     cudaSetValue(mpBufferB, 0, mBufferLayout.SamplesPerChannel);
40
41     MapGuard.Lock();
42     for (int i = 0; i < 16; i++) {
43         bool IsSliding;
44         float Speed;
45         AsyncReadSphereState(i, IsSliding, Speed);
46         if (IsSliding) {
47             if (NextGranule[i] == -1)
48                 NextGranule[i] = MyInterval(Speed);
49             while (NextGranule[i] < mBufferLayout.SamplesPerChannel) {
50                 Events[IDCounter] = new MyEventCUDA(true, Speed * 0.00005f, NextGranule[i]);
51                 IDCounter++;
52                 if (Speed != 0.0f)
53                     NextGranule[i] += MyInterval(Speed);
54             }
55             NextGranule[i] -= mBufferLayout.SamplesPerChannel;
56         }
57         else
58             NextGranule[i] = -1;
59     }
60
61     MyEventMap::iterator Item = Events.begin(), Deleteltem;
62     while (Item != Events.end()) {
63         bool Finish = Item->second->ComputeSignal(mpBufferA, mpBufferB, mBufferLayout.SamplesPerChannel);
64         if (Finish) {
65             Deleteltem = Item;
66             ++Item;
67             Events.erase(Deleteltem);
68         }
69         else
70             ++Item;
71     }
72     MapGuard.Unlock();
73
74     // if not loaded, load the filters
75     if (mpFilterA == NULL) {
76         // yields 1024*64 samples after padding when loading the filter
77         unsigned long NumNonRecCoeffs = 1024*64 - 2*mBufferLayout.SamplesPerChannel;
78         float *b = malloc(NumNonRecCoeffs * sizeof(float));
79         GenerateWhiteNoiseTail(b);
80         mpFilterA = new FilterCUDA(mBufferLayout.SamplesPerChannel, b, NumNonRecCoeffs, a1, NumRecCoeffs);
81     }
82     if (mpFilterB == NULL) {
83         unsigned long NumNonRecCoeffs = 1024*64 - 2*mBufferLayout.SamplesPerChannel;
84         float *b = malloc(NumNonRecCoeffs * sizeof(float));
85         GenerateWhiteNoiseTail(b);
86         mpFilterB = new FilterCUDA(mBufferLayout.SamplesPerChannel, b, NumNonRecCoeffs, a2, NumRecCoeffs);
87     }
88     mpFilterA->GenerateNextBuffer(mpBufferA, mpBufferA);
89     mpFilterB->GenerateNextBuffer(mpBufferB, mpBufferB);
90     cudaCombine(mpBufferA, 1.0f, mpBufferB, 0.10f, pChannel, mBufferLayout.SamplesPerChannel);
91     cudaLimiter(pChannel, mBufferLayout.SamplesPerChannel);
92
93     // copy computed signal on left channel to the right channel
94     cudaMemcpy(pChannel + mBufferLayout.SamplesPerChannel, pChannel,
95         mBufferLayout.SamplesPerChannel * sizeof(float), cudaMemcpyDeviceToDevice);
96 }

```

Listing A.3: Kernel calls for loading and computing the filter using CUDA.

```

1 #define GREATERMULTIPLE(x, per) (((x) + (per) - 1) / (per)) * (per)
2
3 FilterCUDA::FilterCUDA(unsigned long SamplesPerBuffer,
4 float *b, unsigned long P, float *a, unsigned long Q) {
5     mSamplesPerBuffer = SamplesPerBuffer;
6     mInputPosition = 0;
7     mOutputPosition = 0;
8
9     // allocate filter structures
10    mInputSize = mSamplesPerBuffer + GREATERMULTIPLE(mSamplesPerBuffer+P-1, mSamplesPerBuffer);
11    mOutputSize = mSamplesPerBuffer + GREATERMULTIPLE(Q-1, mSamplesPerBuffer);
12
13    A = (float*)malloc(Q * sizeof(float));
14    Bpad = (float*)malloc(mInputSize * sizeof(float));
15    Cpad = (float*)malloc(mInputSize * sizeof(float));
16    Dpad = (float*)malloc(mOutputSize * sizeof(float));
17
18    cudaMalloc((void**)&Xshift, mInputSize * sizeof(float));
19    cudaMalloc((void**)&Yshift, mOutputSize * sizeof(float));
20    cudaMalloc((void**)&Xconv, mInputSize * sizeof(float));
21    cudaMalloc((void**)&Yconv, mOutputSize * sizeof(float));
22
23    cudaMalloc((void**)&Bdft, (mInputSize/2+1) * sizeof(cuFloatComplex));
24    cudaMalloc((void**)&Cdft, (mInputSize/2+1) * sizeof(cuFloatComplex));
25    cudaMalloc((void**)&Ddft, (mOutputSize/2+1) * sizeof(cuFloatComplex));
26    cudaMalloc((void**)&BCdft, (mInputSize/2+1) * sizeof(cuFloatComplex));
27    cudaMalloc((void**)&Xdft, (mInputSize/2+1) * sizeof(cuFloatComplex));
28    cudaMalloc((void**)&Ydft, (mOutputSize/2+1) * sizeof(cuFloatComplex));
29
30    // plan the FFTs (may destroy the contents of the vectors)
31    cufftPlan1d(&PlanB, mInputSize, CUFFT_R2C, 1);
32    cufftPlan1d(&PlanC, mInputSize, CUFFT_R2C, 1);
33    cufftPlan1d(&PlanD, mOutputSize, CUFFT_R2C, 1);
34    cufftPlan1d(&PlanX, mInputSize, CUFFT_R2C, 1);
35    cufftPlan1d(&PlanY, mOutputSize, CUFFT_R2C, 1);
36    cufftPlan1d(&PlanXf, mInputSize, CUFFT_C2R, 1);
37    cufftPlan1d(&PlanYf, mOutputSize, CUFFT_C2R, 1);
38
39    // acquire coefficients
40    for (int i = 0; i < Q; i++)
41        A[i] = a[i];
42    for (int i = 0; i < P; i++)
43        Bpad[i] = b[i];
44
45    // normalize A
46    for (unsigned i = 1; i < Q; i++)
47        A[i] /= A[0];
48
49    // add padding to B
50    memset(&Bpad[P], 0, (mInputSize - P) * sizeof(float));
51
52    // precompute C and D and add padding
53    Dpad[0] = 1.0f;
54    memset(&Dpad[1], 0, (mOutputSize-1) * sizeof(float));
55    for (unsigned long i = 0; i < mSamplesPerBuffer; i++) {
56        Cpad[i] = Dpad[0];
57        for (unsigned long j = 1; j < Q; j++)
58            Dpad[j-1] = Dpad[j] - Cpad[i] * A[j];
59    }
60    memset(&Cpad[mSamplesPerBuffer], 0, (mInputSize - mSamplesPerBuffer) * sizeof(float));
61
62    // compute the FFT of B and C
63    cudaMemcpy(Bdft, Bpad, mInputSize * sizeof(float), cudaMemcpyHostToDevice);
64    cudaMemcpy(Cdft, Cpad, mInputSize * sizeof(float), cudaMemcpyHostToDevice);
65    cudaMemcpy(Ddft, Dpad, mOutputSize * sizeof(float), cudaMemcpyHostToDevice);
66
67    cufftExecR2C(PlanB, (cufftReal*)Bdft, Bdft);
68    cufftExecR2C(PlanC, (cufftReal*)Cdft, Cdft);
69    cufftExecR2C(PlanD, (cufftReal*)Ddft, Ddft);
70
71    // convolve B and C
72    cudaComplexMul(Bdft, Cdft, BCdft, (mInputSize/2+1));
73
74    // clear X and Y to begin processing
75    cudaSetValue(Xshift, 0, mInputSize);
76    cudaSetValue(Yshift, 0, mOutputSize);
77 }
78
79 void FilterCUDA::GenerateNextBuffer(float *Input, float *Output) {
80     // store input
81     cudaMemcpy(Xshift + mInputPosition, Input, mSamplesPerBuffer * sizeof(float), cudaMemcpyDeviceToDevice);
82
83     // perform convolutions
84     cufftExecR2C(PlanX, Xshift, Xdft);
85     cufftExecR2C(PlanY, Yshift, Ydft);
86
87     cudaComplexMul(BCdft, Xdft, Xdft, (mInputSize/2+1));
88     cudaComplexMul(Ddft, Ydft, Ydft, (mOutputSize/2+1));
89
90     cufftExecC2R(PlanXf, Xdft, Xconv);
91     cufftExecC2R(PlanYf, Ydft, Yconv);
92
93     // extract result
94     cudaCombine(&Xconv[mInputPosition], 1.0f / (float)mInputSize,
95               &Yconv[mOutputPosition], 1.0f / (float)mOutputSize, Output, mSamplesPerBuffer);
96
97     // increment pointers

```

```

98     mInputPosition += mSamplesPerBuffer;
99     if (mInputPosition == mInputSize)
100         mInputPosition = 0;
101     mOutputPosition += mSamplesPerBuffer;
102     if (mOutputPosition == mOutputSize)
103         mOutputPosition = 0;
104
105     // store output in the buffer for next step
106     cudaMemcpy(Yshift + mOutputPosition, Output, mSamplesPerBuffer * sizeof(float), cudaMemcpyDeviceToDevice);
107 }
108
109 FilterCUDA::~FilterCUDA(void) {
110     // destroy plans
111     cufftDestroy(PlanYf); cufftDestroy(PlanXf); cufftDestroy(PlanY); cufftDestroy(PlanX);
112     cufftDestroy(PlanD); cufftDestroy(PlanC); cufftDestroy(PlanB);
113
114     // destroy vectors
115     cudaFree(Ydft); cudaFree(Xdft); cudaFree(BCdft); cudaFree(Ddft); cudaFree(Cdft); cudaFree(Bdft);
116     cudaFree(Yconv); cudaFree(Xconv); cudaFree(Yshift); cudaFree(Xshift);
117     free(Dpad); free(Cpad); free(Bpad); free(A);
118 }

```

Listing A.4: Kernel calls for generating sinc pulses using CUDA.

```

1  class MyEventCUDA {
2  protected:
3      long mStartSample, mEventEnd;
4      float mAmplitude, mBandwidth, mSamplingPeriod;
5      unsigned long mTaps;
6      unsigned long mStartSample;
7      bool mWithSurface;
8
9  public:
10     MyEventCUDA(bool WithSurface, float Intensity, unsigned long StartSample, unsigned long Taps = 31) {
11         mSamplingPeriod = 1.0f / 44100.0f;
12         mWithSurface = WithSurface;
13         mAmplitude = Intensity;
14         mBandwidth = Intensity * 20.0;
15         mStartSample = StartSample;
16         mTaps = Taps;
17
18         mStage = 0;
19         mEventEnd = mStartSample + mTaps;
20     }
21
22     bool ComputeSignal(float *BufferA, float *BufferB, unsigned long NumSamples) {
23         unsigned long NumSamplesToGenerate;
24         switch (mStage) {
25             case 0: // skip samples at start
26                 if (mStartSample >= NumSamples)
27                     break;
28                 else {
29                     BufferA += mStartSample;
30                     BufferB += mStartSample;
31                     NumSamples -= mStartSample;
32                     mEventEnd -= mStartSample;
33                     mStartSample = 0;
34                     mStage = 1;
35                 }
36
37             case 1: // pulse body
38                 if (mEventEnd >= NumSamples)
39                     NumSamplesToGenerate = NumSamples;
40                 else
41                     NumSamplesToGenerate = mEventEnd;
42                 if (NumSamplesToGenerate != 0) {
43                     // call the kernel
44                     if (mWithSurface)
45                         cudaSinc(BufferA, BufferB, NumSamplesToGenerate, mStartSample, mEventEnd, mSamplingPeriod,
46                             mAmplitude * 0.5f, mBandwidth);
47                     else
48                         cudaSinc(BufferA, 0, NumSamplesToGenerate, mStartSample, mEventEnd, mSamplingPeriod, mAmplitude,
49                             mBandwidth);
50                 }
51
52                 if (mEventEnd >= NumSamples)
53                     break;
54                 else
55                     return true;
56             }
57
58         mStartSample -= NumSamples;
59         mEventEnd -= NumSamples;
60         return false;
61     }
62 };

```

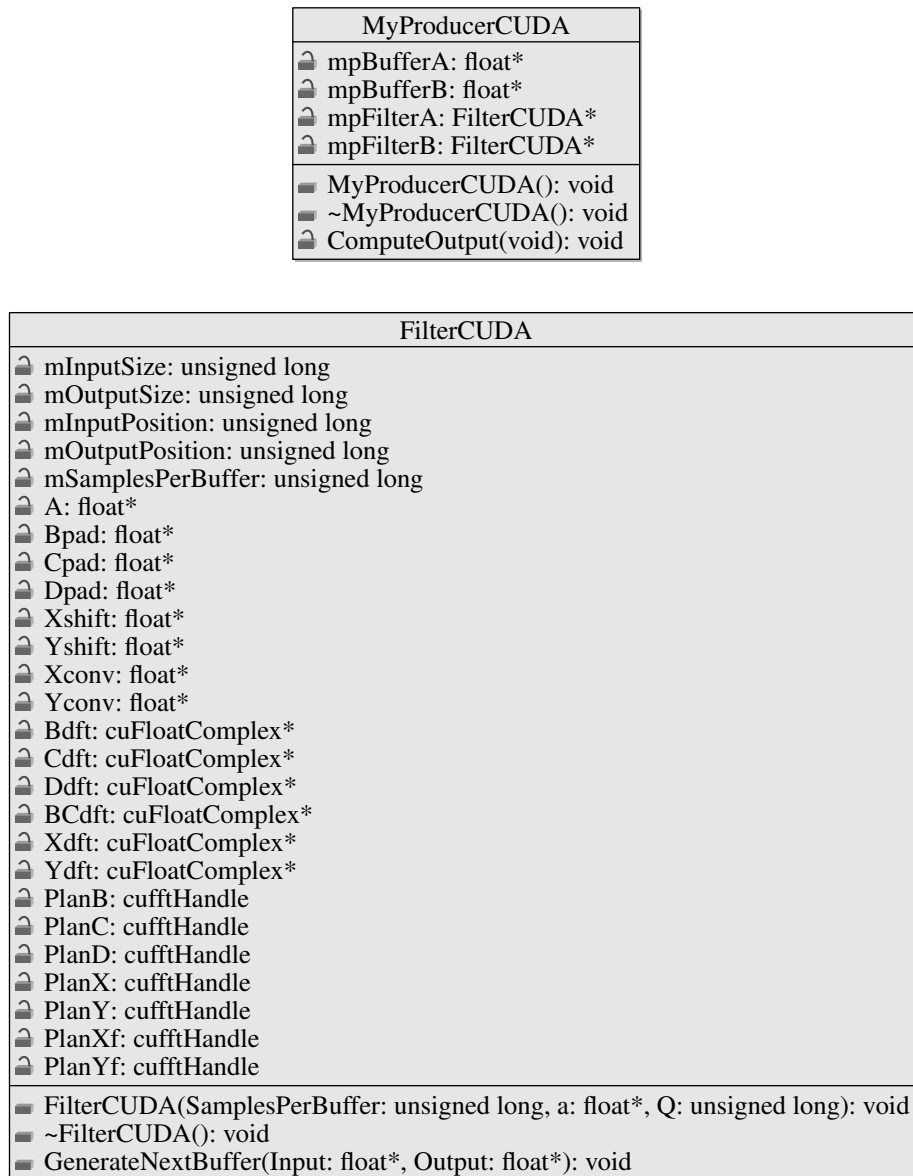


Figure A.1: Class diagram for the implementation of the proposed method.

APPENDIX B UMA IMPLEMENTAÇÃO EFICIENTE BASEADA EM GPUS DE FILTROS LINEARES RECURSIVOS E SUA APLICAÇÃO A RE-SÍNTESE REALÍSTICA EM TEMPO-REAL PARA MUNDOS VIRTUAIS INTERATIVOS

Resumo da Dissertação em Português

Som é parte indispensável da experiência multimídia, transmitindo tanto informação quanto emoção. Avanços tecnológicos nas últimas décadas permitem hoje que um músico efetue todas as etapas de gravações de qualidade industrial usando apenas um computador portátil e poucos equipamentos adicionais. Porém, com uma configuração tão simples, é comum esbarrar em certos limites, especialmente a inflexível não-programabilidade da maioria dos processadores digitais de sinal (DSPs) e o poder computacional limitado das CPUs atuais, que em muitos casos impede o processamento do som em *tempo-real* (e.g., para efeitos sonoros em jogos, na execução musical de um tecladista). Tais limitações são especialmente significativas em muitas aplicações profissionais de áudio, que procuram produzir sons ricos e complexos, tais como *reverberação* por *convolução* (GARCÍA, 2002), *reamostragem* (RUSS, 1996), ou, mais frequentemente, uma combinação das duas em um sistema modular de processamento de sinais.

Por outro lado, processadores gráficos (GPUs) apresentam capacidades computacionais teóricas muito superiores às das CPUs (NICKOLLS et al., 2008; NVIDIA CORP, 2008a), o que levou ao desenvolvimento de tecnologias orientadas ao *processamento de propósitos gerais na GPU* (GPGPU), tais como CUDA e CTM (HOUSTON; GOVINDARAJU, 2007). Usando tais tecnologias, e se explorada corretamente, as GPUs devem poder computar um volume de dados de áudio maior do que muitas CPUs. Apesar de bem explorada para processamento de sinais 2D e 3D (ANSARI, 2003; JARGSTORFF, 2004; MORELAND; ANGEL, 2003; SPITZER, 2003; SUMANAWEEERA; LIU, 2005), e de haverem bibliotecas matemáticas disponíveis tais como CUBLAS e CUFFT (NVIDIA CORP, 2008b,c), pouco foi explorado quanto ao processamento de sinais 1D na GPU. Por exemplo, exceto pela técnica apresentada neste trabalho, não há uma solução geral para filtros lineares recursivos na GPU, exceto pela avaliação em paralelo de um número massivo de instâncias (ZHANG; YE; PAN, 2005), o que é trivial. Este fato tem limitado o uso da GPU para áudio profissional, apesar das evidências de que a GPU possa ser usada para processamento de áudio em tempo-real (TREBIEN; OLIVEIRA, 2008), podendo alcançar um desempenho até 20× maior que o da CPU (TREBIEN, 2006).

Assim, para preencher esta lacuna, apresento uma técnica (TREBIEN; OLIVEIRA, 2009) que permite a implementação eficiente de filtros 1D lineares e invariantes no tempo

em GPUs. O método consiste em desdobrar a equação do filtro até que todas as dependências de dados se refiram a amostras disponíveis. A equação resultante pode ser expressa por duas convoluções, que podem ser computadas eficientemente usando a FFT. Comparada ao estado-da-arte, esta abordagem permite processar filtros com 2 a 4× mais coeficientes em tempo-real. O método é demonstrado em uma aplicação onde sons realísticos são gerados para objetos de diferentes materiais sofrendo colisões.

B.1 Trabalhos Relacionados

Na primeira categoria de trabalhos relacionados, encontram-se artigos da área de processamento de sinais que definem importantes critérios para avaliar a qualidade do método apresentado. Entre eles, Liu (1971) discutiu os problemas de precisão em implementações de filtros lineares digitais usando representações de ponto-fixa e ponto-flutuante. Rader e Gold (1967) discutiram uma implementação alternativa de filtros recursivos de primeira e segunda ordem com erros significativamente reduzidos na posição dos pólos e zeros. Weinstein e Oppenheim (1969) verificaram experimentalmente um modelo para o ruído introduzido em um sinal filtrado como resultado da quantização aritmética. Welch (1969) estabeleceu limites superiores e inferiores ao erro espectral do algoritmo da FFT usando aritmética de ponto-fixa, enquanto Weinstein (1969) estudou a precisão da FFT mas usando aritmética de ponto-flutuante. Por fim, Parhi e Messerschmitt (1989) desenvolveram uma estrutura em bloco para implementação em hardware de filtros recursivos, mas a técnica é dada em forma de pipeline, que contém dependências recursivas e portanto não pode ser implementada eficientemente em máquinas SIMD como a GPU.

A segunda categoria de trabalhos relacionados consiste de aplicações de áudio e processamento de sinais na GPU. Primeiramente, há adaptações do algoritmo da FFT para GPUs (ANSARI, 2003; MORELAND; ANGEL, 2003; SPITZER, 2003; SUMANAWERA; LIU, 2005), algumas das quais já estão disponíveis em bibliotecas (GOVINDARAJU; MANOCHA, 2007). Gallo e Drettakis (2003) apresentaram um método para espacialização baseada em agrupamento de fontes sonoras no espaço 3D e subsequente reamostragem e mixagem na GPU. Gallo e Tsingos (2004) apresentaram uma aplicação similar incluindo simulação de efeito Doppler e filtragem por funções de transferência relativas à cabeça (HRTFs), mas os autores relatam que obtiveram um desempenho 20% menor na GPU em relação à CPU. Robelly *et al.* (2004) apresentaram uma formulação teórica para computação de filtros recursivos invariantes no tempo em arquiteturas paralelas, mas não validam o método com uma implementação. Whalen (2005) avaliou a GPU comparando o desempenho das implementações de um conjunto de processos de áudio simples não-recursivos e sem operação em tempo real, encontrando depenhos até 4× maiores para alguns dos algoritmos avaliados. Zhang *et al.* (2005) usaram GPUs para sonificação de colisões usando *síntese modal* na GPU, computando os modos individualmente, mas não chegaram a aplicar o método a um mundo virtual interativo. Trebien e Oliveira (2008) propuseram um sistema de tempo-real capaz de realizar operações básicas com áudio e processar ondas primitivas e reamostradas. Por fim, Bonneel *et al.* (2008) desenvolveram uma técnica para produzir sons de colisões eficientemente através de síntese modal no domínio frequência, mas, apesar da aplicação à realidade virtual e do desempenho de 5 a 8× maior comparado a técnicas similares, o som foi produzido usando a CPU.

B.2 Avaliação Eficiente de Filtros Lineares Recursivos na GPU

Filtros não-recursivos (caso linear na Equação 4.3a) e processos sem memória, tais como mixagem (Equação 4.2), possuem implementação trivial na GPU que consiste em simplesmente avaliar a equação diretamente. Porém, em um filtro recursivo (caso linear na Equação 4.3), a amostra de saída y_n depende do valor de amostras anteriores (*e.g.*, y_{n-1}, y_{n-2}, \dots). A implementação trivial desse filtro em uma GPU consistiria em forçar a serialização usando sincronização, mas isto não utilizaria todos os processadores paralelos da GPU e certamente teria desempenho inferior à CPU.

Para filtros lineares recursivos e invariantes no tempo, uma possível solução é obtida desdobrando-se a equação do filtro. Se n for substituído por $n - 1$ na Equação 4.3b, obtém-se uma expressão para y_{n-1} e o termo pode ser substituído na equação original. Com a substituição, a dependência do valor de saída em y_{n-1} é eliminada, e outra é acrescentada em y_{n-Q-1} . Repetindo-se este processo para y_{n-2} , depois y_{n-3} e assim por diante, e rearranjando os termos em cada uma delas, observa-se que todas formam o padrão expresso na Equação 6.1, onde os coeficientes c_k e d_j podem ser calculados iterativamente a partir dos coeficientes que definem o filtro, b_i e a_j , usando-se as Equações 6.2 e 6.3. Como o processamento de áudio em tempo real é feito em blocos de m amostras, basta então iterar esse processo até que o termo y_{n-m} tenha sido substituído, de modo que todas as dependências se estabeleçam com amostras de blocos anteriores, as quais podem ser guardadas em memória.

Para uma implementação eficiente, a equação desdobrada pode ser expressa por meio de convoluções de acordo com a definição na Equação 4.5, resultando na Equação 6.5. As convoluções podem ser então calculadas em domínio-frequência de acordo com a propriedade na Equação 4.11, resultando na Equação 6.6. O cálculo de convoluções no domínio-frequência é chamado de *convolução rápida*, com um passo ilustrado na Figura 6.1 e existem métodos conhecidos para implementá-lo (HAYKIN; VEEN, 1998). Neste trabalho, foi usado para convolução rápida o algoritmo de *overlap-save* (OPPENHEIM; SCHAFER, 1975).

Para uma implementação ótima, pode-se armazenar os blocos anteriores necessários em um buffer circular e aproveitar a circularidade da FFT para minimizar o número de transferências de memória. A organização e operação dos buffers para esta implementação no processamento de 1 bloco é ilustrada na Figura 6.2. Ainda, as FFTs de CB e D são pré-computadas, e as FFTs de X e \bar{Y} são computadas na execução em tempo-real.

B.2.1 Extração LPC e Re-Síntese

Para demonstrar o filtro em operação, é necessário obter coeficientes para ele. Um dos métodos mais úteis para isto é a *codificação por predição linear* (LPC) (HARRINGTON; CASSIDY, 1999), o qual obtém um conjunto de coeficientes recursivos (*i.e.*, a_1, a_2 , etc.) a partir de uma gravação. A extração dos coeficientes também obtém um sinal denominado *resíduo*, o qual, se alimentado no filtro com os coeficientes obtidos por este método, recupera exatamente a gravação original. Na prática, o filtro com os coeficientes extraídos modela as ressonâncias da gravação. O resíduo tende a ser uma onda simples, que pode ser substituída por um sinal similar (geralmente uma onda primitiva como um pulso, um trem de pulsos ou ruído branco com algum envelope (COOK, 2002, p. 48–49)). Tal sinal, quando sintetizado e alimentado no filtro, produz na saída um sinal que soa como a gravação original.

Filtros LPC não são garantidamente estáveis. É comum a produção de pólos na fun-

ção de transferência do filtro próximos ao círculo unitário $e^{j\omega}$ no domínio z . Tais pólos, quando implementados em uma máquina com aritmética finita, podem ocasionar realimentação do erro aritmético em certas frequências, tornando o filtro instável. Uma forma de contornar esse problema é reduzir o número de pólos (e, por consequência, coeficientes) do filtro. O mesmo vale para a implementação baseada em FFT aqui apresentada. Diferentes algoritmos de FFT apresentam erros numéricos diferentes (MEYER, 1989), e as bibliotecas de FFT usadas não permitem mudar o algoritmo selecionado pela biblioteca para uma configuração particular (NVIDIA CORP, 2008c). Aparentemente, a biblioteca FFTW3, usada na implementação em CPU, parece sofrer de um erro numérico menor (FRIGO; JOHNSON, 2005).

Neste trabalho, eu escolhi uma gravação particular que produzia um filtro instável na execução com tamanho de bloco 256, porém estável com tamanho de bloco 512. Note que a instabilidade não é o caso mais comum. A Figura 6.3(a) mostra o diagrama de pólos e zeros do filtro obtido por LPC, o qual é, em teoria, estável com precisão infinita. As Figuras 6.3(b) e 6.3(c) mostram a resposta ao impulso do filtro com 256 e 512 amostras por bloco, respectivamente, demonstrando que uma implementação é estável, enquanto que a outra não é. A Figura 6.3(d) mostra a diferença ponto-a-ponto do sinal de saída dos dois casos. Para comparar o conteúdo em frequência dessas duas saídas, eu dividi o sinal de saída em quadros de 256 amostras cada um e apliquei a cada um uma janela de Hanning. As Figuras 6.4(a) e 6.4(b) mostram o resultado da análise do quadro 0, respectivamente, mostrando que a saída é essencialmente a mesma, o que permanece válido até o quadro 5 (Figuras 6.4(c) e 6.4(d)), onde os modos instáveis passam a dominar o sinal. As Figuras 6.4(e) e 6.4(f) mostram o conteúdo do quadro 20.

B.3 Validação da Técnica Proposta

Para demonstrar a implementação do filtro, foi construída uma aplicação consistindo de dois processos separados e concorrentes. Primeiramente, um processo gráfico simples (BARCELLOS, 2008) em C# produz simulação física e colisões entre um conjunto de esferas e uma superfície plana (Figura 7.1), podendo-se aplicar diferentes materiais às esferas (Figura 7.2).

A cada colisão, o processo gráfico envia mensagens a um processo de áudio que implementa o método descrito, que sintetiza o som para o evento. As mensagens contêm informações como a energia liberada em cada impacto e os corpos envolvidos na colisão. Uma área de memória compartilhada é usada para informar, com pouca sobrecarga, quando uma das esferas está rolando sobre a superfície plana. Nesse caso, a velocidade do rolamento é atualizada pelo processo gráfico e lida assincronamente pelo processo de áudio. Um vídeo mostrando a execução da aplicação gravado em tempo-real encontra-se em http://www.inf.ufrgs.br/Audio_on_GPU/en/media/.

Para geração do áudio, um filtro deve ser extraído para cada tipo de som (colisão entre duas esferas de mesmo material e colisão entre esfera e superfície, implicando a utilização de dois filtros), e uma aproximação do resíduo computada para cada evento. Para as esferas, foram feitas gravações em um ambiente pouco controlado, usando objetos de vidro, plástico e madeira, suspensos por barbantes. Um cobertor foi usado para reduzir (sem eliminar) o ruído ambiente. A configuração pode ser vista na Figura 7.3. O som da superfície foi obtido atingindo uma mesa de madeira com um dedo. Uma gravação em um ambiente mais controlado deve melhorar a qualidade dos sons obtidos, mas para garantir uma qualidade mínima, as gravações foram repetidas algumas vezes e a melhor

amostra foi selecionada. Os coeficientes recursivos são extraídos pelo método de LPC. Por inspeção, observei que o resíduo poderia ser aproximado pela convolução entre o pulso *sinc* na Equação 7.3, que passa a ser o sinal de entrada, e o ruído branco amortecido na Equação 7.1, que define os coeficientes não-recursivos do filtro. Um resumo das tarefas envolvidas no pré-processamento e na execução do filtro encontra-se na Figura 7.4. Os parâmetros do pulso são modulados de acordo com a intensidade da colisão (*i.e.*, energia liberada, fornecida via mensagens inter-processo). Quando uma esfera está rolando sobre uma superfície, um trem de pulsos quase periódico é gerado de acordo com uma variável de Poisson. Deve-se notar que, para um realismo completo, outros indicadores sonoros deveriam ser incorporados ao método, tais como atenuação por distância e reverberação ambiente.

Finalmente, a implementação de LPC foi feita em MATLAB e os filtros foram implementados em C++ e CUDA com a biblioteca CUFFT, que acompanha o SDK de CUDA e considerada otimizada para a plataforma. Um diagrama de classes da aplicação encontra-se na Figura A.1. O código-fonte da implementação encontra-se nas Listagens A.1, A.2 e A.3, e da geração de eventos *sinc*, na Listagem A.4.

Para comparar o desempenho da GPU em relação à CPU na implementação do filtro, foi utilizada a biblioteca de baixa latência Steinberg ASIO (ASIO, 2006) com uma modificação para detectar a extrapolação do limite de tempo de cômputo de um bloco (*i.e.*, execução não-tempo-real). Em ambas as aplicações, o código executando na CPU foi escrito em C++. Os testes foram rodados sem o processo gráfico em um processador Intel Core 2 Duo E6300 a 1.86 GHz e com 2 GB de RAM DDR2 a 333 MHz com uma placa-mãe Gigabyte 945GM-S2 e em uma placa gráfica Nvidia GeForce 9800 GTX com 768 MB de memória em um barramento x16 PCI-Express. A versão 2.0 do driver, toolkit e compilador de CUDA com configurações padrões foi empregada. Como o desempenho depende fortemente do algoritmo de FFT, foi escolhida a biblioteca estado-da-arte FFTW3 para a implementação em CPU. A seção não-recursiva do filtro assumiu, durante o experimento, os seguintes tamanhos para convolução, todos múltiplos de 2^{11} : 8.192, 16.384, 32.768, 65.536, 131.072, 262.144, 524.288, 1.048.576, 12.288, 24.576, 49.152, 98.304, 196.608, 393.216, 786.432 e 1.572.864. O sinal consistiu de dois canais de amostras em ponto-flutuante de 32-bits a uma taxa de amostragem de 44,1 kHz. O teste foi repetido para tamanhos de bloco de 128, 256, 512, 1,024 e 2,048, e a Tabela 7.1 mostra os tamanhos máximos de filtro processados em tempo-real para cada configuração.

A GPU conseguiu operar com filtros de 2 a $4\times$ coeficientes do que a CPU. Deve-se notar que as vantagens do método são significativas somente em combinação com outros processos de áudio, evitando transferências de memória desnecessárias entre CPU e GPU.

B.4 Conclusões

É fato que a GPU pode realizar processamento de áudio em tempo real com benefícios de desempenho. Segundo meu conhecimento, a técnica proposta neste trabalho é a primeira na literatura, permitindo a execução de filtros recursivos com tamanhos de 2 a $4\times$ maiores que na CPU. O ganho de desempenho é pequeno se comparado com o esperado devido aos trabalhos anteriores, e a razão mais provável é a eficiência da implementação da FFT, que é limitada fortemente pela velocidade de acesso à memória. Outra limitação é que raramente filtros recursivos têm um grande volume de coeficientes, e filtros pequenos têm implementação sequencial mais rápida na CPU. Porém, o tempo de transferência de memória entre CPU e GPU é grande o bastante para justificar o uso de filtros recursivos

na GPU em combinação com outros processos de áudio mais eficientes. A implementação também não garante a estabilidade do filtro, assim como muitas implementações em CPU. O filtro se comportou de forma menos estável que na CPU, provavelmente pela falta de adesão dos fabricantes ao padrão de ponto-flutuante IEEE-754, com menos erro numérico. Apesar disso, o filtro pode ser usado em grande parte dos casos e em contextos diferentes do apresentado, tais como equalização paramétrica, compressão multi-banda, vocoders, síntese modal e síntese subtrativa.

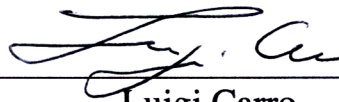
Como trabalho futuro, o problema da estabilidade pode ser resolvido aumentando-se a precisão ou experimentando-se com diferentes algoritmos da FFT. Outra possibilidade seria alterar a resposta ao impulso para que a implementação final tivesse o resultado pretendido, talvez por otimização heurística (HASEYAMA; MATSUURA, 2006). O desempenho poderia ser melhorado mudando-se a arquitetura (*i.e.*, melhorando a velocidade de acesso à memória) ou aplicando-se algoritmos de FFT mais eficientes, tais como os de raiz-8. Ainda resta encontrar uma solução similar à proposta para filtros não-lineares e para filtros variáveis no tempo, tais como o da Equação 8.1.

**“An Efficient GPU-based Implementation of Recursive Linear Filters and
Its Application to Realistic Real-Time Re-Synthesis for Interactive
Virtual Worlds”**

por

FERNANDO TREBIEN

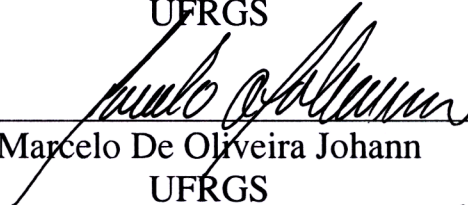
Dissertação Apresentada aos Senhores:



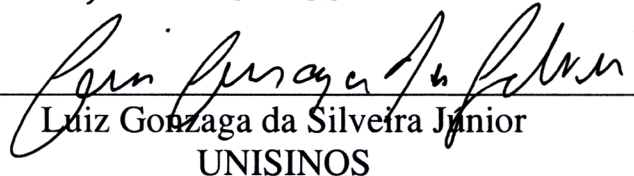
Luigi Carro
UFRGS



João Luiz Dihl Comba
UFRGS



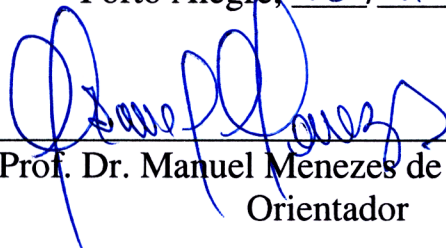
Marcelo De Oliveira Johann
UFRGS



Luiz Gonzaga da Silveira Junior
UNISINOS

Vista e permitida a impressão.

Porto Alegre, 18 / 11 / 2009



Prof. Dr. Manuel Menezes de Oliveira Neto
Orientador