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

Fernando Trebien, Manuel Menezes de Oliveira Neto (orientador)

Programa de Pós-Graduação em Computação (PPGC)
Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{ftrebien,oliveira}@inf.ufrgs.br

Abstract. *In this work, I present a new technique for implementing* digital recursive linear filters *using GPUs supporting real-time processing with 2 to 4$\times$ more coefficients than achieved by an equivalent implementation on CPUs, eliminating the need of CPU-based processing and improving performance by avoiding memory transfers. It consists of unrolling the filter equation and "trading" dependences until an expression containing available samples is obtained. Resulting convolutions are then computed using the FFT. For demonstration, an LPC filter is designed to synthesize sound for a scene parametrically (by object material and collision velocity and angle) and computed using the technique.*

1. Introduction

This work summarizes a Master's thesis [Trebien 2009] (check References for the download URL), and is also associated with one scientific article [Trebien and Oliveira 2009].

From computer games and other interactive graphics applications to mobile phones to the recording studio, there has always been demand for more realistic and pleasing sound effects. Some of these processes present high computational requirements (*e.g.*, reverberation through convolution), and some provide higher sound quality when more processing power is available (*e.g.*, auralization). Also, multiple processes are often combined to build a soundstage. Since many applications also require processing audio streams in real-time (*i.e.*, responding to user inputs), CPU computational power often limits the number and types of processes that can be combined, ultimately forcing the use of lower-quality sound processing methods.

On the other hand, recent GPUs have presented peak throughput capabilities that far exceed those of CPUs. Although 2D and 3D signal processing have been well explored on the GPU [Sumanaweera and Liu 2005], very little been studied on GPU-based 1D signal processing (*e.g.,* no general solution for linear recursive filters on GPUs has been published yet). The lack of such methods has precluded GPUs from being used for serious sound processing, which could greatly benefit graphics applications.

I present a new technique that allows efficient implementations of recursive 1D filters on GPUs, which can be used to re-synthesize and process 1D signals in real time. Its effectiveness and relevance to computer graphics is 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 material properties (thus having a small memory footprint). A method to obtain these coefficients is described. Given its flexible and general nature, this approach replaces with some advantages, although not entirely, the traditional CPU-based techniques that perform playback of pre-recorded sounds.

2. Related Work

Gallo and Tsingos [Gallo and Tsingos 2004] reported an application where 3D sound sources are clustered relative to the listener and mixed using the GPU (also with Doppler shifting and HRTFS filtering), finding that their GPU implementation was 20% slower than their CPU implementation. Jedrzejewski and Marasek [Jedrzejewski and Marasek 2004] used the GPU for computing impulse responses via ray-tracing, with no signal processing on the GPU. Robelly et al. [Robelly et al. 2004] presented a mathematical formulation for computing time-invariant recursive filters on parallel architectures, with high speedups when both filter order and number of parallel processors are high. Trebien and Oliveira [Trebien and Oliveira 2008] mapped audio concepts to graphics concepts and used a graphics API to perform basic audio operations and synthesize basic waveforms, achieving significant speedups. Zhang et al. [Zhang et al. 2005] have used GPUs for *modal synthesis* by synthesizing modes separately and mixing the intermediate results on the GPU. Bonneel et al. [Bonneel et al. 2008] developed a technique for efficient modal synthesis in frequency domain, adequate for sounds composed of narrowband modes, and achieved speedups of 5 to $8\times$ compared to a time-domain solution. Finally, several works present adaptations of the FFT algorithm for GPUs [Sumanaweera and Liu 2005], and FFT implementations are already available in software libraries for GPUs [Govindaraju and Manocha 2007].

3. Recursive Filter Realization for GPUs

A time-invariant causal digital linear filter with a single input signal is defined by

$$w_n = \sum_{i=0}^{P} b_i x_{n-i} \qquad (1a) \qquad\qquad y_n = w_n - \sum_{j=1}^{Q} a_j y_{n-j} \qquad (1b)$$

where b_i are feed-forward coefficients, a_j are feedback coefficients, P is the feed-forward filter order, and Q is the feedback filter order. If all a_j are null, the filter output depends only on its input and is therefore called *non-recursive*. In this case, any input sample affects the value of at most P output samples, and for that reason, non-recursive linear filters are formally called *finite impulse response* (FIR) filters. Otherwise, if any a_j is non-null, a recursion is established on the output signal, propagating the effect of any input sample indefinitely. Because of that, recursive linear filters are called *infinite impulse response* (IIR) filters. The coefficients a_j and b_i are usually constant for each instance of processing, in which case the filter is said to be *time-invariant*. For simplicity, the sets of b_i's and a_j's are referred to as vectors A and B, respectively, so that $A = \begin{bmatrix} 1 & a_1 & \dots & a_Q \end{bmatrix}^T$ and $B = \begin{bmatrix} b_0 & b_1 & \dots & b_P \end{bmatrix}^T$.

3.1. Eliminating Data Dependences

Equation 1a can be implemented with existing methods [García 2002]. To avoid the need for synchronization, Equation 1b is unrolled until all necessary output samples are available from the computation of preceding buffers. Unrolling it one step yields

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

which is an expression for y_n in terms of w_n, w_{n-1} and $y_{n-2}, y_{n-3}, \ldots, y_{n-Q-1}$. By repeating this process, one observes that, at the m-th step, a dependence on y_{n-m} is replaced with a dependence on w_{n-m} and another on $y_{n-m-Q+1}$:

$$y_n = \sum_{k=0}^{m-1} c_k w_{n-k} + \sum_{j=0}^{Q-1} d_j y_{n-m-j} \tag{2}$$

$$\begin{aligned} C &= \begin{bmatrix} c_0 & c_1 & \ldots & c_{m-1} \end{bmatrix}^T \\ D^{(m)} &= \begin{bmatrix} d_0 & d_1 & \ldots & d_{Q-1} \end{bmatrix}^T \end{aligned}$$

$D^{(m)}$ and C are computed after m iterative steps from

$$\begin{aligned} D^{(0)} &= \begin{bmatrix} 1 & 0 & \ldots & 0 \end{bmatrix}^T \\ D^{(k+1)} &= D^{(k)}[0]\, A' + S D^{(k)} \\ A' &= \begin{bmatrix} -a_1 & -a_2 & \ldots & -a_Q \end{bmatrix}^T \\ C &= \begin{bmatrix} D^{(0)}[0] & D^{(1)}[0] & \ldots & D^{(m-1)}[0] \end{bmatrix}^T \end{aligned}$$

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

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

Note that $SD^{(k)}$ represents a positional right shift on vector $D^{(k)}$, so it can be optimized. Since all samples up to y_{n-m} are now available, one can define an intermediate value $\overline{y}_k = y_{k-m}$ if $k \leq n$ and $\overline{y}_k = 0$ otherwise. Since Equation 1a can be expressed as $w_n = (b * x)_n$ (*i.e.*, $w = b * x$), Equation 2 can then be expressed in terms of $\overline{y}_n$ as

$$y_n = (c * b * x)_n + (d * \overline{y})_n \tag{3}$$

where $*$ represents the convolution operator, c and d represent signals respectively equal to vectors C and $D^{(m)}$ in every defined position and null otherwise. Since c and d only depend on the a_j coefficients, $c * b$ and d can be precomputed. x is obtained by accessing the current and previous input buffers, and $\overline{y}$ is obtained from previous output buffers.

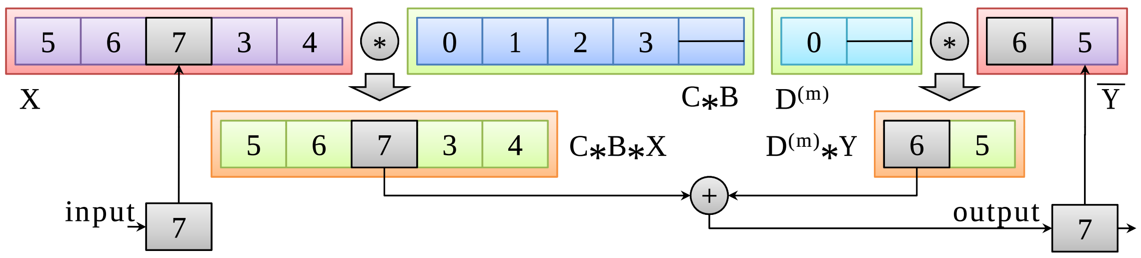


Figure 1. Summary of operations for processing an input audio block.

The DFT can be used to compute a convolution between two discrete signals, or, for efficiency, an FFT, but care must be taken when using FFTs for fast convolution since they assume that the inputs are periodic signals, which is not the case. There are several known ways to implement fast convolution using FFTs [Haykin and Veen 1998]. For this work, I have implemented *overlap-save*, which adds padding (*i.e.,* null samples) to the input vectors to obtain a partial output of the acyclic convolution.

To apply overlap-save to Equation 3, let r represent the number of samples in a block of audio, CB a vector containing the linear convolution of c and b with at least r null samples at its end, and D representing d with at least r null samples at the end. The FFTs $\Im(CB)$ and $\Im(D)$ of CB and D, respectively, are precomputed. During the actual processing, the FFTs $\Im(X)$ and $\Im(\overline{Y})$ of X and $\overline{\text{Y}}$ are computed (those do not have padding), then multiplied by the spectra of CB and D, respectively. The output is obtained by applying the IFFT to obtain time-domain signals, then adding the two:

$$Y_n = (\, \Im^{-1}(\Im(CB) \odot \Im(X)) \; + \; \Im^{-1}(\Im(D) \odot \Im(\overline{Y})) \,)_n$$

The symbol $\odot$ represents the element-wise multiplication and $\Im^{-1}$ is the IFFT. All vectors X, Y, CB and D are maintained in memory as circular buffers of blocks of audio. The output block replaces one position inside the circular buffer containing part of Y. Figure 1 illustrates this process in time domain.

4. A Demonstration Scenario

The algorithms described in the previous sections are implemented using CUDA and the CUFFT library. To demonstrate the relevance of this approach to real-time graphics applications, a simple application was created to simulate the collisions among a set of spheres and a piecewise planar surface. 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 is modelled as made of wood and the spheres as either glass, wood or plastic. In the demo application, all spheres have the same material, but one could modify this sound generating process to accommodate a greater number of materials simultaneously. Figure 2 shows two screenshots of the actual application.

As objects collide, the application sends messages to the audio synthesis process, which then synthesizes the appropriate sound for each event. The messages contain information such as energy lost in impacts and type of contacting bodies.

In order to generate realistic sounds using filtering instead of sample playback, one filter for each type of material must be designed. This can be obtained by computing

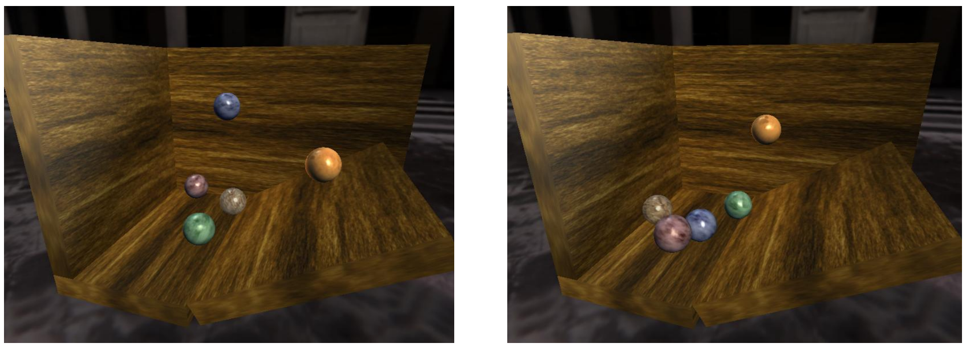


Figure 2. Snapshots of the demonstration scene.

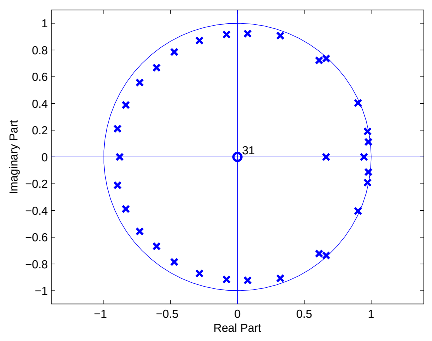
the filter coefficients from actual sound of the material. Such a filtering-based approach has advantages over sample playback since by changing a few parameters, one can re-synthesize variations of the original sound. For instance, the sound of impacts between objects at different speeds and contact angles can be simulated.

4.1. Computing Filter Coefficients from Actual Sounds

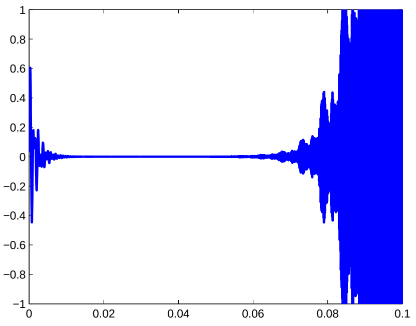
One of the most useful methods for computing the required filter coefficients is re-synthesis through *linear predictive coding* (LPC) [Harrington and Cassidy 1999]. LPC designs a filter that predicts the output from part of the input by minimizing the error ϵ in $x_n = \sum_{j=1}^{Q} a_j x_{n-j} + \epsilon_n$. Note that this is actually a recursive linear filter where x is the output and ϵ is the input. The original signal can be exactly re-synthesized given the filter coefficients a_j and the error signal ϵ (a *residual* signal), which is usually simple enough to be replaced by an approximate primitive waveform (*e.g.,* a pulse, a pulse train or white noise with envelope). While perfect re-synthesis is not achieved, the resulting signal sounds similar to the original because the filter models resonances of the sound source. This method not only saves memory but also allows the generation of several similar sounds by synthesizing slightly different residuals.

For the demo application, coefficients are precomputed for a number of sounds representing the strike between two spheres and the strike with a wooden surface. I recorded waveforms for each material in a loosely controlled environment (without noise dampening, for instance) but with acceptable results, since physical accuracy is not a requirement. By inspection, I noticed that residuals 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, suggesting that this combination would suffice. To synthesize the residual, a granule of a Hann-windowed $sinc$ function is generated for each sound event (*i.e.,* collision). The resulting signal is then convolved (by the filter) with another signal B consisting of unfiltered exponentially-decaying white noise. Convolution with a windowed $sinc$ is a form of low-pass filtering, simulating the attenuation in high frequencies observed in actual sounds of less intense collisions.

The filters that the demo application executes to re-synthesize collision sounds are formed by the non-recursive coefficients from B and the recursive coefficients A extracted by LPC. 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 a sphere-on-plane collision, the pulse is fed



(a) Pole-zero plot for the filter. Many poles lie close to the unit circle, suggesting that a digital implementation may be unstable.



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

Figure 3. Behavior of LPC-designed filter with unstable GPU implementation.

to both filters, but for sphere-on-sphere, it is fed only to SS. The amplitude parameter α and the bandwidth parameter β are proportional to impact intensity.

For every sphere that is rolling over a surface, a quasi-periodic train of pulses is generated with amplitude and bandwidth proportional to the sphere's speed. The interval between each pulse has a periodic component and a random Poisson component.

5. Filter Stability

Stability is not guaranteed for filters designed with LPC. Generally, increasing the number of coefficients generates poles in the filter's transfer function that are inside but very close to the unit circle $e^{\imath\omega}$ 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 LPC extraction with fewer coefficients. Figure 3(a) shows the pole-zero plot of a filter designed with LPC, and Figure 3(b) shows the impulse response of the filter when running with 256 samples per block, demonstrating that this particular implementation is not stable. The same filter running with 512 samples per block is stable.

This problem happens particularly due to the use of FFTs, whose main source of error is the implementation of floating point operations, which 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 in CUFFT, which does not specify the algorithms it uses or their error bounds. An equivalent CPU-based implementation using FFTW3 appears to be less subject to this problem.

A simple heuristic can be given to test for stability: run an *impulse signal* through the filter, record the output, and analyze it using the STFT in search of decaying and increasing modes. All frequencies are expected to be decaying, so if a number n of frames contain less energy in all frequencies than the first frame, the filter is highly likely to be stable. Greater values of n provide greater accuracy.

6. Results

The LPC implementation for computing the filter coefficients from the sounds of the materials (Section 4.1) is written in MATLAB. The physically-based simulation depicted in Figure 2 is written in C# as an independent application and integrated with the CUDA implementation using inter-process communication. The sound generating process creates a shared memory block to hold a vector with information about each sphere, and a message queue for collisions. The use of shared memory represents a very small overhead, allowing both processes to run in real-time with no perceptible delays. The sound generating process reads this information once per input block and updates the pulse-train generation parameters. 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/`.

To compare the performance of the GPU approach against a CPU implementation, a low overhead modification was added to the procedure called by the low-latency audio interface (Steinberg ASIO). This callback receives and provides buffers to the audio device and with this modification additionally checks if a buffer is available for playback. If this condition is sustained for consecutive calls (at least 10 seconds of audio), the system signals that the real-time test has passed. This method is then used to compare an implementation in C++ with the CUDA implementation running most of its computation on the GPU. Since the technique relies heavily on the performance of the FFT algorithm, we used the FFTW and CUFFT, both considered highly optimized, for the CPU and the GPU implementations, respectively. Performance was measured by increasing the size of vectors involved in convolutions until real-time operation could not be sustained anymore. 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 different buffer sizes (which affect the number of convolutions being performed) and revealed an increase of a maximum 2 to $4\times$ more coefficients when running the GPU-based implementation.

7. Conclusion

A new technique for efficient implementations of 1D digital filters on GPUs is presented which may enable new 1D signal processing applications that require real-time filtering with a large number of coefficients. This solution is the first presented in the literature.

The relevance of these new GPU algorithms for computer graphics is demonstrated using a real-time 3D application performing physically-based collision detection. At a collision event, the application re-synthesizes realistic sounds for the colliding objects based on materials, speed, and collision angles. The re-synthesis process uses coefficients (computed from recordings of actual sounds) specifying a recursive filter that describes the materials' acoustic properties. 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 appropriate parameters, these techniques can re-synthesize a large variety of realistic sounds from a reference one, greatly benefiting games and interactive applications that require immediate auditory feedback. Other audio applications that may benefit from recursive filtering include equalization, multi-band compression, vocoder, modal synthesis and subtractive synthesis.

Most of the filter theory and of filter applications are based on time-invariant filters. The presented technique, however, cannot be easily extended for time-varying filters (where coefficients are replaced by functions of n) and for *non-linear* filters. This is especially troublesome with non-linear recursive filters, and finding alternative solutions to this problem is an interesting challenge for future exploration.

References

Bonneel, N., Drettakis, G., Tsingos, N., Viaud-Delmon, I., and James, D. (2008). Fast modal sounds with scalable frequency-domain synthesis. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, volume 27, pages 1–9, New York, NY, USA. ACM.

Gallo, E. and Tsingos, N. (2004). Efficient 3d audio processing with the gpu. electronic proceedings of the acm workshop on general purpose computing on graphics processors, pp. c-42.

García, G. (2002). Optimal filter partition for efficient convolution with short input/output delay. In *Proceedings of the Audio Engineering Society*, number 113.

Govindaraju, N. K. and Manocha, D. (2007). Cache-efficient numerical algorithms using graphics hardware. *Parallel Computing*, 33(10-11):663–684.

Harrington, J. and Cassidy, S. (1999). *Techniques in Speech Acoustics*, chapter 8, pages 211–238. Kluwer Academic Publishers.

Haykin, S. and Veen, B. V. (1998). *Signals and Systems*. John Wiley & Sons, New York, NY, USA.

Jedrzejewski, M. and Marasek, K. (2004). Computation of room acoustics using programmable video hardware. In *International Conference on Computer Vision and Graphics ICCVG'2004*.

Meyer, R. (1989). Error analysis and comparison of fft implementation structures. *Acoustics, Speech, and Signal Processing, 1989. ICASSP-89., 1989 International Conference on*, pages 888–891 vol.2.

Robelly, J., Cichon, G., Seidel, H., and Fettweis, G. (2004). Implementation of recursive digital filters into vector simd dsp architectures. *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 5:165–168.

Sumanaweera, T. and Liu, D. (2005). *Medical Image Reconstruction with the FFT. In GPU Gems 2, Matt Pharr (editor)*, pages 765–784. Addison Wesley.

Trebien, F. (2009). An efficient GPU-based implementation of recursive linear filters and its application to realistic real-time re-synthesis for interactive virtual worlds. Masters thesis, UFRGS, Porto Alegre, Brazil. `http://www.inf.ufrgs.br/~ftrebien/monograph.pdf`.

Trebien, F. and Oliveira, M. M. (2008). *ShaderX6: Advanced Rendering Techniques*, chapter Real-Time Audio Processing on the GPU, pages 583–604. Charles River Media, Inc., Hingham, MA, USA, first edition.

Trebien, F. and Oliveira, M. M. (2009). Realistic real-time sound re-synthesis and processing for interactive virtualworlds. *The Visual Computer*, 25(5–7):469–477.

Zhang, Q., Ye, L., and Pan, Z. (2005). Physically-based sound synthesis on gpus. In Kishino, F., Kitamura, Y., Kato, H., and Nagata, N., editors, *ICEC*, volume 3711 of *Lecture Notes in Computer Science*, pages 328–333. Springer.