Multichannel Massive Audio Processing for a Generalized Crosstalk Cancellation and Equalization application using GPUs

Jose A. Belloch[†], Alberto Gonzalez[†], F. J. Martínez-Zaldívar[†⋆], Antonio M. Vidal[‡]

[†]Instituto de Telecomunicaciones y Aplicaciones Multimedia, 8G Building
[‡]Departamento de Sistemas Informáticos y Computación, 1F Building
Universitat Politècnica de València
Camino de Vera s/n, Valencia, 46022 Spain,
[⋆]Corresponding author e-mail: fjmartin@dcom.upv.es
Tel: +34 963877302, Fax: +34 963877309

September 20, 2012

Abstract

Multichannel acoustic signal processing has undergone major development in recent years due to the increased complexity of current audio processing applications, which involves the processing of multiple sources, channels, or filters. A general scenario that appears in this context is the immersive reproduction of binaural audio without the use of headphones, which requires the use of a crosstalk canceler. However, Generalized Crosstalk Cancellation and Equalization (GCCE) requires high computing capacity, which is a considerable limitation for real-time applications. This paper discusses the design and implementation of all the processing blocks of a multichannel convolution on a GPU for real-time applications. To this end, a very efficient filtering method using specific data structures is proposed, which takes advantage of overlap-save filtering and filter fragmentation. It has been shown that, for a real-time application with 22 inputs and 64 outputs, the system is capable of managing 1408 filters of 2048 coefficients with a latency time less than 6 ms. The proposed GPU implementation can be easily adapted to any acoustic environment, demonstrating the validity of these co-processors for managing intensive multichannel audio applications.

1 Introduction

The growing need to incorporate new effects and to improve the hearing experience [37] has increased the development of multichannel sound applications. People want to collaborate through communication with the feeling of being together and sharing the same environment. Communication environments of this kind are considered to be Immersive Audio Schemes [19]. This phenomenon comes from the mix of several acoustic effects: 3D spatial sound [28], crosstalk cancellation, room compensation [33], loudspeaker equalization, etc.

Until now, most of these effects could be achieved only in theaters or funfairs, which usually use very powerful computers and consume a large amount of energy. Masoumi, in [23], goes so far as to propose a new mathematical model of crosstalk to reduce computational costs. The use of GPUs makes it possible to achieve such effects, sometimes even faster, while saving CPU resources. Moreover, the combination of personal computers and GPUs have the potential to replace dedicated powerful computers.

Multichannel sound signal processing is mainly based on combining the output signals resulting from convolution operations in such a way that a given special acoustic effect is achieved.

We define C_{in} and C_{out} as the total number of sources and loudspeakers, respectively. Fig. 1 contains all the operations carried out between channels and filters of a generalized multichannel reproduction system where x_j is the j-th source, y_i is the i-th loudspeaker signal, and the filter implemented between them has an impulse response given by h_{ij}, with $j = 0, \ldots, C_{\text{in}} - 1$ and $i = 0, \ldots, C_{\text{out}} - 1$. All operations of the multichannel reproduction system are reflected in (1), where $*$ denotes the convolution operation.

$$y_i = \sum_{j=0}^{C_{\text{in}}-1} (h_{ij} * x_j). \tag{1}$$

Moreover, parameter C_{tot} represents the number of filters involved in the application. As there is a filtering path from every source to every loudspeaker, the number of filters implemented is $C_{\text{tot}} = C_{\text{in}} \cdot C_{\text{out}}$.

1.1 Generalized Crosstalk Cancellation and Equalization (GCCE)

One application that is especially important in the context of multichannel acoustic signal processing is the reproduction of binaural audio without the use of headphones. GCCE plays an important role in this phenomenon by inverting the transmission paths between loudspeakers and listeners. Assuming a reproduction scenario with Q listeners, each listener would receive contributions from every loudspeaker at both ears. The aim of the application is to create a pair of desired signals that are not disturbed by these contributions at the ears of the listeners. Fig. 2 shows the placement of $2 \cdot Q$ desired signals, one signal per ear (represented by d_{zR} and d_{zL}, $z \in [0, Q-1]$, L=Left ear and R=Right ear) in a room. An application example would be a scenario where there are several people watching a movie in the same room and each of them is capable of listening to the audio in a different language without the use of headphones. The block *Crosstalk Canceler and Equalizer* is a filter bank with the same structure as the block shown in Fig. 1. Signal y_i is reproduced through the loudspeaker i, which belongs to an array of loudspeakers. This signal is the sum of C_{in} convolutions that have to be carried out in real time.

1.2 State of the art and Objectives

The use of GPU [7] has always been related to graphic or image applications since it offers the possibility of carrying out the same operations over multiple data, such as applications of object tracking [26], image sequences analysis [11], crowd simulation [42], or even a new approach for automatic human skin segmentation [22].

All the audio convolutions and sums shown until now can be performed independently. In this sense, if GPU is used as a co-processor that carries out audio processing tasks, CPU resources can be released and used in other tasks [18]. Obviously, this could only happen if there are free GPU resources and no graphic intensive application is overusing the GPU.

GPU computing has already been applied to different problems in acoustics and audio processing. Studies of computing room acoustics were carried out by Webb and Bilbao in [44] and [43], as well as geometric acoustic modelling like ray-tracing [20] [29]. Two inmersive-audio technologies: wave field synthesis (WFS) [6] and beamforming (BF) [41] have also been implemented on GPU in [35], where a comparison among different platforms of these applications can be found. Detailed GPU-implementations of WFS and BF are reported in [34] and [24], respectively. Spatial sound through head-related transfer functions (HRTFs) has also made use of GPUs. The first studies date from 2004 by [14]. In [31], the GPU is used for delay, gain, air absorption, and HRTFs filtering in real-time auralization. A Comparison of HRTFs performance between CPU and GPU can be seen in [9] and [8]. An overview of more applications is enumerated by Tsingos in [39] and [40].

Focusing on the convolution operation, there are some publications in the literature in which convolution on GPU is involved. In 2005, Smirnov and Chiueh presented an early investigation of FIR filtering on a GPU [32]. Development of recursive filters in a GPU was presented by Trebien and Oliveira [38]. Cowan and Kapralos implemented a convolution algorithm on GPU [10] using the OpenGL shading language [2]. Moreover, the study of [30] reveals that at a buffer size of 1024 samples, the maximum length for a single channel FFT on a GPU was around 4 million samples. The convolution algorithm shown in [45] has the feature of reducing the latency of the system by subdividing the filters into several subfilters. Their GPU-implementation is able to convolve 352 channels in a time of 10.53 ms.

First approaches to our real-time convolution algorithm were carried out in [5]. In that article, it was presented a GPU implementation that executes multiple convolutions concurrently. That work had a lot of limitations regarding the latency times and number of channels that a multichannel application would require. Also, that implementation on the GPU did not use the CUDA resources efficiently, since there were many accesses to the GPU *global-memory* instead of using GPU *shared-memory* [1]. It offered a good approach to multiple convolutions but it did not fit with a GCCE application where not only is it necessary to execute multiples convolutions but also it is required combining the convolution results. Furthermore, the implementation only focused on the possibility of using pre-defined sizes of data and of filters, without considering other common scenarios in which the filters sizes are larger than the data sizes.

On the other hand, filters sizes larger than data sizes were neither considered in [3]. In that article, it was implemented on GPU a crosstalk application, which is a particular case of a GCCE application with two input channels and two output channels with fixed sizes of both filters and data. The purpose of the article was to validate the use of the GPU as a co-processor that frees up CPU resources.

A similar work was showed in [4]. The peculiarity of that article was that the filters changed in real-time. It was described a spatial audio application that interacts with the user who was able to change the location of the sound.

The implementation of the multichannel convolution is a new step in order to develop audio applications that requiere to combine the result of multiple real-time convolutions. A thorough analysis regarding the temporal characteristics, the number of sources and the number of loudspeakers had not been performed so far using a GPU accelerator in a real environment. This kind of analysis is mandatory in order to asses the performance of the GPU in a real-time audio applications.

The main contribution of this paper consists of configuring a complete study that extrapolates the implementation of multiple convolutions to a GCCE application on a GPU attending to different and common situations: the size of data buffers that are much larger than the size of filters and the size of data buffers that are much smaller than the size of filters.

The paper is organized as follows: Section 2 is devoted to tackling audio real-time applications on GPU, offering a brief description of the architectural characteristics of the GPU. Section 3 describes suitable data structures for efficient convolution on GPU. Section 4 advances one step further and extrapolates the convolution algorithm presented in Section 3 to GCCE applications. Section 5 shows the performance of the practical developed systems. Finally, some concluding remarks are presented in Section 6.

2 Real-Time Application on GPU

Dealing with real-time audio applications on GPU requires a basic understanding of the GPU programming features. This section provides a basic description of the GPU data flow and some relevant issues that must be taken into account when programming a real-time application.

2.1 Graphics Processing Units

Compute Unified Device Architecture is a software programming model property of NVIDIA [1] that presents the massive computation potential offered by the programmable GPU. GPUs can have multiple stream multiprocessors, where each stream multiprocessor consists of either eight cores if CUDA capability is 1.x, or 32 cores in the case of 2.x (GPU with Fermi architecture), or 192 cores in case of 3.x (GPU with Kepler architecture). GPU devices may have a large amount of off-chip device memory (*global-memory*) and have a fast on-chip memory (*shared-memory*, *registers*). Devices of compute capability 2.x come with an L1/L2 cache hierarchy that is used to cache *global-memory* accesses. The L2 cache is 768 kB, whereas L1 is selected by the programmer, between 16 kB and 48 kB. Sizes of L1 cache and *shared-memory* are related. Therefore, if L1 cache is set to 16 kB, the *shared-memory* size is 48 kB, whereas if L1 cache is set to 48 kB, the *shared-memory* size is 16 kB. Depending on the application, one of the two previous options will give better performance.

Following Flynn's taxonomy [13], from a conceptual point of view, a GPU can be considered as an SIMD machine (Single Instruction, Multiple Data); that is, a computer in which a single set of instructions is executed on different data sets. Implementations of this model usually work synchronously, with a common clock signal. An instruction unit sends the same instruction to all the processing elements, which then simultaneously execute this instruction on their own data. In the CUDA model, the programmer defines the kernel function. The code that will be executed on GPU is written in the kernel.

For the implementations, we used the Nvidia TESLA C2070 GPU with 2.0 CUDA capability. Because there are variables that are used by multiple threads, the selected configuration for L1 cache is 16 kB and 48 kB for *shared-memory*. The CUDA toolkit and SDK version is 4.0. The operations carried out in the algorithm use intensive floating-point arithmetic in single precision.

2.2 Real-Time Application

In a real-time audio application, audio samples come from their respetive sources and are saved in different input-data buffers, one per source. Once the input-data buffers are filled, they are transferred through the PCI express bus [25] to the GPU where all the processing of the block *Crosstalk Canceler and Equalizer* of Fig. 2 is carried out. Once the execution on GPU ends, audio samples are saved in output-data buffers and are subsequently sent back to the CPU in order to be played by the loudspeakers. The processing algorithms implemented on the GPU are based on block filtering and take into account the size of buffers and filters seeking the most efficient performance in any situation [36].

The size of input-data buffer n determines the time spent to fill it, which we call t_{buff}. Time t_{buff} is equal to n/f_s, where f_s is the audio sampling frequency. We define t_{proc} as the execution time since the input-data buffer is sent to the GPU until the output-data buffer comes back to the CPU, which includes the transfer times GPU $\Leftrightarrow$ CPU and the buffer processing on the GPU. The times t_{proc} and t_{buff} allow us to calculate two important parameters on audio signal processing: *latency* and *throughput*. Latency indicates the time from which the processing starts until an output-response is given after processing, while throughput expresses the number of input samples processed per second. Fig. 3 shows a time diagram of the processes involved for a multichannel application with C_{in}=4 and C_{out}=2. It can be observed that in order to avoid losing audio samples in a real-time application, the time t_{proc} must be less than t_{buff}. Therefore, the minimum throughput in this case is achieved when $t_{\text{proc}} = t_{\text{buff}}$.

3 GPU data structure for efficient convolution

The most relevant operation in a generalized crosstalk Cancelation is convolution. We implement the convolution on GPU focusing on two different environments based on the size of the filter (lf represents the size of the filter) and the size of the input-data buffer (defined previously as n). An implementation where the size of the input-data buffer is much larger than the size of the filter ($n \gg lf$) is described in Scheme 1 and is based on the the fragmentation of the input-data buffer. On the other hand, Scheme 2 deals with the opposite case, ($n \ll lf$) and is based on the fragmentation of the filter. The main goal of fragmentation is to obtain the best performance from the resources on the GPU, which maximally exploits the parallelism. The selected data structures in both schemes seek to obtain maximum benefit of the coalesced access to *global-memory* [21]. Note that, although both approaches are described independently here, the user does not have to be aware of this issue since the system would choose the most efficient one in a real application for the given task. The following subsections describe both schemes in the easiest situation, a simple convolution of one source with one filter.

3.1 Scheme 1: Fragmentation of the input-data buffer.

The implementation we present is based on the overlap-save technique [27]. A matrix **S** is configured using the samples within the input-data buffer. Matrix **S** has P rows and L columns. This kind of configuration is detailed in [5]. The value P indicates the number of the overlap-save frames that is configured from the n samples of the input-buffer. The value L is the size of the frames. In order to exploit GPU resources, P must be properly selected; its value determines L, which also depends on n. The filter must have the same size as the frames. Thus, the filter length will be zero-padded from lf to L. In this scheme, the filter is also considered to be a matrix, which we call **F**. Hence, matrix **F** has 1 row and L columns. The reason for configuring data in a matrix structure is to allow the same operation to be executed with different data portions and to allow data to be reused when an element-wise multiplication is carried out between the frames and the filter (Convolution Theorem, [27]). For this operation, matrix **S** stays in *global-memory* and matrix **F** is moved to *shared-memory* on GPU, since filter values are shared for all the frames during the element-wise multiplication (see Fig. 4 (a)). It is important to point out that the element-wise multiplication must be carried out in the frequency domain. To this end, FFTs of all rows of matrix **S** (P FFTs) and of matrix **F** (one FFT) had to be executed previous to the element-wise multiplication.

There are some recent publications about FFT in GPU as in [12], but the NVIDIA FFT library, CUFFT [1], is used for our application. This GPU library allows multiple one-dimesional FFTs to be obtained simultaneously.

3.2 Scheme 2: Fragmentation of the filter.

This scheme occurs in applications where latency plays an important role and the filter size is much larger than the size of input-data buffer. Therefore, it is necessary to split the filter into blocks in order to obtain a fast system response. Fragmentation could be done uniformly as in [15] and [16] or non-uniformly as in [17]. For this implementation, we use the algorithm presented in [46] which is based on *the Uniformly-partitioned fast convolution algorithm using the overlap-save technique.* The filter is uniformly fragmented into blocks whose size is the same as the size of the input-data buffer. Hence, the sizes of the matrices of Scheme 1 change in Scheme 2. Matrix **F** now has P rows and L columns, where $P{=}lf/n$ is the number of fragments obtained from the filter and L is twice the size of the input-buffer $L{=}2{\cdot}n$, that is, each subfilter is zero-padded to length L. In this case, matrix **S** has one row and L columns and contains samples of the current input-data buffer and the previous one.

One of the operations of this algorithm refers to an element-wise multiplication in the frequency domain between all the fragments of the filter and the input-data buffer. Matrix **F** stays in *global-memory* and matrix **S** is moved to *shared-memory* for this operation since input samples are shared for all the element-wise multiplications with the filter fragments (see Fig. 4 (b)). As in Section 3.1, FFTs of all rows of matrix **S** and of matrix **F** had to be executed previous to the element-wise multiplication.

4 GPU data structure for GCCE applications

This section analyzes and describes in detail the implementation of the two schemes on GPU extrapolating to a multichannel system. In the case of a GCCE application, tridimensional structures are used. The implementations are generalized for any value of sources C_{in} and loudspeakers C_{out}. In order to make the configuration and the implementation on GPU more understandable, the figures presented throughout this section illustrate a multichannel application with C_{in}=4 sources, C_{out}=2 loudspeakers and, therefore, C_{tot}=8 different filters. Thus, following (1), the output signals in the two loudspeakers are:

$$\begin{aligned} y_0 &= h_{00} * x_0 + h_{01} * x_1 + h_{02} * x_2 + h_{03} * x_3, \\ y_1 &= h_{10} * x_0 + h_{11} * x_1 + h_{12} * x_2 + h_{13} * x_3. \end{aligned} \tag{2}$$

As in Section 3, we distinguish two schemes, but now the fragmentation will be carried out in every input-data buffer (Scheme 1: Fragmentation of multiple input-data buffers) and every filter (Scheme 2: Fragmentation of multiple filters).

4.1 Scheme 1: Fragmentation of multiple input-data buffers.

Matrix **S** turns into a tridimensional matrix whose dimensions will be ($P \times L \times C_{\text{in}}$) for multichannel convolution, where overlap-save frames from the C_{in} input-data buffers are located in different layers, see Fig. 5 (a). The matrix **F** also turns into a tridimensional structure whose dimensions are ($1 \times C_{\text{out}} \cdot L \times C_{\text{in}}$). Filters h_{00} and h_{10} are placed on the same layer because their respective operations refer to different outputs. In contrast, filters h_{00}, h_{01}, h_{02}, and h_{03} are located on different layers because they take part in calculating the output y_0. The same occurs with the output y_1. This can be checked in (2) and Fig. 5 (a).

Following the overlap-save technique, the FFT of each frame in matrix $\mathbf{S}$ must be carried out (In Fig. 5 (b), X_j represents samples of sound source j, input-data samples of x_j in the frequency domain). The same occurs with the filters h_{ij}, which are transformed into H_{ij} in the frequency domain. Thus, $C_{\text{in}} \cdot P$ FFTs are calculated for each new input-data buffer while C_{tot} FFTs of filters will be executed (one for each filter) only once at the beginning of the algorithm. Two different kernels are launched to carry out the rest of the algorithm.

4.1.1 Kernel 1

Once the data are in the frequency-domain, the placement of matrix $\mathbf{F}$ in the *shared-memory* allows each frame to be simultaneously element-wise multiplied by its corresponding filter. Taking into account the content in Section 2, and using a thread for processing a sample of input-buffer, we use the grid configuration shown in Fig. 6 (a). This kernel launches $C_{\text{in}} \cdot P \cdot C_{\text{out}} \cdot L$ threads. Each thread will only make a complex multiplication between a value of matrix $\mathbf{S}$ and its corresponding complex-component in matrix $\mathbf{F}$. Each component of the filter is accessed P times, while each component of a frame is accessed C_{out} times. The result of the operation causes that $\mathbf{S}$ has now these dimensions ($P \times C_{\text{out}} \cdot L \times C_{\text{in}}$) (see Fig. 5 (b)).

4.1.2 Kernel 2

The next step consists of adding up all the layers in order to calculate the outputs in the frequency domain Y_i (Y_i represents samples of loudspeaker i, output-data samples of y_i in the frequency domain). In this case, we use a bidimensional grid configuration where a thread processes an output sample. Thus, $P \cdot C_{\text{out}} \cdot L$ threads are required to sum the layers (see Fig. 6 (b)). Each thread will make C_{in} complex sums reducing all the layers to one layer (see Fig. 7). Now, matrix $\mathbf{S}$ shows two dimensions given by $(P \times C_{\text{out}} \cdot L)$.

Finally, the CUFFT library is applied again $C_{\text{out}} \cdot P$ times in order to obtain IFFT from all the output frames of all the outputs y_i, according to the multichannel system in (2). All the frames in the time domain are then sent back to the CPU to be reproduced.

4.2 Scheme 2: Fragmentation of multiple filters.

In this scenario, the size of the input-data buffers is much smaller than the size of filters. Following the algorithm presented in [46], the filters h_{ij} are split into P fragments (as in Scheme 2 of section 3), each of which has the same size as the input buffers. As in Scheme 1, matrix $\mathbf{F}$ turns into a tridimensional matrix with dimensions $(P \times C_{\text{out}} \cdot L \times C_{\text{in}})$. All the fragments that belong to the same filter are placed within the same layer. The filters used for calculating the same output y_i remain in different layers. Matrix $\mathbf{S}$ configures another tridimensional structure with dimensions $(1 \times L \times C_{\text{in}})$. Fig.8 (a) clarifies the setting of data on GPU.

In this scheme, C_{in} FFTs are carried out every time the input-data buffers are transferred to GPU. At the beginning of the processing, $C_{\text{out}} \cdot P$

FFTs are executed in matrix $\mathbf{F}$ only once. As in the previous scheme, two different kernels are executed on GPU.

4.2.1 Kernel 1

Once the data are in the frequency-domain, the placement of matrix $\mathbf{S}$ in the *shared-memory* allows every fragment in matrix $\mathbf{F}$ to be simultaneously element-wise multiplied by its corresponding input-data buffer, thus obtaining a resulting matrix $\mathbf{R}$ with the same structure as matrix $\mathbf{F}$. If the processing in GPU is carried out on the k-th input-data buffer, the resulting matrix $\mathbf{R}$ is called $\mathbf{R}_{\mathrm{k}}$ (see Fig. 8 (b)). This matrix $\mathbf{R}_{\mathrm{k}}$ must be accumulated with the previous one, $\mathbf{R}_{\mathrm{k}-1}$, which was obtained from the *(k-1)*-th input-data buffer. However, this element-wise sum is not straightforwardly carried out but depends on a parameter that we call $\mathit{PointOut} \in [0, P-1]$. This parameter is a modular counter that increases incrementally with each new input-data buffer $\mathit{PointOut}{=}k\%P$ (% represents the rest of the division). It indicates that a generic row of matrix $\mathbf{R}_{\mathrm{k}}$ *Rkrow* ($\mathit{Rkrow} \in [0, P-1]$) must be element-wise sum with the row $(\mathit{Rkrow} + \mathit{PointOut})\%P$ of $\mathbf{R}_{\mathrm{k}-1}$. We carry out the addition between matrices $\mathbf{R}_{\mathrm{k}-1}$ and $\mathbf{R}_{\mathrm{k}}$ in this peculiar way because of the audio processing with partitioned filters [46]. Fig. 9 (a) exhibits the particular case when $\mathit{PointOut}{=}1$. For these operations, $C_{\mathrm{in}} \cdot P \cdot C_{\mathrm{out}} \cdot L$ threads are used. Each thread performs a complex multiplication between a value of matrix $\mathbf{F}$ and its corresponding complex component in matrix $\mathbf{S}$, and then accumulates the result with the corresponding value in $\mathbf{R}_{\mathrm{k}-1}$. As a thread per sample of every fragment is used, the same grid configuration as kernel 1 from Scheme 1 is applied, (Fig. 6 (a)).

4.2.2 Kernel 2

The next step consists of adding up all the layers; however, in this case, only the values on the row indicated by *PointOut* are used. The resulting vector is copied to other memory positions called *OutVect*. This vector represents the output-data buffers in the frequency-domain. After the IFFTs are applied, the outputs y_i are obtained, and are sent back to the CPU. The matrix $\mathbf{R}_{\mathrm{k}}$ takes the role of matrix $\mathbf{R}_{\mathrm{k}-1}$ for the next input-data buffer. Nevertheless, to take this role, the row indicated by *PointOut* will be set to 0 and the parameter *PointOut* will be increased incrementally after the copy to *OutVect* from the matrix $\mathbf{R}_{\mathrm{k}}$. Fig. 9 (b) reflects all these operations. This kernel launches $C_{\mathrm{out}} \cdot L$ threads. Each thread sums C_{in} complex values, saves the result in *OutVect*, and sets its corresponding elements to 0 on all layers of the row marked by *PointOut*. In this case, a unidimensional grid configuration is used where there is one thread for each processing sample (see Fig. 6 (c)).

5 Results

Several tests can be carried out with both implementations in order to find the best performance for a given environment. There are many parameters to set, both in terms of computation (CUDA parameters such as grid size, block size, number of threads, block dimensions) and of the audio signal processing aspect (latency, size of input-data buffer, number of sources, number of loudspeakers).

In this article, two different schemes have been presented depending on the size of both the input-data buffer and the filter (n and lf, respectively). When the input-data buffer is much larger than the filter size, it is fragmented into different overlap-save frames (Scheme 1). On the other hand, when the input-data buffer is much smaller than the filter size, the filter is the one that is fragmented (Scheme 2).

This second scheme aims to reduce the latency time by reducing the time of response of the system t_{proc}. Note that t_{proc} contains not only the execution time of the kernels but also the data transfers between GPU and CPU and all the data overhead in order to carry out a real-time application.

5.1 CUDA aspects

Before testing our acoustic multichannel application, it was necessary to set two CUDA parameters: the number of threads per block and the distribution of the threads within the block (*blockDim.x*, *blockDim.y*, and *blockDim.z*). The choice of these two parameters has a vast impact over timing in experiments. The different configurations tested varying these two parameters offer results that follow the same tendency. As a summary, Table 1 shows the t_{proc} obtained in a specific multichannel application with $C_{\text{in}} = 72$ and $C_{\text{out}} = 32$ using different numbers and distributions of threads within a block.

The best performance was achieved when *blockDim.x*=32, *blockDim.y*=8, and *blockDim.z*=2. As can be observed, the block is configured with 512 threads and not with 1024, which is the maximum number of threads per block in CUDA with Fermi architecture. According to [21], setting the number of threads to 512 makes the thread blocks be executed faster by the SMs on GPU.

On the other hand, the greater the number of *blockDim.x*, the greater the performances. Therefore, not only is the number of threads per block important, but also how they are distributed. Hence, for the experiments related to audio aspects, we will use the configuration *blockDim.x*=32, *blockDim.y*=8, and *blockDim.z*=2.

5.2 Audio aspects

1. $n \ll lf$

   Once the CUDA parameters are set, we will leave aside the computational aspects and focus on the scheme when the size of input-data buffer is much smaller than the size of filters. The most significant test revolves around the maximum number of filters C_{tot} that a GPU, given a specific latency time t_{buff}, can manage in a real-time multichannel GCCE. Among the different tests, we detail the time t_{proc} used by the GPU to process a system configured with a different number of sources C_{in} combined with a specific number of loudspeakers C_{out} (2, 4, 8, 16, 32, 64, and 96) using filters whose size is lf=2048 coefficients.

   The first test was done setting an input-buffer size n of 128 samples, with $t_{\text{buff}} = 2.9$ ms. The results in Fig. 10 (a) show that the obtained t_{proc} times increase linearly as the number of sources increases. Focusing on real-time applications, the maximum number of filters of this size that this implementation can manage is 1408 filters, which is obtained when C_{in}=22 and C_{out}=64.

The system can also carry out applications involving more filters, but they would not satisfy the real-time condition $t_{\text{proc}} < t_{\text{buff}}$. Therefore, the configurations below the dotted line (t_{buff}) in Fig. 10 (a),(b),(c), and (d) allow real-time applications to be carried out. Fig. 10 (a) also shows the maximum number of filters that can be achieved with a specific number of loudspeakers, 1344 filters for 96 sources and 32 loudspeakers among others. In any case, every configuration would work for off-line processing; even the ones that are above the dotted line. For example, as Fig. 10 (a) shows, for a C_{in}=38, C_{out}=64, C_{tot}=2432, the processing time t_{proc} is 4.830 ms. This means that using C_{in}=38 audio wav file sources of mono systems of 2 MB each (a mono audio wav file is composed of audio samples of `short int`, 2 bytes), the time spent to process 38 audio wav file sources with 64 loudspeakers using buffers of 128 samples would be 158.27 s. This time could be used as a processing reference for other kinds of applications that do not require real-time.

If we increase the number of input-data buffer samples to 256, as Fig. 10 (b) shows, the maximum number of filters increases to 3136, obtained when C_{in}=98 and C_{out}=32. By doubling the input-buffer size, the limit is achieved with 6336 filters (see Fig. 10 (c)).

Fig. 10 (d) shows the maximum number of filters with input-data buffer sizes of 1024. The maximum number of filters is obtained using C_{out}=96, which achieves up to 12480 filters in a GCCE.

Following the operations shown in Fig. 3, the latencies and throughputs from the maximum number of configurations are shown in Table 2. The latencies are calculated as $t_{\text{proc}} + t_{\text{buff}}$, where t_{buff}=(input-buffer size)*(1/44.1) ms. It can be observed that the latency values are approximately double the t_{proc}. Generally, the greater the number of sources C_{in}, the greater t_{proc}, and the greater throughput, whose values revolve around 1 and 10 million samples processed per second. When the input-buffer is 128 samples, maximum throughput achieves $9.977 \cdot 10^5$ samples/s; when the input-buffer is 2048 samples, maximum throughput achieves $8.185 \cdot 10^6$ samples/s.

2. $n \gg lf$

   In this case, the input-data buffer is divided into overlap-save frames. The test we show looks for the most efficient number of frames in order to exploit GPU parallelism. Among the different configurations tested in a multichannel application, we selected the one that fixes a $t_{\text{buff}} = 92.86$ ms (4096 samples) and a filter size of 129 coefficients. Fig. 11 shows t_{proc} in multichannel applications with 2, 4, 32, and 64 loudspeakers. In each implementation, a sweep of number of sources was carried out dividing the input-data buffer into a different number of overlap-save frames (2, 4, 8, 16, and 32). The best performances, which exploit the maximum GPU resources, are obtained when the input-data buffer is divided into 4 overlap-save frames, as Fig. 11 shows in (a), (b), (c), and (d).

6 Conclusions

This work analyzes different environments in the field of multichannel audio processing and implements them using Graphics Processing Units. Achieving the best performance using GPUs is not as easy as just enumerating threads and executing them; it also requires a meaningful analysis of CUDA aspects (number of threads per block and distribution of threads within the blocks) and how to set data on GPU. In this paper, we have detailed the implementation of a multichannel convolution on GPU using tridimensional data structures, (tridimensional blocks and tridimensional grids). The algorithm implemented on GPU responds to a massive convolution or a generalized crosstalk cancellation and equalization. The placement of data inside the GPU changes depending on the size of the input-data buffer and the size of the filters. When the size of filters is much larger than the size of input-data buffer, the filters are fragmented and the parallelism is exploited by the element-wise multiplication of the fragments with the input-data buffer. The evaluated tests show that, with only an input-data buffer of 128 samples, it is possible to achieve up to real-time multichannel applications with 1408 filters of 2048 coefficients. This number gets larger as the input-data buffer increases. Otherwise, when the size of the filters are much smaller than the size of the input-data buffers, these buffers are fragmented into ovelap-save frames. In this case, parallelism is exploited by the element-wise multiplication of the frames with the filter in the frequency domain. The figures shown for this test indicate that when the input-data buffers are fragmented into four frames, minimun t_{proc} time is achieved.

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

7 Acknowledgements

This work has been partially funded by Spanish Ministerio de Ciencia e Innovacion TEC2009-13741, Generalitat Valenciana PROMETEO 2009/2013 and GV/2010/027, and Universitat Politècnica de València through Programa de Apoyo a la Investigación y Desarrollo (PAID-05-10).

References

[1] *NVIDIA Developer Zone*, online at: http://developer.nvidia.com.

[2] *openGL*, online at: http://www.opengl.org/.

[3] J. Belloch, A. Gonzalez, F. Martinez-Zaldivar, and A. Vidal, *A real-time crosstalk canceller on a notebook GPU*, in Multimedia and Expo (ICME), 2011 IEEE International Conference on, Barcelona, Spain, july 2011, pp. 1 –4.

[4] J. A. Belloch, M. Ferrer, A. Gonzalez, F. Martinez-Zaldivar, and A. M. Vidal, *Headphone-based spatial sound with a GPU accelerator*, Procedia Computer Science, 9 (2012), pp. 116 – 125. Proceedings of the International Conference on Computational Science, ICCS 2012.

[5] J. A. Belloch, A. Gonzalez, F. J. Martínez-Zaldívar, and A. M. Vidal, *Real-time massive convolution for audio applications on GPU*, Journal of Supercomputing, 58 (2011), pp. 449–457.

[6] A. Berkhout, D. de Vries, and P. Vogel, *Acoustic control by wave field synthesis*, J. Acoustic. Soc. Amer, 93 (1993), pp. 2764–2778.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, *A performance study of general-purpose applications on graphics processors using CUDA*, Journal of Parallel and Distributed Computing, 68 (2008), pp. 1370–1380.

[8] B. Cowan and B. Kapralos, *Spatial sound for video games and virtual environments utilizing real-time GPU-Based Convolution*, in Proc. 2008 Conf. on Future Play: Research, Play, Share, Ontario, Canada, November 2008.

[9] ——, *Efficient 3d audio processing with the GPU*, in FuturePlay 2009 at GDC Canada International Conference on the Future of Game Design and Technology, Vancouver, Canada, May 2009.

[10] ——, *GPU-Based One-Dimensional Convolution for Real-Time Spatial Sound Generation*, Loading...: The Journal of the Canadian Game Studies Association, 3 (2009), pp. 1–14.

[11] L. D'Amore, D. Casaburi, A. Galletti, L. Marcellino, and A. Murli, *Integration of emerging computer technologies for an efficient image sequences analysis*, Integrated Computer-Aided Engineering, 18 (2011), pp. 365–378.

[12] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju, *Auto-tuning of fast Fourier transform on graphics processors*, in Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, PPoPP '11, 2011, pp. 257–266.

[13] M. Flynn, *Some computer organizations and their effectivness*, IEEE Transactions on Computers, 21 (1972), pp. 948–960.

[14] E. Gallo and N. Tsingo, *Efficient 3D Audio Processing with the GPU*, in GP2: ACM Workshop on General Purpose Computing on Graphics Processors, Los Angeles, USA, August 2004.

[15] G. Garcia, *Optimal filter partition for efficient convolution with short input/output delay*, in Proceedings of the 113th AES Convention, Los Angeles,U.S.A., October 2002.

[16] W. G. Gardner, *Efficient convolution without input-output delay*, Journal of the Audio Engineering Society, 43 (1995), pp. 127–136.

[17] P. M. Gerald and P. C. W. Sommen, *A new method for efficient convolution in frequency domain by nonuniform partitioning for adaptive filtering*, IEEE Transactions on signal processing, 44 (1996), pp. 127–136.

[18] C. Gregg and K. Hazelwood, *Where is the data? Why you cannot debate CPU vs. GPU performance without the answer*, in ISPASS 2011 - IEEE International Symposium on Performance Analysis of Systems and Software, Austin, U.S.A., April 2011.

[19] Y. A. Huang, J. Chen, and J. Benesty, *Immerse audio schemes*, IEEE Signal Processing Magazine, 28 (2011), pp. 20–32.

[20] M. Jedrzejewski and K. Marasek, *Computation of room acoustics using programmable video hardware*, Computational Imaging and Vision, 32 (2006), pp. 587–592. http://dx.doi.org/10.1007/1-4020-4179-9_84.

[21] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors*, Morgan Kaufman, 2010.

[22] L. Lattari, A. Montenegro, A. Conci, E. Clua, V. Mota, M. Bernardes-Vieira, and G. Lizarraga, *Using graph cuts in GPUs for color based human skin segmentation*, Integrated Computer-Aided Engineering, 18 (2011), pp. 41–59.

[23] N. Masoumi, S. Safavi-Naeini, and M. I. Elmasry, *A methodology for substrate crosstalk evaluation for system-on-a-chip*, Integrated Computer-Aided Engineering, (2002).

[24] C. I. C. Nilsen and I. Hafizovic, *Digital beamforming using a GPU*, in IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings, Taipei, Taiwan, April 2009, pp. 609–612.

[25] A. Nishida, *Building cost effective high performance computing environment via PCI express*, in Proceedings of the 2006 International Con-

ference on Parallel Processing Workshops, Columbus,U.S.A., August 2006, pp. 195–198.

[26] J. C. Noyer, P. Lanvin, and M. Benjelloun, *Correlation-based particle filter for 3d object tracking*, Integrated Computer-Aided Engineering, 16 (2011), pp. 165–177.

[27] A. V. Oppenheim, A. S. Willsky, and S. Hamid, *Signals and systems*, Processing series, Prentice Hall, 2nd ed., 1997.

[28] R. Rabenstein, S. Spors, and P. Steiffen, *Wave field synthesis techniques for spatial sound reproduction*, Topics in Acoustic Echo and Noise Control, 5 (2006), pp. 517–545. http://dx.doi.org/10.1007/3-540-33213-8_13.

[29] N. Rober, U. Kaminski, and M. Masuch, *Ray acoustics using computer graphics technology*, in Conference on Digital Audio Effects (DAFx-07) proceedings, Bourdeaux, France, June 2007.

[30] L. Savioja, V. Välimki, and J. O. Smith, *Audio signal processing using graphics processing units*, J. Audio Eng. Soc, 59 (2011), pp. 3–19.

[31] S. Siltaten, T. Lokki, and L. Savioja, *Frequency domain acoustic radiance transfer for real-time auralization*, Acta Acustica/Acustica, 95 (2009), pp. 106–117.

[32] A. Smirnov and T. Chiueh, *Implementation of a fir filter on a gpu*, tech. report, Experimental Computer Systems Lab. Stony Brook University, 2005.

[33] S. Spors, R. Rabenstein, and W. Herbordt, *Active listening room compensation for massive multichannel sound reproduction system using wave-domain adaptive filtering*, J. Acoustic. Soc. Am., 122 (2007), pp. 354–369.

[34] D. Theodoropoulos, C. Ciobanu, and G. Kuzmanov, *Wave field synthesis for 3D audio: Architectural prospectives*, in Proc. ACM Int. Conf. Computing Frontiers, Ischia,Italy, May 2009, pp. 127–136.

[35] D. Theodoropoulos, G. Kuzmanov, and G. Gaydadjiev, *Multi-core platforms for beamforming and wave field synthesis*, IEEE Transactions on multimedia, 3 (2011), pp. 235–245.

[36] A. Torger and A. Farina, *Real-time partitioned convolution for ambiophonics surround sound*, in IEEE Workshop on the Applications of Signal Processing to Audio and Acoustics, New York,U.S.A., October 2001, pp. 195–198.

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

[38] F. Trebien and M. Oliveira, *Realistic real-time sound re-synthesis and processing for interactive virtual worlds*, The Visual Computer, 25 (2009), pp. 469–477. 10.1007/s00371-009-0341-5.

[39] N. Tsingo, *Using programmable graphics hardware for acoustics and audio rendering*, in Proceedings of the 127th AES Convention, New York, USA, October 2009.

[40] ——, *Using programmable graphics hardware for auralization*, in Proceedings of EAA Symp. on Auralization, Espoo, Finland, June 2009.

[41] B. V. Veen and K. Buckley, *Beamforming: A versatile approach to spatial filtering*, IEEE ASSP Mag., 5 (1988), pp. 4–24.

[42] G. Vigueras, J. M. Orduña, M. Lozano, and Y. Chrysanthou, *A distributed visualization system for crowd simulations*, Integrated Computer-Aided Engineering, 18 (2011), pp. 349–363.

[43] C. J. Webb and S. Bilbao, *Computing room acoustics with CUDA - 3D FDTD schemes with boundary losses and viscosity*, in IEEE International Conference on Acoustics, Speech and Signal Processing, Prague, Czech Republic, May 2011.

[44] C. J. Webb and S. Bilbao, *Virtual room acoustics: A comparison of techniques for computing 3D-FDTD schemes using CUDA*, in Proceedings of the 130th AES Convention, London, U.K., May 2011.

[45] F. Wefers and J. Berg, *High-Performance real-time FIR-filtering using fast convolution on graphics hardware*, in Proc. of the 13th Conference on Digital Audio Effects, Graz, Austria, September 2010.

[46] F. Wefers and M. Vorländer, *Optimal filter partitions for real-time FIR filtering using uniformly-partitioned FFT-based convolution in the frequency-domain*, in Proc. of the 14th Conference on Digital Audio Effects, Paris, France, September 2010.

Table 1: t_{proc} in a multichannel application with $C_{\text{in}} = 72$ and $C_{\text{out}} = 32$ using different distributions of thread blocks: *blockDim.x*, *blockDim.y*, and *blockDim.z*

blockDim.x	blockDim.y	blockdDim.z	threads per block	t_{proc} (ms)
64	8	2	1024	5.219
32	16	2	1024	5.243
32	**8**	**2**	**512**	**4.573**
16	16	2	512	4.742
16	8	2	256	4.822
8	16	2	256	5.258

Table 2: Latencies and Throughputs from the maximum number of C_{tot} that are obtained under real-time conditions.

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

- Fig. 1: The signal at loudspeaker y_i is composed of a combination of all the sources x_j filtered through their respective h_{ij}.
- Fig. 2: $2 \cdot Q$ desired signals are set to each ear of Q listeners in a room. Cross paths and room effects are canceled because of the use of the *Crosstalk canceler and Equalizer* block.
- Fig. 3: Important parameters in a real-time multichannel application, with C_{in}=4, C_{out}=2 and C_{tot}=8.
- Fig. 4: (a) shows Scheme 1 where matrix **S** is located in *global-memory* and matrix **F** in *shared-memory*; (b) shows the opposite case, Scheme 2 where matrix **F** is located in *global-memory* and matrix **S** in *shared-memory*.
- Fig. 5: (a) shows matrices **S** and **F** in GPU. Then, frequency-domain transform and element-wise multiplication are applied. (b) shows that the resulting matrix is stored at the same memory position.
- Fig. 6: (a) Tridimensional thread block grid structure launched in kernel 1. There is a thread for every component of the frames. (b) Thread block grid structure for accumulating the resulting values obtained from kernel 1 in Scheme 1. (c) Thread block grid structure for accumulating the resulting values obtained from kernel 1 in Scheme 2.
- Fig. 7: Addition of all the planes to obtain the different outputs (in this case, Y_0 and Y_1).
- Fig. 8: (a) shows matrices **S** and **F** in GPU. Then, frequency-domain

transform and element-wise multiplication are applied. (b) shows that the resulting matrix $\mathbf{R}_{\mathrm{k}}$ is stored in a different memory position.

- Fig. 9: (a) Element-wise sum between $\mathbf{R}_{\mathrm{k}}$ and $\mathbf{R}_{\mathrm{k}-1}$. Row 0 of $\mathbf{R}_{\mathrm{k}}$ is element-wise sum with the row indicated by *PointOut*; row 1 is element-wise sum with the row indicated by *PointOut+1*; and so on. (b) Copy of the row indicated by *PointOut* in $\mathbf{R}_{\mathrm{k}}$ to *OutVect*, which is later set to 0. *PointOut* increases incrementally and gets prepared for the next input-data buffer.

- Fig. 10: t_{proc} used by GPU in a GCCE for different values of sources C_{in} and loudspeakers C_{out}, using a sampling frequency of f_s=44.1 kHz with: t_{buff}=2.9 ms in (a), t_{buff}=5.8 ms in (b), t_{buff}=11.6 ms in (c), and t_{buff}=23.2 ms in (d).

- Fig. 11: t_{proc} in a multichannel application fragmenting the input-buffer in different overlap-save frames: (a) for 2 loudspeakers; (b) for 4 loudspeakers; (c) for 32 loudspeakers; and (d) for 64 loudspeakers.

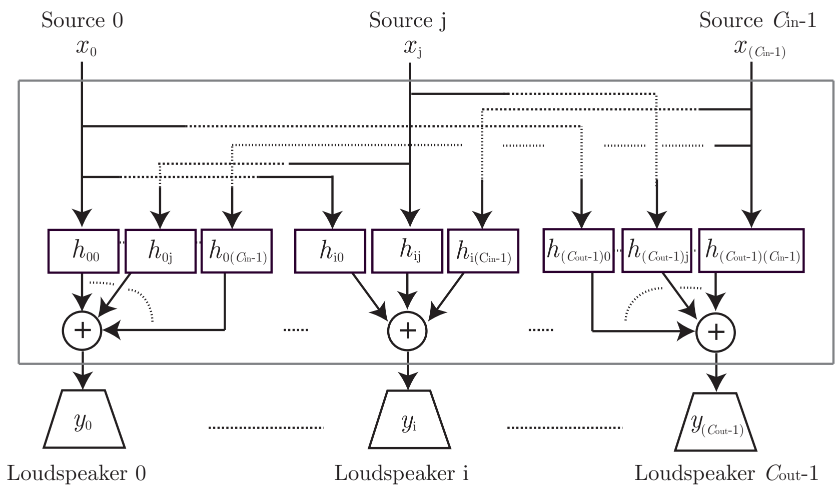

Source 0
x_0
Source j
x_j
Source C_{in}-1
$x_{(C_{in}-1)}$
h_{00}
h_{0j}
$h_{0(C_{in}-1)}$
h_{i0}
h_{ij}
$h_{i(C_{in}-1)}$
$h_{(C_{out}-1)0}$
$h_{(C_{out}-1)j}$
$h_{(C_{out}-1)(C_{in}-1)}$
y_0
y_i
$y_{(C_{out}-1)}$
Loudspeaker 0
Loudspeaker i
Loudspeaker C_{out}-1

Fig 1:

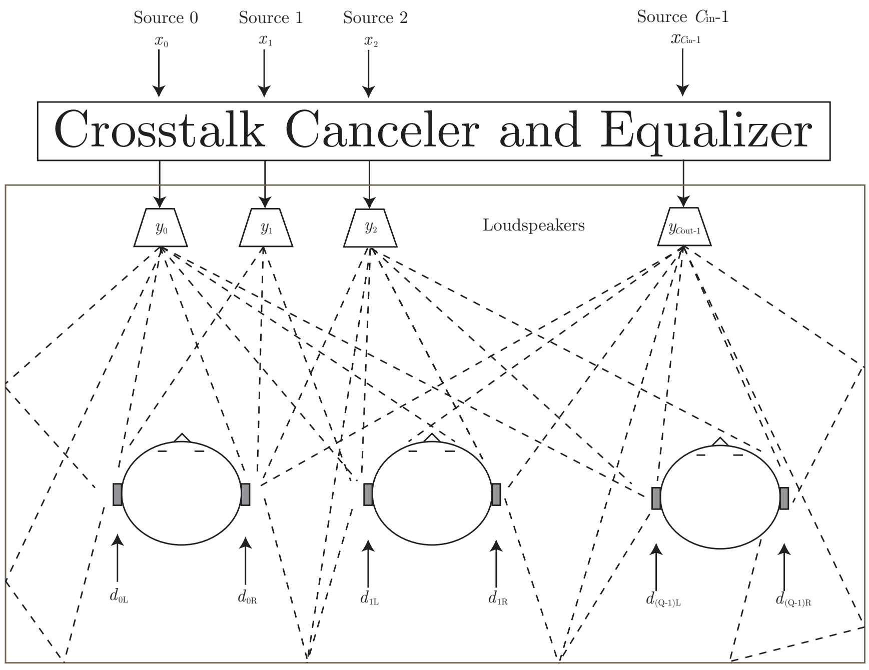

Source 0
x_0
Source 1
x_1
Source 2
x_2
Source C_{in}-1
$x_{C_{\text{in}}\text{-}1}$
Crosstalk Canceler and Equalizer
y_0
y_1
y_2
Loudspeakers
$y_{C_{\text{out}}\text{-}1}$
$d_{0\text{L}}$
$d_{0\text{R}}$
$d_{1\text{L}}$
$d_{1\text{R}}$
$d_{(\text{Q-1})\text{L}}$
$d_{(\text{Q-1})\text{R}}$

Fig 2:

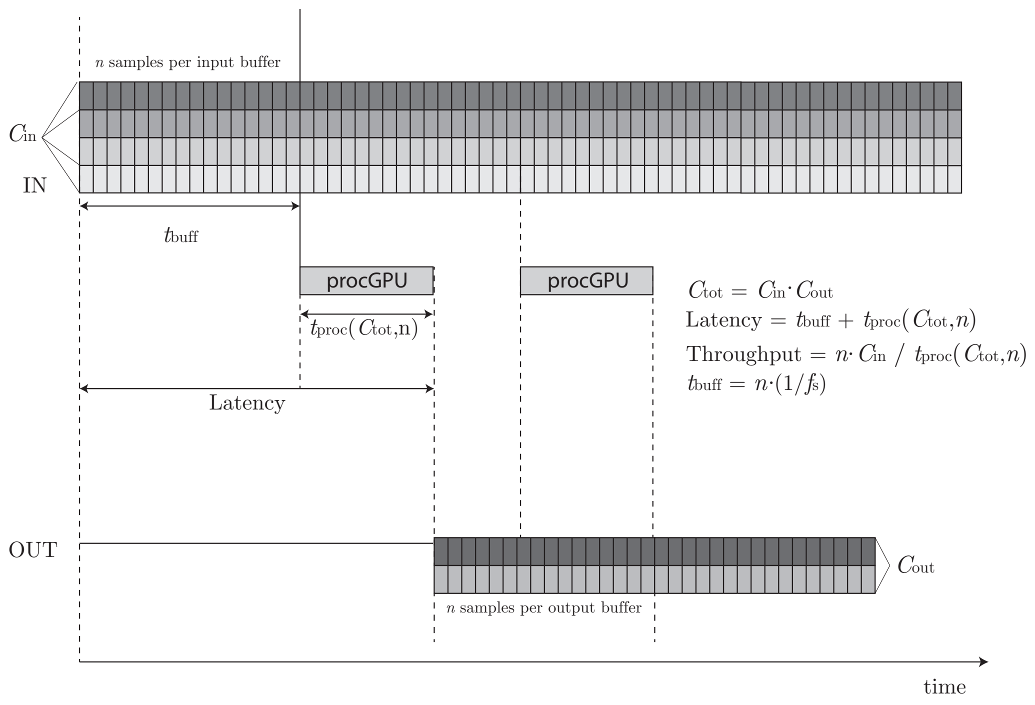

n samples per input buffer
C_{in}
IN
t_{buff}
procGPU
procGPU
$t_{\text{proc}}(C_{\text{tot}},\text{n})$
Latency
$C_{\text{tot}} = C_{\text{in}} \cdot C_{\text{out}}$
Latency $= t_{\text{buff}} + t_{\text{proc}}(C_{\text{tot}},n)$
Throughput $= n \cdot C_{\text{in}} \,/\, t_{\text{proc}}(C_{\text{tot}},n)$
$t_{\text{buff}} = n \cdot (1/f_s)$
OUT
C_{out}
n samples per output buffer
time

Fig 3:

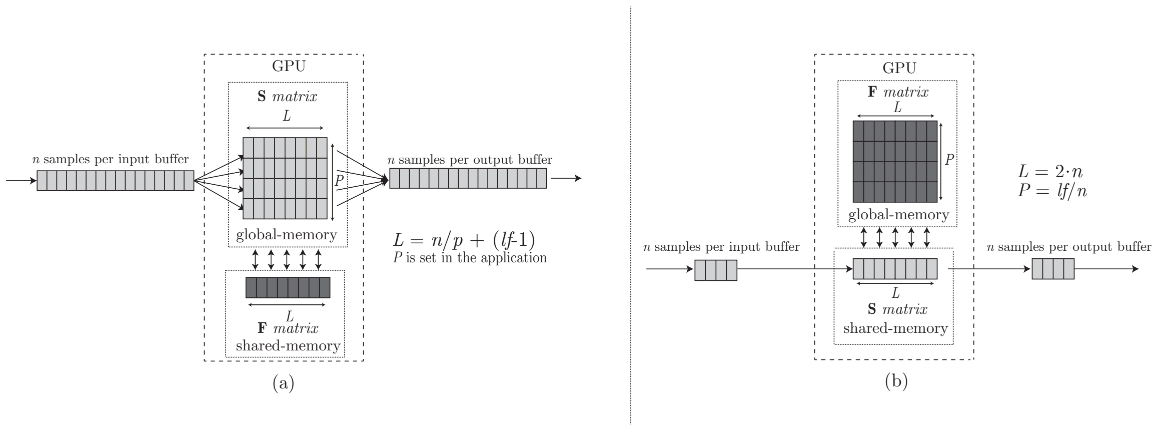
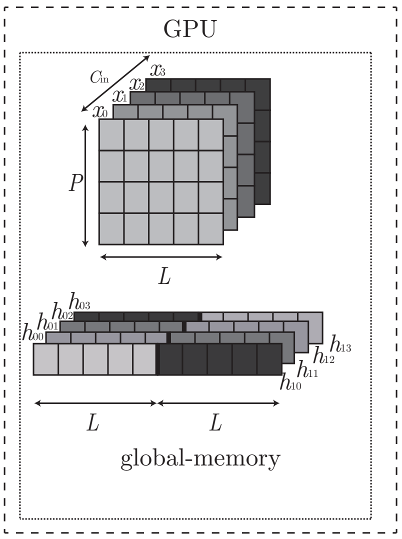
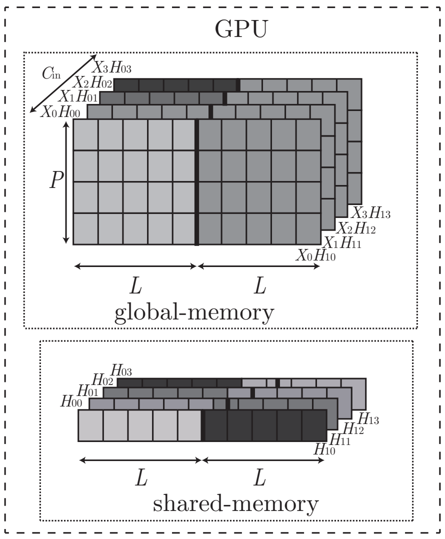

GPU
$\mathbf{S}$ *matrix*
L
n samples per input buffer
P
n samples per output buffer
global-memory
$L = n/p + (lf\text{-}1)$
P is set in the application
L
$\mathbf{F}$ *matrix*
shared-memory
(a)
GPU
$\mathbf{F}$ *matrix*
L
P
$L = 2{\cdot}n$
$P = lf/n$
global-memory
n samples per input buffer
n samples per output buffer
L
$\mathbf{S}$ *matrix*
shared-memory
(b)

Fig 4:

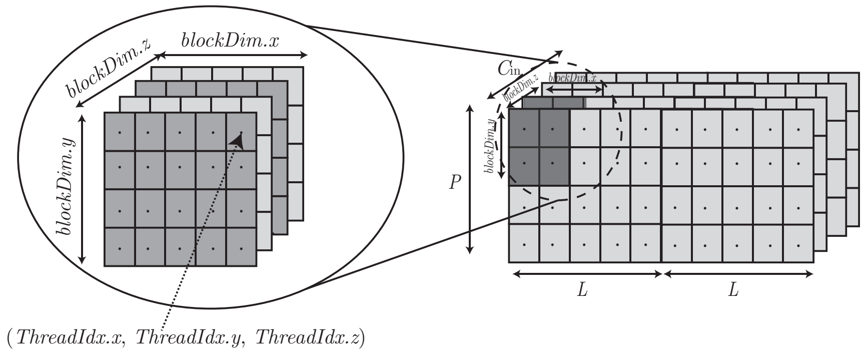

GPU
C_{in}
x_0
x_1
x_2
x_3
P
L
h_{00}
h_{01}
h_{02}
h_{03}
h_{10}
h_{11}
h_{12}
h_{13}
L
L
global-memory

(a)


GPU
C_{in}
X_0H_{00}
X_1H_{01}
X_2H_{02}
X_3H_{03}
P
X_0H_{10}
X_1H_{11}
X_2H_{12}
X_3H_{13}
L
L
global-memory
H_{00}
H_{01}
H_{02}
H_{03}
H_{10}
H_{11}
H_{12}
H_{13}
L
L
shared-memory

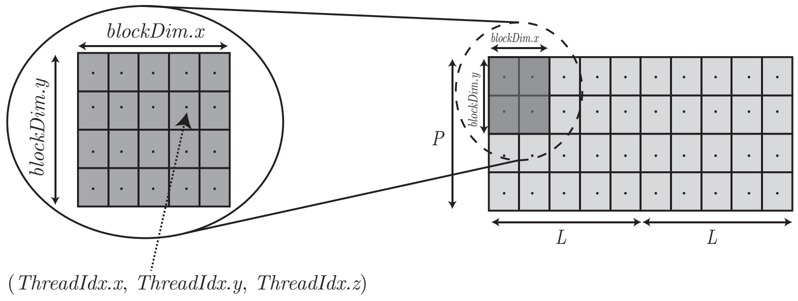
(b)

Fig 5:



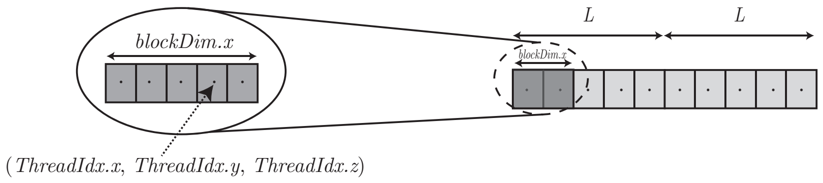
blockDim.x
blockDim.z
blockDim.y
C_{in}
blockDim.z
blockDim.x
blockDim.y
P
L
L
(ThreadIdx.x, ThreadIdx.y, ThreadIdx.z)

(a)



blockDim.x
blockDim.y
blockDim.x
blockDim.y
P
L
L
(ThreadIdx.x, ThreadIdx.y, ThreadIdx.z)

(b)



blockDim.x
L
L
blockDim.x
(ThreadIdx.x, ThreadIdx.y, ThreadIdx.z)

(c)

Fig 6:

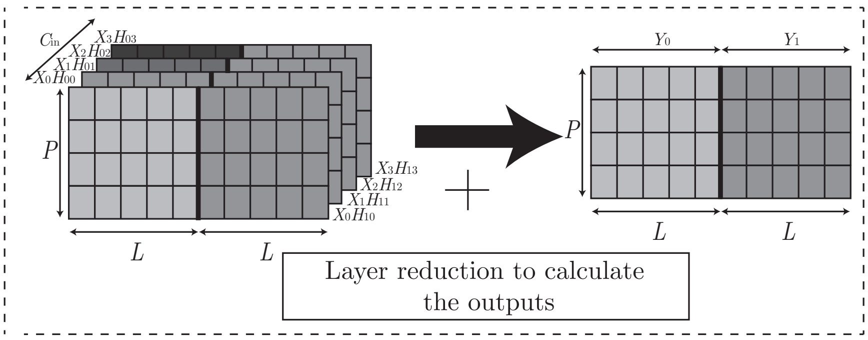

C_{in}
X_3H_{03}
X_2H_{02}
X_1H_{01}
X_0H_{00}
P
L
L
X_3H_{13}
X_2H_{12}
X_1H_{11}
X_0H_{10}
$+$
Y_0
Y_1
P
L
L
Layer reduction to calculate the outputs

Fig 7:

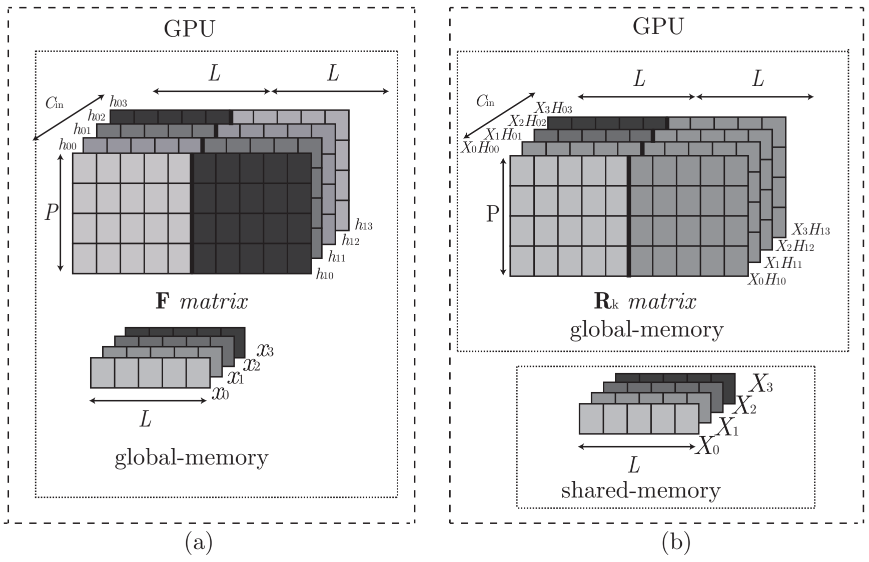
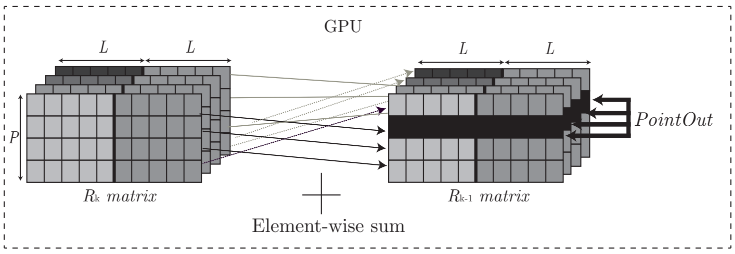
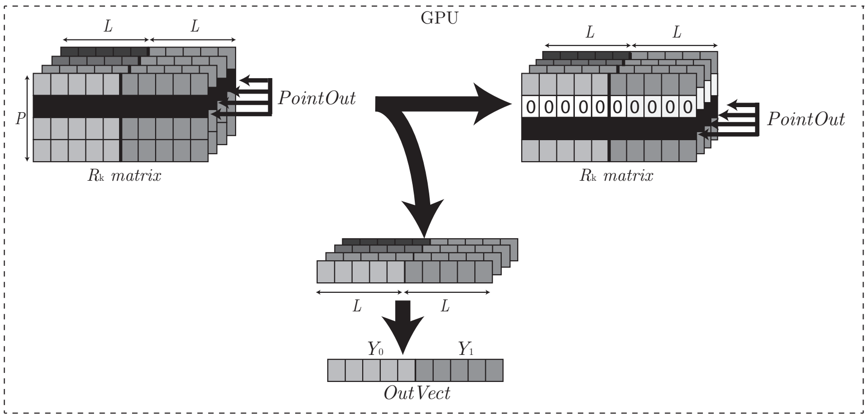

GPU
L
L
C_{in}
h_{03}
h_{02}
h_{01}
h_{00}
P
h_{13}
h_{12}
h_{11}
h_{10}
$\mathbf{F}$ *matrix*
x_3
x_2
x_1
x_0
L
global-memory
(a)
GPU
L
L
C_{in}
X_3H_{03}
X_2H_{02}
X_1H_{01}
X_0H_{00}
P
X_3H_{13}
X_2H_{12}
X_1H_{11}
X_0H_{10}
$\mathbf{R}_k$ *matrix*
global-memory
X_3
X_2
X_1
X_0
L
shared-memory
(b)

Fig 8:

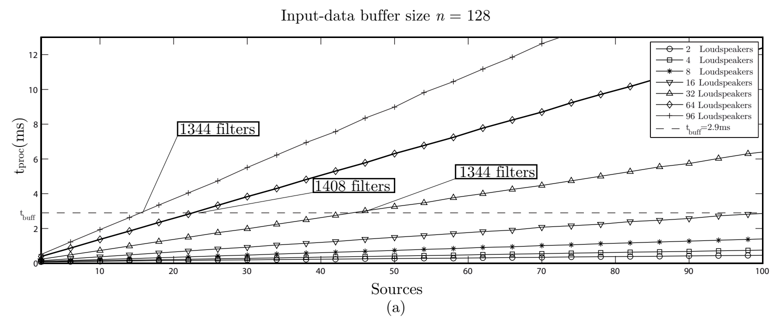

GPU
L
L
P
R_k matrix
L
L
PointOut
$R_{k\text{-}1}$ matrix
Element-wise sum

(a)

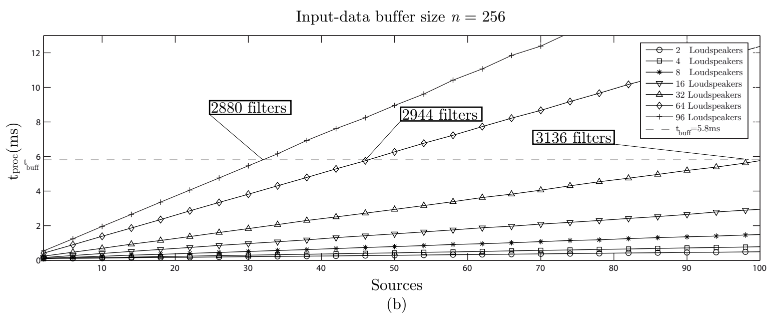

GPU
L
L
P
PointOut
R_k matrix
L
L
0 0 0 0 0 0 0 0 0 0
PointOut
R_k matrix
L
L
Y_0
Y_1
OutVect

(b)

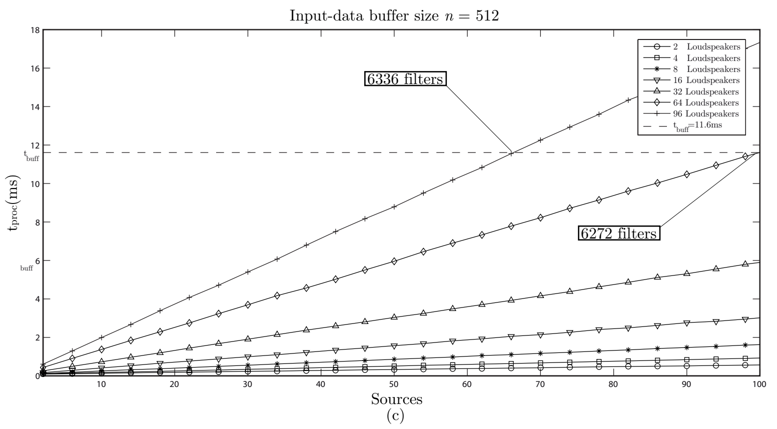
Fig 9:


Input-data buffer size $n = 128$
t_{proc}(ms)
Sources
1344 filters
1408 filters
1344 filters
t_{buff}
2 Loudspeakers
4 Loudspeakers
8 Loudspeakers
16 Loudspeakers
32 Loudspeakers
64 Loudspeakers
96 Loudspeakers
t_{buff}=2.9ms

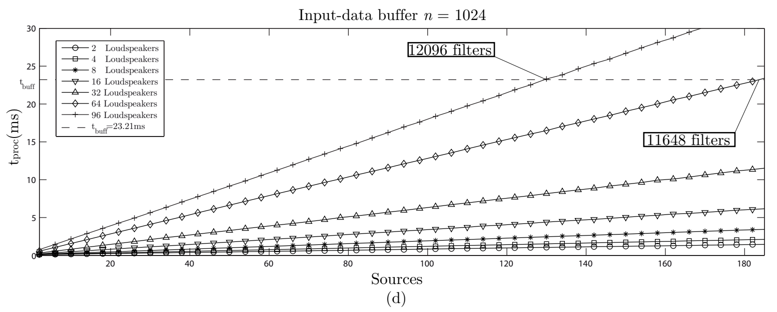
(a)


Input-data buffer size $n = 256$
t_{proc}(ms)
Sources
2880 filters
2944 filters
3136 filters
t_{buff}
2 Loudspeakers
4 Loudspeakers
8 Loudspeakers
16 Loudspeakers
32 Loudspeakers
64 Loudspeakers
96 Loudspeakers
t_{buff}=5.8ms

(b)

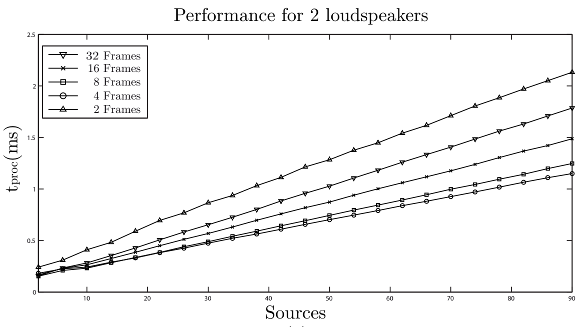

Input-data buffer size $n = 512$
t_{proc}(ms)
Sources
6336 filters
6272 filters
t_{buff}
2 Loudspeakers
4 Loudspeakers
8 Loudspeakers
16 Loudspeakers
32 Loudspeakers
64 Loudspeakers
96 Loudspeakers
t_{buff}=11.6ms

(c)

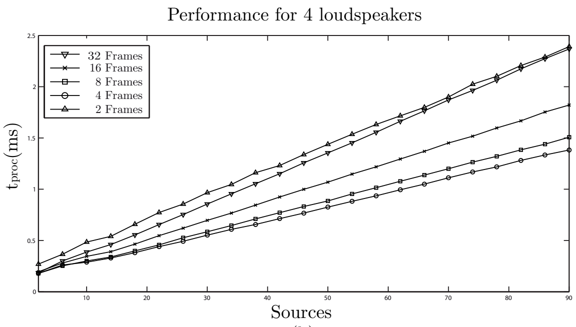

Input-data buffer $n = 1024$
t_{proc}(ms)
Sources
12096 filters
11648 filters
t_{buff}
2 Loudspeakers
4 Loudspeakers
8 Loudspeakers
16 Loudspeakers
32 Loudspeakers
64 Loudspeakers
96 Loudspeakers
t_{buff}=23.21ms

(d)

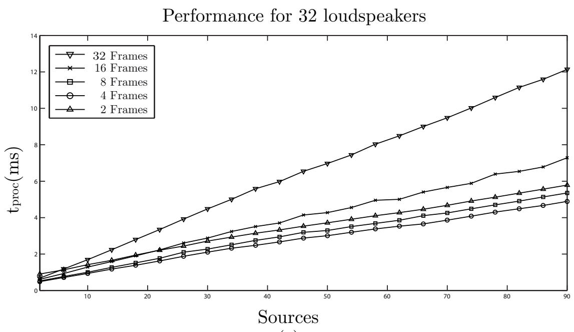
Fig 10:


Performance for 2 loudspeakers
32 Frames
16 Frames
8 Frames
4 Frames
2 Frames
t_{proc}(ms)
Sources

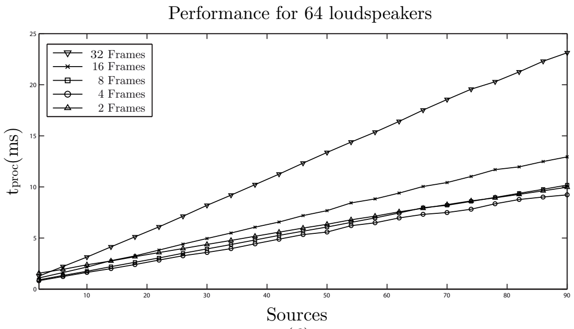
(a)


Performance for 4 loudspeakers
32 Frames
16 Frames
8 Frames
4 Frames
2 Frames
t_{proc}(ms)
Sources

(b)


Performance for 32 loudspeakers
32 Frames
16 Frames
8 Frames
4 Frames
2 Frames
t_{proc}(ms)
Sources

(c)


Performance for 64 loudspeakers
32 Frames
16 Frames
8 Frames
4 Frames
2 Frames
t_{proc}(ms)
Sources

(d)

Fig 11: