Politecnico di Milano

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE
Corso di Laurea Magistrale in Ingegneria Informatica
Master of Science in Computer Science And Engineering

Tesi di Laurea Magistrale - Master's Thesis

Spectral Manipulation of Audio using General-Purpose Graphics Processing Units

Relatori - Advisors
Prof. Augusto Sarti
Prof. Victor Lazzarini

Candidato - Candidate
Andrea Gianfranco Crespi
Matr.813390

Anno Accademico 2015-2016

Politecnico di Milano:
www.polimi.it

Scuola di Ingegneria Industriale e dell'Informazione:
www.ingindinf.polimi.it

Acknowledgements

First, I would like to thank my hosting research supervisor in Ireland, Victor Lazzarini, for accepting my unusual request to carry out a project under his supervision, and for welcoming me in the facilities of Maynooth University. I would like to thank him for the extraordinary support he has given me throughout all the stages of this project, both during my stay in Dublin and later. I acknowledge that he has dedicated a lot of energies and attentions to this project, being truly helpful and patient. He introduced me to the exciting world of *Csound* and to some critical aspects of audio programming and debugging. Finally, I thank him for assisting me in the writing process of this thesis by means of meticulously correcting multiple draft versions of each chapter.
Along with Victor Lazzarini, I would also like to thank the whole Music Department of Maynooth University for providing me with the tools for the development of this project: a quiet room to work in, two desktop computers, and two brand new graphics cards, specifically bought for this project.

Then, I would like to thank my advisor, Augusto Sarti, for supporting the idea of a project to be carried out abroad, outside the scope of Politecnico di Milano, and for his assistance in the final stages of the drafting process.

Not least, I would like to thank all the competent and passionate professors I was lucky to meet on my path at Politecnico di Milano, from whom I have had the pleasure to learn the foundations of such fascinating fields as engineering and computer science. In particular, I would like to thank: Marco Bramanti (calculus), Lorenzo Mezzalira (computer science), Paolo Dulio (algebra and geometry), Marco Sampietro (electronics), Marco Tagliasacchi (digital signal processing), Matteo Matteucci (statistical learning) and, again, Augusto Sarti (sound analysis, synthesis and processing). Without their teachings and stimuli, this work could not be even conceived.

Milano, July 2016 Andrea Crespi

If this word "music" is sacred
and reserved for eighteenth- and nineteenth-century instruments,
we can substitute a more meaningful term:
organization of sound.

John Cage
'The Future of Music: Credo' (1937)

Contents

List of Figures

List of Tables

Sommario

L'ambito dell'elaborazione digitale di segnali audio è caratterizzato da una ineusaribile necessità di dispositivi di calcolo sempre più potenti e di programmi sempre più efficienti. Figure professionali quali il *sound designer*, l'ingegnere del suono, il musicista e il compositore sono alla costante ricerca di strumenti sempre meno limitanti, di tecniche di elaborazione e sintesi del suono che restituiscano una maggiore qualità audio, nonché di tecniche di analisi più sofisticate, possibilmente a costi contenuti. La comparsa sul mercato, nel corso dell'ultimo decennio, di processori altamente paralleli e liberamente programmabili (come le moderne unità di elaborazione grafica), ha innescato una nuova tendenza nella ricerca scientifica volta allo sviluppo di processi audio ad alte prestazioni: in molti casi lo sfruttamento della suddetta categoria di processori, ampiamente diffusa in diverse tipologie di dispositivi elettronici, consente di eseguire un maggior numero di applicazioni contemporaneamente e/o di ottenere risultati di maggiore qualità. In questa tesi viene analizzata e commentata la prassi di delegare l'elaborazione di processi audio di vario tipo a processori grafici. Inoltre, particolare attenzione viene dedicata allo studio di un'applicazione specifica, l'elaborazione spettrale del suono: il contesto operativo fornito da *Csound* per la manipolazione di segnali audio basata sulla rappresentazione *streaming phase vocoder* viene declinato in termini di calcolo parallelo su unità di elaborazione grafica, attraverso lo sviluppo di moduli operativi in ambiente *CUDA*. La funzionalità di questi moduli è stata verificata su due sistemi di riferimento e comparata con le versioni originali, basate su un modello di calcolo tradizionale, ovvero centralizzato. I risultati mostrano che la manipolazione spettrale di segnali audio su unità grafiche non è solo realizzabile in tempo reale ma è anche caratterizzata da migliori prestazioni se confrontata con il metodo tradizionale impostato su unità di elaborazione centrali, almeno per quanto riguarda i due sistemi in esame.

Parole chiave: GP-GPU, Elaborazione Spettrale, HiPAC, CUDA, Csound

Abstract

The scope of audio computing is in a never-ending need for more processing power and more efficient software. Sound designers, engineers, musicians and composers are constantly seeking for less restrictive tools, for higher-quality synthesis and processing techniques, and for more sophisticated sound analysis techniques, possibly at a lower cost. The appearance on the market of general-purpose, highly parallel processors (like modern GPUs) has triggered, in the last decade, a new trend in the scientific research aimed at developing better performing audio processes: in many cases, harnessing this kind of hardware, which is widespread in a variety of electronic devices (or it can be conveniently added to most computer systems), allows for the execution of more tasks simultaneously and/or for the achievement of better sound quality results. In this thesis, the practice of casting audio computing tasks to general-purpose graphics processing units is reviewed and discussed. In addition, a specific field of application is thoroughly investigated, namely that of spectral manipulation of audio signals: the streaming phase vocoder framework provided in the *Csound* environment is ported to a GPU-based parallel computing model by means of developing *CUDA*-based processing modules. These modules are tested and compared against the original CPU-based versions on two target computer systems. The results show that real-time spectral manipulation of audio on GPUs is not only feasible but it is also computed potentially faster with respect to the standard centralised approach.

Keywords: GP-GPU, Spectral Processing, HiPAC, CUDA, Csound

Introduction

Thesis Overview and Motivations

This thesis investigates new ways of improving the performance and the efficiency of audio processing algorithms on digital systems by adopting a *general-purpose GPU computing* scheme (GP-GPU[1]). In particular, the attention is focused on real-time spectral audio manipulations based on the phase vocoder model[2], and the framework chosen for this study is that of commodity desktop computers equipped with a dedicated graphics processor.

Even though, in the scope of digital media, the subclass of audio computing and processing is generally considered one of the less demanding in terms of computational power, this is only true for what concerns everyday applications that are intended for the average user, who is not involved in a creative[3] process and does not play an active role in the very definition of the processing steps. Nevertheless, today's technology standards can still be limiting for the achievement of certain results, even when bleeding edge tools are employed, especially in those applications that involve non-standard procedures and in all those cases where an always higher level of audio quality[4] is required. Real-time spectral processing in particular is an application field that tends to quickly saturate the available processing load of an average digital system.
Thus, since the early days of digital audio, the hardware and software industries have never stopped developing more and more powerful processing tools, both in terms of audio quality capabilities and in terms of the processing power and efficiency that is needed to actually make better audio possible. As a consequence, during the last decades, the constant improvement of digital audio, along with a

[1] *GP-GPU computing*: the combined use of latency-oriented CPUs (based on a sequential computing model) and throughput-oriented graphics processors (based on a parallel computing model) in order to optimise the performance of a digital system when executing parallel algorithms. For an insight on this topic, see section 1.1. *GPU*: graphics processing unit.

[2] *Phase vocoder*: a particular representation of audio signals in the frequency domain. See section 1.2.

[3] The term *creative* is used here in a wide sense and it is not only referring to artistic processes but also to any audio application (being it for analysis, processing or synthesis purposes) that requires an active approach in order to carry out a desired task. This could apply to situations of artistic, technological or scientific nature. This kind of approach could be necessary, for example, in the scope of music production applications, in the scope of sound design for videogames, cinema or virtual reality, as well as that of sound analysis for scientific purposes.

[4] Here the words *audio quality* do not refer only to the quality of music reproduction but have to be interpreted in a wider sense, and can refer to such things as superior sound synthesis possibilities (for musical or non-musical purposes) or enhanced sound telecommunication capabilities and low-latency real-time applications, to name a few.

more general burst of digital technology on the whole, has eventually made audio manipulation available to a wider public, with an ever growing level of quality and possibilities, even without the need for dedicated hardware. The personal computer, and, generally speaking, the introduction of general-purpose computers, have had a major role in this transition: audio processing can now be performed at will on machines that are not specifically designed for this purpose. Mobile devices such as tablets and phones have also recently gained attractive general-purpose capabilities and they *de facto* expand the possibilities for audio processing even further. Interestingly enough, all kinds of general-purpose devices have seen an increasing utilisation for many specific audio-related tasks[5], even in professional environments: from electronic music performance to acoustics simulations.
Still, regardless of the level of computational power and generality of a specific digital machine (from DIY-oriented microcontrollers to smartphones, from laptops to application-specific supercomputers), there will always be a need for more computational power in order to make more and better achievements possible in a given framework: in fact, there will always be a demand for more realistic and aesthetically pleasing sounds and effects, as well as for a more precise analysis of sound sources. Indeed, countless audio applications can provide higher sound quality or better results, in general, when more processing power is available.

In order to optimise the overall audio processing performance of a given device or application (without sacrificing its versatility), a new paradigm has emerged in the course of the last decade: it consists of the exploitation of a *heterogenous computing* scheme, where the usual scalar processors are flanked by vector processors (like GPUs) in the computation of parallel algorithms. This direction is consistent with a more general shift of the whole computer industry, from sequential to parallel computing ([1]). In [2] and [3], Dobson, ffitch and Bradford defined the *High-Performance Audio Computing* (HiPAC) domain of study as descriptive of new classes of computationally demanding audio processes implemented by means of the next generation of parallel processing platforms and tools. HiPAC is seen as a way to potentially transform the current landscape of audio synthesis, processing and music composition. As a matter of fact, the GP-GPU model (a subclass of *heterogenous computing*) has been successfully applied to a wide variety of audio-related fields (see chapter 2).

This work investigates the possibilities offered by the GP-GPU computing scheme, for the manipulation (or the synthesis) of audio signals. This investigation is twofold: on one side, a brief review of the existing scientific literature about this topic will be presented (chapter 2); on a more empirical side, a GPU-based implementation for a few real-time spectral processing algorithms will be developed (chapter 3) and tested (chapter 4). The specific field that will be addressed in the experimental section of the thesis is that of *phase vocoder processing*[6]. In particular, nine basic and well-known phase vocoder algorithms will be developed and tested on a desktop computer system. The choice of this specific field for the experimental part (i.e., the frequency-domain manipulation of sound) has a double motivation: it is both known for being particularly demanding in terms of computational resources

[5]This is actually true for all kinds of digital media.

[6]For an insight on the phase vocoder, see section 1.2.

and it is generally well suited for a parallel approach, which is essential for an effective use of the GPU.

Brief Description of the Project

Starting from a review of the scientific literature about GP-GPU computing for audio-related applications (in a very broad sense), this project aims at assessing its potentials for a performance improvement in the much narrower field of real-time phase vocoder processing. The report that can be found in chapter 2, in fact, deals with an assorted range of different fields in the macro category of sound-related research (for instance, ray tracing [4], wave-based modelling [5], spectral model synthesis [6], finite difference physical models [7], to cite but a few) and explores the ways these can relate with the GP-GPU framework. This report aims at understanding conventional techniques and typical issues involved in the implementation of GP-GPU programs for audio-related purposes. It also summarises the potentials of this computing model in terms of performance improvement. Much of the information contained in this review has been useful for the development stage, for which nine phase-vocoder-based algorithms are translated from a traditional (i.e., sequential) scheme to a parallel computing design. Specifically, the considered algorithms are *unit generators*[7] within the *Csound*[8] programming language for sound design and computer music. This project provides for the transposition of these algorithms to a parallel computation paradigm by means of the *CUDA*[9] platform, to be used in combination with NVIDIA[10] GPUs.

The starting point for this project is to be found in a publication by Victor Lazzarini et al., namely “Streaming Spectral Processing with Consumer-Level Graphics Processing Units” ([8]). This paper describes the implementation of a streaming spectral processing system for real-time audio that is intended to be executed in a GP-GPU fashion by means of the *CUDA* platform. It explores, among other processes, the implementation of standard phase vocoder analysis and re-synthesis (in the form of *Csound unit generators*), and it investigates the potentials, in terms of computational speed, of harnessing GPU resources in this context. The results obtained in [8] show that GPU-based full phase vocoder analysis and re-synthesis can be run quite efficiently (and in real-time) on an off-the-shelf laptop computer equipped with an on-board GPU. The authors point out that, even though this solution, on average, does not provide for a speed-up in the execution time on the target system (at least for the phase vocoder analysis/re-synthesis program), it can actually serves as a means of freeing up some computation load from the CPU in a multicore/multiprocessor operation scenario. Also, they expect the *CUDA* version to outperform the original version of this algorithm when executed on more

[7] *Unit generator*: the basic formal unit in *MUSIC-N*-style computer music programming languages (like *Csound*). *Unit generators* form the building blocks for designing synthesis and signal processing algorithms in software. The unit generator theory of sound synthesis was first developed and implemented by Max Mathews and his colleagues at Bell Labs in the 1950s.

[8] *Csound*: `http://csound.github.io/about.html`. For more information about *Csound* please refer to section 3.1.1.

[9] *NVIDIA CUDA*: `http://www.nvidia.com/object/cuda_home_new.html`. For more information about *CUDA*, please refer to section 3.1.2.

[10] *NVIDIA*: `http://www.nvidia.com/`

advanced GPUs (or, alternatively, with less powerful CPUs).
The study conducted by Lazzarini et al. in [8] eventually opens up a new direction for further research. In fact, as it has been proven that phase vocoder analysis and re-synthesis can be efficiently carried out in a GP-GPU fashion, it is worth wondering whether phase vocoder processing for spectral manipulations can be conveniently implemented in the same framework. After all, most frequency-domain processing algorithms are theoretically well suited for parallel computation. In addition, the possibility of keeping spectral data in the GPU, instead of bouncing data between host memory and device memory, is extremely appealing, as it would critically reduce the occurrence of memory transfers, which are very costly in terms of latency. Thus, this thesis investigates the possibility of expanding the *CUDA* framework for spectral processing in *Csound*, started by Lazzarini et al. in [8], so that the GPU is thoroughly employed for a full spectral processing chain (from analysis to re-synthesis), including the actual frequency-domain manipulation stage (or stages).

The project consists in the translation of nine phase-vocoder-based spectral processing *Csound* modules from their original form to a *CUDA*-based implementation for parallel computing. The specific algorithms addressed in this project are the following (see section 3.4 for more details about the specific *unit generators*):

- Frequency-domain amplitude scaling (`pvscale`)
- Frequency-domain filtering (`pvsfilter`)
- Frequency-domain adaptive filtering (`pvstencil`)
- Pitch scaling (`pvscale`)
- Pitch shifting (`pvshift`)
- Spectral flux smoothing via IIR filters (`pvsmooth`)
- Spectral flux smoothing via FIR filters (`pvsblur`)
- Frequency-domain selective overlapping of two signals (`pvsmix`)
- Frequency-domain interpolation between two signals (`pvsmorph`)

The resulting *plugin opcodes*[11] are tested within basic *Csound* scripts on two target systems equipped with two different NVIDIA GPUs. In these testing scripts the new *opcodes* are inserted in the processing chain in between the *CUDA*-based phase vocoder analysis and re-synthesis *unit generators* developed for [8]. Their execution times are compared with those obtained by the original modules, which are designed to run on the CPU. The results are analysed and commented in order to asses the conditions that, in this framework, endow the GP-GPU approach with superior efficiency. The goal of this project is precisely that of implementing efficient algorithms that could take advantage of idle hardware resources (in the GPU, specifically) in order to either free up CPU load for other tasks or even boost the performance of the very same processes, when possible. Yet, the same study can also be useful for investigating the overall potentials of parallel computing in a spectral audio framework, regardless of the actual processor employed. In fact, very similar implementations could be also applied to dedicated co-processor systems in high-performance computing applications.

[11] *Plugin opcodes*: custom *Csound* modules.

Thesis Outline

The content of the thesis is structured as follows:

In the first chapter the thesis is contextualised and background information is provided. This chapter is designed to open the way for the implementation of GPU-based phase vocoder processing programs. Hence, it provides a general review of the two main topics involved, namely: the GP-GPU computing framework and the phase vocoder framework. As a matter of fact, chapter 1 is boldly split into two separate sections.
The first section introduces the concepts of *heterogeneous computing* and the GPU as a powerful co-processor for general purpose computing, as well as for graphics-related tasks. A focus is set on modern NVIDIA GPUs, their architecture and the software abstractions employed in order to program them and use them as general purpose parallel processors.
The second section is devoted to an introduction to the phase vocoder representation of audio signals. The mathematics involved in the transition from a time domain signal to its phase vocoder representation (and back) is faced and explained, with a focus on the *short time Fourier transform* (STFT) approach/interpretation. The convenience of the phase vocoder representation for certain classes of signal manipulation (including those that are considered in chapter 3) is addressed and made clear.

The second chapter moves away from the very specific topic of phase vocoder processing and takes a step back, analysing the wider scope of GP-GPU computing for audio applications. A variety of audio-related research fields are explored with the aim of summarising the potentials of GP-GPU computing when applied to each area. These include: additive synthesis, spectral model synthesis, physically-based synthesis via finite difference methods, room acoustics modelling, headphone-based spatial sound via HRTFs, digital audio effects aimed at music production (with a focus on reverberation) and recursive filters. In order to bring the discussion back to this thesis' main topic, the last section is dedicated to a few publications in the area of GPU-based spectral audio processing. Some attention is also dedicated to the very first examples of GPU-based audio processing, before graphics processors were designed for general purpose programming.

The third chapter describes the implementation stage of nine *CUDA*-based phase vocoder processing modules that are designed to be integrated in the *Csound* environment. To begin with, the employed tools (i.e., *Csound* and *CUDA*) are presented. In particular, *Csound*'s spectral signal processing framework (also called *fsig* framework) is thoroughly illustrated. Then, an *a priori* analysis of *CUDA*-based spectral processing is carried out, highlighting potential benefits as well as plausible trade-offs and limitations.
Finally, after a quick display of the nine original *unit generators* that are meant to be translated to a parallel computing model, the actual implementation of *CUDA*-based *plugin opcodes* is addressed. Ultimately, these are shared objects

written in *CUDA C*[12] that are designed to be integrated in *Csound*'s source code in order to selectively cast computations to the GPU. The translation of each original *unit generator* is thoroughly described with the support of *CUDA C* code snippets. Each resulting *plugin opcode* is deeply analysed in its very structural details.

The fourth chapter describes the testing stage of the *plugin opcodes* developed for this project. Here, the specifications of the two target systems (two average desktop computers) are reported, with a particular focus on their graphics processors. The testing procedure is described, and so are the *Csound* scripts employed for testing purposes. The aim of the tests is that of recording the execution time of small sound processing programs under different conditions and parameter settings (such as DFT size and hop size of the phase vocoder model). These programs are designed in a modular fashion so that different versions can be analysed, each version having different stages of the processing chain cast to the GPU. The corresponding execution times are compared to those resulting from the fully sequential version of the same programs (running on the CPU only) in order to obtain *speed-up factors*. Module after module, these results are illustrated with the support of convenient graphs and they are successively analysed and commented. The specific conditions under which a GP-GPU approach seems to give a substantial performance benefit are thus identified.
Finally, a comparison between the performance of the two target GPUs is carried out and a few options for future code optimisation are considered and described.

[12] *Cuda C*: a set of extensions to the *C/C++* programming language designed to allow the programmer to cast computations to the graphics processors.

Chapter 1

Background: the General-Purpose GPU Computing Framework and the Phase Vocoder

This chapter presents the general topics on which this work is based. These are essentially two: general-purpose computing on graphics processing units (usually abbreviated as "GP-GPU computing") and the phase vocoder technique. While the former is a remarkably wide subject that applies to an extremely broad range of fields, the latter is a quite specific technique, mainly employed in the field of signal processing for audio applications. In this work, both topics are brought together in order to investigate the potential of a synergistic application of the two, in the scope of digital audio processing.

1.1 General-Purpose Computing on GPUs

General-purpose GPU computing is the practice of employing the graphics processors of a digital system for non-graphical purposes. GP-GPU computing is a subcategory of a broader field of computer science, namely *heterogeneous computing*, which studies complex digital systems that use more than one kind of processing unit in order to gain performance and/or efficiency from a synchronous and symbiotic action of more and diverse cores. Each processing unit is placed in the system either to carry out specific tasks, for which it is specialised, or to perform general-purpose computations that are particularly congenial to its specific architecture. The aim of a *Heterogeneous System Architecture* (HSA) is precisely that of exploiting different kinds of computing units for different tasks, making sure that this behaviour is hidden from the ultimate user of the system: in fact, the user does not need to know which unit has to be used for the management of each single sub-operation that is involved in the execution of a desired "macro task"[1].

The units that make up a heterogeneous architecture system typically belong to some of the following classes ([9]): latency-oriented cores (i.e., classic CPUs, *scalar* or *superscalar* architectures), throughput-oriented cores (i.e., *vector processors* and/or massively parallel architectures like modern GPUs), intellectual property

[1] This is actually taken care of by the heterogeneous system programmer.

cores, digital signal processing cores for specific media-related tasks (DSPs) and configurable logic/cores. This work focuses on graphics processors in particular, and their potentially relevant role in the field of audio signal processing on heterogeneous systems.

1.1.1 The GPU

In a heterogeneous computing framework, the GPU plays a very important role, as it is an extremely widespread kind of processing unit which is frequently found in a variety of systems, including systems that are very common in the everyday life of many people: from personal computers to mobile phones, from embedded systems to gaming consoles. Modern graphics processors are placed in heterogeneous systems in order to accommodate two possible kinds of need: firstly they are specialised electronic circuits designed to execute graphics-related computations such as digital image (and video) rendering and processing, mainly (but not solely) for display purposes; secondly, when they are paired with and coordinated/directed by a central processing unit, they can be exploited to carry out some of the most burdensome computations required by any general-purpose task. Actually, while it is true that GPUs can handle *any* task if properly programmed, not any task is well suited for this kind of processors. This is because of the way modern GPU architectures are designed as extremely wide vector processors with hundreds of simple, unsophisticated[2] cores. Typically, computations are cast to the GPU when they involve the very same chain of operations to be done in parallel on large blocks of data, in a *Single-Instruction Multiple-Data* (SIMD) fashion.

In a heterogeneous system, a better utilisation scheme is potentially achieved when sequential computations on the CPU are interleaved with parallel SIMD operations on the GPU. This setting ultimately corresponds to what is labelled as *General-Purpose GPU Computing*. There are two possible reasons for which GP-GPU computing could be needed inside a heterogeneous system: sometimes it could make sense to cast specific processes to the GPU (provided they are at least moderate candidates to the parallel scheme) so that the CPU is relieved of excessive computational loads, and this is especially true in those cases when the GPU is at a low level of utilisation or it is not being utilised at all; alternatively, and most importantly, modern GPUs have superior computing potentials, with respect to CPUs, when an application involves a high degree of data parallelism and a massively parallel processing scheme can be applied. In these cases[3] the internal architecture of the GPUs allows for compute-intensive tasks to be completed in a fraction of the time required by a CPU of comparable market value. Speed-up

[2]GPU cores are kept simple by design, mainly for circuit area economy and energy dissipation considerations, as the chip needs to fit hundreds of cores. With respect to CPUs, GPU cores typically feature small caches (to boost memory throughput), lower clock frequencies and very basic control mechanisms, lacking branch prediction techniques and data forwarding. The idea is to generate and maintain thousands of threads in flight, in contrast to the use of large caches in order to hide memory latencies in CPU designs. Computing parallelism is mainly physical (many cores effectively operate at the same time on different data) and virtual parallelism is mostly found at thread level.

[3]Image and video processing are in fact particular cases of a massively parallel processing scenario, which includes applications from virtually any field.

factors in the dozens or even hundreds are to be expected for processes that are particularly well suited for the parallel scheme.
Countless examples could be adduced to prove the successful employment of GPUs in a plethora of fields and applications that are not related to graphics. Just to give an idea, these include[4]: financial analysis, scientific and engineering simulation, numerical methods for mathematical problems, statistical modelling, machine learning, data mining, biomedical informatics, computational chemistry, ray tracing rendering, interactive physics, cryptography, electronic design automation and digital audio processing. Chapter 2 of this thesis is precisely dedicated to a review about the applications of the GP-GPU scheme to the field of digital audio processing.

1.1.2 A Brief History of GPU Architecture

In order to have a complete view on the topic of GP-GPU computing, a basic understanding of the GPU[5] architecture and its evolution throughout the last decades is needed. What follows is a quick recap of the main concepts regarding the history of GPUs (see also [10]), with a particular focus on the critical years of birth and development of the general-purpose approach, which roughly coincides with the first decade of the century.
The main objective of this section is to show how GPU hardware architectures evolved from a specific single core, fixed function hardware pipeline designed solely for graphics, to a set of hundreds parallel and highly programmable cores for general-purpose computing. Prior to a certain stage in the evolution of GPUs (i.e., the early 2000s), it would have been nonsense to think of casting audio processing to the GPU, simply because of a lack of programmability. Furthermore, the huge benefits that will be exploited from modern GPUs in the development of this thesis mainly rely on the concept of hardware parallelism, which was introduced in the late 1990s. Parallelism is in fact the main drive for any GP-GPU project, in any field.

The original GPUs were modelled after the concept of a graphics pipeline: this is a conceptual model of stages that graphics data is sent through to transform coordinates from a 3D space and other environment information (specified by the programmer) into the 2D pixel space on the screen[6]. The graphics pipeline was initially implemented via a combination of hardware (the GPU itself) and software running on the CPU (thanks to APIs like *OpenGL*[7] and *DirectX*[8]). The graphics pipeline can be generalized out of two main stages: geometry and rendering, the former being mostly associated with software applications and the latter being mostly carried out by hardware stages on the GPU.

[4]See `http://gpgpu.org/` for a comprehensive catalogue about the current and historical use of GPUs for general-purpose computation.

[5]Although the term "GPU" would not be introduced until 1999 by NVIDIA, it will be used throughout this report in order to refer to any chip designed for graphics computations and to send data to a screen.

[6]The graphics pipeline will be addressed with more details in section 2.1

[7]*Khronos Group OpenGL*: `https://www.opengl.org/`

[8]*Microsoft DirectX*:
`https://msdn.microsoft.com/en-us/library/windows/apps/hh452744`

The early 1980s have generally been credited with being the roots of the modern era of computer graphics, although the GPUs of the time were really just integrated *framebuffers*[9]. During the 1980s and early 1990s the trend was to design co-processors that featured circuits to handle more and more graphics pipeline stages (starting from the last rendering stages, those conceptually closer to the frame buffer), so as to free up more and more CPU cycles. During the 1990s (and beyond), the videogame industry was a huge driving force for higher performance chips: even with deep hardware pipelines, early GPUs could still output only one pixel per clock cycle, meaning that, especially for gaming applications, the CPU could still send more information to the GPU that it could handle. This lead to the need of adding more pipelines in parallel to the GPU, so that multiple pixels could be processed in parallel each clock cycle.
In 1999, the first cards to implement a set of complete pipelines at consumer level were released by both NVIDIA and ATI[10]. Still, this hardware was based on "fixed function" circuits: once the programmer sent data into the GPU, the data had to be processed in a pre-established fashion. While faster, this model involved an annoying lack of flexibility for graphical effects and it could not have a flourishing future: as newer features were added to graphics APIs, the fixed function model could not take advantage of the new standards.
Starting from 2001, more and more programmability inside of the GPU pipelines was added to newer models: as a result, the programmer was given the possibility of sending, along with the usual data, *vertex* and *fragment shader programs* that operate on data while in the pipeline. These *shader programs* were small *kernels*, written at assembly-level or in C-like languages like NVIDIA's *Cg*[11], *OpenGL Shading Language* (*GLSL*) and Microsoft's *High Level Shading Language* (*HLSL*[12]).
As this trend of extending the GPU programmability went on, in 2003 the first wave of non-graphics GPU computing started to come about with the introduction of *DirectX 9*. In 2004, early high level GPU languages such as *Brook*[13] and *Sh*[14] started to appear. These languages provided, among other features, dynamic flow control in *shader programs*.
In 2006, a critical step in the evolution of GPU architecture was made with the introduction of NVIDIA's GeForce 8 series and its new massively parallel scheme: the "unified design"[15] (Figure 1.1). The hardware version of the graphics pipeline was abandoned in favour of a series of simple, general-purpose, all alike parallel cores called "Streaming Processors". These cores were grouped in a set of "Streaming Multiprocessors" working in a SIMD fashion. Ten years later, this is still the basis for the most recent GPU architectures. Of course, for graphical applications, the traditional graphics pipeline (meaning the conceptual model) still applies to the

[9]A *framebuffer* is a portion of memory reserved for holding the complete bit-mapped image that is sent to the monitor.
[10]*ATI*: `https://en.wikipedia.org/wiki/ATI_Technologies`
[11]*NVIDIA Cg*: `https://developer.nvidia.com/cg-toolkit`
[12]*Microsoft HLSL*:
`https://msdn.microsoft.com/it-it/library/windows/desktop/bb509561(v=vs.85).aspx`
[13]*Brook (Stanford University)*:
`https://graphics.stanford.edu/projects/brookgpu/lang.html`
[14]*Sh (University of Waterloo)*: `http://www.libsh.org/`
[15]Otherwise known as "unified shader architecture".

unified design: it simply does not appear as a hardware scheme any more, as it becomes purely a software abstraction.

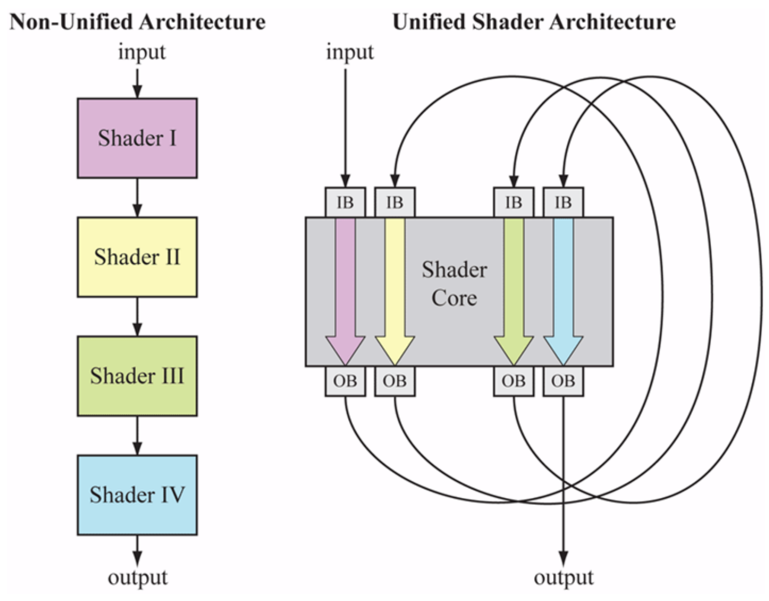

Figure 1.1: Vertex shaders, pixel shaders, etc. become threads running different programs on a flexible core. (Image from [11])

To harness all this general-purpose GPU power NVIDIA developed a new set of extensions to *C/C++*: *CUDA*. This was made available in 2007 and it was designed to work in combination with NVIDIA GPUs only. Not much later, ATI *Stream*[16] for ATI cards and *DirectX 10* for either card (though Microsoft Windows only) were introduced. In 2009, the *OpenCL*[17] framework for writing programs for heterogeneous parallel platforms (including any CPU-GPU model) was released by the Khronos Group in collaboration with technical teams at NVIDIA, AMD[18], IBM[19], Qualcomm[20] and Intel[21].
In 2010, NVIDIA's Fermi was the first GPU architecture designed specifically for GP-GPU computing: it featured true hardware cache hierarchy, ECC, unified memory address space and concurrent kernel execution.

Later evolutions in the GPU architecture are beyond the scope of this thesis but, for completeness, it is interesting to mention a new trend of fusing CPU-like cores and GPU-like vector processors on the same die. This trend was started in 2011 by AMD with the introduction of their APUs ("Accelerated Processing Unit"): these processors are designed so that a standard x86 processor for scalar workloads and a DX11 GPU for vector workloads are brought together on the same die. Heterogeneous parallel processing can be done on this hardware via *OpenCL*. Intel followed with similar architectures. On a similar direction, but

[16] *AMD Stream Computing*: http://developer.amd.com/partners/training-partners/streamcomputing/
[17] *Khronos Group OpenCL*: Open Computing Language, https://www.khronos.org/opencl/
[18] *AMD*: http://www.amd.com/
[19] *IBM*: http://www.ibm.com/
[20] *Qualcomm*: https://www.qualcomm.com/
[21] *Intel*: http://www.intel.com/

regarding smaller form factors and a lower power consumption target, NVIDIA has been producing the Tegra[22] system-on-a-chip line since 2008. These chips are designed as efficient multimedia/gaming processors[23] that combine an ARM[24] CPU with a Maxwell-based[25] (or Kepler-based[26]) GPU on the same die. Of course, heterogeneous parallel processing can be also done on this hardware via *CUDA*. It seems that future GPU generations will look more and more like wide-vector, general-purpose processors, and eventually there will be no real distinction between CPUs and GPUs, as they might combine in one single entity.

1.1.3 Modern NVIDIA GPUs: Architecture and Programming Model

This work is focused on the use of NVIDIA GPUs in particular (via the *CUDA*[27] API) and, since many terms and concepts that only apply to this specific framework will be used extensively throughout this thesis, it is required to present these in more detail. Actually, many of the concepts that are going to be discussed in the following paragraphs also apply to graphics units from other manufacturers.

NVIDIA GPUs belonging to the *unified shader architecture* design can be described at two levels of abstraction: the hardware structure and the software abstractions used to program it, i.e. the *CUDA* programming model. While the software abstractions are common to all generations of *CUDA*-enabled GPUs, the hardware microarchitecture has varied from one generation to the next: this discussion will be based on NVIDIA's Maxwell microarchitecture which is the latest available at the time of writing. In particular, the first generation Maxwell (GeForce-Maxwell 107) will be addressed, as the main graphics card employed throughout the implementation and benchmarking stages of this thesis (GeForce GTX750Ti) belongs to this category.

At hardware level, NVIDIA GPUs are structured in a hierarchical fashion. At the highest level, one ore more *Graphics Processing Clusters* (GPCs) can be found[28], as well as a *GigaThread Engine* (see figure 1.2). The former is a collection of vector processors containing the actual cores, the latter is a component that marshals data and instructions between the GPC, the raster operations processors (ROPs), the cache memory, the memory controllers, the bus interface, and the display I/O ([12]).
Each GPC contains a variable number[29] of *Streaming Multiprocessors* (SMs). These are vector processors on which blocks of threads are spawned by the *GigaThread*

[22] *NVIDIA Tegra*: http://www.nvidia.com/object/tegra.html

[23] Apart from being installed in smartphones, tablets and gaming consoles, Tegra is also the core of the Jetson board for embedded systems, which is designed for developing low-power and compute-intensive embedded projects.
NVIDIA Jetson: http://www.nvidia.com/object/embedded-systems.html

[24] *ARM*: https://www.arm.com/

[25] See section 1.1.3 for more details about NVIDIA Maxwell microarchitecture.

[26] Older versions of the Tegra units featured a VLIW-based VEC4 architecture instead.

[27] For a better insight about *CUDA*, please refer to section 3.1.2.

[28] Entry-level and medium range GPUs, like the GeForce GTX750Ti, are typically equipped with only one GPC.

[29] There are five SMs in the single GPC of GeForce-Maxwell 107 microarchitecture.

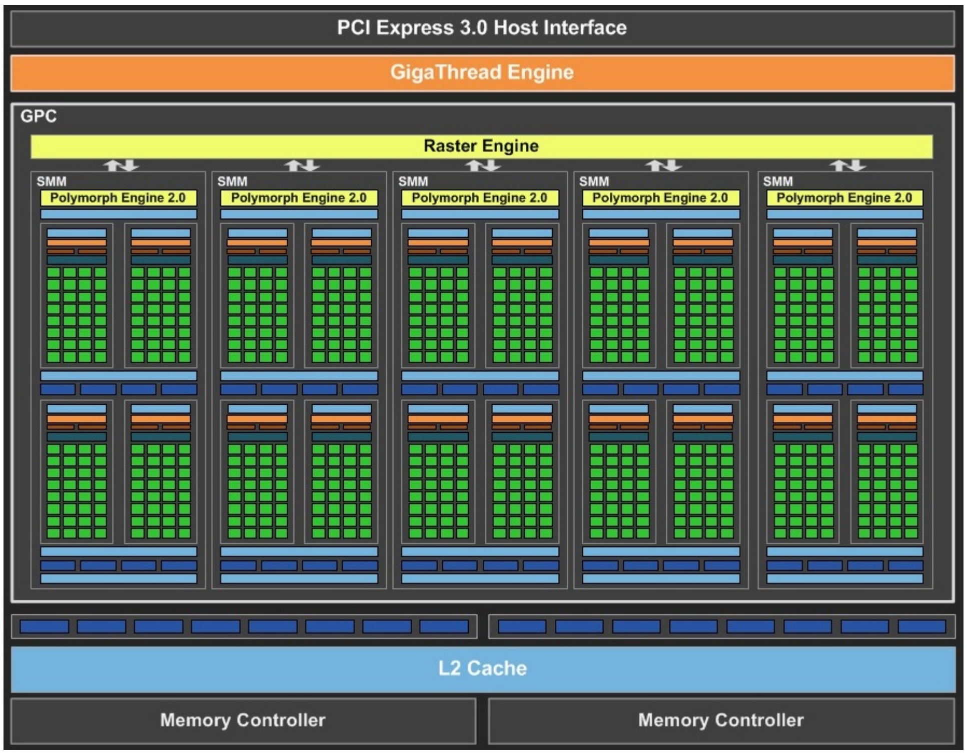

Figure 1.2: General structure of an NVIDIA GPU (GeForce-Maxwell 107 microarchitecture in particular): the three main blocks are the *Graphics Processing Cluster*, the *GigaThread Engine* and the L2 cache. (Image from [12])

Engine. Unlike other types of vector processors, SMs not only contain a large number of registers, but they also have a considerable number of ALUs: each SM is in fact made of an array of basic, scalar processing cores, termed *CUDA cores* (pictured as light green squares in figure 1.2 and 1.3), that constitute the lowest level of the hierarchical hardware structure: each thread generated by a *CUDA* program is eventually assigned to run on one of these cores. *CUDA cores* are physically and conceptually grouped in blocks of 32 units that share the same register file, the same instruction buffer and the same *warp*[30] scheduler. In some previous microarchitectures a *Streaming Multiprocessor* would consist of just one group of 32 cores, together with the corresponding extra resources and units, but in GeForce-Maxwell 107 microarchitecture there are four groups of these per SM, summing up to 128 cores per SM, resulting in turn in a total number of 640 *CUDA cores* in the GeForce GTX750Ti GPU (figure 1.2 and 1.3).
In any case, all *CUDA cores* have direct access to (and can communicate through) an SM-specific, on-chip and user-managed cache memory[31] termed *shared memory*. As it can be seen in figure 1.2 and 1.3, there can be other levels of cache memory, in addition to *shared memory*.

At the software abstraction level, there is another hierarchical structure which partially overlaps with the hardware one. The *CUDA* programming model is based on a *Single-Instruction Multiple-Thread* (SIMT) scheme for which a single function in

[30]In the *CUDA* jargon, *warps* are groups of 32 threads. See below.
[31]Cache memory was actually absent in older microarchitectures.

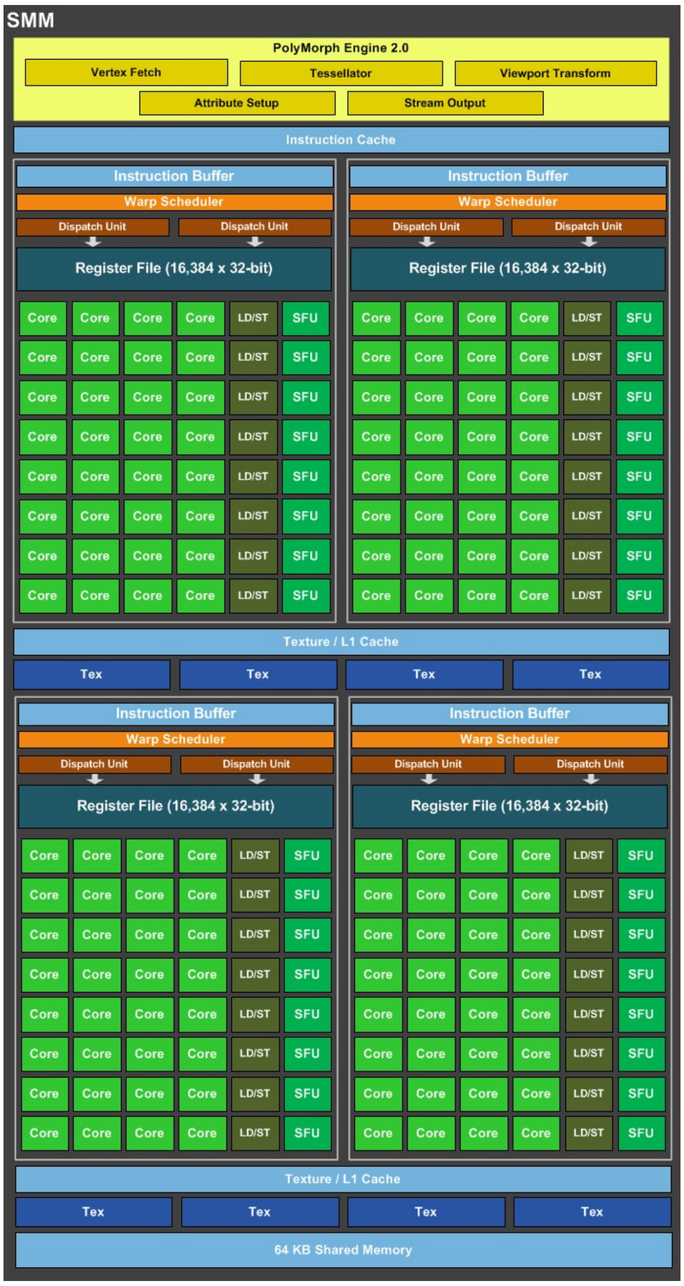

Figure 1.3: Internal structure of a *Streaming Multiprocessor* as in GeForce-Maxwell 107 microarchitecture: groups of 32 cores each share the same resources, including a *warp* scheduler, an instruction buffer, a large register file, some load/store units and some special function units. In Maxwell GPUs the *Streaming Multiprocessor* is called "SMM". (Image from [12])

a *kernel* (i.e., a self-contained block of source code) spawns a set of identical threads that are designed to execute the very same machine-level instructions on multiple data, in a SIMD fashion. Thus, each thread is ultimately assigned to a *CUDA core* on the hardware side. Threads are conceptually grouped in multidimensional *thread blocks* and each *thread block* is assigned to only one SM on the hardware side. Finally, *thread blocks* are in turn placed in a multidimensional *grid* for software abstraction convenience. The way this software model is built on top of the hardware model

allows for a great degree of code portability and scalability: the same code can run seamlessly on different NVIDIA GPU microarchitectures from very distinct generations and with a different number of cores, of course resulting in a varied performance[32].
A key concept of the *CUDA* programming model is the *warp*, i.e. a group of 32 threads that execute in lockstep in a SIMD fashion. *Warps* are assigned to the aforementioned groups of 32 cores and, because these share a single instruction buffer, a *warp* is the smallest unit of work a GPU issues[33]. Machine-level instructions belonging to a *warp* are executed simultaneously inside the SM, in a truly parallel way[34]. For full efficiency, it is advised that all threads in a warp have a single execution path: divergence via conditional branches will force each branch to be executed sequentially until these converge back into the same path. For this reason it is important to minimise divergent conditionals in the GPU code.

1.1.4 Heterogeneous Computing via the CPU-GPU Pair

As already pointed out, general-purpose computing on GPUs can be seen as a particular case of heterogeneous computing. In this case, there is a specific relationship between the two entities involved, the CPU and the GPU: the first has the role of the master, and it is typically called "host" in this scenario, while the second is operated as a slave device, and indeed it is called "device".
Under this model, serial sections of code, running on the host, are interspersed with parallel ones running on the device (figure 1.4). GP-GPU code is supposed to start execution on the host as a single-threaded process. As soon as some kind of vectorised parallel computation is needed, the host calls the device into play via a *kernel call*. Most of the times, the data on which calculations need to be carried out reside on host memory: when this is the case, the CPU also takes care[35] of transferring this data to the device memory by means of the DMA[36] controller. Then, through the *kernel call*, the host spawns and schedules a number of threads on the device, and these are always grouped in *warps*.

[32]A device with less SMs will need to assign more *thread blocks* to the same SM, hence increasing the level of *warp* competition for hardware resources and eventually resulting in less physical parallelism and longer execution times; conversely, a device with more SMs can assign less *thread blocks* to each SM, increasing the number of threads that are processed simultaneously and decreasing the execution time.

[33]As a result, problems that involve a number of elements which is not an integer multiple of 32 will still spawn a number of threads corresponding to the nearest multiple of 32 (rounded up): some of these threads simply will not do any work and they will occupy resources, resulting in a non ideal utilisation scheme.

[34]Usually, a *thread block* is made of more than one *warp* and this is why a *warp scheduler* unit is needed to direct the order of execution of different *warps*.

[35]Since the release of *CUDA 6* and the introduction of *Unified Memory*, this operation can be made in a transparent way to the programmer. From [14]: "Unified Memory creates a pool of managed memory that is shared between the CPU and GPU, bridging the CPU-GPU divide. Managed memory is accessible to both the CPU and GPU using a single pointer. The key is that the system automatically migrates data allocated in Unified Memory between host and device so that it looks like CPU memory to code running on the CPU, and like GPU memory to code running on the GPU".

[36]*Direct Memory Access*.

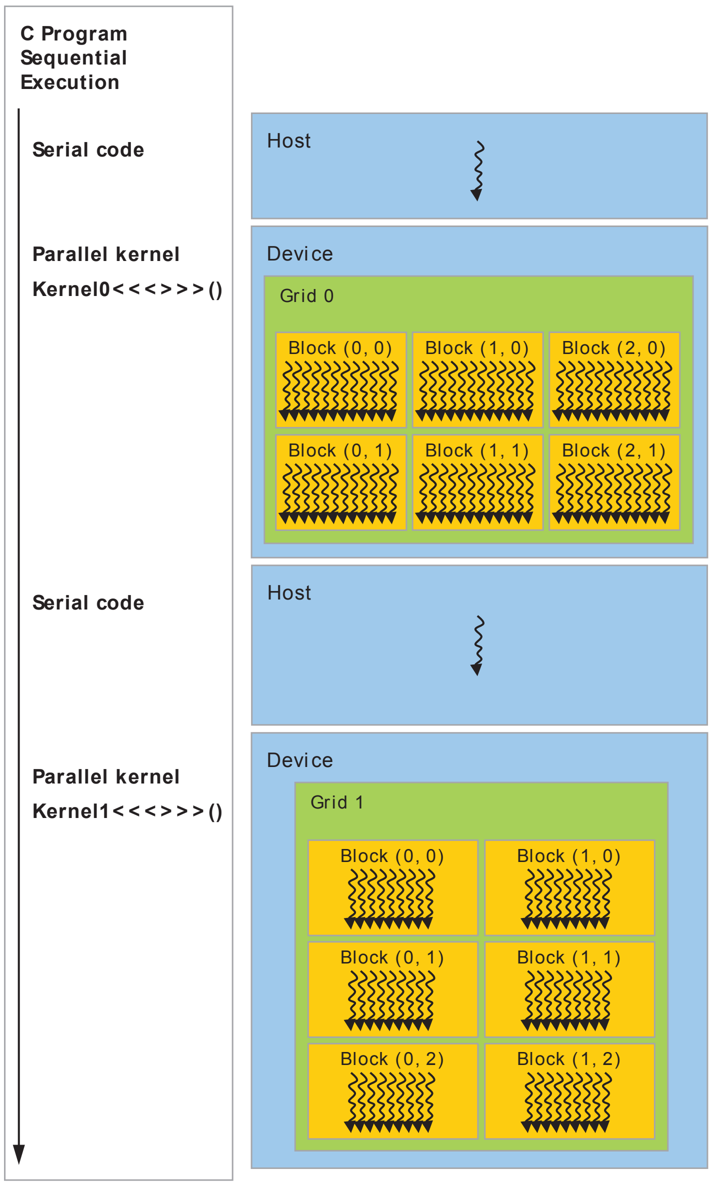

Figure 1.4: Heterogeneous programming model as in *CUDA*: each curly arrow represents a thread. (Image from [13])

As the CPU is waiting for memory transfers to complete and for vectorised operation to be carried out by the GPU, the former is actually free to manage other threads from other concurrent processes. In this way, the two processors are optimally exploited for what they can do best: on one hand the CPU works as a heavily multitasking general-purpose processor intended to make a lot of context changes; on the other hand, the GPU is employed for massively parallel operations on data arrays, harnessing its vectorial structure. In practice, not many algorithms are purely data parallel: typically, there is a need for branching as well as for some kind of communication among threads, which often implies the need for synchronisation barriers. This is all implementable in an SIMT model, and with *CUDA* in particular, but care must be taken so that the performance is not degraded unnecessarily.

1.2 The Phase Vocoder

The phase vocoder is a mathematical tool which is meant to provide an alternative representation of generic signals, even though it was originally developed specifically for speech modelling and it works best for a specific class of signals (see below).
This technique was in fact proposed by Flanagan and Golden in 1966 [15] as a way to enhance a pre-existing speech signals encoding method, the channel vocoder[37]. In this publication, the authors show how this method not only leads to a certain economy of transmission bandwidth for speech signals (with a relatively low impact on voice quality), but also provides a framework for time compressing and expanding the same kind of signals, while keeping the spectral content substantially intact[38]. The concepts of the phase vocoder were then developed further by many other researchers, and it eventually proved to be a powerful tool for many other processing tasks, not only for speech signals but also for other kinds of musical signals. Most prominently, this technique is employed for high fidelity time-scale modifications or pitch transposition of a wide variety of sounds, but it can actually provide for many other classes of spectral manipulation, as it basically allows to arbitrarily control individual sound harmonics.

The phase vocoder falls under the umbrella of *analysis-synthesis* techniques. The idea is that of modelling a signal $x_{(t)}$ as the sum of a finite number N of sinusoids whose instantaneous amplitudes and frequencies vary slowly with time. In particular, frequency values are expressed in terms of a set of fixed and harmonically related central frequencies $k\omega_0$ (all multiples of a fundamental frequency $\omega_0 = 2\pi/N$) and a set of time-varying frequency deviations $\Delta\omega_{k(t)}$:

$$\begin{aligned} x_{(t)} \simeq \sum_{k=0}^{N-1} x_{k(t)} &= \sum_{k=0}^{N-1} \rho_{k(t)} \cos(k\omega_0 t + \varphi_{k(t)}) = \\ &= \sum_{k=0}^{N-1} \rho_{k(t)} \cos\left(k\omega_0 t + \int_0^t \dot{\varphi}_{k(\tau)}\, d\tau\right) = \\ &= \sum_{k=0}^{N-1} \rho_{k(t)} \cos\left(k\omega_0 t + \int_0^t \Delta\omega_{k(\tau)}\, d\tau\right) \end{aligned} \tag{1.1}$$

In equation 1.1, $\rho_{k(t)}$ is the amplitude envelope in time for component k while $\varphi_{k(t)}$ is the corresponding phase envelope[39]. In musical terms, however, a more intuitive representation is given by extracting the time evolution of the frequencies of each component, $\tilde{\omega}_{k(t)}$. In this scenario, these frequencies are obtained by the sum of two components, namely bins' central frequencies (known a priori and static) and

[37]The name *vocoder* itself is indeed a contraction of the term "voice coder".

[38]Artefacts can be actually introduced by this process, and their intensity depends on the amount of time stretching required, by the nature of the original signal and by the settings used for the phase vocoder.

[39]Technically speaking, the true *phase envelope* $\mathit{\Phi}_{k(t)}$ would be the whole argument of the cosine function, thus $\mathit{\Phi}_{k(t)} = k\omega_0 t + \varphi_{k(t)}$. In this context, however, since the phase component related to the bins' central frequencies is fixed by design at $k\omega_0 t$, what really characterises phase envelopes is $\varphi_{k(t)}$. Thus it makes sense to "overload" the term *phase envelope* to indicate that component of $\mathit{\Phi}_{k(t)}$ that really determines frequency deviations from the bins' central frequencies, i.e. $\varphi_{k(t)}$.

time-varying frequency deviations:

$$\tilde{\omega}_{k(t)} = k\omega_0 + \Delta\omega_{k(t)} \ , \qquad k = 0, 1, 2, \ldots, N-1. \tag{1.2}$$

It can be shown that the frequency deviations are related to the phase envelopes by means of time differentiation:

$$\begin{aligned} \Phi_{k(t)} &= k\omega_0 t + \varphi_{k(t)} = \int_{-\infty}^{t} \tilde{\omega}_{k(\tau)} d\tau + \varphi_0 = \int_{-\infty}^{t} (k\omega_0 + \Delta\omega_{k(\tau)}) d\tau + \varphi_0 = \\ &= k\omega_0 t + \int_{-\infty}^{t} \Delta\omega_{k(\tau)} d\tau + \varphi_0 \ , \qquad k = 0, 1, 2, \ldots, N-1. \end{aligned} \tag{1.3}$$

$$\begin{aligned} \tilde{\omega}_{k(t)} &= \frac{d\Phi_{k(t)}}{dt} = \frac{d}{dt}(k\omega_0 t) + \frac{d\varphi_{k(t)}}{dt} = k\omega_0 + \frac{d}{dt}\left(\int_{-\infty}^{t} \Delta\omega_{k(\tau)} d\tau + \varphi_0 \right) = \\ &= k\omega_0 + \frac{d}{dt} \int_{-\infty}^{t} \Delta\omega_{k(\tau)} d\tau \ , \qquad k = 0, 1, 2, \ldots, N-1. \end{aligned} \tag{1.4}$$

Thus, solving equation 1.4 for $\Delta\omega_{k(t)}$ with respect to $\varphi_{k(t)}$:

$$\begin{aligned} k\omega_0 + \frac{d\varphi_{k(t)}}{dt} &= k\omega_0 + \frac{d}{dt} \int_{-\infty}^{t} \Delta\omega_{k(\tau)} d\tau \Rightarrow \\ \Rightarrow \Delta\omega_{k(t)} &= \frac{d\varphi_{k(t)}}{dt} \ , \qquad k = 0, 1, 2, \ldots, N-1. \end{aligned} \tag{1.5}$$

This representation is very significant from a musical point of view: in fact, psychoacoustics and physiological studies show that this description is closer to a perceptual model of the human hearing system, with respect to other representations (for instance, with respect to a basic time domain description or a straightforward Fourier transform description).
As with the channel vocoder, the original phase vocoder representation of [15] is based on the Fourier transform model but it deviates from the typical source/filter utterance generation model[40] used by other speech encoding techniques. In fact, the phase vocoder is designed to exploit the short-time phase spectrum of the input signal (as well as the short-time amplitude spectrum) in order to recover information on its harmonic nature; additionally, it does not need a supplementary pitch tracking process or a decision making rule for voiced or unvoiced segments as this information is somewhat embedded in the phase vocoder representation itself.

As it was shown in the previous paragraphs, the phase vocoder is based on a sinusoidal model assumption for the input signal. This means that, in order for this method to be effective, the input signal is required not to be too dissimilar[41] from a finite sum of sinusoids with time-varying amplitudes and time-varying frequencies

[40]The source/filter model consists of representing a speech signal as an alternately voiced (periodic) or unvoiced (aperiodic and "noisy") excitation signal that is fed into a dynamic system which is meant to model the vocal tract. Even though the phase vocoder technique does not explicitly consider this structure into its model, the very same concepts are somehow embedded in the phase vocoder itself, in an implicit way.

[41]Actually, as far as basic analysis/re-synthesis is concerned, good results can be achieved even when the input signal bears little relation to the sinusoidal model. Still, similarity to this model becomes crucial when any kind of alteration is intended in between analysis and re-synthesis stages [16].

that are well separated from each other. Yet, it is important to point out that these frequencies are not required to be harmonically related; indeed, the possibility for the phase vocoder to track and modify inharmonic partial components of a sound, as well as harmonic ones, is attractive for many applications.

As discussed in [16], the phase vocoder can be faced from two complementary (but mathematically equivalent) viewpoints: the *filterbank* interpretation and the *Fourier-transform* interpretation. In addition, each viewpoint can be handled by means of a few slightly different attitudes. In this chapter the phase vocoder will be explained only from a particular perspective, ultimately the one upon which the *Csound* implementation is based[42], as this is the one that will be eventually used in this project. This perspective falls under the *Fourier-transform* interpretation of the phase vocoder and it is of course based on a discrete-time/discrete-frequency digital model. It will be referred to as "STFT-based phase vocoder".

1.2.1 STFT-Based Phase Vocoder Analysis

The phase vocoder analysis stage can be explained using the *short-time Fourier transform* (STFT) as a starting point. This is a well known and widespread mathematical tool that consists of a succession of overlapping Fourier transforms taken over finite-duration windows in time. In the discrete domain of digital signals, the STFT $X_{[k,l]}$ of the time-domain signal $x_{[n]}$ at time instant l can be defined as:

$$X_{[k,l]} = \sum_{n=-\infty}^{\infty} w_{[n-l]} x_{[n]} e^{-j2\pi kn/N} \ , \qquad k = 0, 1, 2, \ldots, N-1. \tag{1.6}$$

This definition implies the use of a window function $w_{[n]}$ with limited support over a number of samples M which is usually set equal to the DFT size, i.e. the number of frequency bins N (thus, $M = N$ and this will be the assumption from now on). A handful of different shapes can be employed for the window function, depending on the desired specifications for the well known trade-off between frequency resolution of spectral peaks and bin-leakage/aliasing problems. Essentially, the window function carries out two different tasks: it is responsible for selecting a specific time segment of the input signal at each analysis step and, unless a rectangular window is chosen, it also takes care of eliminating discontinuities at its edges by means of smooth bell-like shapes, in order to minimise spectral smearing.
The analysis time instants l are not required to be as closely spaced as the time samples that make up the input signal: indeed, they are usually chosen in such a way that each window is overlapped with only a few other windows (at least one) and they are equally spaced in time. The number of samples between analysis instants is called *analysis hop size* and will be referred to by means of capital L[43]. The choice of this parameter is usually left to the user, together with the choice of the DFT size, and these parameters strongly depend on the nature of each specific application.

[42]This perspective also coincides with the one used in Chapter 9 of *The Audio Programming Book*, [17].

[43]As a result, l can only have values that are multiple of L.

By comparison to the *filterbank* interpretation[44], rather than putting emphasis on the temporal evolution of magnitude and phase (or frequency) values in each filter band, this STFT arrangement focuses attention on the complex spectrum values for all of the different filters bands (i.e., frequency bins) at a single point in time. In reality, these are only two different viewpoints for the same underlying concept.

Starting from the STFT representation, a few more steps are needed in order to get to the phase vocoder representation described above. Namely, amplitude and frequency functions of time for each component are desired. The former are simply obtainable by converting each complex spectrum into polar coordinates; the latter can be obtained via an extra step: phase differentiation over time. This step is referred to as *frequency tracking*, because it provides, step by step, the time-varying deviation of each component from the central frequency of the DFT bin to which it is associated.
Before proceeding to these steps, it is advisable to rewrite equation 1.6 in a way that is closer to what can be interpreted by a digital machine. In particular, it would be wise to lead back this formulation to the one used for applying fast algorithms (FFTs) any time a DFT is required.
Thanks to the limited support of the window function, the infinite summation of equation 1.6 can be reduced to a sum of N elements:

$$\begin{aligned} X_{[k,l]} &= \sum_{n=-\infty}^{\infty} w_{[n-l]} x_{[n]} e^{-j2\pi kn/N} = \\ &= \sum_{n=l}^{l+N-1} w_{[n-l]} x_{[n]} e^{-j2\pi kn/N} \;, \qquad k = 0,1,2,\ldots,N-1. \end{aligned} \tag{1.7}$$

In order to lead this back to a summation from zero, a change of variables is applied, namely $n = m + l \Rightarrow m = n - l$:

$$\begin{aligned} X_{[k,l]} &= \sum_{m=0}^{N-1} w_{[m]} x_{[m+l]} e^{-j2\pi k(m+l)/N} = \\ &= e^{-j2\pi kl/N} \sum_{m=0}^{N-1} w_{[m]} x_{[m+l]} e^{-j2\pi km/N} \;, \qquad k = 0,1,2,\ldots,N-1. \end{aligned} \tag{1.8}$$

Now the last summation appearing in 1.8 exactly represents the DFT of a signal portion (properly reshaped by the window contour) and can be computed via an FFT algorithm. Since, for the shift theorem of the DFT, the presence of the exponential phase term preceding the summation is equivalent to shifting the input samples circularly to the right according to $l \mod N$, rather than actually computing the multiplication by this complex exponential, the input (windowed) signal $s_{[n]} = w_{[n]} x_{[n+l]}$ can actually be shifted before computing the DFT:

$$X_{[k,l]} = e^{-j2\pi kl/N} DFT\{s_{[n]}\} = DFT\{\tilde{s}_{[n]}\} \tag{1.9}$$

[44]In a nutshell, this consists of analysing the input signal by feeding it into a parallel set of band-pass filters centred at equally spaced frequencies throughout the audible spectrum. From the resulting signals, the amplitude and phase (or frequency) time envelopes are extracted. These will be eventually processed (if needed) and used for re-synthesise an output signal. See [18] and [16] for more details.

, where $\tilde{s}_{[n]}$ is the circularly shifted version of $s_{[n]}$ (by $l \mod N$ samples to the right).
For the sake of more elegant syntactics, a different index t can be used instead of l for identifying the slower time rate of analysis windows with respect to the sample rate: $t = l/L$. In this setting, the STFT of a signal $x_{[n]}$ becomes $X_{[k,t]}$, with t increasing of just one unit at each analysis step.
Back to the problem of extracting the amplitude and frequency time functions for each component, let $A_{[k,t]} = Re\{X_{[k,t]}\}$ and $B_{[k,t]} = Im\{X_{[k,t]}\}$ be the real and imaginary part of the STFT representation of equation 1.6:

$$X_{[k,t]} = A_{[k,t]} + jB_{[k,t]} \tag{1.10}$$

Then, employing a heuristic approach under the assumption that the amplitudes $\rho_k[t]$ are "slowly varying"[45], the amplitude functions of time for each component k of the original signal $x_{[n]}$ can be simply extracted from the the magnitude of $X_{[k,t]}$:

$$\rho_{k[t]} \simeq |X_{[k,t]}| = \sqrt{A^2_{[k,t]} + B^2_{[k,t]}}\ , \qquad k = 0, 1, 2, \ldots, N-1. \tag{1.11}$$

Here, the time evolution is set by the t variable, which is increased by one every hop size samples L: for higher time resolution a lower analysis hop size parameter of the STFT scheme needs to be chosen, and vice versa.
In order to generate the frequency trajectories, the phase envelopes are needed first:

$$\varphi_{k[t]} \simeq \arg(X_{[k,t]}) = \arctan\left(\frac{B_{[k,t]}}{A_{[k,t]}}\right)\ , \qquad k = 0, 1, 2, \ldots, N-1. \tag{1.12}$$

Again, equation 1.12 holds only under the assumption of slowly varying phases and, again, this is a heuristic approach that does not always guarantee perfect reconstruction[46] (see the section 1.2.3).
Then, in order to obtain the frequency deviations, time differentiation of these function needs to be carried out (equation 1.5). In the digital domain this is usually approximated by means of a finite difference between phase values of consecutive frames[47] (*backward Euler* method):

$$\Delta\omega_{k[t]} \simeq \frac{\varphi_{k[t]} - \varphi_{k[t-1]}}{L}\ , \qquad k = 0, 1, 2 \ldots, N-1. \tag{1.14}$$

[45] "Slowly varying" means that the amplitude envelopes are relatively constant over the time window over which the DFT is taken. For a more detailed explanation of the nature of this heuristics see [19] under the section "Short-time Fourier transform of a sinusoidal signal".

[46] Nevertheless, these approximations offer some powerful tools for spectral processing and sound encoding, at least from a perceptual point of view.

[47] An alternative solution could be that of explicitly differentiate equation 1.5:

$$\begin{aligned}\Delta\omega_{k[t]} &= \frac{d\varphi_{k[t]}}{dt} = \frac{d}{dt}\arctan\left(\frac{B_{[k,t]}}{A_{[k,t]}}\right) = \frac{1}{1+\left(\frac{B_{[k,t]}}{A_{[k,t]}}\right)^2}\frac{d}{dt}\left(\frac{B_{[k,t]}}{A_{[k,t]}}\right) = \\ &= \frac{A^2_{[k,t]}}{A^2_{[k,t]}+B^2_{[k,t]}}\left(\frac{\dot{B}_{[k,t]}}{A_{[k,t]}} - B_{[k,t]}\frac{\dot{A}_{[k,t]}}{A^2_{[k,t]}}\right) = \frac{A_{[k,t]}\dot{B}_{[k,t]} - B_{[k,t]}\dot{A}_{[k,t]}}{A^2_{[k,t]}+B^2_{[k,t]}} = \\ &\simeq \frac{1}{L}\frac{A_{[k,t]}(B_{[k,t]} - B_{[k,t-1]}) - B_{[k,t]}(A_{[k,t]} - A_{[k,t-1]})}{A^2_{[k,t]}+B^2_{[k,t]}}\ , \qquad k = 0, 1, \ldots, N-1.\end{aligned} \tag{1.13}$$

The result of this operation needs to be brought down to principal values (in the interval between $-\pi$ and π). This operation is called *wrapping*.
After phase difference wrapping, the true frequency values[48] are computed by simply adding the corresponding bin frequencies to each frequency deviation obtained by means of equation 1.14:

$$\tilde{\omega}_{k[t]} = k\omega_0 + \Delta\omega_{k[t]} \ , \qquad k = 0, 1, 2, \ldots, N-1. \tag{1.15}$$

Sometimes, it is preferred to express frequencies in hertz rather than in digital terms (i.e., in radians). To do so, it suffices multiplying $\Delta\omega_{k[t]}$ by $\frac{F_S}{2\pi}$, where F_S is the sampling rate employed in the system.

To sum up, the steps involved in phase vocoder analysis are:

1. Extract N samples from a signal and apply an analysis window.
2. Rotate the samples in the signal frame according to the analysis time instant $l \mod N$.
3. Take the DFT of the resulting array of real values (possibly exploiting an FFT algorithm).
4. Convert the obtained complex spectrum from rectangular to polar format.
5. Compute the phase differences and bring these values to principal values (wrapping).
6. Add the obtained difference values to the bands' central frequencies and convert to hertz.

Thus, phase vocoder analysis results in a time stream of N amplitude values (from step 4) and N frequency values (from step 6). Each pair of values is related to one of the N components that make up a sound in the sinusoidal model of equation 1.1. In this way an important goal is achieved, namely that of separating temporal information from spectral information. The resulting benefits are truly precious when high fidelity temporal and/or spectral manipulations are desired.

1.2.2 ISTFT-Based Phase Vocoder Re-Synthesis

After a signal has been expressed in terms of the phase vocoder representation provided by equation 1.1, any kind of manipulation can be applied (many of these will be actually discussed in Chapter 3). When all the processing has been performed[49], the obtained phase vocoder data usually needs to be brought back to a time domain representation for playback or storage. To do so, a variety of methods can be applied and most of these are derivatives of the two possible views of the phase vocoder: on one side, an additive synthesis framework can be employed by means of a bank of oscillators driven by phase vocoder data (acting as time-varying

[48]For completeness, it is due to point out that an alternative method for extracting the true frequency values from STFTS is also widespread. This is the *Instantaneous Frequency Distribution* algorithm proposed by Toshihiko Abe in [20].

[49]Of course, in a speech encoding scenario there can be no need for any processing. Still, a re-synthesis stage at receiver side needs to be performed. The steps involved are the same, regardless of the nature of the application.

amplitude and frequency controls); on the other side, inverse Fourier transforms overlapping in time are employed (ISTFT). The latter is usually a more efficient method, thanks to the possibility of using fast algorithms for computing inverse Fourier transforms, and it is the one that will be discussed here.

In the ISTFT re-synthesis scenario, all the steps discussed for the analysis stage need to be retraced back.
To begin with, the frequency values need to be transformed back to phase values. To do so, the bin frequencies are subtracted from $\tilde{\omega}_{k[t]}$ values and the result is integrated over time:

$$\Delta\omega_{k[t]} = \tilde{\omega}_{k[t]} - k\omega_0 \ , \qquad k = 0, 1, 2, \ldots, N-1. \tag{1.16}$$

$$\varphi_{k[t]} = L \sum_{\tau=0}^{t} \Delta\omega_{k[\tau]} = \arg(X_{[k,t]}) \ , \qquad k = 0, 1, 2, \ldots, N-1. \tag{1.17}$$

Then, the STFT data needs to be converted back from polar to rectangular coordinates:

$$\begin{aligned} A_{[k,t]} &= Re\{X_{[k,t]}\} = |X_{[k,t]}| \cos(\varphi_{k[t]}) \\ B_{[k,t]} &= Im\{X_{[k,t]}\} = |X_{[k,t]}| \sin(\varphi_{k[t]}) \end{aligned} \tag{1.18}$$

Now, the inverse DFT of each frame can be taken in order to obtain time-domain frames that partially overlap in time:

$$x_{t[m]} = \frac{1}{N} \sum_{k=0}^{N-1} X_{[k,t]} e^{j2\pi km/N} \ , \qquad \begin{aligned} &m = 0, 1, 2, \ldots, N-1. \\ &t = 0, 1, 2, 3, \ldots \end{aligned} \tag{1.19}$$

Each of these $x_{t[m]}$ frames is meant to be placed on the time axis having the first sample at time instant t[50].
Then, the obtained frames need to be cyclically rotated in the opposite direction with respect to what was done in the analysis stage, hence to the left (again, the shift amount is $l \mod N$ if no time-scaling modification is applied, but this may vary).
Finally, a standard overlap-and-add method is employed to obtain the full time-domain signal:

$$x_{[n]} = \sum_{t} w_{[n-tI]} x_{t[n-tI]} \qquad \forall n \tag{1.20}$$

, where $w_{[n]}$ is a windowing function with a support of N samples (from $n = 0$ to $n = N-1$) and $x_{t[n]}$ is the "extended version" of $x_{t[m]}$ such that it is defined to be zero-valued outside of the same interval. I is the *re-synthesis hop size.* The re-synthesis stage is thus complete.

To sum up, the steps involved in phase vocoder re-synthesis are:

1. Bring the frequency values back to frequency deviations by subtracting them from the bins' central frequencies.

[50]Actually, this is true only if no time scaling is desired: for time scaling operations, a different spacing between successive frames (with respect to the analysis hop size L) has to be employed. In these cases, a *synthesis hop size* $I < L$ is needed for time compression and $I > L$ for time expansion.

2. Convert frequency deviations back to phase values by accumulating them over time and scaling them by the analysis hop size[51].
3. Perform a polar to rectangular conversion.
4. Take the IDFT of the obtained spectral data (possibly exploiting an FFT algorithm).
5. Left-shift (circularly) the obtained frames.
6. Overlap-and-add consecutive frames using re-synthesis windows.

1.2.3 Limitations of the Phase Vocoder

In spite of the many successes of the phase vocoder as it has been presented here, numerous problems have limited its use. Still, many of these problems can be solved by means of improved techniques that are based on the same scheme (see [18]).

The main issue is the already mentioned need for the input to be close enough to the sinusoidal model. This requirement is not always met. The best kind of sounds are the so-called *quasi-periodic* signals like steady speech vowels and sustained musical notes. Noise-like signals and transients (i.e., impulsive sounds like musical attacks and speech plosives) are less congenial to phase vocoder analysis and processing. They can be still approximately represented by means of a sum of sine waves to some extent but they are more likely to produce artefacts when re-synthesised. Especially (but not only) for these kinds of sounds, enhanced techniques have been developed in the last decades (see [18]).
Assuming that enough frequency resolution is achieved when computing DFTs, such that there is a sufficient number of bins to track all the sinusoidal components of a sound, there are still two critical issues that a basic phase vocoder fails to address. Firstly, even though each partial is varying its frequency slowly enough, it can always happen that, as they move in the audio spectrum, two sinusoidal components end up in the same DFT bin. Unfortunately, phase vocoder analysis will be able to resolve a maximum of one sinusoidal component per frequency band. Thus, this method will fail to output the right values for the amplitudes and frequencies of each conflicting partial. Instead, the result of the analysis will be an amplitude-modulated composite output, in many ways similar to beats. Also, a particular sine wave may not be adequately estimated when it falls between two adjacent DFT bins.
Secondly, amplitude and frequency values of one ore more partials may vary too rapidly. This results in violating the assumptions that were made in section 1.2.1 and perceivable artefacts are likely to sprout in the re-synthesised signal.

Yet another problem is implicitly brought about by the STFT framework[52]: the universal issue of the time-frequency resolution trade-off. Large analysis windows are needed for accurate frequency resolution but this is in contrast with the need for correctly analysing short sound events or signals with rapidly varying parameters. Again, some variations of the phase vocoder approach have been proposed in the past to address this issue ([18]).

[51]If time scaling is desired, phase values also need to be scaled further by the scaling ratio ([16]).
[52]Yet, this problem is embedded in the *filterbank* approach with no difference.

Despite the problems that have been highlighted in this section, the phase vocoder remains an extremely powerful tool for sound manipulation, even in its basic form. As a matter of fact, by means of this technique, an outstanding array of processing tasks can be applied to a variety of sound sources with high fidelity.

1.3 Conclusions

This chapter provides the background information that is needed in order to fully understand the development phases of this project. The principles of GP-GPU computing were presented in the first half of the chapter together with basic information about GPU architectures (with a brief historical *excursus*) and the related software abstractions (mainly focusing on NVIDIA processors). This information is essential for the development of *CUDA*-based applications.
In the second half of the chapter, the phase vocoder mathematical model for audio signals was presented and discussed, focusing on the steps involved in the conversion from a time domain signal to a phase-vocoder domain one (and back). Also, the phase vocoder's potentials were reviewed, together with its limitations. In chapter 3 these two topics will be brought together for the development of phase-vocoder-based spectral manipulation programs that are designed to harness the computing power of *CUDA*-enabled GPUs. In fact, as it has been shown in the phase vocoder section of this chapter, the mathematical passages that make up phase vocoder analysis and re-synthesis are very congenial to a parallel processor. As a matter of fact, they comprise FFT analysis and synthesis stages (which can be efficiently computed in a parallel fashion[53], as discussed, for instance, in [23], [24] and [25]) and a few other operations that can be carried out independently at bin level. This is exactly what encouraged Lazzarini et al. to carry out their analysis in [8]. In addition, most typical manipulation algorithms that are based on a phase vocoder representation can be easily implemented in a parallel fashion as well. This very topic will be addressed in chapter 3.

[53]In a *CUDA* scenario, this is typically done via the cuFFT [21] library but other implementations also exist (see [22]).

Chapter 2

GP-GPU Computing for Audio Applications: a Review

What follows is a brief review based on the scientific literature that have shaped the field of GP-GPU computing for audio applications. The author does not claim, and does not even presume, this review to be a complete inspection of the literature at all: even though the birth of this field was relatively recent (it first sprang around the year 2005), a remarkable number of scientific publications has been already published and a complete covering of this topic would require more than one chapter (besides, it is not the main purpose of this work). In fact, this field has received a lot of attention from many researchers in the scope of audio computing: on one side the GP-GPU method can give outstanding results and opens up a whole new scene of possibilities and applications that were not even possible before; on the other side, it presents intrinsic challenges that are hard to overcome and need to be investigated. These issues are especially related to the limitations implied by the need of transferring data to the GPU and to the fact that some basic audio signal processing operations do not fit very well the parallel computation scheme.
In this review, it will be shown how the GP-GPU model has been explored by researchers for a wide variety of different applications and how their approaches have changed during the years, the critical shift occurring after the introduction of the *unified shader architecture* (see section 1.1.2) and that of GP-GPU APIs, *in primis CUDA*. A few examples from each application field are thus examined and their results are reported and compared to each other. This overview will also be useful to understand the typical issues involved in GP-GPU computing for audio applications and to learn useful tricks in preparation for the implementation of GPU-enhanced spectral processing programs.

2.1 Early Examples of GP-GPU Computing for Audio Applications

In order to show the methodologies used in the early years of GP-GPU computing, and the power gained by exploiting parallel processing even on older architectures (i.e., on the hardware graphics pipeline), a few studies from the pre-*CUDA* period are going to be reported next.

A Brief Description of the Graphics Pipeline

In order to fully understand the working principles behind some of the earlier works that will be described in this review, a basic knowledge of the graphics pipeline is needed. What follows is a brief description[1] of this conceptual model (and its hardware counterpart).

The graphics pipeline, as it is used (actually, as it *was* used) by GP-GPU programmers, can be roughly divided into six stages: the *vertex processor*, the *primitive assembly* stage, the *clipping-and-culling* stage, the *rasterization* stage, the *fragment processor* and the *raster operation* stage ("ROP"). The first three stages are conceptually related to geometric operations, while the last three are related to rendering. Typically, only a few of these stages are (were) actually programmable on hardware implementations and only two are actually useful for non-graphics purposes: the *vertex processor* and the *fragment processor.* The programmability of these elements is somewhat limited if compared to a general-purpose CPU and, still, all computations need to be expressed in graphics terms.

The *vertex processor* handles geometric transformations and lightning: it receives *vertices* (i.e., points) represented in homogeneous coordinates as four-dimensional arrays (for 3D scenes) and applies a transformation so that the position in the object coordinate system is translated in the eye coordinate system, according to the position of the eye:

$$\begin{bmatrix} x_o \\ y_o \\ z_o \\ w_o \end{bmatrix} = M_{4\times4} \begin{bmatrix} x_i \\ y_i \\ z_i \\ w_i \end{bmatrix} \tag{2.1}$$

, where x_i, y_i, z_i, and w_i are the input coordinates, x_o, y_o, z_o, and w_o are the output coordinates and M_{4x4} is the trasformation matrix. The fourth dimension is needed in order to exploit a *projective geometry* framework rather than an *Euclidean geometry* one: this allows affine transformations and, in general, projective transformations to be easily represented by matrices. The colour of the new vertices is computed from position and intensity of light sources, position of the eye, vertex normal and, of course, original colour.

In the *primitive assembly* stage, *vertices* are assembled into *triangles* and/or *lines* (when needed), in preparation for further operations.

In the *clipping-and-culling* stage, *vertices* outside the viewing frustum are discarded and *triangles* which face away from the eye are tagged so that they can be treated differently (or even discarded in some cases).

In the *rasterization* stage, the *primitives* that make up the virtual scene (i.e., *triangles*, *lines* and *points*) are finally projected on a plane, thus converted into a set of *fragments*, i.e. data that describes the characteristics of a *pixel* and its relations with the scene which is being rendered (not yet a simple color value to be displayed). For *lines* and *triangles*, the color values of single *vertices* are interpolated to give smooth transitions.

In the *fragment processor*, any kind of operation can be applied to *fragments* in

[1]The information that is reported in the following paragraphs was mostly taken from "A Crash Course on Programmable Graphics", by Li-Yi Wei ([26]).

order to obtain a new set of modified *fragments* (typically: *texture mapping*). This is of course the most important stage for general-purpose computing.

The *ROP* brings all the information held by each *fragment* down to a simple color value assigned to a *pixel*. In this way, a 2D picture is rendered. The *pixel* value information is eventually sent to the *frame buffer* for display.
Before the introduction of the *unified shader architecture* and that of general-purpose programming APIs, the *vertex processor* and the *fragment processor* could only be programmed by means of specialised language extensions and compilers. Some strong programming limitations were imposed by GPU processors. Still, most of the main features needed for audio processing were available on older GPUs: the ability to perform conditional branching (and, consequently, looping), the ability to read and write to and from memory locations (in a *ping-pong* fashion), and, finally, the availability of a set of highly efficient vector and trigonometric operations, which are very convenient for parallel DSP applications. Of course all these features can also be exploited by modern GP-GPU APIs while releasing the software developer from the burden of translating the addressed processes into the graphics rendering scheme, with all the limitations and complexities implied. All the examples that follow are based on NVIDIA's *Cg* language for GPU programming.

Whalen - 2005

An early example (March 2005) can be found in Sean Whalen's paper "Audio and the Graphics Processing Unit" [27], where a few digital audio effects were implemented as *Cg* kernels and executed on a GPU. Namely, these effects are simple versions of chorus, dynamic range compression, delay, FIR high-pass filtering, FIR low-pass filtering, noise gate and amplitude normalization. Being one of the first studies on GPU audio processing, the author chose an off-line framework to begin with, as he was concerned about the limited bandwidth of the AGP bus being not enough for real-time operations.
In Whalen's program (and in the other examples described next) digital audio elements and processes are translated into graphics *primitives* and graphical operations, i.e. into rendering terms: 32-bit floating-points mono samples are loaded into the red channel only of a square *texture* in order to be processed (the full RGBA space was not used, in order to simplify translating between a 1D sample array and a 2D *texture*). Very little information is given about the way kernel operations are performed on the GPU and the reader is invited to refer to later examples for a better understanding of typical methods.

Tests were run on a 105000 samples audio file (16-bit short PCM converted to 32-bit floats before processing in order to avoid distortion). The system used for the benchmarks consisted of a 3GHz Pentium 4 and an NVIDIA GeForce FX 5200 / AGP, accelerated by NVIDIA's kernel driver 6629 and *Cg* 1.3. The execution times of the program computing different audio effects were compared with a CPU version of the same program written in *C*. Results are reported in figure 2.1.
Although the GPU scores better in overall execution time, it is actually outperformed by the CPU in more than half of the algorithms. In fact, these results reinforce the notion that GPU performance is strongly dependent on the specific algorithm that is analysed and its particular implementation (i.e., whether it fits the GPU architecture

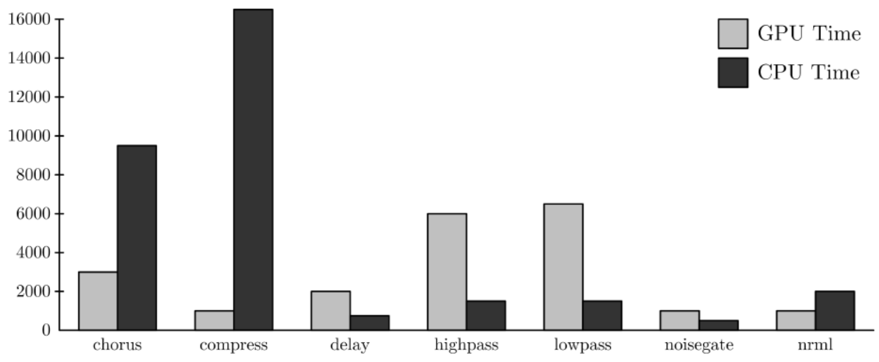

Figure 2.1: Results obtained in [27] for a CPU versus GPU comparison (execution time in microseconds). Seven basic audio effects are considered and 105000 audio samples are processed in each experiment.

well or not). Since very little information is provided on the implementation details of these algorithms, it makes little sense trying to carry out a deeper analysis of Whalen's results and trying to compare the algorithms that were tested. Nevertheless, the author himself points out that the main reason for the significantly poor performance of the GPU versions of the high-pass and low-pass filtering algorithms (implemented through two 5-tap masks) is to be found in the particular way data needed to be mapped from one-dimensional arrays to 2D textures. Noteworthy, this is a problem that is completely absent in modern GPUs. In all other tests, GPU performance is either comparable or remarkably better[2]. About the possibility of porting his program to a real-time scenario, Whalen states that: "Real-time effects processing on the AGP bus seems unlikely. To achieve low latency, short chunks of audio must be sent to the GPU and back to system memory. The readback performance of the AGP bus will likely limit GPU acceleration to offline audio processing. PCI-X should remove this limitation". Indeed, many examples of real-time audio GPU processing applications have been developed in the following years.

Smirnov and Chiueh - 2005

In [29], Smirnov and Chiueh design and implement a GPU algorithm for basic FIR filtering and test it in the scope of audio processing. In this work, the authors use NVIDIA's *Cg* compiler in order to program fragment processors on a GeForce 6600 GPU. They embed the resulting codes in already existing *GNU Radio*[3] building blocks. Smirnov and Chiueh used their FIR filter implementation in order to compute:

[2]The best performance is achieved on the dynamic range compression algorithm, where an impressive speed-up factor is brought about. This is a peculiar result, especially if compared to the much less exciting results obtained in [28] (see section 2.7). This difference can be explained by the fact that countless different versions of compression algorithms can be implemented, based on very distinct designs. Whalen's and Fabritius' versions are very likely based on incomparable designs.

[3]*GNU Radio* (`http://gnuradio.org/`) is a free and open-source software development toolkit that provides signal processing blocks to implement software radios.

- simple FIR filtering of a real input signal $x_{[n]}$:

$$y_{[n]} = \sum_{j=0}^{l} x_{[n+j]} h_{[j]} \tag{2.2}$$

, where h are the taps of the desired impulse response.

- the *Hilbert transformation* of a real input signal $x_{[n]}$:

$$\begin{cases} \Re(y_{[n]}) = x_{[n+k/2]} \\ \Im(y_{[n]}) = \sum_{j=0}^{l} x_{[n+j]} h_{[j]} \end{cases} \tag{2.3}$$

, where h are specially defined taps.

- the *frequency translating FIR filter* of a real input signal $x_{[n]}$:

$$y_{[n]} = \left(\sum_{j=0}^{l} x_{[n+j]} h_{[j]} \right) e^{2\pi f n} \tag{2.4}$$

, where f is the desired amount of frequency shift.

Vector elements (both input/output ones and impulse response ones) are stored in textures: each pixel contains either four real numbers or two complex numbers $(Re_n, Im_n, Re_{n+1}, Im_{n+1})$ and consecutive samples are stored sequentially along rows. The computations needed for evaluating equation 2.2 are carried out in the time domain, without the use of FFTs. Yet, a convenient matrix representation is employed in order to reduce the number of GPU instructions (see [29] for details).

The performance of the three processes was tested on a Pentium 4 HT 3.2GHz with a GeForce 6600 video card. The GPU implementation was compared with an SSE-optimized CPU version of *GNU Radio*. The results of testing the plain FIR filter are shown in figure 2.2.

These results imply that the GPU implementation outperforms the CPU implementation for vector sizes greater than 60000. This is a typical behaviour of GPU-based programs: a substantial improvement over the CPU is only observed when the desired process is applied to a large number of data. To confirm this, Smirnov and Chiueh report that the GPU-based Hilbert transformation program, which involves a short impulse response of only 31 taps, performs much worse than its CPU-based counterpart. It turns out that the plain FIR filter and the frequency translation FIR filter (which also performs well on the GPU) are computation-bound programs. Thus, being also congenial to parallel computation, they are well suited for GP-GPU processing. On the other hand, the Hilbert transformation is a memory bandwidth-bound program and it fails to give any improvement over a standard CPU implementation.

All the three developed modules are finally put together in a more complex "radio receiver" application (see the paper for details): a comparison is made between the time it takes to generate 500 output samples on by means of CPU and with a GP-GPU approach. The results are shown in figure 2.3.

These results again corroborate the conclusion that implementing certain blocks on the GPU can improve the application performance, especially for FIR filters

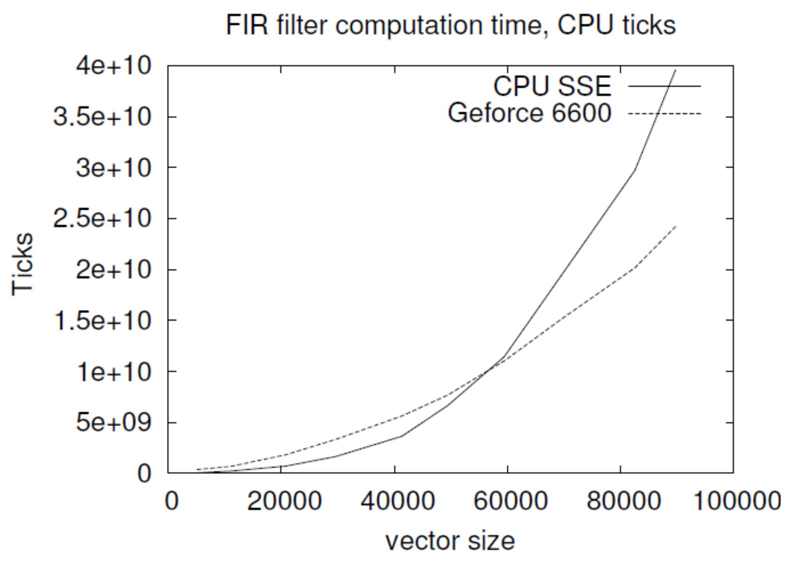

Figure 2.2: Performance comparison of CPU and GPU implementations of a plain FIR filter with a number of taps equal to the number of input (and output) elements. Results from [29].

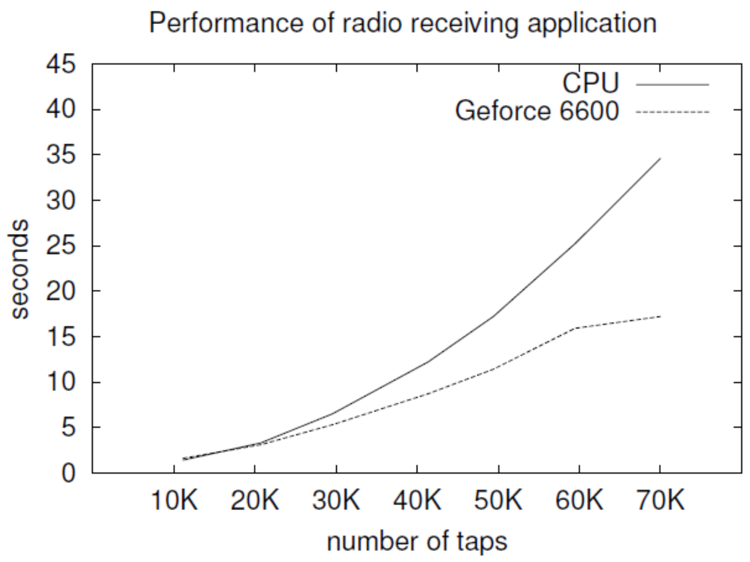

Figure 2.3: Performance comparison (CPU vs GPU) of a radio receiver application for different number of taps used for the internal FIR filters. The measured quantity is the time to generate 500 output samples. Results from [29].

with a larger number of taps. Of course this whole framework is not addressing the possibility of real-time execution, even with low sample rates. The hardware used in this study is not powerful enough for being able to filter in real-time with impulse responses as long as those considered here. Yet, more recent studies show a significant improvement and they prove that real-time FIR filtering with long impulse responses is possible and it can be faster if computed on a GPU (see [28], section 2.7, for instance). Current GPUs are quite different than at that time and the break-even point has shifted towards much shorter filter lengths with respect to the 60000 taps reported here. FIR filtering is a key functional block for countless audio-related applications: this review will be dealing with many more examples of this computational block.

Trebien - 2006

The real-time framework is addressed, for example, in Fernando Trebien's undergraduate thesis [30], which features a description of graphics-pipeline-level audio processing in the GPU. Trebien implements an embryonic prototype of a GPU-based modular audio system designed for real-time operation. His aim is to show how the limitations of CPUs in handling computationally intensive audio processing tasks (especially when multiple processing chains are meant to be executed concurrently) can be overcome by running certain processes on a GPU. The objective is therefore quite similar to that of this work but, while Trebien starts from scratch, building a whole audio system based on RtAudio[4] and ASIO[5] drivers, this work is arranged as an integration into the *Csound* environment and focuses on very specific spectral processes as opposed to the basic but fundamental building blocks implemented by Trebien.

The author sets up a mapping from a network model of virtually interconnected software modules to the graphics pipeline. This operation was accomplished via the *OpenGL* graphics library as an accessory to *C++* and *Cg* programming languages, where *Cg* is used for writing *shader programs*. It is noteworthy to stress that the author cites a few GP-GPU "systems" like *Brook* (a programming language for stream processing), *Sh* and *Shallows* (*C++* libraries), but he states that "none of them is an established standard, and all of them are still under development. In any case, using any of those systems is limiting to some extent and generally less efficient than direct programming of the graphics system". Trebien loads audio sample values into *texels* (mapping samples from different audio channels of multi-channel streams onto different color components of a single *texel*) and then posts tasks for computation on the GPU (typically, the task would be a horizontal *line* for an audio frame). Memory indexing for reading source *textures* and writing result *textures* is performed in a *ping-pong* fashion over successive *shader* calls: this is because of an intrinsic limitation in old GPUs for which a *texture* could not be set as source and target at the same time.
The modules that make up the prototype audio system implemented by Trebien are the following:

- A waveform synthesis program which is able to generate four different types of waveform by evaluating four different functions at successive time instants (sine wave, sawtooth wave, square wave, and triangle wave *shaders*).
- A mixing program to sum different *lines* in *textures*, i.e. mixing audio streams into one.
- An interpolating wavetable synthesis program.
- A delay effect program.
- A FIR filter program.

[4] *RtAudio (McGill University)*: `https://www.music.mcgill.ca/~gary/rtaudio/`
[5] *Steinberg ASIO*: `http://www.steinberg.net/en/company/developers.html`

No IIR filter programs were implemented because of their intrinsic recursivity, meaning that a straightforward implementation as a *shader* would not benefit from the GPU architecture[6].

Benchmarking of this system was carried out on an Athlon64 3000+ (1.81GHz) with 1GB of memory paired with an NVIDIA GeForce 6600 with 256MB of memory. Tests were run with stereo blocks of 256 32-bit samples at a sampling rate of 48kHz. Unfortunately, only the waveform synthesis module was actually tested: the author counted the maximum number of times this program could be invoked in real-time without causing buffer under-run in order to assess the application behaviour on practical usage. He compared this data with the results obtained by running similar programs on the CPU and consequently obtained a *speed-up factor* for each type of waveform. The *speed-up factors* reported by Trebien are the following:

- 28x for the triangle waveform
- 30x for the sinusoid waveform
- 36x for the sawtooth waveform
- 59x for the square waveform

These are exceptional results, perhaps too optimistic. In fact, analyzing the results of other studies regarding real-time audio processing on the GPU, speed-up factors as high as these can be hardly found. In addition, the video card used for these experiments hosted a medium-range GPU with only 8 *fragment processors*. These facts may lead to believe that in Trebien's experiments there might have been some kind of bias in favor of GPU computing. In any case, there is no reason to question that at least a good level of speed-up was achieved. Of course these results only apply to waveform synthesis by function evaluation, which is a very limited sub-scope of audio signal processing, but the framework in which they were obtained can be easily extended to many other applications, as this review demonstrates by illustrating numerous examples.

Zhang et al. - 2005

Another pre-*CUDA* example of GP-GPU computing applied to sound synthesis can be found in [31]. In this publication, Zhang et al. describe a method to exploit the processing power of a GPU in the context of *modal synthesis*[7] of sounds produced by colliding objects. In this case, the GPU is used in order to achieve real-time synthesis of a higher number of sounding objects simultaneously. Modal synthesis is indeed a very appealing target for parallel computing, considering that each mode can be synthesized independently of the others.
The approach used by Zhang et al. consists in pre-calculating modal models and storing them in a 2D *texture* so that they can be used as input data for *fragment programs* in order to compute the actual response to simulation stimuli. This

[6]This is a topic that will be actually faced in the section 2.8.

[7]*Modal synthesis* is a physical modelling synthesis technique which consists in modelling the sound produced by a vibrating object as a bank of damped oscillators that are excited by an external stimulus. More recent examples of the same technique have also been addressed in this review, see section 2.8.

approach has the downside of being memory-intensive[8] but it benefits in terms of computational complexity. The modal model used in [31] is represented by the set of parameters $\{\omega_i, d_i, A_i^j\}$, where $1 \leq i \leq N$ represents the mode number (for a total of N modes) and $1 \leq j \leq K$ represents the sampling contact location index (for a total of K sampling points). ω_i is the mode frequency, d_i is the decay rate that characterizes each mode and A_i^j is the mode's initial amplitude at each particular location after an impulsive excitation. All these parameters are meant to be given by some prior empirical analysis of the objects or by other computations based on their geometric and physical properties.
Zhang et al. show that, under the assumption of a linear model, the final response under an arbitrary force $F_{[t]}$ at a particular sampling location, can be expressed as:

$$y_{[t]} = \sum_{i=1}^{N} y_{i[t]} = \sum_{i=1}^{N} \left(R_i y_{i[t-1]} + G_i y_{i[t-2]} + B_i^j F_{[t-1]} \right) \tag{2.5}$$

Where $R_i = 2e^{-d_i/S} cos(\omega_i/S)$, $G_i = -e^{-2d_i/S}$, and $B_i^j = e^{-d_i/S} sin(\omega_i/S) A_i^j$ (where S is the audio sampling rate). All the R_i and G_i values can be pre-calculated before any actual computation, whereas B_i^j depends on the simulation-related factors A_i^j (which are related to specific contact locations). The authors actually pre-calculate all the K possible B_i^j at each desired sampling location; in this way, the computation of the B_i^J goes down to selecting the proper set of coefficients which corresponds to the sampling location closest to the actual contact location.
At the beginning of the simulation, all the modal models of the sounding objects (i.e, all their parameters) are stored in a 2D *texture* of the GPU and this is actually the only data transfer that is taking place from host to device. The names chosen for the parameters in equation (2.5) are not fortuitous, in fact, the R_i, G_i and B_i^j parameters are loaded into the red, green and blue channel of each *texel*. Parameters related to different modes are stored in successive columns of the *texture* while data related to different sampling locations and different objects is stored in successive rows[9] (see figure 2.4).

The actual sound synthesis is then performed in two steps: first, the current response sample of each individual mode from each object is computed, then all the modes' samples are summarized in a single one. This procedure is repeated at audio rate for each output sample. The first operation corresponds to selecting the desired row of the *texture* for each object (depending on the contact locations) and evaluating[10]

$$y_{i[t]} = R_i y_{i[t-1]} + G_i y_{i[t-2]} + B_i^{j*} F_{[t-1]} \tag{2.6}$$

for each mode i. In graphics terms, this is equivalent to rendering a rectangle through a *fragment program*: the total number of rows in the resulting rectangle is

[8]Still, the memory capacity of the GPUs of the time allowed for the synthesis of enough sounding objects and modes to recreate typical real life situations.

[9]There is indeed some data redundancy in this representation, since each R_i and G_i do not depend on contact locations but they are stored anyway in multiple rows. This redundant representation was chosen in order to simplify computational logic in the *fragment program*.

[10]Equation 2.6 is actually a dot product between two vectors, i.e. $[R_i, G_i, B_i^{j*}]^T$ and $[y_{i[t-1]}, y_{i[t-2]}, F_{[t-1]}]^T$. Since this is a very common operation in the graphics framework, it is carried out very efficiently on GPUs.

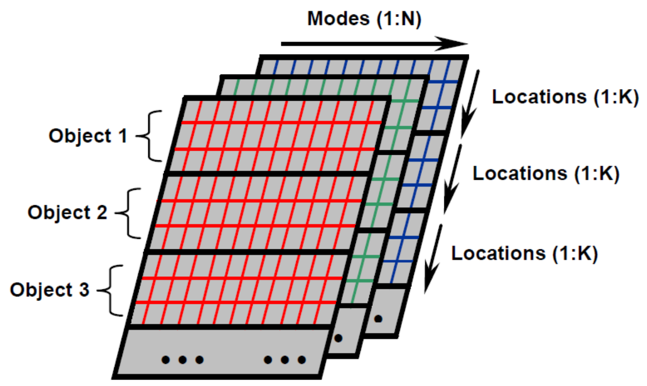

Figure 2.4: In [31], modal models are stored in a 2D *texture*. Different channels are used for storing different modal parameters.

equal to the number of objects that are currently oscillating due to past collisions. The rendered rectangle is stored in one channel of a 2D *texture* and, since only two past samples per mode are needed at each step of the computation (according to equation 2.6), only two channels of *texture memory* are used for storage (for instance, red and green channels are shown in figure 2.5).

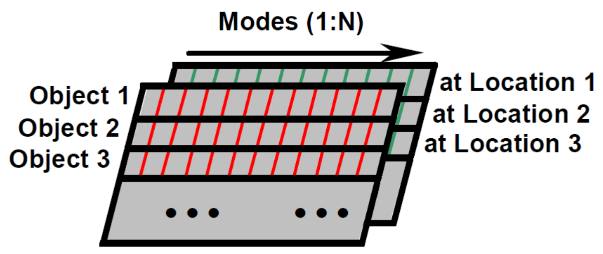

Figure 2.5: In [31], the current samples from each modal response are stored in one channel of a 2D *texture* after computation.

Together with $y_{i[t-1]}$, the result $y_{i[t]}$ is written back to the same *texel*, which will be used in the next cycle.
Summarizing all responses in one total response is achieved through a reduction operation. Due to the fact that there was no global register or hardware accumulator on older GPUs, reductions needed to be implemented as a multi-pass *ping-pong* process: starting with the initial rectangle, in each following step a rectangle scaled by a factor of 0.5 (both column-wise and row-wise) is rendered. For N sounding objects with N modes each, one channel of an $N \times N$ 2D *texture* is transformed in one channel of a single *pixel* in $log_2(N)$ rendering passes. The resulting *pixel* is then read back by the CPU[11].

This method was tested on 64 sounding objects in an acoustic scene. Each object is modelled using 512 modes and 16 sampling contact locations. This is equivalent to synthesize 32K modes simultaneously. Tests were run on a Pentium IV 2.8GHz CPU flanked by an NVIDIA GeForce 6800 GT with 256MB of video memory. The audio stream consists of 32-bit floating-point samples but the sampling rate used

[11]Probably in a block fashion, but this is not specified by the authors

in the tests is not declared. The GPU version of the code was written in *Cg* and allows to synthesize in real-time more than 6 times as many modes as the CPU version, which could handle less than 5000 modes.
This approach is particularly efficient since it needs data to be moved from host to device only once, i.e. at the beginning of the simulation. This is at the expense of memory occupation and lack of flexibility (the properties and number of objects must be defined beforehand). Concerning data transfer in the opposite direction, the authors do not specify how often output samples were moved from device to host, but they point out that data trasfers were indeed the performance bottleneck (for a real-time scenario) and that PCI-Express could have helped improving the total number of modes achievable in real-time (they were using an AGP 8X interface instead).

2.2 Additive Synthesis

Additive synthesis on general-purpose computers has always been quite limited by processing power: when high quality synthesis of arbitrary sounds is desired, the need arises for a number of sinusoids that is too high for any consumer-level CPU to compute. It has been typical to perform high quality additive synthesis with specially designed hardware. Yet, the task of generating and mixing sine waves of different frequencies can be accomplished in a fully parallel fashion and it is thus suitable for GPU computing.

Savioja et al. - 2010

In [32], Svioja et al. set up a testing framework in order to find out how many sine waves can be computed in real-time for the synthesis of general sounds on a commodity GPU, using *CUDA*. Two versions of an additive synthesis algorithm are implemented:

- A *plain* version concentrates on pure sinusoid computation at random frequencies.
- A *full* (more realistic) version defines specific starting phase and gain parameters in addition to the frequency parameter. Also, there is a possibility to slide each sinusoid from one frequency to another, causing more memory transactions.

For each version, two sine wave generation methods are investigated: table lookup with linear interpolation and `sinf` function.
The *CUDA* implementation assigns to each thread the task of computing a subset of the total number of sinusoids. In addition, only a fraction of the buffer size is computed by each thread (for instance, 128 samples are computed by one thread when the buffer size is 1024). This way, as it is shown in figure 2.6, given a specific buffer size and a target number of sinusoids to be computed, each thread is designed to compute samples related to a limited spectral region and to a limited time interval (inside the buffer time interval). Each thread also accumulates all the sinusoids' samples related to the same time instant and to the same spectral region.

A second set of threads accumulates, again in the vertical direction, the results computed in the first set in order to obtain a number of output samples which is equal to the buffer size. The whole process is repeated for each successive output buffer.

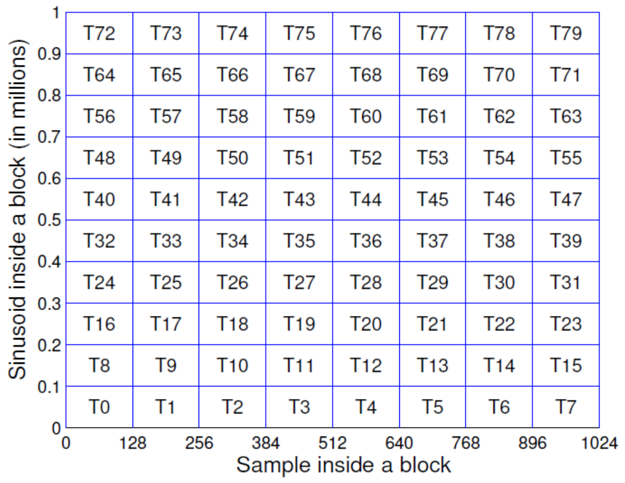
Sinusoid inside a block (in millions)
Sample inside a block
T72 T73 T74 T75 T76 T77 T78 T79
T64 T65 T66 T67 T68 T69 T70 T71
T56 T57 T58 T59 T60 T61 T62 T63
T48 T49 T50 T51 T52 T53 T54 T55
T40 T41 T42 T43 T44 T45 T46 T47
T32 T33 T34 T35 T36 T37 T38 T39
T24 T25 T26 T27 T28 T29 T30 T31
T16 T17 T18 T19 T20 T21 T22 T23
T8 T9 T10 T11 T12 T13 T14 T15
T0 T1 T2 T3 T4 T5 T6 T7
0 128 256 384 512 640 768 896 1024
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1

Figure 2.6: Thread to data mapping in the *CUDA* implementation of additive synthesis by Savioja et al., [32]. Each box corresponds to one thread which is in turn assigned to multiple spectral components and multiple time samples.

Savioja et al. tested this implementation on a commodity PC with a 2GHz Intel Pentium Dual E2180, 2GB of RAM and an NVIDIA GeForce GTX480 with 1.5GB of RAM memory. The operating system running on this machine was Microsoft Windows XP Pro SP3 and audio was managed by the Windows WaveOut API (using 44.1kHz as the target sampling rate).

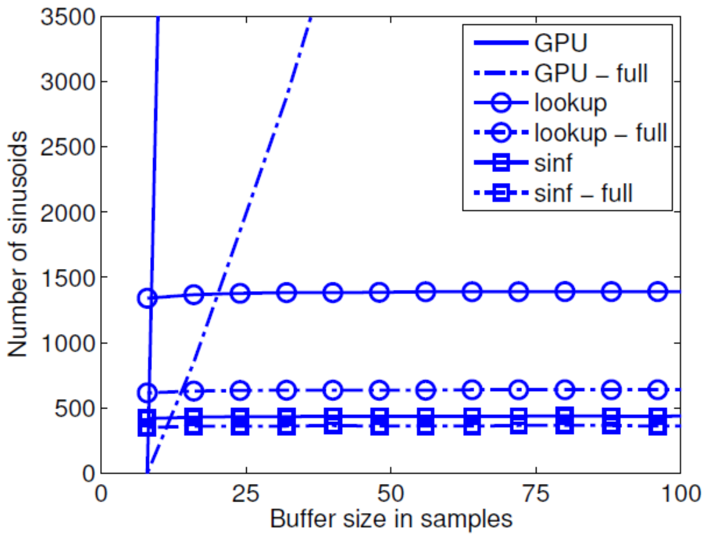

Figure 2.7: Number of sinusoids that can be computed and mixed for real-time additive synthesis on a commodity computer (CPU vs GPU) as a function of the buffer size (results from [32], small buffers).

Tests were run with a variety of buffer sizes, from 8 to 2048 samples in 8 samples steps. These experiments clearly show that the GPU implementation is superior with respect to a standard CPU implementation for all buffer sizes over 16 samples (figure 2.7). Analysing figure 2.7, it is important to stress that, while the CPU versions show a fixed performance that does not depend on the buffer size (it makes sense in a sequential computing framework), the GPU versions strongly depend on this parameter. This can be explained by noting that, as long as there are idle *CUDA cores* available, more threads can be actually instantiated on the GPU, simultaneously, until the graphics hardware is fully busy, thus increasing the total achievable number of sine waves that can be synthesised. Figure 2.8 shows the GPU results for longer buffers. The maximum performance of 1.9 million sinusoids is achieved with a buffer size of 2016 samples.

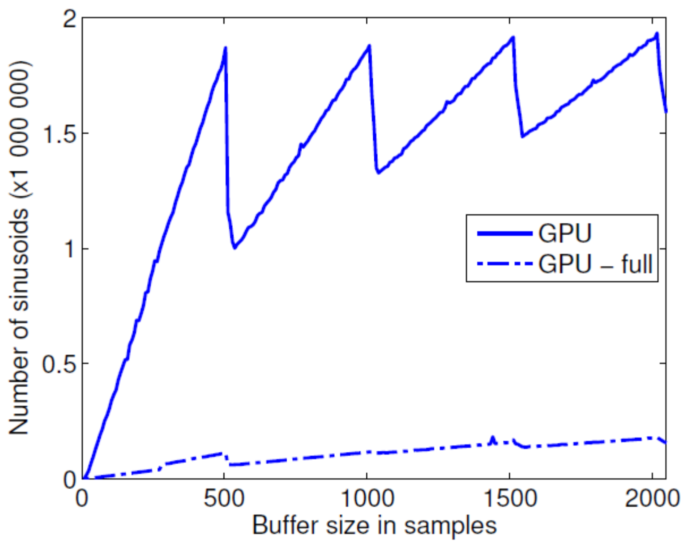

Figure 2.8: Number of sinusoids that can be computed and mixed for real-time additive synthesis on a commodity GPU as a function of the buffer size (results from [32], large buffers).

It is interesting to note that there are local maxima in the trends shown in figure 2.8. These are probably related to the way *CUDA* manages the mapping between threads and hardware resources, and they highlight the need for the programmer to know some details about the underlying hardware. Also note that the GPU version suffers[12] most from the incorporation of the individual gains, starting phases and slides, but still the performance at 500-samples-long buffers (a typical setting) provides a speed-up factor of more than 6 times over the fastest CPU implementation.

[12]This is caused by the increased number of memory transactions, since for each sinusoid there are four fetches instead of one, as in the plain version.

2.3 Spectral Model Synthesis

Spectral model synthesis (SMS) is an effective sound analysis/synthesis technique that can provide for high fidelity imitations of pre-recorded sound sources and it is often chosen for modelling some specific categories of sounding objects in a software environment. Nevertheless, high fidelity is only achieved when a remarkable number of spectral components is included in the model: in these cases SMS can get computationally intensive and very hard to manage in real time with commodity CPUs. Still, if the characteristics of the spectral components (amplitude and frequency) do not change too rapidly, and they can be considered constant on some time intervals, this operation turns out to be well suited for parallel processing. Thus, a GP-GPU computing approach can be employed in order to increase the number of spectral components that can be managed in real-time by a heterogeneous system. This approach shares a lot of similarities with the GPU-based additive synthesis framework analysed in section 2.2.

Tsai et. al. - 2010

In [6], Tsai et al. present a GPU-based technique for implementing real-time spectral model synthesis, using *CUDA*. They design the sound they want to achieve as a superposition of deterministic component (s_h) made of piecewise-stable and harmonically related sinusoids and a stochastic component (s_n), represented by a residual signal:

$$s_{[n]} = s_{h[n]} + s_{n[n]} = \sum_{p=1}^{P_{[t]}} A_{p[t]} cos(p\omega_0 n + \varphi_p) + s_{n[n]} \tag{2.7}$$

, where $P_{[t]}$ is the (slowly) varying number of harmonics[13], $A_{p[t]}$ is the (slowly) varying amplitude related to the p^{th} spectral component, ω_0 is the fundamental frequency of the synthesised sound and φ_p is the initial phase related to the p^{th} component.

The residual signal can be completely characterised by a low-order autoregressive LPC model:

$$s_{n[n]} = -\sum_{q=1}^{Q} a_q s_{n[n-q]} + w_{[n]} \tag{2.8}$$

, where Q is the LPC order, a_q are the predictor coefficients and $w_{[n]}$ is a white noise. Note that equation 2.8 represents a recursive filter and it cannot be computed in parallel in a straightforward way.

Tsai et al. set up a synthesis framework for two audio channels: the sound in each channel is obtained by mixing 50 SMS instruments. Each instrument is modelled by means of 50 spectral components with time-varying parameters that update every 10ms. The sampling rate is set to 44100Hz.

Two *CUDA* implementations are presented for the deterministic part, with different approaches about the way data is stored in the device memory and with a different

[13] P is actually considered constant over R samples. The time index t is such that it grows at a slower rate than n: $n = Rt$.

level of sophistication: *algorithm 1* is more straightforward, whereas *algorithm 2* employs shared memory a more clever *thread-to-data* mapping (see [6] for more details). In both implementations, the idea is that of generating 882 samples for each iteration of the external loop: these samples represent a 10ms portion of the final signal (441 samples for each channel). Since the model's parameters are considered constant over this interval, each instrument can be computed independently of the others and the same goes for the computation of the final samples, which are obtained by mixing the partial results related to each instrument. A similar discussion can be applied to the stochastic signal, with the exception that the partial results related to each instrument cannot be computed in parallel and are thus computed sequentially. Actually, also lower time granularities are investigated (up to 10 seconds).

Tsai et al. use the following computer system for testing this framework and comparing the execution times achieved by their *CUDA* implementation to a plain CPU version of the same synthesis process: an Intel Quad Core Q6600 at 2.4GHz flanked by an NVIDIA Tesla C1060 at 1.3GHz (240 *CUDA cores*, compute capability 1.3). The GPU turns out to be capable of achieving the goal of synthesising 5000 sinusoids plus residual noise in real-time for all time granularities above 10ms. On the other hand, the CPU-based sequential implementation fails in all cases (it has to be said that this version was not SSE-optimised). The speed-up factors obtained in these tests are reported in table 2.1.

Table 2.1: Speed-up factors (GPU vs CPU) for the sound synthesis of 100 SMS instruments (as in [6]) with different time granularities for parameters update. CPU performance is compared to the two *CUDA* implementations.

Granularity (s)	*Algorithm 1*	*Algorithm 2*
0.01	50.74	76.11
0.05	52.13	80.11
0.1	53.41	81.40
0.5	53.86	82.19
1	54.00	82.15
5	54.12	82.40
10	54.27	82.29

In a nutshell, the performance increase is quite remarkable, as speed-up factors vary between 50 and 80. Analysing this table a little further, it can be learned that, in this case, a well-tuned and sophisticated *CUDA* implementation (*algorithm 2*) turns out to be approximately 1.5 faster than a straightforward implementation (*algorithm 1*), and this difference can be enough to justify some extra effort when implementing *CUDA* applications. Table 2.1 also shows that the speed-up factors tend to increase as bigger and bigger chunks of data are processed each time the GPU is called into action (i.e., decreasing granularity): this is actually a typical behaviour in the GP-GPU computing scope.
Tsai et al. also tried to push the Tesla GPU to the limit, by increasing the desired

number of SMS instruments[14]: with a granularity of 10ms, the maximum number of instruments that this GPU could handle in real time was 1700.

2.4 Physically-based Synthesis via Finite Difference Methods

Finite difference methods can be the basis for implementing physically modelled musical instruments. However, such methods can get very compute-intensive, especially when realistic sounds are desired and, consequently, the need arises for complex virtual structures and for a high level of spatial resolution in the definition of these physical models. Yet, most of the times, the resulting large simulations can be expressed by means of a parallel paradigm. Thus, the possibility of harnessing the massively parallel architecture of modern GPUs gets very attractive.

Sosnik and Hsu - 2010 - 2013

In [33], Sosnik and Hsu implement a GPU-based real-time[15] finite difference-based simulation for a two-dimensional membrane. They point out that finite difference-based sound synthesis for large and/or fine-grained membranes and plates is too expensive to run in real time on CPUs and that the architecture of the GPU is particularly well suited for this type of algorithm.
Sosnik and Hsu consider a square (but elastic) membrane modelled as a vertical displacement continuous function of space and time $u_{(x,y,t)}$ with respect to a horizontal x-y grid. The shape of this function and its evolution in time are governed by the wave equation (with dissipation):

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{\partial^2 u}{\partial t^2} + \eta \frac{\partial u}{\partial t} \tag{2.9}$$

where η is the viscosity coefficient. After the model is moved in a discrete space, discrete time framework, the authors end up obtaining the following solution for the one-step-ahead value of the displacement function in each grid point:

$$\begin{aligned} u_{i,j}^{n+1} = \left(1 + \frac{\eta \Delta t}{2}\right)^{-1} \Bigg\{ & \rho \, (u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n) + \\ & + 2u_{i,j}^n - \left(1 + \frac{\eta \Delta t}{2}\right) u_{i,j}^n \Bigg\} \end{aligned} \tag{2.10}$$

where $\rho = [v(\Delta t/\Delta x)]^2$, such that v is the propagation speed of the wave in the given medium and Δt and Δx are the temporal and spacial sampling units, respectively. In order to obtain different sounds, the size of the grid is varied, together with the values of η and ρ and, if desired, those of the boundary conditions. Of course, the displacement function u also needs to be initialised accordingly, on the basis of the desired kind of excitation: in [33], for all simulations, the excitation is defined as a

[14]Each instrument is composed of 50 harmonics.
[15]Good responsiveness is considered by the authors as a key requirement in this context.

drastic shift from a perfectly flat surface (at $n = -1$) to a Gaussian initialisation (at $n = 0$). The audio output is obtained by simply considering the evolution in time of the central point in the grid.
A parallel implementation of equation 2.10 is implemented via *CUDA*: each grid point update is mapped to a single *CUDA thread* via 2D thread blocks (note that boundary points need to be treated accordingly). At each step, the value of u corresponding to the chosen output grid-point is saved to a specific array which is sent to the audio buffer (at host side) at the desired rate. Note that the parallelism exploited here involves the two spatial dimensions only: each output sample is still computed sequentially via a for loop. Also noteworthy is the fact that, even though the amount of data which is handled by the GPU can be quite remarkable when a fine-grained spatial resolution is set, the memory transfers between device and host only involve the output buffer, which ultimately consists of a one dimensional array of no more than a few hundreds elements (audio samples). In addition, no memory displacement in the opposite direction (from host to device) is needed, apart from a preliminary transfer for the initialisation of the membrane's shape.

The *CUDA* implementation was tested on three systems and compared to a standard serial implementation.

- System 1: a desktop computer with 2.5GHz Intel Core 2 Quad running Ubuntu 9.10, flanked by an NVIDIA GeForce GTX285.
- System 2: a Mac Book Air with a 1.86GHz Intel Core 2 Duo running OS 10.5.8, flanked by an integrated NVIDIA GeForce 9400M.
- System 3: a macPro with dual 3 GHz Intel Quad-Core Xeon running OS 10.5.8, flanked by an NVIDIA GeForce 8800GT.

A series of tests was run, each test being characterised by a different grid size (from 15x15 to 21x21) and a different target buffer size (from 8 to 4096). The sampling rate was set at 44.1kHz and computation was carried out with single precision (4-byte) floats. The simulations consisted in playing the virtual instrument for 5 seconds and recording the total execution time of the two versions. The results obtained from the simulations with a fixed grid size of 21x21 points are shown in figure 2.9.
While the performance on the CPU remains almost constant for all buffer sizes, on the GPU side there is a clear trend for the execution time to decrease with increasing buffer size. This is probably due to the fact that, with higher buffer sizes, fewer memory transfers and fewer kernel calls are needed. The comparison between the sequential implementation and the parallel one really depends on the specific system, as well as on the employed buffer size. However, it is worth noting that, on system 1, the GPU-based version outperforms the CPU-based one, with speed-ups of 1.2 to 2.9.
Figure 2.10 shows the results obtained with a fixed buffer size of 4096 samples but with varying grid sizes. As with the previous test, the parallel implementation is faster than the serial one on the GTX285 system for all tested grid sizes; it can be seen in 2.10 that timings for the CPU show an approximate $O(n^2)$ increase with grid size, while GPU timings increase significantly more slowly. In all cases, speed-up improved with larger grid-sizes.

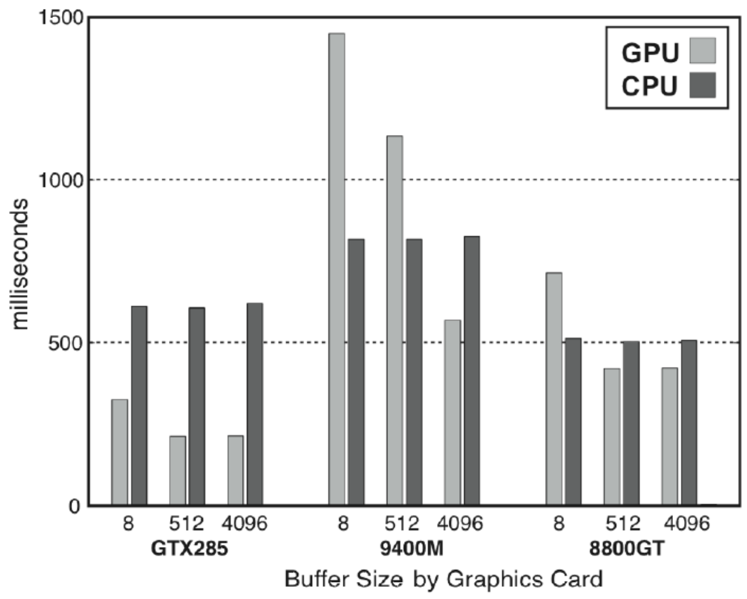

Figure 2.9: Execution speed of finite difference-based membrane sound synthesis programs with a constant grid size of 21x21 points and varying output buffer sizes. Comparison between three systems producing a 5 seconds output ([33]).

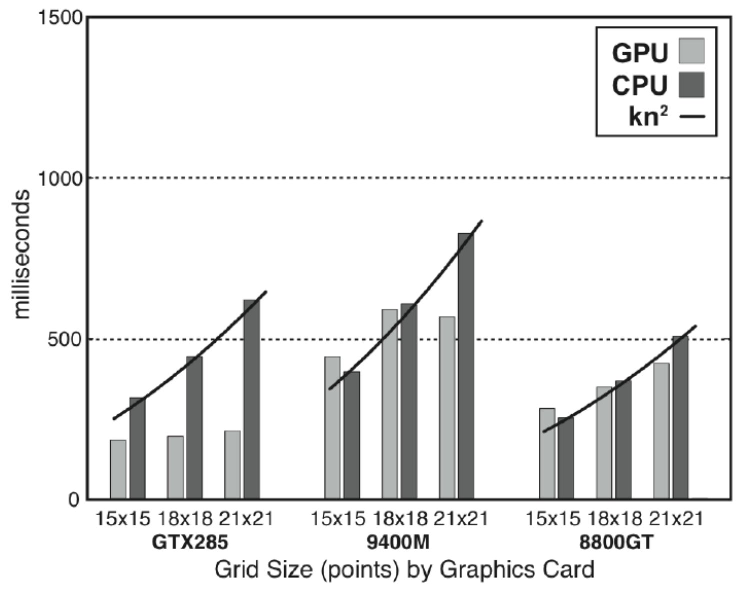

Figure 2.10: Execution speed of finite difference-based membrane sound synthesis programs with a constant buffer size of 4096 samples and varying grid sizes. Comparison between three systems producing a 5 seconds output ([33]).

In [33], Sosnik and Hsu prove that, in an adequate system, better performance can be obtained when casting finite difference sound synthesis of a membrane to the GPU. On less powerful systems, a performance gain can be obtained only for high grid sizes (which actually translate into more realistic sounds) and high buffer sizes (which can undermine real-time applications). In general, a GP-GPU approach to sound synthesis via finite difference methods is promising, even when dealing with mid-range GPUs. Larger simulation grid sizes could be achieved by exploiting the parallelism of multiple GPUs or that of more recent GPUs with more memory and more cores.

In [34], Sosnik and Hsu reprised the same project after a few years and tested a very similar application on more recent hardware. This time, much larger grids could be simulated, giving a better spatial resolution and a more realistic sound. The new tests were run on mid 2012 MacBook Pro Retina with a 2.7GHz Intel Core i7 processor, 16GB of RAM and a built-in 900MHz NVIDIA GeForce GT 650M GPU (one of the slower Kepler-based GPUs). The operating system is OS X 10.8.2 running *CUDA 5.0*.
Figure 2.11 shows the time needed to generate one 512-sample buffer of audio for the *finite difference synthesizer* running on the CPU and on the GPU (execution times above 11ms get out of the real-time region).

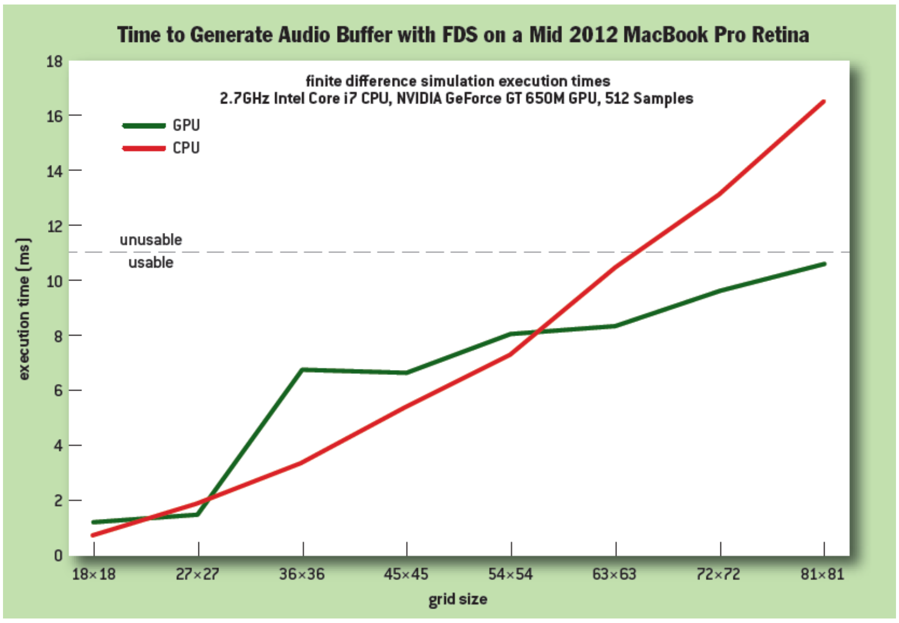

Figure 2.11: Execution time for the synthesis of one 512-samples buffer by means of a finite difference-based membrane synthesizer set to different grid sizes. CPU and GPU performance are compared ([34]).

For this system, the CPU load gets too high for simulations based on grids that are larger than 69x69 points. However, the GP-GPU version of the algorithm is capable of providing real-time execution for grid sizes as large as 84x84: this corresponds to a 48% improvement in the maximum grid size supported.

2.5 Room Acoustics Modelling

GP-GPU computing has always been an attractive approach for what concerns the research field of room acoustics modelling. As a matter of fact, GPU-enhanced room acoustics computation is both one of the older audio-related fields to be investigated ([35], for instance, dates back to 2004) and one of the most studied.

Of course, for practical reasons, only a very small fraction of the existing literature about this topic will be reported here, just to give an idea. For a thorough analysis of this research area please refer to [5] and [36]
The reason for which this field was, in the early days of GP-GPU computing, an obvious candidate for GPU processing is that it actually shares several similarities with graphics-related problems such as global illumination modelling. This is especially true for the higher portion of the audio spectrum, where the sound wavelength is short compared to the dimensions of reflecting surfaces and sound can be assumed to behave similarly to light and modelled as rays (this is called *ray tracing*).
In the lower part of the spectrum, where the ray assumption fails, room acoustics modelling is usually done by means of wave-based methods, for which the wave equation is solved numerically or analytically. Different approaches are possible: these include *finite-element method* (FEM), *boundary-element method* (BEM), *finite-difference time-domain technique* (FDTD), and *digital waveguide mesh*. It turns out that all these techniques can be implemented using highly parallel algorithms and they are thus perfect candidates for GP-GPU computing.
An important study about the GPU-based ray tracing technique applied to sound propagation can be found in [4]. However, this paper is not going to be summarised here because it has a very theoretical and qualitative approach (the theoretical side is of course beyond the scope of this work) and it does not delve into the details of the GPU implementation. A similar argument can be made for [37], which in turn deals with a waveguide-based approach to room acoustics and reports speed-up factors from 4.5 to 69-fold in the 2D case, depending on the resolution of the employed lattice.

Hamilton and Webb - 2013

In [7], Hamilton and Webb present a room acoustics simulation based on a finite difference approximation on a face-centred cubic (FCC) grid with finite volume impedence boundary conditions. The authors aim at coupling finite volume boundary conditions to a 13-point finite difference scheme on the FCC grid for large-scale room acoustics simulations enhanced by GPU acceleration.
Hamilton and Webb start from a classic dispersion-free 3D wave equation:

$$(\partial_t^2 - c^2\Delta)u_{(t,x,y,z)} = 0 \tag{2.11}$$

, where c is the speed of the wave (constant), Δ is the 3D Laplacian operator ($\Delta = \partial_x^2 + \partial_y^2 + \partial_z^2$) and u represents a *velocity potential*, such that air pressure p and particle velocity $\mathbf{v}$ fields are given by:

$$\begin{aligned} p_{(t,x,y,z)} &= \rho\ \partial_t u_{(t,x,y,z)} \\ \mathbf{v}_{(t,x,y,z)} &= -\nabla u_{(t,x,y,z)} \end{aligned} \tag{2.12}$$

From there, by means of defining discrete approximations for the first-order and second-order differential operator and for the Laplacian operator (using a 13-point *stencil*), they end up with the following explicit recursion for the approximated solution:

$$\hat{u}_{(t+T,\mathbf{x})} = -\hat{u}_{(t,\mathbf{x})} - \hat{u}_{(t-T,\mathbf{x})} + \frac{1}{4}\sum_{i=1}^{12} \hat{u}_{(t,\mathbf{x}+\boldsymbol{v}_i)} \tag{2.13}$$

, where T is the time-step, $\mathbf{x}$ represents the 3D spatial coordinates and $\boldsymbol{v}_i$ are vectors in $\mathbb{R}^3$ that are accordingly chosen.
For what concerns the boundary points of the room, a finite volume scheme with frequency-dependent absorption at the walls is employed. This approach leads to the following update equation for the boundary cells:

$$\hat{u}_{(t+T,\mathbf{x})} = \frac{1}{k_1}\left(k_2 + k_3 \sum_{i=1}^{12} q_i \delta_{\boldsymbol{v}_i^+}\right)\hat{u}_{(t,\mathbf{x})} - k_4 \hat{u}_{(t-T,\mathbf{x})} \tag{2.14}$$

, where k_1, k_2, k_3 and k_4 are scaling constants, q_i is either 1 or 0 (depending on the considered cell). The operator $\delta_{\boldsymbol{v}_i^+}$ is defined as:

$$\delta_{\boldsymbol{v}_i^+} = \frac{1}{L}(\hat{u}_{(t,\mathbf{x}+\boldsymbol{v}_i L)} - \hat{u}_{(t,\mathbf{x})}) \tag{2.15}$$

, where L is the spatial step.
By looking at the two update equations (2.13 and 2.14), it is evident that, at each time step, updating the grid points is data-independent and the whole process can clearly benefit from parallel execution. The main concern in terms of GPU efficiency is ensuring memory coalescence: a clever technique for avoiding too many uncoalesced accesses is displayed (see the full text for more details).

The *CUDA* implementation was tested against a plain CPU version in a simulation based on a grid of 195,520,000 points (800x520x470), computed for 44100 time steps, using double precision arithmetic. The computer system used for these tests comprised of an Intel Xeon E5-2620 and an NVIDIA Tesla K20 GPU. The resulting execution times were: 30 hours for the serial code running on the CPU and 38.5 minutes for the parallel code running on the GPU. The parallel version is thus 46 times faster.

2.6 Headphone-based Spatial Sound (HRTF)

Spatial audio systems are becoming more and more popular in the entertainment industry, including applications in the scope of videogames, virtual reality, cinema and music performances. One particular sub-class of spatial audio that has shown remarkable potentials is that of headphone-based auralization by means of HRTF[16] filtering: this technique allows a listener to perceive the virtual position of sound sources in space by means of using only two real sources (headphones). This kind of effect is obtained by filtering the sound sources with special filters whose coefficients shape the sound with spatial information. As a matter of fact, the impulse responses of HRTFs describe how sound waves are filtered by the scattering properties of the individual body shape (i.e., pinna, head, shoulders, neck and torso) before the sound reaches the listeners eardrum. The realisation of this effect eventually

[16] *HRTF*: Head Related Transfer Functions

corresponds to the computation of two FIR filters per virtual sound source (one for the left ear and one for the right ear). This is doable in real-time applications on commodity computer system as long as sound sources are kept limited and filter lengths are kept short. However, too often the need arises for more virtual sources than an off-the-shelf CPU can handle. To solve this problem, HRTF filtering can benefit from parallel processing (FIR filters are easily cast to a data-parallel scheme) and it can be successfully implemented on GPUs for rendering a higher number of virtual sound sources in real-time.

Belloch et al. - 2012

In [38], Belloch et al. discuss the design, the implementation and the performance of a headphone-based spatial audio application whose processing is totally carried out on a GPU, via *CUDA*. They use an HRTF database of 384 filters (187 for the left earphone and 187 for the right earphone) made of 512 coefficients each (in the frequency domain). Their objective is that of increasing the total number of virtual sound sources that can be rendered in real-time on a commodity notebook.
Given N virtual sources x_i and two filters ($h_L^{\mathbf{r}_i}$ and $h_R^{\mathbf{r}_i}$) for each possible direction $\mathbf{r}_i$ in the 3D space, the output signals y_L and y_R are computed as:

$$\begin{aligned} y_{L[n]} &= \sum_{i=0}^{N} h_{L\,[n]}^{\mathbf{r}_i} * x_{i[n]} \\ y_{R[n]} &= \sum_{i=0}^{N} h_{R\,[n]}^{\mathbf{r}_i} * x_{i[n]} \end{aligned} \tag{2.16}$$

These convolutions are usually implemented in the frequency domain, and this is also the case in [38]. At the beginning of the process, all the HRTF filters (frequency-domain coefficients) are transferred to the GPU memory. Then, the following operations are performed in a *CUDA* environment:

1. N memory buffers of 512 samples each (from different sound sources) are filled at host side.
2. When filled, these frames are transferred to the device.
3. Position information vectors are transferred to the device.
4. Convolution is started:
   (a) Each input frame is Fourier transformed via cuFFT library ([21]).
   (b) Element-wise multiplication is performed between transformed input frames and the corresponding filter vectors.
   (c) Element-wise summation of all the outputs related to the left channel is performed (the same for the right channel).
   (d) The two resulting frames are inverse transformed (again via cuFFT).
   (e) In case of moving sources, additional filters are computed (related to the new positions) and the results are weighted by smoothing time functions and summed to smoothed versions of the results obtained from the old positions.
5. The two resulting frames are transferred back to the host for playback.

Belloch et al. make use of *CUDA streams*[17] to efficiently manage the real-time motion of sound sources in space. Also, particular care is taken so that global memory accesses are coalesced.

Tests were run on an NVIDIA Tesla GTS-360M (*CUDA* compute capability 1.2) with 12 *streaming multiprocessors* (no information is given about the rest of the system, other than it is a notebook computer). The results are shown in figure 2.12, where the execution time of the whole process is plotted as a function of the number of virtual sources employed for the simulation. The experiments were carried out in an extreme worse situation, that is, when all the sources are constantly moving through the 3D scene.
Figure 2.12 shows that a maximum of 2500 moving sources can be rendered for

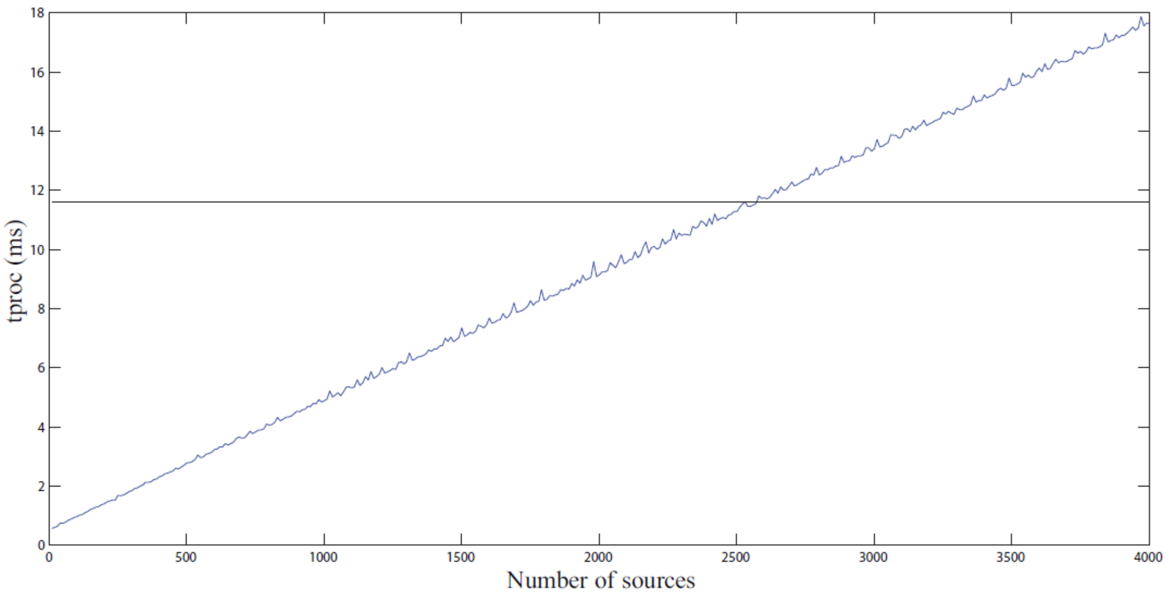

Figure 2.12: Execution time of the *CUDA*-based HRTF rendering application discussed in [38] for a variable number of virtual sources. In this simulation, sources are moving constantly around the scene (worse condition). The horizontal line represents the real-time threshold.

headphone-based spatial audio on the target system. Note that the buffer size used (512 samples) is quite low, thus insuring low latency. No comparison with sequential, CPU-based processing is reported in this work. In fact, the authors are not interested in this comparison: instead, they aim at assessing how much the utilisation level of the CPU can be kept low while rendering a HRTF simulation on the GPU. It turns out that when 2500 moving sources are simulated, the CPU task manager reaches barely 10% of the available computing potential.

[17]See section 4.5.1 for more details about *CUDA streams*.

2.7 Using the GPU in the Context of Music Production

Knowing the potential power offered by GPUs combined with the possibility of using them for general-purpose computing, it is right and proper for audio programmers to wonder whether it makes sense to harness this power in any project that is meant to run on a GPU-equipped device (nowadays, not only computers). One field in particular where GP-GPU computing has not really become popular[18] (yet) is that of music production software, meaning software specifically designed to accomplish some tasks inside the last stages of the digital music recording chain, i.e. tracking, mixing and mastering. In this scenario, when sufficiently complex projects are considered, high processing power is often needed: actually, the CPU can often be the limiting factor when a project runs too many filters/effects on too many tracks. In fact, some vendors sell dedicated processors to handle audio DSP. A variety of DSPs for audio effects[19] are available for the music producer, ranging from soundcards with on-board effects to dedicated DSP expansion cards (usually PCIe cards) that provide no audio I/O, instead concentrating on a range of high-quality effects. Unfortunately, this hardware is often very expensive and these dedicated audio processors are typically closed proprietary devices that only allow certain audio processes to be computed, keeping the amount of programmability very limited. By contrast, GPUs are widespread in the majority of computer systems and are sold in a broad market, with an extensive range of price tags, from entry level to hi-end processors, and, most importantly, they are now fully programmable.

[18]Although GPU-supporting audio software does exist, the practice of developing parallel-processing-oriented versions of code has not asserted yet. One essential exception is Acustica's Nebula VST series, an emulator for analog hardware like pre-amps, equalizers, compressors, tapes, filters, effects and reverbs. In Nebula plug-ins, the model identification scheme of non-linear and time-variant systems is based on Vectorial Volterra Kernels. From Acustica's website: "The Acqua Engine VST Plug-in is "Designed For *CUDA*", which means that it is able to take advantage of this technology to provide enhanced performance for those who use NVIDIA graphics cards that are *CUDA* enabled" (`http://www.acustica-audio.com/`). Another example can be found in Liquidsonics' Reverberate LE, a highly efficient convolution reverb processor VST which is available in a version taking advantage of *CUDA* for the main convolution processing tasks, reducing CPU usage. It is interesting to report here a note which can be found on Liquidsonics' website: "Zero latency mode is not supported by the GPU Edition due to a combination of factors including the mechanisms involved with transferring data to and from the GPU being much more efficient with larger blocks, a current requirement to run *CUDA* VST plug-ins in a separate thread (making larger block processing more efficient) and the lack of coherency in buffering schemes used by various different VST hosts complicating the above issues. It may become more practical to implement this feature in future using newer GPU architectures" (`http://www.liquidsonics.com/`).

[19]To name a few (older and newer) devices: Universal Audio's UAD series and Apollo series (`http://www.uaudio.com/`), TC Electronic's PowerCore (`http://www.tcelectronic.com/powercore-pci/`), Digidesign's HD Accel (now Avid's Pro Tools HD, `http://www.avid.com/products/protools-hdx`) and countless more. More than real "co-processors" most of these are actually DSP "external processor" and their role is indeed quite different to that of using the GPU for similar tasks.

Fabritius - 2009

Precisely to answer the question about how much gain in performance it is possible to achieve, if any, in the execution of the algorithms involved in a typical music production scenario, Frederik Fabritius carries out a thorough analysis [28] on the efficiency of porting to the parallel programming scheme some of the most common (and basic) processing algorithms, namely: equalizer, delay, reverberation and dynamic range compression. This is done in order to study the possibility of “expanding the computational potential for audio hosts in music production, by taking advantage of the otherwise unused GPU” and understand in which cases the parallel versions of these widespread algorithms are faster. This is a good starting point for studying the possibility of integrating GPU processing (or some other form of parallel processing) into digital audio workstations by default.
Fabritius designs a program for applying to a sound file at least one prototype version for each of the four processes: the equalizer family is represented by applying a second order low-pass Butterworth filter (IIR), the delay is designed by means of memory lookups involving a simple sum evaluation and balance factors, the reverberation family is represented by a convolution reverb (i.e., an applied FIR filter) and the compressor is based on a target gain envelope design. CPU-based and GPU-based versions of the same processes are implemented and tested in order to make a comparison between the two programming models.
Noteworthy, most of Fabritius’ work is based on an off-line utilization scheme as opposed to what will be presented in the next chapters of this thesis, which will focus on exploiting parallel computing for real-time sound design and live music applications. Fabritius’ work, instead, aims at building a faster framework mainly for mixing and mastering stages where the same process is applied in a batch fashion to a whole audio file. Of course, most of the considerations which will be made in the next paragraphs also apply to real-time processes.

As the author points out, some of the algorithms are better suited for parallel processing than others: the equalizer, for instance, is inherently sequential and recursive and should not benefit from being parallelised (at least in its straightforward implementation, see the section dedicated to parallel implementations of recursive filters for more information). On the other hand, the convolution reverb (FIR filtering) should be very efficiently computed in parallel, as it has been already shown in this review (see 2.6, for instance). Delay and dynamic compression are likely to show an intermediate level of achievable parallelism, as they can be broken down into different steps, some of which are suited for parallelism and some others are not. In any case, Fabritius points out that whenever an algorithm shows a similar level of execution speed on the CPU and on the GPU, it is still worth considering the GPU implementation so to free the CPU for other tasks.

About the delay, it can be seen from the update equation[20]

$$y_{[n]} = (1-\alpha)\ x_{[n]} + \alpha\ \beta\ y_{[n-L]} \tag{2.17}$$

that recursion problems can be avoided if processing is carried out in a block fashion with a block length of L samples. In this way, L parallel threads can be launched

[20]In equation 2.17, α is the “wet”parameter, β is the amount of feedback and L is the delay length in samples.

on the GPU for each frame, but each pair of consecutive frames must be computed sequentially. This means that the longer the delay the more efficient the GPU version of the algorithm. A copy and sum design is also implemented: this approach is completely parallel (with the exception of one atomic operation needed for each iteration) but it has the disadvantage of increasing the time complexity from $O(n)$ to $O(n^2)$ (see [28] for details). Note that this approach is presented even though floating point atomic operations were not available at the time of development and it thus scores poorly on benchmarks (barrier synchronization is used instead of atomic operations).
For convolution reverb, the fast convolution method is used: the signal and the impulse response are Fourier transformed, multiplied in the frequency domain and transformed back to the time domain. Three variants of this approach are implemented and tested: standard *fast convolution*, i.e. the batch processing on the whole signal, *basic overlap-and-save* design, based on successive processing of overlapped input signal STFT[21] frames, and *partitioned convolution*, where both the signal and the impulse response are split up into segments, convolved in an *overlap-and-save* fashion and finally combined by means of proper delays (See [39],[40] and [41]). Each of these methods is very well suited for parallel computation; however, even though *basic overlap-and-save* has a lower time complexity with respect to *fast convolution* ($O(Nlog2(N_{block}))$ against $O(Nlog2(N))$, and should be therefore more efficient for larger problems, as the FFT size is decreased the GPU version is likely to give a smaller proportional speed-up with respect to a batch implementation (this is because of the usual overhead introduced by GPU processing). Furthermore, even though partitioned convolution is slightly more costly than *basic overlap-and-save*, a smaller amount of memory is referenced during computation and this results in a more efficient use of cache memory.
The dynamic compressor examined by Fabritius uses a target gain envelope based on an RMS input level detector. The RMS input level is computed via a truncated infinite impulse response (TIIR) filter ([42]), using a delay line with a length in samples which corresponds to the chosen RMS time window (N_{RMS}). For each sample n:

$$\begin{aligned} SumOfSquares[n] &= SumOfSquares[n-1] + x[n]^2 \\ &\quad - SumOfSquares[n - N_{RMS}] \\ level[n] &= SumOfSquares[n]/N_{RMS} \end{aligned} \tag{2.18}$$

This approach is thus similar to the algorithm used for the delay and similar considerations apply: the presence of $SumOfSquares[n-1]$ in the first equation, however, implies a computation pattern that must be fully sequential. Nonetheless, other parts of the compressor algorithm can be computed in parallel, i.e. the computation of the sample by sample difference between input level and threshold (the so-called “delta level”, used both for switching the compressor on and off and in the computation of the gain envelope) and the final application of the gain envelope (which must be otherwise computed sequentially before application).

Fabritius states he would have used *OpenCL* had it been available when he was developing his wok: *OpenCL* is seen as an ideal framework thanks to its

[21] *Short Time Fourier Transform.*

cross-platform nature, preventing vendor lock-in. He states his choice went down to selecting between NVIDIA's *CUDA* and AMD's *Brook+* as GP-GPU programming APIs, and he eventually chose *CUDA* for two reasons: first, being *CUDA* more similar to *OpenCL*, porting his work to a hypothetical *OpenCL* version should be easier, secondly, he already owned an NVIDIA graphics card.
Note that the *CUDA* implementation of Fabritius' work does not consider asynchronous memory operations because the testing platform is limited to compute capability 1.0 (*CUDA*'s zero copy feature is also not supported).

The testing stage of Fabritius' thesis is quite articulated and a thorough report of each result data is beyond the scope of this work; nevertheless, an abstract of the most interesting results will be provided in the following paragraphs. The testing system used by Fabritius is the following: Intel Core 2 Quad Q6600 (2,66GHz) with 4MB of shared L2 cache per core pair and 32KB of L1 cache per core, 2048MB of RAM and an NVIDIA Geforce 8800 GTS (compute capability 1.0) with 320MB of device memory. Tests are run on a mono signal with a sampling rate of 44.1kHz and 32-bit resolution.

As it is implemented as a single-threaded process, it comes as no surprise that the equalizer performs very poorly on the GPU: when doing tests on a 8399608 samples long file, the "speed-up" factor φ given by the ratio of the running time of the CPU version over the running time of the GPU version turns out to be a very disappointing $\varphi = \frac{106.565ms}{4629.41ms} = 0.023$.

The performance tests made on the delay algorithm are quite interesting to analyse: it turns out that both the straightforward design and the circular buffer design are slightly slower when executed on the GPU, with no particular influence given by varying the delay length (which is kept below 10 seconds for musical plausibility). A few explanations for these behaviours are reported: the reduced execution speed is explained by the fact that the memory transfer overhead implied by GP-GPU computing is not compensated by parallel processing as the kernels are very simple (with low time complexity and few arithmetic operations) and the delay length is not enough, which means that not enough threads are spawned on the GPU, even in the case of a 10 seconds delay. It is in fact proved that kernel execution time of the circular buffer design is more than 5 times faster for the GPU when a 4 seconds delay is used, but this is not enough to compensate for an overall memory transfer overhead of almost 40ms. Overall, the "speed-up factor" φ turns out to be $\varphi = \frac{25.367ms}{44.555ms} = 0.57$. Note that the CPU version was optimized for multi-core computing as it exploits the *Streaming SIMD Extensions* (SSE) through the *OpenMP*[22] API. It is also important to stress that the GP-GPU implementation used by Fabritius in this work is quite "rudimentary" for today's standards: as explained by the author himself, the lack of the zero copy feature and the lack of asynchronous memory transfers really spoil the potential of the GPU version, and testing the same algorithm on a more recent device would probably result in the GPU version being faster than the CPU version for delay lengths above a certain size. Nevertheless, even in the analysed situation, it would still make sense to offload these computations to the GPU in order to make room for other tasks to use CPU resources. This is a key point of GP-GPU audio programming in general.

[22] *OpenMP*: `http://openmp.org/wp/`

All versions of the reverb algorithm turn out to be very efficiently run on the GPU. They are all suited for parallel processing and they are all computationally intensive enough, so that parallel processing compensates for memory transfer overheads. Making a comparison between GPU versions, *partitioned convolution* results to be the fastest method: this is due to a good level of achieved memory coalescence (this applies to *basic overlap-and-save* as well) and to a computation shift from Fourier transforms to complex multiplications, which are very fast on GPUs. Among the CPU versions, *basic overlap-and-save* is found to be the fastest; thus, a final comparison is made between the *partitioned convolution* design (on the GPU) and the *basic overlap-and-save* design (on the CPU). Not surprisingly, the speed of execution of these two implementations is found to be highly dependent on the length of the impulse response used for filtering. For large filter lengths (from a quarter of a second upwards) the GPU version is faster, getting better and better with respect to the CPU version as the filter length grows. For smaller filter lengths (below 0.2 seconds) the performance of the CPU compared to the GPU is roughly equal. Again, the bottleneck of the parallel version is to be found in the memory transfers and running the same test on a more recent system (i.e. on a device with higher compute capability) would probably result in faster execution times on the GPU side, even for quite small decay times. Analysing the case of a fixed decay time (a typical value of 0.75 seconds) and varying the length of the input signal, the tests show that, as this variable grows, the running time increases more slowly for the GPU implementation. However, the speed-up given by GP-GPU processing for a typical application (0.75 seconds decay time reverb applied to a 190 seconds audio signal) is not really outstanding: $\varphi = \frac{326.119ms}{233.538ms} = 1.40$. FFTs are computed through the cuFFT library ([21]) on the GPU, while the FFTW library ([43]) is used for the CPU versions.

Finally, the GPU version of the dynamic compressor turns out to be outdone by the CPU version. In addition, the tests show that, while the CPU version is somewhat independent of the RMS window size, the running time of the GPU version grows linearly with this variable. For a very small RMS window size of 5ms the execution time for the GPU is 3 times as long as for the CPU implementation but, if the CPU resources are scarce, it could still make sense to offload compressor computations to the GPU.

To sum up, Fabritius'study shows that the only process that clearly benefits from GP-GPU computing is the convolution reverb algorithm (and, consequently, any kind of FIR filtering). The delay and the dynamic compressor are shown to be slower when executed on the GPU but, in some cases, it could still be beneficial to offload computations from the CPU to the GPU for both these tasks. On the other hand, the equalizer performs very badly when using a (single-threaded) straightforward design, but in other sections of this review it is shown how other, much more efficient designs are available for this kind of process (See section 2.8). These results are more than promising for the introduction of GP-GPU computing in everyday music production systems: it is in fact quite safe to assume that running the very same tests on more recent devices, making sure to exploit features like asynchronous memory transfers and zero copy, would make the GPU versions of all analysed algorithms much faster (even faster than CPU versions when this is not already the case).

Savioja et al. - 2011

A slightly more recent approach to GPU-based reverberation for musical purposes can be found in [44]. Here, Savioja et al. discuss two versions of a reverberating algorithm, one in the frequency domain (using an STFT approach) and one in the time domain (plain convolution). For both versions, a GP-GPU implementation (based on *CUDA*) is compared to a standard CPU implementation. The frequency-domain approach is also used as a way to assess the efficiency of GPU-based Fourier transforms in general.

For what concerns the frequency-domain approach, the authors investigate what is the maximum DFT size that can be handled in real-time by a commodity computer for a given number of channels and with various buffer sizes. In each channel, the processing is managed one buffer at a time with 50% overlap. Each buffer is Hamming windowed and zero padded to the DFT length under examination. The FFT of the input buffer is computed, the multiplication in the frequency domain is evaluated and the result is inverse transformed and overlap-added with the other results in order to give two channels for stereo output. The CPU implementation makes use of the highly optimised FFTW library ([43], which employs SSE instructions) while the GPU implementation is based on the cuFFT library ([21]).

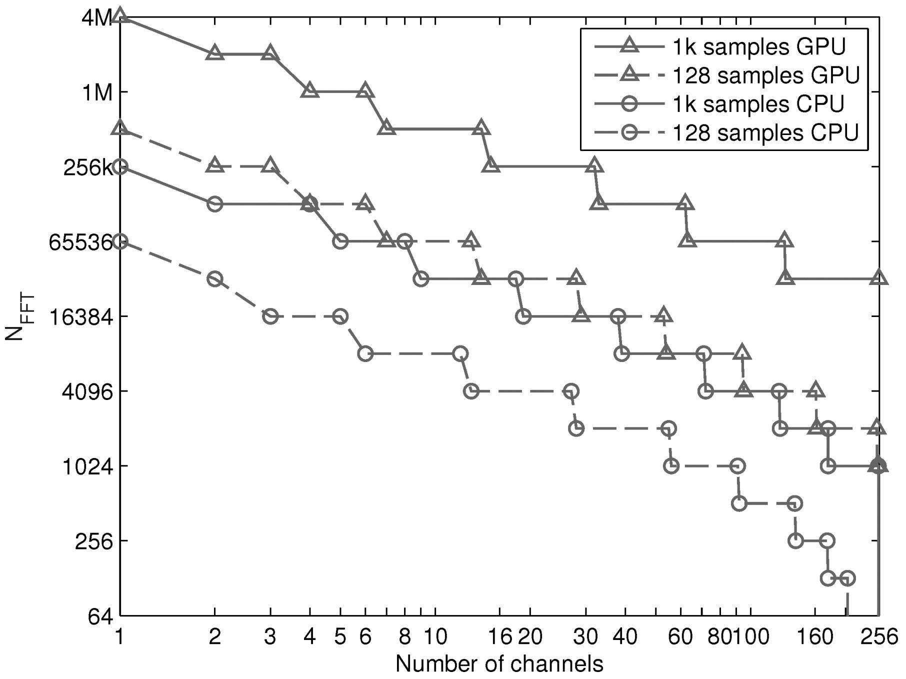

Figure 2.13: Maximum FFT length for real-time frequency-domain convolution as a function of the number of channels to which it is applied. CPU vs GPU comparison as in [44], where two possible buffer sizes are considered (128 samples and 1024 samples).

The tests were run on the very same system employed in [32] (see 2.2), except for the use of a higher sampling rate of 48kHz in these tests. Figure 2.13 shows the maximum DFT size (i.e., the maximum impulse response length) as a function of

the number of the effected channels (up to 256).
These results show that the GPU is quite consistently able to handle transforms eight time as long as the CPU version in the same time, no matter how many channels are considered (even with one single channel the GPU still performs better). By means of reading figure 2.13 it is possible to grasp an idea of the practical limits for multi-channel FFT processing on a GPU. For instance, when 16-channel audio is processed at 48kHz with small latency (buffer size of 128 samples), the maximum reverberating time achievable with this framework corresponds to 32768 samples.

For what concerns the convolution in the time domain, the authors show that the computation time of an FIR filter does not depend linearly on the filter length when it is computed on a GPU. In addition they investigate what is the maximum filter length that can be used in real-time as a function of the number of input channels. Each channel is processed with two different filters for stereo output (so this framework could be valid for studying HRTF[23] applications in addition to reverberation).

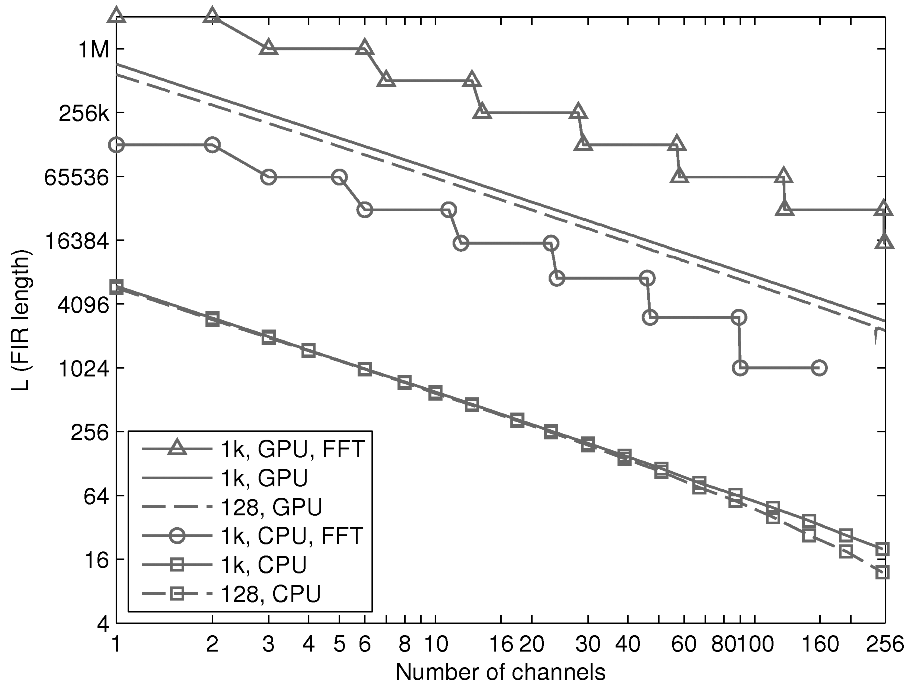

Figure 2.14: Maximum FIR length for two real-time convolutions in the time domain as a function of the number of channels to which they are applied. CPU vs GPU comparison as in [44], where two possible buffer sizes are considered (128 samples and 1024 samples).

The *CUDA* implementation consists of two phases: in the first phase, $2 \times (N + L - 1)$ threads are mapped to each input buffer (where N is the buffer size and L is the filter length). In the second phase the results of the first phase are summed together for final stereo output. This *CUDA*-based implementation turns out to be faster than a straightforward CPU implementation in all examined cases. Figure 2.14 reports the maximum length of the FIR filters in order to achieve real-time

[23] *HRTF*: Head Related Transfer Function.

processing. This is plotted as a function of the considered number of channels for two possible buffer sizes.
The figure shows that the GPU is able to handle 130 times longer filters than those the CPU can handle in real-time. It also shows that the computation time is nearly independent of the buffer size: in practice, quite short buffers can be efficiently used for time-domain convolution on the GPU.

2.8 About Recursive Filters in a Parallel Computing Scenario

The parallel computing scenario brings many outstanding features to the audio programmer. Some operations are immediately recognizable as excellent candidates for being parallelised and cast to the GPU (e.g., FIR filtering with long impulse responses, as used in convolution reverb). Unfortunately, some other operations raise legit doubts about what can be earned in terms of performance, since they do not fit the GPU architecture well (at least not immediately): this is especially true for recursive operations where data dependency prevents the possibility of instantiating a set of parallel threads on independent data (i.e., each step of the computation relies on data which resulted from the previous steps). This kind of operations are likely to perform worse on the GPU than on the CPU simply because the resulting thread must be performed on a single (much simpler) core. Conversely, GP-GPU computing is based on harnessing many simple cores simultaneously, performing the very same instructions in parallel.

In the digital audio processing scope, a very common example of an operation which is not a good candidate for GPU computing is the application of IIR filters to an input signal x:

$$y_{[n]} = \sum_{i=0}^{P} b_i x_{[n-i]} - \sum_{j=1}^{Q} a_j y_{[n-j]} \tag{2.19}$$

When considering a GPU implementation, the problem with IIR filters is indeed their *recursivity*: at any time n, any IIR filter equation relies on past outputs to compute the current output. This means that it is not possible to exploit data parallelism using many independent threads on the GPU to compute different output samples: each but the (conceptually) first thread would be "tied" to the result of the (conceptually) previous thread. As a result, a straightforward implementation of the filter equation would imply a single, sequential thread on the GPU: not only this would not at all exploit the computing resources of the GPU, but it would also result in terrible performance, given that single GPU cores are not at all comparable with any modern CPU in terms of single-thread speed of execution, not to mention the unavoidable memory transfer latency implied by GP-GPU computing.
Providentially, researchers have studied alternative methods to implement the recursive equation of IIR filters with an arbitrary level of parallelism, in order to exploit parallel architectures. In this section, two techniques in particular are going to be presented.

Trebien - 2009

The first is an approach presented by Fernando Trebien in his master thesis [45] and reprised in [46]. In these publications Trebien proposes a clever method to eliminate, up to a certain degree, the data dependencies that would emerge from a parallel execution of a linear time-invariant[24] recursive filter equation[25], making the filter implementation more suitable for GPU processing: "the problem is solved by unrolling the equation and "trading" dependences on samples close to the current output for preceding ones, thus requiring only the storage of a limited number of previous output samples".
Trebien shows that this technique not only provides an adequate increase in the number of coefficients that, in many cases, can be used for real time filtering, but it also allows to eliminate time consuming data transfers from device memory to host memory and back when the filtering operation is part of a complex processing chain and it is coupled with other audio processes to be executed on the GPU. The core of the technique can be synthesized by the following relations:

$$y_{[n]} = \sum_{i=0}^{P} b_i x_{[n-i]} - \sum_{j=1}^{Q} a_j y_{[n-j]} \;\Rightarrow \cdots \Rightarrow$$
$$\Rightarrow\; y_{[n]} = \sum_{k=0}^{m-1} c_k \left(\sum_{i=0}^{P} b_i x_{[n-i-k]} \right) + \sum_{j=0}^{Q-1} d_j y_{[n-m-j]}$$

Where P and Q are the feedforward and feedback filter orders, respectively, b_i and a_j are the coefficients of the feedforward and feedback parts, respectively, $x_{[n]}$ is the input signal, m is an arbitrary number of samples that describes the chosen amount of unrolling and c_k and d_j are coefficients that can be related to a_j and b_i by means of the following relations:

$$C = [c_0, c_1, \dots, c_{m-1}]^T = [D^{(0)}_{[0]}, D^{(1)}_{[0]}, \dots, D^{(m-1)}_{[0]}]^T \tag{2.20}$$

$$\textit{with} \quad D^{(m)} = [d_0, d_1, \dots, d_{Q-1}]^T \tag{2.21}$$

$$\textit{where} \quad D^{(k+1)} = -D^{(k)}_{[0]} A + S_{Q\times Q} D^{(k)} \quad \textit{and} \quad D^{(0)} = [1, 0, \dots, 0]^T \tag{2.22}$$

$$\textit{where} \quad S_{Q\times Q} = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ 0 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \quad \textit{and} \quad A = [a_1, a_2, \dots, a_Q]^T \tag{2.23}$$

$D^{(k)}$ represents the vector D at the m-th step of unrolling and the expression $D^{(k)}_{[i]}$ is the i-th coefficient of vector $D^{(k)}$. The reader may refer to [45] for the missing

[24]The method presented in this publication is valid for time-invariant filters only and cannot be easily extended to handle time-variant filters in general.

[25]The topic of IIR filters had been quickly faced in his bachelor thesis but this feature was not actually implemented due to the lack of a straightforward way of writing such a shader program, as it would not fit the GPU architecture well. See section 2.1 for more details on Trebien's bachelor thesis.

intermediate steps and for a more detailed explanation. As the author points out, “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”. In this way, once the appropriate value of m is chosen, all c_k and d_k coefficients can be precomputed via the algorithm reported above: these coefficients and the b_j coefficients can then be treated as impulse responses and the filtering can be applied in the frequency domain:

$$y_{[k]} = \mathcal{F}^{-1}[\mathcal{F}(c_n)\mathcal{F}(b_n)\mathcal{F}(x_{[n]})] + \mathcal{F}^{-1}[\mathcal{F}(d_n)\mathcal{F}(y_{[n-k]})] \tag{2.24}$$

, where all signals are conveniently zero-padded in order to get linear convolution instead of circular convolution and the Fourier transform operator is to be interpreted as a DFT operated via the cuFFT library ([21]). Note that, as the c_n, b_n and d_n coefficients are known beforehand, the transform of these coefficients needs to be computed only once (i.e., it can be pre-computed): only two transforms need to be computed at each processing step ($\mathcal{F}(x_n)$ and $\mathcal{F}(y_{[n-k]})$); samples of $x_{[n]}$ are obtained by accessing the current and previous input blocks while samples of $y_{[n-k]}$ are obtained from previous output blocks only. Block-wise filtering was performed by the author in an *overlap-and-save* fashion.

To demonstrate the effectiveness of this method, the author developed a program that synthesizes in real time the sounds triggered by events emerging from a simultaneous run of an object motion and collision simulation program: on the GPU, two LPC[26] all-pole shaping filters (recursive by nature) are applied to conveniently synthesized excitation signals. The two filters were designed and computed in MATLAB using, for the recursive section, a number of coefficients between 16 and 128: these coefficients were extracted from recorded sounds through linear predictive coding. The residual signals extracted from the LPC method were modeled appropriately so that approximate versions could be synthesized in real time. The samples of the synthesized residuals were eventually used during the final synthesis stage as coefficients of the non-recursive section of the two filters. The synthesis of the residual signals and the filtering were implemented in *CUDA C*[27] (the computations to take place on the GPU) and they were designed to be triggered by events (impulse signals) in the object collision simulation, also running on the GPU simultaneously.
The benchmarking stage of this system aimed at testing how many filter coefficients could be handled in real-time by the GP-GPU application, using different frame sizes, and comparing these results to those obtained by running the audio processes (synthesis and filtering) on the CPU only[28]: while the order of the recursive section

[26] *LPC*: Linear Predictive Coding

[27] As it was published in 2009, for the development of his master thesis Trebien could rely on version 2.0 of NVIDIA *CUDA* platform (drivers, toolkit and compiler); AMD/ATI’s *Close To Metal* platform was also available at the time but it was not chosen due to the lack of a native FFT library (needed for filtering in the frequency domain) and was considered more complex to work with. *OpenCL* was made available to developers only when this project was already in progress, so it was not really an option.

[28] While, for the filtering process, the *CUDA* application was based on the cuFFT library provided by NVIDIA ([21]), the CPU version exploited the FFTW library ([43]). Both libraries are considered highly optimized for their particular platforms.

of the filter was kept fixed, different numbers of coefficients for the non-recursive section were tested (starting from 8192, up to more than one million coefficients). Unfortunately, the exact number of coefficients used for the denominator of the transfer function is not clear; however, it is possible to speculate that 32 coefficients were used, as the author states that using 32 coefficients produced a stable transfer function for the recorded sounds while modelling the sources' resonances acceptably[29]. The signal stream consisted of two channels of 32-bit floating-point samples at a sampling rate of 44.1kHz.
Tests were run on a 1.86Ghz Intel Core 2 Duo E6300 with 2GB of DDR2 RAM at 333MHz, flanked by an NVIDIA GeForce 9800 GTX graphics card with 768MB of memory (x16 PCI-x), all mounted on a Gigabyte 945GM-S2 motherboard.
The results show that, on average, the GPU was capable of handling, in real-time, filters with 2 to 4 times more coefficients than the CPU (See table 2.2).

Table 2.2: Number of coefficients achievable for the non-recursive part of a real-time IIR filter, for different block sizes. CPU-based results are compared to GPU-based results ([45]).

Block Size	Non-recursive order (CPU)	Non-recursive order (GPU)
128	32768	-
256	65536	131072
512	131072	262144
1024	131072	524288
2048	262144	1048576

Even though this might not be an outstanding improvement *per se*, it is important to stress that these results show how even recursive filters (by nature not suited for straightforward parallel computing) can be implemented on the GPU in a real-time application, possibly with even more coefficients. The groundbreaking aspect of this result is to be found in those applications that need recursive filters inside a composite signal processing chain: with this method, it is possible to cast the whole computation chain to the GPU, removing any need of breaking it for data transfer between host and device, which is typically the bottleneck for GP-GPU programs.
It is worth noting that, for a block size of 128 samples, the GPU implementation could not achieve real-time performance for any number of coefficients, probably due to the overhead of *CUDA* kernel launch calls. This confirms a trend in real-time audio GP-GPU applications: small block sizes are usually more likely to be problematic. On the other side, when large block sizes are used, GPU application typically achieve better results, as confirmed by data in table 2.2 for increasing frame sizes (from 256 samples up).
Other audio applications that may benefit from this technique include equalization, multi-band dynamic compression, vocoder, modal synthesis and subtractive

[29]Moreover, in audio applications, using more than 16 coefficients in the recursive section of a filter is rare in practice.

synthesis.

It is interesting to compare this work to that of Zhang, Ye and Pan [31]: while Trebien finds a smart way of exploiting the high throughput capabilities of graphics processors for computing the results of a single recursive filter in a single step launching many different threads that cooperate for the achievement of one common result, Zhang et al. exploit the same feature in a complementary way, as they launch many recursive filters in parallel on independent data, relying on the straightforward implementation of the filter which computes the current output using saved values of past outputs obtained from previous steps[30].

Bradford - 2015

It is quite interesting to compare Trebien's method with Russel Bradford's simpler solution to the very same problem, as illustrated in his "A short note on long recursion" [47]. Bradford aims at computing a "long" recursive filter

$$y_{[n]} = x_{[n]} + a_0 y_{[n-1]} + a_1 y_{[n-2]} + \cdots + a_{[N-1]} y_{[n-N]} \tag{2.25}$$

in a parallel fashion (in particular, he is interested in cases where N is large, e.g., $N = 176400$). The method proposed by Bradford starts from the assumption of a typical audio architecture that works using a block-wise processing scheme with b samples per block; considering the b output samples obtained by processing one block:

$$\begin{aligned}
y_{[n]} &= x_{[n]} + (a_0 y_{[n-1]} + a_1 y_{[n-2]} + \cdots + a_{N-1} y_{[n-N]}) \\
y_{[n+1]} &= x_{[n+1]} + a_0 y_{[n]} + (a_1 y_{[n-1]} + \cdots + a_{N-1} y_{[n-N+1]}) \\
y_{[n+2]} &= x_{[n+2]} + a_0 y_{[n+1]} + a_1 y_{[n]} + (\cdots + a_{N-1} y_{[n-N+2]}) \\
&\vdots \\
y_{[n+b-1]} &= x_{[n+b-1]} + a_0 y_{[n+b-2]} + a_1 y_{[n+b-3]} + \cdots + \\
&\qquad + (a_{b-1} y_{[n-1]} + \cdots + a_{N-1} y_{[n-N+b-1]})
\end{aligned} \tag{2.26}$$

the operations highlighted by parentheses are independent of all $y_{[i]},\ i \geq n$. These groups of operations can actually be computed in parallel, both "horizontally" and "vertically". For large N this is the majority of the terms of the computation (note that, evidently, a small sequential part to top up the partial sums is needed). The total time complexity of this method is $O(b^2 + logN)$ per block (or $O(b + \frac{1}{b}logN)$ per output sample) when $O(Nb)$ processors are used.
Bradford also shows an improved method featuring the use of FFTs to compute partial sums. In order to do this, a slightly different blocking structure needs to be

[30]More details on [31] can be found in Section 2.1

considered:

$$
\begin{aligned}
y_{[n]} &= x_{[n]} + a_0 y_{[n-1]} + a_1 y_{[n-2]} + \cdots + \\
&\qquad + a_{b-1} y_{[n-b]} + \big(a_b y_{[n-b-1]} + \cdots + a_{N-1} y_{[n-N]}\big) \\
y_{[n+1]} &= x_{[n+1]} + a_0 y_{[n]} + a_1 y_{[n-1]} + \cdots + \\
&\qquad + a_{b-1} y_{[n-b+1]} + \big(a_b y_{[n-b]} + \cdots + a_{N-1} y_{[n-N+1]}\big) \\
y_{[n+2]} &= x_{[n+2]} + a_0 y_{[n+1]} + a_1 y_{[n]} + \cdots + \\
&\qquad + a_{b-1} y_{[n-b+2]} + \big(a_b y_{[n-b+1]} + \cdots + a_{N-1} y_{[n-N+2]}\big) \\
&\vdots \\
y_{[n+b-1]} &= x_{[n+b-1]} + a_0 y_{[n+b-2]} + a_1 y_{[n+b-3]} + \cdots + \\
&\qquad + a_{b-1} y_{[n-1]} + \big(a_b y_{[n-2]} + \cdots + a_{N-1} y_{[n-N+b-1]}\big)
\end{aligned}
\tag{2.27}
$$

Now the groups of operations in the parenthesis (i.e., the partial sums) are all independent of $y_{[n-1]}$ and up; they can be seen as convolution sums (or, equivalently, as an FIR filter applied to past output samples) and they can be computed in the frequency domain, using FFTs[31]. Note that these partial sums are shorter than before, and thus a lower level of parallelism is exploited, but at the same time frequency domain multiplication can be used (which is faster for a large N). The resulting time complexity turns out to be the same as the non-FFT method, but using only $O(N)$ processors as opposed to $O(Nb)$. Thus, this is theoretically more efficient.

Bradford tested these two approaches on quad-core Intel i7 920 at 2.67GHz coupled with a GeForce GTX 470 with 448 cores at 1.22GHz. He measured the time to process one second of double precision samples (44100 samples) with an IIR filter with 176400 coefficients for the recursive part (and just one coefficient for the non-recursive part, as in the equations above). The block size was set at b = 512 and *CUDA* 5.0 was used for the GPU implementation. The results can be found in table 2.3.

Table 2.3: Processing times (in seconds) for applying an IIR filter to 44100 samples, using two different methods (dot product and FFT-based method) on the CPU and on the GPU ([47]).

	CPU version (sequential)	**GPU version (parallel)**
Dot product	8.7 sec	1.1 sec
FFTs	1.1 sec	0.14 sec

A 4-threaded parallel FFT on the CPU was also tested with a resulting time of 0.7sec. Note that the GPU implementation using FFTs greatly outperforms all other implementations and it is the only one capable of real-time processing. Theoretically, a 24 seconds echo (N = 1058400) could be applied in real-time. Using single precision floats, a 56 seconds echo (N = 2469600) could be applied in real-time.

[31]CPU version is based on FFTW ([43]), while GPU version is based on cuFFT [21].

2.9 Spectral Processing

This thesis is focused on the specific sub-field of phase vocoder processing[32] on GPUs. Although not extensively, this very topic has already been investigated in the past and a few examples are going to be reported here. Most of the widespread spectral processing frameworks rely on the Fourier transform in some way. Once again, the possibility of employing, in this scenario, SIMD implementations of the FFT algorithm is very attractive.

Bianchi and Queiroz - 2012

In [48], Bianchi and Queiroz implement GPU-based versions of plain FFT analysis and full phase vocoder analysis/re-synthesis in the *Pure Data*[33] environment. Their aim is that of learning the maximum frame sizes for which each task is feasible in real-time on commodity GPUs. They exploit *Pure Data*'s extensible design to implement the interaction with the GPU using *C* and *CUDA C* code compiled as shared libraries, in a very similar way to what will be presented in chapter 3 (in a *Csound* environment, instead).
The FFT analysis module is very basic: it simply takes a signal frame, transfers it to device memory, applies an FFT transformation and transfers the resulting spectral data back to the host. The full phase vocoder module uses the same amount of data transfer but comprises a lot more computations. In fact, after the forward FFT is performed, it needs to translate amplitude/phase information into amplitude/frequency information and the whole process is reversed for re-synthesis, using additive synthesis instead of inverse Fourier transforms. Different methods for additive synthesis are investigated: table lookup with 4-point cubic interpolation, table lookup with 2-point linear interpolation, table lookup with truncated index, direct use of the `sinf()` primitive and table lookup with linearly interpolated texture fetching.

The test system is a 2.67GHz 8-core Intel Core i7 CPU 920 with 6GB of RAM running Ubuntu 11.04. Two NVIDIA GPUs are employed for the tests: a GeForce GTX275 (240 *CUDA cores*, 896MB RAM, 127.0GB/s memory bandwidth) and a GeForce GTX470 (448 cores, 1280MB RAM, 133.9 GB/s). FFTs are implemented via the cuFFT library ([21]) but not many more details are provided on the *CUDA C* implementation of the phase vocoder.
The authors run the two *Pure Data* modules for a period equal to 100 DSP blocks, with various frame sizes of 2^i (with $6 \leq i \leq 17$) and they measure the mean time for data transfers alone and for the full round-trips. The results regarding the plain FFT analysis module are reported in figure 2.15 and those regarding the full phase vocoder analysis/re-synthesis module are reported in figure 2.16.

[32]See section 1.2 for a detailed discussion about phase vocoder processing.
[33]*Pure Data*: `https://puredata.info/`

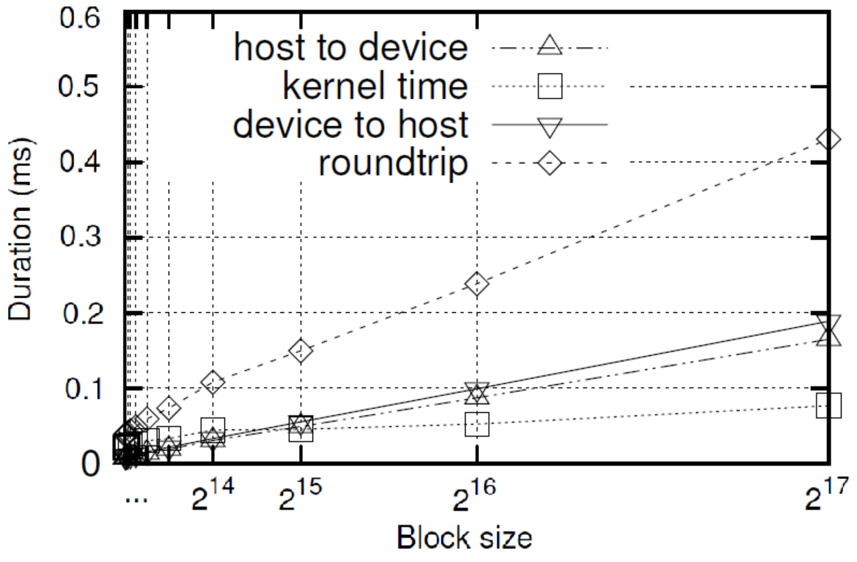

Figure 2.15: Memory transfer times and kernel times for a *Pure Data* FFT analysis module running on a GeForce GTX470 for different block sizes, as in [48].

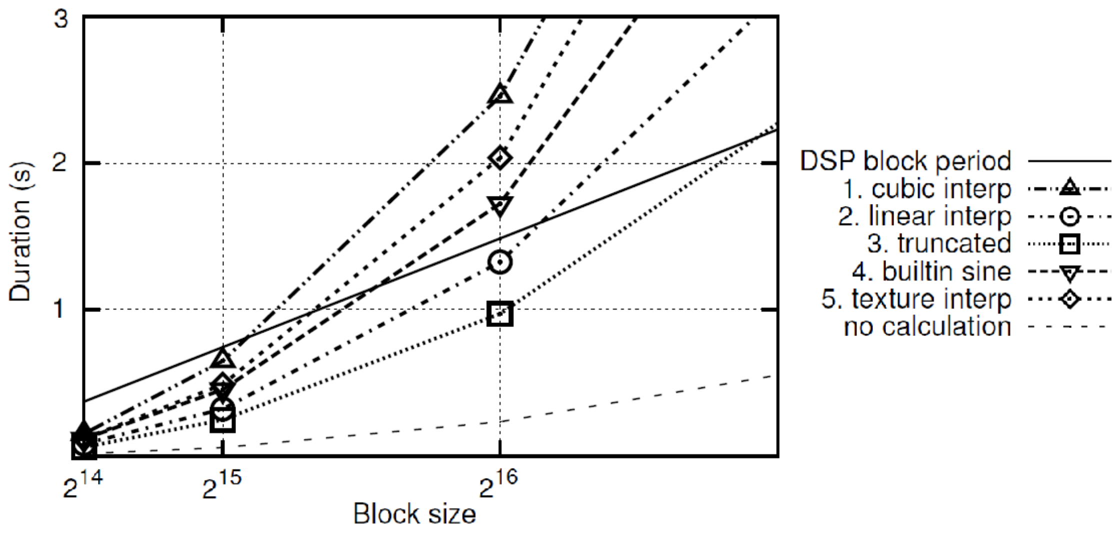

Figure 2.16: Execution time for a *Pure Data* full phase vocoder analysis/re-synthesis module running on a GeForce GTX470 for different block sizes, as in [48]. Different lines indicate different approaches to re-synthesis.

By looking at figures 2.15 and 2.16, it can be learned that there is a noticeable difference, of some orders of magnitude, between the time it takes to run the plain FFT module and the full phase vocoder module: hundreds of pure FFT executions could occur in a DSP cycle while only a few phase vocoder round-trips can actually be performed in the same time. Memory transfer times seem to grow linearly with the employed block size. A comparison between the two GPUs (not shown in figure 2.16) reveals that the GTX275 is slower in all cases and it fails to operate in real-time for any block size bigger than 2^{15}. The GTX470 performs better and it can operate phase vocoder round-trips in real-time also for a block size of 2^{16} (but not bigger). The maximum block size achievable in each scenario is reported in table 2.4.

Table 2.4: Maximum block size achievable in real-time for full phase vocoder round-trips using different methods for the re-synthesis stage, as reported in [48] (1: table lookup with 4-point cubic interpolation; 2: table lookup with 2-point linear interpolation; 3: table lookup with truncated index; 4: direct use of the `sinf()` primitive; 5: table lookup with linearly interpolated texture fetching). The performance of two NVIDIA GPUs is compared.

Model ╲ Method	**1**	**2**	**3**	**4**	**5**
GTX 275	2^{14}	2^{15}	2^{15}	2^{15}	2^{14}
GTX 470	2^{15}	2^{16}	2^{16}	2^{15}	2^{15}

No comparison between CPU-based and GPU-based execution is provided. Yet, the feasibility of a real-time GPU-based phase vocoder framework in *Pure Data* is assessed.

Bradford et al. - 2011

In [49], Bradford et al. implement an SIMD version of the *sliding phase vocoder* (SPV) framework, which is designed to run on graphics processors via *CUDA*. The SPV is a variation on the classic frame-based streaming phase vocoder that makes use of the Sliding Discrete Fourier Transform (SDFT, see next) instead of standard STDFTs[34] in order to perform a frame update for each output sample, thus reducing the hop size to one single sample. This approach offers musical advantages ([50]), including lower latency and the potential for new classes of effects like *transformational FM* (see [51], for instance). On the downside, the SDFT (and hence the SPV) is characterised by a very high computational cost, which makes it intractable for real-time use, even on high-end consumer workstations. However, massively parallel architectures are excellent candidates for accelerating this algorithm, as it is perfectly congenial to parallel computing.
The authors show that, given the complex values $X_{n[k]}$ of a frequency-domain frame at time instant n, the DFT of the input signal at time $n+1$ can be expressed as:

$$\begin{aligned} X_{n+1[k]} &= DFT\{x_{[n+1]}\} \\ &= \sum_{l=0}^{N-1} x_{l+n+1} e^{-j2\pi l \frac{k}{N}} \\ &= \sum_{l=1}^{N} x_{l+n} e^{-j2\pi (l-1) \frac{k}{N}} \\ &= \left(\sum_{l=0}^{N-1} x_{l+n} e^{-j2\pi l \frac{k}{N}} - x_n + x_{n+N} \right) e^{j2\pi \frac{k}{N}} \\ &= \left(X_{n[k]} - x_n + x_{n+N} \right) e^{j2\pi \frac{k}{N}} \end{aligned} \tag{2.28}$$

[34] *Short Time Discrete Fourier Transform.*

Thus, the cost of hopping by just one sample is N complex multiplications and $2N$ additions (the value of $e^{j2\pi\frac{k}{N}}$ can be precomputed), where N is the DFT size. Note that, for each frequency bin k, the computation of the new value is independent of the other bins. The computation of 2.28 for each k is thus very much congenial for a SIMD-like parallel scheme. In this work, time domain samples are obtained by means of the basic IDFT formula. A problem with propagating numerical errors may arise at typical sampling rate (44.1kHz) when single precision is employed: the authors suggest using double precision for these computation (which is now available in modern GPU cores).

A few programs are written by Bradford et al. in a combination of *C* and *CUDA C*. The first application is a full SPV analysis/re-synthesis program running at a sampling rate of 44.1kHz, with tunable number of audio channels and fixed DFT size (1024 bins). Table 2.5 shows the execution times recorded for a few tests with varying number of channels. The system used by the authors for these tests is a 4-core Intel i7 desktop machine running at 2.67GHz accelerated by an NVIDIA GeForce GTX470 (14 SMs, for a total of 448 *CUDA cores* and 1.3GB RAM).

Table 2.5: Execution times for full sliding phase vocoder round-trips producing a 1s sound on a GeForce GTX470 GPU. Results from [49].

No of channels	Time to produce a 1s sound
1	0.56s
2	0.59s
4	0.70s
8	0.88s

All these results are under the real-time threshold. This confirms that real-time SPV processing is effortlessly achieved when cast to a consumer-level GPU. This is a groundbreaking accomplishment as, before this implementation, SPV processing was only possible on expensive and highly specialised plug-in vector hardware ([52]). Note that the execution time grows sublinearly with the amount of data involved (number of channels): this means that memory transfer latency is having a critical impact on the tests with a low number of channels, which is more and more softened as the number of channels is increased, which in turn increases the utilisation level of the GPU cores.
In order to demonstrate the efficacy of this GPU-based real-time SPV framework, Bradford et al. implemented two other applications: a high quality[35] pitch shifting program and a *transformational FM* program. They do not show execution times but they state that both applications can be run in real-time for eight channels of audio on the target system.

[35]The fact that pitch shifting is implemented in a sliding framework as opposed to a “hopping” one provides for a better quality, i.e., less artefacts.

Lazzarini et al. - 2014

In [8], Lazzarini et al. design and test a *CUDA*-based phase vocoder framework (both sliding and standard versions) in a *Csound* environment (similarly to what was done in [48], even though Bianchi and Queiroz dealt with a *Pure Data* integration). In addition, the authors design an additive synthesis program as an alternative to inverse FFT for phase vocoder re-synthesis and they compare these two implementations.
The phase vocoder synthesis and re-synthesis *Csound* modules are implemented by means of *CUDA* kernels that directly apply the processing steps described in section 1.2, exploiting an SIMD model (see [8] for a detailed explanation of each single kernel). The sliding phase vocoder is implemented on the basis of the discussion presented in [49] and incorporates a *transformational FM* audio effect in the processing chain. Additive synthesis is accomplished in a similar way to what is presented in [32] (see section 2.2).

Table 2.6: Execution times for a 60s run of phase vocoder analysis (standard) running on a laptop on-board GPU with different DFT sizes and hop sizes. Results from [8]. Sampling rate: 44.1kHz.

(DFT size, hopsize)	Time (s)
(1024, 128)	2.95
(1024, 256)	1.68s
(2048, 256)	2.20s
(2048, 512)	1.28s

Tests were executed on an NVIDIA GT650M GPU (compute capability 3.0) with 384 cores (at 900MHz) and 1024MB RAM in a *Csound 6* environment running on OSX10.9. The results are compared with standard (sequential) CPU versions of the same algorithms running on a 2.8GHz Intel i7 processor. The results of these tests are reported in tables 2.6, 2.7, 2.8.

Table 2.7: Execution times for a 60s run of phase vocoder synthesis (standard) running on a laptop on-board GPU with different DFT sizes and hop sizes. Results from [8]. Sampling rate: 44.1kHz.

(DFT size, hopsize)	Time (s)
(1024, 128)	3.30
(1024, 256)	1.84
(2048, 256)	2.65
(2048, 512)	1.44s

The results reported in table 2.6 and 2.7 validate the possibility of offloading phase vocoder analysis and/or re-synthesis stages to a (cheap) on-board GPU without having to worry about performance issues (they run 20x faster than the real-time

limit). Of course, this leads to a gain in the available CPU resources, to be employed for other tasks.
The results of table 2.8 show that the CPU performs better in almost every case. Yet, the difference is never huge, even though the comparison is between an on-board GPU and a high-end processor. It is worth noting that the execution time of the full PV round-trip on the GPU is remarkably lower than the sum of the two separate processes (analysis and re-synthesis). This is probably due to a certain latency that is met when the GPU is called into action.

Table 2.8: Execution times for a 60s run of full phase vocoder round-trips (standard) running on a laptop on-board GPU and on a high performance CPU with different DFT sizes and hop sizes. Results from [8]. Sampling rate: 44.1kHz.

(DFT size, hopsize)	GPU time (s)	CPU time (s)
(1024, 128)	4.72	1.24
(1024, 256)	2.57	0.69
(2048, 256)	3.03	1.28
(2048, 512)	1.73	0.70
(4096, 512)	1.98	1.34
(4096, 1024)	1.20	0.74
(8192, 1024)	1.64	1.36
(8192, 2048)	1.01	0.75
(16384, 2048)	1.38	1.40
(16384, 4096)	0.86	0.77

Table 2.9 reports the results regarding the additive synthesis module (which also includes a phase vocoder analysis stage used for determining the time-varying amplitude a frequency parameters).
Additive synthesis proves to be more expensive in terms of computational load, but, overall, the results are still well within the range of low-latency, real-time performance. It can be acknowledged that additive synthesis is a good match for the GPU, especially when the parallel computational load is high (synthesis involving a high number of frequency bins) and the process granularity is low (high hop sizes). In general, GPUs are more suited to processing larger batches of data, which is of course in opposition to the requirements of streaming audio processing. Nevertheless, the results are very promising.
Finally, table 2.10 shows the results regarding the sliding phase vocoder.
While real-time SPV processing with a sequential computing approach is clearly (and predictably) out of reach on an off-the-shelf laptop computer, the adoption of the parallel computing model drastically changes this scenario, making SPV processing possible even with an on-board GPU. Overall, with these tests, Lazzarini et al. prove that consumer-level GPU processing can be harnessed for spectral audio and additive synthesis applications.

Table 2.9: Execution times for a 60s run of phase vocoder analysis plus additive re-synthesis running on a laptop on-board GPU and on a high performance CPU with different DFT sizes, hop sizes and number of frequency bins employed for the additive synthesis. Results from [8]. Sampling rate: 44.1kHz.

(DFT size, bin, hopsize)	GPU time (s)	CPU time (s)
(1024, 128, 128)	4.93	3.28
(1024, 128, 256)	3.70	3.01
(1024, 256, 128)	7.20	5.77
(1024, 256, 256)	3.37	5.46
(1024, 512, 256)	4.20	10.76
(2048, 256, 512)	3.04	5.65
(2048, 512, 512)	3.94	10.55
(2048, 1024, 512)	6.87	20.89

Table 2.10: Execution times for a 60s run of full sliding phase vocoder round-trips running on a laptop on-board GPU and on a high performance CPU with different DFT sizes. Results from [8]. Sampling rate: 44.1kHz.

DFT size	GPU time (s)	CPU time (s)
512	33.05	68.794
1024	37.98	138.29
2048	54.99	272.33

2.10 Conclusions

This chapter reported a few examples of research studies that investigate the GP-GPU computing framework applied to a wide variety of audio-related applications such as sound synthesis or audio processing. The congeniality of many audio-related algorithms to parallel computing is indeed very attractive and provides for impressive performance speed-ups if carefully exploited on GPUs.

Section 2.1 is dedicated to older research studies dealing with audio programming based on GPU architectures from the early days of heterogeneous parallel programming, before the appearance of the *unified shader architecture*. Here, the focus is set on the techniques employed for GPU programming, by means of exploiting the hardware graphics pipeline, rather than on specific audio topics. The key aspect of these works is the need for twisting the original problem in order to recast it in terms of graphics programming. Despite the low level of parallelism achievable by older GPUs, these studies provide promising results and prove that GPU-enhanced audio processing is very attractive for some applications (e.g., FIR filtering, waveform synthesis and modal synthesis).

Sections 2.2 to 2.4 are dedicated to sound synthesis, by means of different methods. Section 2.2 deals with the specific topic of additive synthesis with an

extremely high number of sine waves. The GPU proves to be an excellent candidate for executing this kind of applications with better performances, allowing the real-time synthesis of audio signals obtained from the superposition of almost 2 million sine waves.
Section 2.3 deals with spectral model synthesis and shows that complex simulations involving an extremely high number (tens of hundreds) of high quality instruments (50 harmonics each) can be run in real time on a GPU (speed-ups in the order of a few dozens with respect to CPU execution).
Section 2.4 is dedicated to finite difference methods for physically-based sound synthesis. This technique benefits a great deal from parallel computing, especially when extremely large simulations are considered (with high spatial resolution), for high quality sound rendering. For instance, the sound of a percussed membrane can be simulated in real-time on a GPU using a grid of 84x84 sampling points.

Section 2.5 is dedicated to the wide area of room acoustics modelling. This field is one of the most attractive for GP-GPU computing, and one of the most studied. A variety of methods can be applied, and all of these can benefit from a parallel computing model. Real-time simulations are still out of reach for large rooms with decent spatial resolution. However, much faster simulations can be achieved when casting the computations on the GPU, with speed-up factors in the order of a few dozens.

HRTF rendering is discussed in section 2.6. This is actually a particular application of signal convolution (typically computed via FIR filtering). FIR filtering can be done very efficiently on parallel architectures, especially when it is applied in the frequency domain, thanks to the existence of fast parallel versions of the FFT algorithm and to the possibility of performing parallel element-wise multiplications of frequency bins. Tens of hundreds moving virtual sources can be filtered on the GPU in order to get headphone-based spatial sound, while keeping the CPU load to minimum levels. The efficiency of data-parallel FFT implementations is a crucial aspect in countless audio-related GP-GPU applications, not only in FIR filtering processes. As a matter of fact, the motivations for the development of GPU-based spectral processing algorithms, the main objective of this thesis, strongly rely on the existence of fast data-parallel FFT implementations.

Section 2.7 deals with the possibility of using the GPU in the context of music production (i.e., for tracking, mixing and mastering purposes). It turns out that some applications benefit quite a lot in terms of speed-up with respect to CPU-based implementations (e.g., reverberation[36]) while others are not very congenial to parallel processing (e.g., equalisation via recursive filters). Still, in those cases that do not show a clear improvement over CPU execution, it can make sense to employ the GPU anyway, in order to free up CPU resources for other possible concurrent tasks, provided that the performance of the GPU implementation is somewhat comparable to that of the CPU.

Section 2.8 delves into the delicate topic regarding the implementation of recursive (IIR) filters on parallel computing architectures. This is a kind of algorithm that is not implementable on parallel processors in a straightforward way, due to its intrinsic recursive nature which causes data dependencies. Nevertheless, in the

[36]Which is again an application of FIR filtering.

past years, researchers have designed alternative approaches that are more suited for parallel computing architectures like GPUs. Thanks to these techniques, high order IIR filters can be computed in real time on the GPU even faster than on the CPU, with speed-ups in the order of one dozen.

Finally, section 2.9 brings the attention back to the topic of spectral processing applications (like the phase vocoder framework). This section presents a few works that are very close to the research task that will be addressed in the following chapters of this thesis. In these publications, the feasibility of a GPU-based real-time phase vocoder framework is assessed. More importantly, the possibility of executing sliding phase vocoder analysis and re-synthesis in real-time on affordable and general-purpose (parallel) hardware is proven as well. This feature in particular allows audio programmers to design a number of novel real-time digital effects on unspecialised hardware (modern GPUs).
The GP-GPU computing approach has recently been explored for other audio-related applications that have not been reported in this review. Some of these include: wavefield synthesis ([53]), beamforming with microphone array techniques ([54]) and acoustic likelihood computations for speech recognition ([55]).

To summarise, GPUs are well suited for computationally demanding audio applications that can be expressed in a parallel (SIMD) fashion. Only a few limitations can be highlighted. For instance, it is essential to spawn enough concurrent threads on the GPU so that it is fully (or at least abundantly) utilised[37], and, in practice, this often raises the need for higher buffer sizes, which unfortunately translate in increased input-output latencies. However, in many practical cases this cost is acceptable and more recent hardware is providing lower and lower latencies. Then, for some key applications (e.g., gaming), a very limited amount of GPU resources is generally available so care must be taken when designing GP-GPU audio processes in these contexts. The key factor that limits GPU efficiency for audio processing is certainly the limited communication bandwidth between the GPU and the CPU, even with current PCI-x buses. Anyway, some applications (e.g., most synthesis processes) do not actually need any data to be transferred to the device, since all the data could be generated on the device itself. Also, for a very limited number of applications it could be possible to skip the *device-to-host* memory transfer and output the processed/synthesised sound directly to the HDMI port of the video card.

[37]This is a key concept of *throughput-oriented* computing: there is no point of using GPU cores if only a small number of them is involved in the computations.

Chapter 3

Research Task and Implementation

In this chapter, the very core of this work will be presented: the research task will be faced and explained thoroughly, together with the actual implementation stage of the target applications, the employed tools and the overall framework.

3.1 Tools

Two main tools were used in the development of this work: *Csound*[1], as the main audio programming environment, and NVIDIA's *CUDA*[2], as an API to gain access to the GPU, manage host-device data transfers and cast computations to the device. The objective is that of writing and testing *Csound* modules that implement spectral processing algorithms on *CUDA*-enabled GPUs, in order to seek a performance improvement.

3.1.1 *Csound*

Csound is a domain-specific programming language (DSL) for audio computing, sound design and *computer music* composition. It can also be described as an *audio compiler* in the sense that it interprets textual instructions in the form of source code and translates them into digital audio. It is a *C*-language-based cross-platform and open source free software[3], available under the GNU Lesser General Public License (LGPL[4]). It is maintained and expanded by a core of developers with support from a wider global community.

Csound can be thought as a series of layers, with various 'modes of entry' for different kinds of users and for related applications [56]. At the lowest level, *Csound* is a self-contained audio programming language implemented in a cross-platform library, with a well-defined API which allows software developers to create programs for audio synthesis and processing. It supports a variety of synthesis techniques and

[1] *Csound*: `http://csound.github.io/about.html`
[2] *NVIDIA CUDA*: `http://www.nvidia.com/object/cuda_home_new.html`
[3] *Csound*'s source code is available at `http://csound.github.io`
[4] *GNU Lesser General Public License*: `https://www.gnu.org/copyleft/lesser.html`

allows various means/levels of internal and external control. As it will be shown in a few lines, it is also widely and easily extensible via custom modules.
At the middle layer, the *Csound* language is used for writing programs for performance, composition, and other audio processing tasks such as *sonification*. At this level, the system allows the composer to design virtual instruments, and to control them in real time or deferred time.
At the highest level, the user unconsciously exploits *Csound*'s features while accessing applications that are based on the middle and lower levels. This is seen, for instance, in some frontends, such as *Cecilia*[5] and *blue*[6], where the user might only need to deal with parameter setting in the graphical interface, in plugins or applications generated by *Cabbage*[7], in bundled packages such as *CsoundForLive*[8], or in mobile applications for iOS[9] and Android[10].

Csound's syntax is based on the use of *unit generators* (also called "*ugens*", or "*opcodes*" in the *Csound* jargon), i.e. basic formal units that constitute the building blocks for designing synthesis and signal processing algorithms. There are *unit generators* for very basic tasks and operations (e.g., for the communication with A/D and D/A converters for input-output purposes), as well as for more involved applications, such as applying effects to an audio stream or even playing a virtual instrument based on physical modelling synthesis principles. There are currently nearly 1700 *unit generators* available to the user and directly interpreted by the compiler itself but one of *Csound*'s greater strengths is that it is completely modular and extensible by the user, as new and custom *unit generators* are very easily implemented. Custom *unit generators* can be developed through two basic mechanisms: *user-defined opcodes* (UDOs), written in the *Csound* language itself and, for an even greater range of possibilities, pre-compiled/binary[11] *opcodes* written in *C* or *C++* (these are known in *Csound* parlance as "*plugin opcodes*"). A detailed guide to the *opcode* development API for writing custom *plugin opcodes* can be found in [57] (in the same publication, a subsection addresses *opcodes* that are meant to comprise spectral signals in particular).

The objectives of this work involve writing a few custom *plugin opcodes* (or, rather, "translating" existing *unit generators* to a massively parallel scheme) in order to validate the possibilities offered by GP-GPU computing in the scope of spectral processing of audio signals.

3.1.2 *CUDA*

CUDA (Compute Unified Device Architecture) is a computing platform and API model created by NVIDIA with the aim of allowing software developers to use any *CUDA*-enabled graphic processing unit (i.e. NVIDIA GPUs) for general

[5] *Cecilia*: `http://ajaxsoundstudio.com/software/cecilia/`
[6] *blue*: `http://blue.kunstmusik.com/`
[7] *Cabbage*: `http://cabbageaudio.com/`
[8] *CsoundForLive*: `http://csoundforlive.com/`
[9] *Apple iOS*: `http://www.apple.com/ios/`
[10] *Google Android*: `https://www.android.com/`
[11] In order to add the custom *opcode* to the *Csound* environment the user simply needs to compile and build the *C* (or *C++*) code as a shared, dynamic library.

purpose computing, giving direct access to the GPU's virtual instruction set[12]. The computing power of NVIDIA GPUs is in fact accessible to software developers through *CUDA*-accelerated libraries and extensions to *C/C++*[13] or *Fortran*, relieving them of the need for advanced skills in graphics programming and making it easier to translate general purpose algorithms in a massively parallel scheme.

The *CUDA* software development kit[14] is free and cross-platform (it is available for Microsoft Windows, Linux and Mac OSX). Nevertheless, it must be stressed that, unlike *OpenCL*, the other main framework for heterogeneous computing, *CUDA* is only usable in conjunction with NVIDIA GPUs: in particular, it works with NVIDIA GPUs from the G8x series onwards, including GeForce, Quadro and Tesla line. Notwithstanding, the *CUDA* platform supports other computational interfaces, including *OpenCL*, and third party wrappers are also available for *Python*, *Perl*, *Fortran*, *Java*, *Ruby*, *Lua*, *Haskell*, *R*, *MATLAB*, *IDL* and *Mathematica*.

CUDA plays a crucial role in this work: spectral audio computations are cast to an NVIDIA GPU by means of writing *Csound plugin opcodes* in the *CUDA C* extended language and compiling them via NVCC as dynamically loaded libraries (also known as "shared objects" in UNIX systems).

3.2 Spectral Signal Processing in *Csound*

The choice of working on spectral (or, more precisely, phase vocoder) processing *unit generators* is not arbitrary: these algorithms are in fact expected to be well suited for parallel processing[15], the working principle behind *CUDA* and GP-GPU computing in general. This kind of *unit generators* usually involves executing the very same operation on multiple phase vocoder bins without any inter-bin dependency.

Csound features a solid streaming[16] frequency-domain signal framework [58], based on the *fsig* variable type. In the *fsig* framework, *Csound* instruments can manipulate any input signal in the frequency domain: this is done by transforming the time-domain input signal in a streaming frequency-domain representation through a phase vocoder analysis *unit generator*, proceeding with the actual manipulation stage based on a spectral processing *unit generator* (or perhaps a chain of these) and finally going back to the time domain via a phase vocoder re-synthesis *unit generator*. For the streaming phase vocoder kind of processing, the analysis stage is typically carried out through the `pvsanal` *unit generator*[17], which is loosely modelled on

[12]GPUs are programmed via complex instruction sets and, for proprietary hardware related reasons, these are not publicly available. The native assembly language is usually only accessible to software developers through standardized higher level languages and APIs. The *OpenGL* virtual instruction set and *CUDA* are examples of such hardware abstraction layers on top of the specialised processor native instruction set.

[13]The resulting extended language is called *CUDA C* and it is compiled via *NVCC*, NVIDIA's LLVM-based *C/C++* compiler. *LLVM Project*: `http://llvm.org/`

[14]The *CUDA Toolkit* is available for free download at https://developer.nvidia.com/cuda-downloads.

[15]See also the following section: *Spectral signal processing with CUDA*.

[16]Here the term *streaming* is used to specify an STFT scheme where a new data frame is generated every hop size input samples.

[17]Other *unit generators* are available for similar purposes, i.e. `pvsfread`, used for reading a

the original CARL phase vocoder [16] (see section 1.2 for more details). The re-synthesis stage can be performed according to two basic principles: overlap-and-add inverse DFT (via the `pvsynth`[18] *unit generator*) and additive synthesis (mainly via `pvsadd`). The former is generally the most efficient way of re-synthesizing bin-frame amplitude-frequency data and it is the one technique that was used throughout the development of this work, since additive synthesis is better suited for *partial tracks* manipulation, which was not addressed in this work.

Csound provides a comprehensive set of streaming spectral processing *unit generators* that are designed to receive input *fsigs* and produce transformed output *fsigs* in return. They can be loosely classified in four categories: frequency transformations, amplitude transformations, modifications that involve both amplitude and frequency data and, finally, cross-synthesis effects (i.e., modules that receive two or more *fsigs* and produce a single *fsig* obtained by some kind of combination of the inputs). In this work, a few *unit generators* from each category have been examined and translated in a parallel computing scheme, namely: `pvsgain`, `pvsfilter`[19] and `pvstencil` from the amplitude transformation category, `pvscale` and `pvshift` from the frequency transformation category, `pvsmooth` and `pvsblur` from the amplitude-frequency modification category and `pvsmix` and `pvsmorph` from the cross-synthesis category. All these *unit generators* and their parallel processing (i.e. *CUDA*) counterparts will be discussed thoroughly in the next sections of this chapter.

It is meaningful to report that the phase vocoder framework is known in the *Csound* community (but also in the broader community of audio signal processing) to be a very compute-intensive one. When too many phase vocoder processes are carried out simultaneously on multiple streams, audio programmers working on resource-limited systems are often forced to deal with real-time execution barriers. This is mainly ascribable to the analysis and re-synthesis stages that involve the computation of DFTs, as described in 1.2. Given the already discussed suitability of FFTs (and spectral processes) to the parallel scheme, this concern eventually drove the development of GPU versions for the phase vocoder analysis and re-synthesis stages ([8]).

3.2.1 The *fsig* Framework

Before proceeding to an in-depth analysis of each single *unit generator*, a little more insight is needed about the working principles behind the support for streaming frequency-domain signals in *Csound* (defined by the *fsig* variable type).
From [58]:

pre-existing non-streaming phase vocoder analysis (*PVOC-EX* [59]) file (.pvx extension, generated by the `pvanal` utility), and `pvsifd`, an implementation of the instantaneous frequency distribution analysis [60]. Yet, for the purposes of this work, `pvsanal` has been sufficient and no other analysis units were used.

[18]Before applying the inverse DFT, the `pvsynth` *unit generator* operates the inverse steps of phase vocoder analysis: it takes the amplitude-frequency pairs, integrates the frequencies to obtain current phase values and converts the data from polar to rectangular representation. It then applies the inverse DFT and the resulting time-domain signal block is overlapped and added to the correct time-aligned position at the output.

[19]Actually, `pvsfilter` could be considered as belonging to the cross-synthesis category as well.

> Such signals are processed at a rate that is dependent on the size of the DFT analysis frame and the number of overlapping frames (or the hop size), effectively the rate of generation of new spectral frames. The ‘perform’-method[20] of a spectral processing *opcode* is called every control period, but it only outputs a new frame if there is a new frame at its input. The *fsig* framework provides support for such checks. Consequently, the *fsig* rate is independent of the control rate[21].
> *Fsigs* are self-describing. Unlike time-domain audio and control signals, they are furnished with the extra information about their features: DFT length, number of overlaps (N over hop size), window size, window type, data format and frame count (current frame number, starting from 0). The actual format of the spectral data can vary, currently three types are being used: `PVS_AMP_FREQ`, amplitude and frequency pairs as produced by the phase vocoder and IFD[22]; `PVS_AMP_PHASE`, amplitude and phase (polar DFT) data; and `PVS_TRACKS`, partial tracks of amplitude, frequency, phase and track ID. Of these, the first two will have a fixed size, namely the DFT size plus two extra values (holding the positive side of the spectrum generated by the DFT of a real signal plus the Nyquist frequency), or $N/2 + 1$ bins.

All the frequency-domain processing that has been done in the scope of this work is based on the `PVS_AMP_FREQ` format.
The way *fsig* variables will be defined in the development of new *plugin opcodes* is by means of the `PVSDAT` data structure, which is in turn defined in the *pstream.h* header file as:

```
typedef struct pvsdat {
  int32   N;          // DFT size
  int     sliding;    // Flag to indicate sliding case
  int32   NB;         // Number of samples in a frame in the sliding
                      // case (used instead of N)
  int32   overlap;    // Number of overlaps on each frame
  int32   winsize;    // Window size in samples
  int     wintype;    // Type of the window used for STFT analysis
                      //  0 =  HAMMING, 1 =  VonHann, 2 = Kaiser
  int32   format;     // PVS_AMP_FREQ, PVS_AMP_PHASE or PVS_TRACKS
  uint32  framecount; // Current frame number
  AUXCH   frame;      // Actual PV data
} PVSDAT;
```

The `AUXCH` data structure is in turn defined in *csoundCore.h* as:

```
typedef struct auxch {
  struct auxch* nxtchp;
  size_t size;
  void*  auxp;    // PV data is stored from this memory location...
  void*  endp;    // ...to this memory location
} AUXCH;
```

[20]For explanations about *Csound*’s internal 2-phases working mechanism, see the chapter *Initialization and Performance Pass* of the *Csound Floss Manual* [61].

[21]The only caveat is that the *fsig* framework requires the control period in samples (“`ksmps`”) to be smaller or equal to the analysis hop size.

[22]*Instantaneous Frequency Distribution* [60].

In spectral processing *unit generators*, phase vocoder data is stored in memory using a single precision floating-point representation (float) and according to the following layout: amplitude-frequency information related to successive channels is stored in an interleaved fashion, with amplitude first, followed by frequency[23]. The total number of phase vocoder channels is equal to $N/2+1$ which corresponds to twice as many floats (i.e. $N+2$, where N is the DFT size): the last channel contains data related to the Nyquist frequency[24].

3.3 Spectral Signal Processing with *CUDA*

Regardless of the specific audio programming environment adopted (*Csound* in this case), spectral signal processing is a kind of task that is usually well suited for parallel computing, thus for GP-GPU computing in particular. Frequency-domain manipulations, in fact, often involve the same operations to be repeated for each DFT bin[25]. In a latency-oriented processing scenario (e.g., typical CPUs) this is performed by implementing a `for` loop which spawns a very small number of threads (just one thread for a single-core CPU), thus scheduling the loop iterations over a relatively large number of clock cycles when the DFT size is quite large. Conversely, in a throughput-oriented processing scenario (e.g., modern GPUs) these operations can be performed no longer in a loop fashion but in a massively parallel model instead, allowing tens or even hundreds of operations to be performed simultaneously[26]. The difference between these two approaches becomes more and more critical as the number of iterations needed by the original loop increases, and, moreover, as the number and complexity of the operations to be carried out in each iteration increases as well.
With its parallel computing features, *CUDA* is thus an appropriate tool for managing the transition from a latency-oriented approach for the implementation of spectral processing algorithms (as envisaged by a more traditional programming scheme) to a throughput-oriented one.

3.3.1 About Performance: Limitations, Trade-offs and Improvements

It has to be noted that one of the strongest limitations of GP-GPU computing, i.e. host/device memory transfer latency, applies to this scenario with no exceptions. More specifically, the larger the DFT size, the worse the latency that is to be expected, but, at the same time, the more computations can be carried out in parallel. This fact implies a remarkable trade-off between the performance gain

[23]So, for instance, `myPVstream->frame.auxp` points to the lowest channel's amplitude value and `myPVstream->frame.auxp[1]` is the lowest channel's frequency value, i.e. 0Hz.

[24]Thus, for instance, `myPVstream->frame.auxp[2*N]` is the amplitude of the Nyquist frequency. Note that the Nyquist frequency itself (`myPVstream->frame.auxp[2*N+1]`) is usually ignored as it does not really carry any information: it is always equal to half the sample rate. The same applies to the 0Hz component.

[25]Of course, this also applies to the particular case of the phase vocoder.

[26]See also section 1.1, *GP-GPU Computing*.

that comes from implementing parallel-computation-congenial algorithms that involve large amounts of data (like spectral manipulation processes, usually) and the performance loss that comes from having to move large amount of data from host to device and back. More so, these issues typically show non-linear trends and depend on many other factors. Often, what makes the difference in this trade-off is the actual amount of computations involved in between data transfers: when a higher computational load is needed in each thread, and the number of parallel threads is high (i.e., large DFT size), the resulting outstanding level of throughput may hide the latency issues caused by memory transfers and make the GPU implementation easily outperform the CPU implementation (if this is not already the case).
In the spectral processing framework, this reasoning brings two noteworthy implications: first, computationally intensive algorithms are more likely to bring higher *speed-up factors* when implemented with *CUDA*, provided they are suited for parallel processing; second, chaining more parallel-implemented spectral processing algorithms (when the desired application needs them) should decrease the negative impact of memory transfers on GPU performance, since these would only come into play at the very beginning (host to device) and at the very end of the chain (device to host). Both implications will be apparent in the benchmarking stage of this work. While the former implication is straightforward, the latter actually needs the processing modules to be "tuned" with the analysis/re-synthesis modules in such a way that data is kept in device memory after the analysis stage (a feature that was not provided for in [8]) as well as after each processing stage, and eventually the re-synthesis module needs to read data from device memory (again, this was not provided for in [8]).

Of course, the other important limitation of GP-GPU computing in this scope is that not all processing algorithms may be perfectly congenial to the parallel scheme. There are actually many levels and shades of suitability for parallel processing: some algorithms may be more suitable than others but often there is not an absolute line of separation between the two cases. Moreover, if an algorithm is not well suited to parallel implementation in a straightforward way, it does not necessarily mean that there are no possible work-arounds in order to exploit the GPU: these might include theoretical solutions to re-cast the original problem to a parallel scheme (see for instance section 2.8, *About Recursive Filters in a Parallel Computing Scenario*, for a few solutions to a classical problem in audio GPU processing) as well as operative solutions involving the use of *CUDA* special features and optimizations[27] (above all, *atomic operations*[28]). In the development of this work, all the faced algorithms were actually well suited for parallel execution, with very few exceptions.

3.4 Selected *Unit Generators*

In order to assess the consistency of developing parallel computing versions of pre-existing spectral processing *Csound unit generators* running on off-the-shelf GPUs, a set of nine algorithms was chosen as a test array. These were selected from different categories of frequency-domain manipulation algorithms: `pvsgain`,

[27]See also section 4.5.
[28]See section 3.6.4

`pvsfilter` and `pvstencil` from the amplitude transformation category, `pvscale` and `pvshift` from the frequency transformation category, `pvsmooth` and `pvsblur` from the amplitude-frequency modification category and `pvsmix` and `pvsmorph` from the cross-synthesis category.

3.4.1 pvsgain

Syntax: `fsig   pvsgain   fsigin, kgain`

This is probably the most basic spectral processing algorithm: it simply scales the amplitude of the input phase vocoder stream according to a control-rate input argument ("`kgain`"). Its effect is equivalent to that of multiplying a time-domain signal by a scalar (thus affecting the signal's perceived volume) but this is done in the frequency domain instead. It is particularly useful when chained with other spectral manipulations.

Given its simplicity, this unit generator was mainly chosen to experiment with the general approach for GPU implementation in the first stages of this project. It also serves as a reference for the lightest spectral manipulation algorithm that can be thought of, among those with a straightforward and unquestionable musical meaning.

3.4.2 pvsfilter

Syntax: `fsig   pvsfilter   fsigin, fsigfil, kdepth [, igain]`

This *unit generator* receives two *fsig* variables: the first ("`fsigin`") is the input phase vocoder stream to be processed while the second ("`fsigfil`") is a filtering phase vocoder stream. This module operates frequency-domain (possibly time-varying) filtering: it multiplies the amplitudes of the two streams, channel by channel, with the possibility of controlling the extent of this operation[29] via the "`kdepth`" control-rate parameter. Finally, the resulting amplitudes are scaled further (all by the same amount) according to an initialization-time parameter, "`igain`".
Note that the frequency values of the input stream are left unchanged, and those belonging to the filtering stream are completely ignored.

3.4.3 pvstencil

Syntax: `fsig   pvstencil   fsigin, kgain, klevel, iftable`

This *unit generator* transforms a phase vocoder stream according to a masking function table ("`iftable`"): for each channel, if the amplitude falls below the value of the corresponding element of the function table, a certain control-rate gain ("`kgain`") is applied to that channel. Prior to this operation, the values in the

[29]The original input signal is actually mixed with the filtered signal in a dry/wet fashion, `kdepth` being the tuning parameter for the final mix. `kdepth = 1` results in an output which solely consists of the processed signal while `kdepth = 0` returns the unprocessed signal (acting as a *bypass* switch).

masking table can be scaled by another control-rate parameter (“`klevel`”) in order to increase or decrease the depth of the effect.
Applications of `pvstencil` include noise reduction, filtering and inverse-masking (see *The Canonical Csound Reference Manual* [62] for more details).

3.4.4 `pvscale`

Syntax: `fsig   pvscale   fsigin, kscal [, kkeepform, kgain, kcoefs]`

This *unit generator* scales the frequency components of a phase vocoder stream, resulting in pitch scaling. It multiplies each channel’s frequency value by the desired control-rate scaling parameter “`kscale`”, that can be either between 0 and 1 (downward scaling) or higher than 1 (upward scaling). As a result, pitches in the input stream are moved to new spectral positions, still maintaining all harmonic relations. In addition, amplitude components can be optionally modified in order to attempt formant preservation, a feature which might be particularly appealing, for example, in speech signal processing. By means of an optional parameter (“`kkeepform`”) a spectral envelope estimation process is triggered on the input stream. This is achievable via two possible methods: a classic cepstrum low-pass liftering mechanism [63] (with a step-shaped mask whose cut-off quefrency is optionally specified by the user with “`kcoefs`”[30]) or an iterative envelope detection mechanism, also known as *true envelope estimator* ([64] and [65]).
Finally, consistent amplitude scaling on all channels can also be performed optionally (as if `pvsgain` was added in the processing chain) thanks to the optional parameter “`kgain`” (which is set to 1 by default).

3.4.5 `pvshift`

Syntax: `fsig   pvshift   fsigin, kshift, klowest [, kkeepform, igain , kcoefs]`

This *unit generator* shifts the frequency components of a phase vocoder stream, resulting in pitch shifting. From a certain channel upwards (specified by the user via the “`klowest`” parameter), it offsets frequency values by the desired control-rate shifting parameter “`kshift`”, that can be either positive (upward shifting) or negative (downward shifting) and is expressed in hertz. As a result, pitches in the input stream are moved to new spectral positions and harmonic relations are altered. In addition, amplitude components can be optionally modified in order to attempt formant preservation in the exact same way as it is carried out in `pvscale` (see above). Finally, as in `pvscale`, consistent amplitude scaling on all channels can also be performed optionally.

3.4.6 `pvsmooth`

Syntax: `fsig   pvsmooth   fsigin, kacf, kfcf`

[30]This parameter actually corresponds to the number of cepstrum coefficients that are meant to be kept, while all the higher coefficients are brought down to zero in order to annihilate the high quefrency components of the cepstrum.

This *unit generator* smooths the spectral flux of an input signal by low-pass filtering the time evolution functions of the amplitude and frequency components of the input phase vocoder stream. On each channel, a first order low-pass IIR filter with time-varying cut-off frequency is applied to the time functions of both amplitude and frequency components. The cut-off frequency parameters ("`kacf`" for the amplitude time functions and "`kfcf`" for the frequency time functions) run at control-rate and are expressed as fractions of half the frame rate (which is actually the phase vocoder stream sampling rate). This means that the highest cut-off frequency is 1 and the lowest 0; the lower the frequency the smoother the resulting spectral flux and more pronounced the effect will be. The effects produced are more or less similar to `pvsblur` (see below), but with two important differences: first, smoothing of amplitudes and frequencies use separate sets of filters; second, there is no increase in computational cost when a higher amount of 'blurring' (smoothing) is desired (and no latency is introduced).

3.4.7 pvsblur

Syntax: `fsig   pvsblur   fsigin, kblurtime, imaxdel`

As `pvsmooth`, this *unit generator* smooths the spectral flux of an input signal by low-pass filtering the time evolution functions of the amplitude and frequency components of the input phase vocoder stream. Instead of using IIR filters, though, filtering is accomplished by a moving time-average (a kind of FIR filter) of successive phase vocoder frames. Both amplitude and frequency time functions are processed the exact same way, using a single tuning parameter "`kblurtime`" which sets the length of the averaging window[31] in seconds: longer periods will result in a more pronounced effect.
The "`imaxdel`" parameter is used to set the maximum expected averaging window to be used in the specific application (this is needed in order to reserve enough memory for processed data).

3.4.8 pvsmix

Syntax: `fsig   pvsmix   fsigin1, fsigin2`

This *unit generator* mixes two input phase vocoder streams in a seamless way: for each channel in the output stream, the corresponding amplitudes in the two input streams are compared and the strongest one is selected while the weakest is discarded. Also, the frequency information paired to the strongest amplitude is used in the output channel. This operation is essentially different from that of mixing two signals in the time domain but the perceived effect is actually similar.

3.4.9 pvsmorph

Syntax: `fsig   pvsmorph   fsig1, fsig2, kampint, kfrqint`

This *unit generator* performs interpolation (or "morphing") between two input phase vocoder streams. The amplitude and frequency components of the two sources

[31]As a side-effect the output stream will be delayed by the same amount.

are linearly interpolated. Optionally, emphasis can be given to one input stream or the other depending on the control-rate values of "`kampint`" and "`kfrqint`", for the amplitude and frequency components respectively.

3.5 Development of GPU-operating *Plugin Opcodes*

The nine algorithms described above have been transposed to the *CUDA* model. In this framework, input phase vocoder data is either transferred to the GPU, together with other parameters, or read directly from device memory, depending on the considered version of the *plugin opcode* (see below). Computations are then carried out on the GPU itself in a parallel fashion by the *CUDA cores* and the output is either transferred back to host memory or stored in device memory for future use. The original algorithms are written in *C++* and the GPU versions have been written in *CUDA C*, placing the computational core of these processes inside *CUDA kernels*. As the nature of these algorithms implies, GPU processing is invoked every hop size samples.

3.5.1 *Host Memory Input-Output* Version and *Device-Only* Version

All the *plugin opcodes* that have been developed in the scope of this work were written in two different versions.
The first version was developed with a certain level of abstraction from the bigger picture and intended as a typical GP-GPU problem, in some way isolated from the framework it was supposed to be placed in. In this version, the whole GP-GPU processing cycle is covered: host-to-device memory transfers are followed by parallel computation, which in turn is followed by device-to-host memory transfers, once computation is completed. This version of the algorithm was used both as a starting point for the author in order to practice the basics of *CUDA* development, and as an initial prototype for testing the correctness of the resulting code (comparing the resulting signals to those generated by the original CPU code). The *plugin opcodes* created in this way were given the name of the original *unit generators*, preceded by the prefix "`cuda`" (for instance, the GPU version of `pvsgain` was named `cudapvsgain`, and so on).
This version of the new *CUDA plugin opcodes* can be used in a *Csound instrument* in conjunction with standard `pvsanal`/ `pvsynth` pairs as well as with the *CUDA* versions of these (`cudanal` and `cudasynth`, developed by Victor Lazzarini et al. in [8]). The key concept here is that these new *plugin opcodes* are meant to find in host memory the data to be processed and they will manage memory transfers themselves through the `cudaMemcpy()` API function. They will eventually transfer the output phase vocoder frame back to host memory after the conclusion of all computations for the specific frame under analysis, again via `cudaMemcpy()`. As it was already pointed out, this might not be the ideal strategy for achieving the best results from the GPU in terms of performance. As already discussed in section

3.3.1, the GPU's computing power might get overshadowed by memory transfer latency, even in the case of an algorithm which is congenial to parallel processing and even in the case of a minimum possible number of displacements (i.e. two: one host-to-device and one device-to-host). If, computationally-wise, a specific algorithm is not particularly demanding at thread level, the potential improvement given by the GPU might be undermined by memory transfer latency. This is most probably the reason why some of the tested *plugin opcodes* belonging to this version actually perform slightly worse than the CPU code[32], even though they should be good candidates for parallel processing.

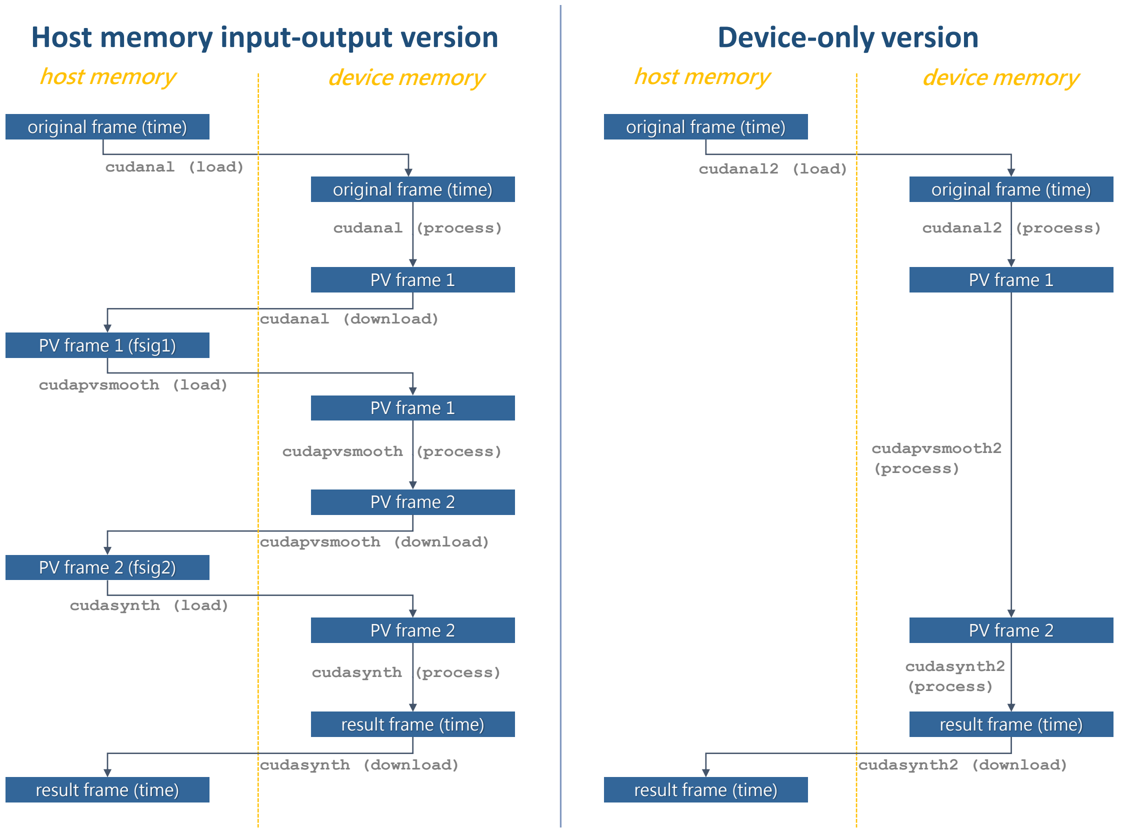

Figure 3.1: Data flow diagrams of a practical utilisation example, comparing the two versions that were developed for each algorithm. The *device-only* version involves much less memory transfers and is expected to perform better in any case.

In order to enhance the speed-up potentials of using the GPU, a *device-only* version of each algorithm was developed as well. This version is designed to minimise host/device memory transfers in the whole processing chain, considering the analysis and re-synthesis stages as well as the actual manipulation stage. Consequently, the new *plugin opcodes* were designed to find phase vocoder frames inside device memory already, and they only need to care about the actual computation of the output

[32]Obviously, this happens especially when these *plugin opcodes* are used in conjunction with the original CPU *unit generators* for analysis and re-synthesis, since they are always slower than `cudanal` and `cudasynth`. See chapter 4 for a more detailed comparison between the performances of all implementations.

phase vocoder frame, which is also kept in device memory (and not transferred back to host memory). In order for this to be possible, the code for the analysis and re-synthesis stages needed to be revised as well, and this was taken care of by Victor Lazzarini, who modified the `cudanal` and `cudasynth` *opcodes* accordingly, introducing `cudanal2` and `cudasynth2`. All the processing *plugin opcodes* were hence modified, eliminating all memory transfers, and taking care that memory pointers are correctly handled[33]. The resulting *plugin opcodes* were named using the additional suffix ‘2’, hence `cudapvsgain2`, `cudapvsfilter2` and so forth. In this new framework, the number of memory transfers is drastically reduced: now only two memory transfers are needed, regardless of the length of the processing chain, whereas the previous version of the *CUDA plugin opcodes* involves a number of transfers which is equal to twice the number of processing modules plus 4 (2 transfers for the analysis stage and 2 for the re-synthesis stage).

A graphical comparison of the two approaches is shown in figure 3.1. Here, a practical application is depicted showing data transfers and processes involved in the two versions. This example consists of phase vocoder analysis, spectral processing (using `cudapvsmooth` and `cudapvsmooth2` modules in this case[34]) and re-synthesis. Each module is conceptually divided in three stages: a loading memory transfer (host to device), a processing stage and, finally, a downloading memory transfer (device to host). Of course, in the *device-only* versions, the spectral processing modules lack the memory transfer stages and execute in one single conceptual block.

3.5.2 A General Scheme for *CUDA Plugin Opcodes*

The general scheme used for the development[35] of *CUDA plugin opcodes* is as follows[36]. In the initialisation function, information about the device is collected via the `cudaGetDeviceProperties()` function. This information (namely, number *Streaming Multiprocessors* and maximum number of threads per block) is then used to define[37] the block size and the grid size that have to be used for launching computation kernels, depending on the particular GPU architecture under analysis. In the same initialisation function, a device memory allocation stage is performed via `cudaMalloc()` for input-output data and possibly for other useful data. Also, output data is usually set to zero right after allocation (via `cudaMemset()`).

In the performance function, a check[38] for the presence of new input data is

[33]In this new framework, in-place processing for phase vocoder frames should be avoided because what is read on device memory is the original copy of the analysis data, and that might be needed by other *unit generators*.

[34]Note that `pvsmooth` also needs the previous output frame (in addition to the current input frame) to compute the current output frame. In order to keep the picture readable and clean, this dependency is not shown in figure 3.1. In any case, regardless of the specific version, this data is always read from device memory.

[35]For a deeper insight about the rules for the development of *Csound plugin opcodes* in general, see [57].

[36]The following scheme applies to the first version of *plugin opcodes* (the *host memory input-output version*) which is more general. The *device-only* scheme can be easily derived from that.

[37]See the final codes in appendix B for more details.

[38] `if (p->lastframe < p->inputPVstream->framecount) { ... }`, where `p` is the data structure that contains all the data related to a particular instance of the *unit generator* and

made (as usual with *fsig*-based *opcodes*). Only when new input data is detected, the following actions are scheduled: input data is transferred to device memory, the parallel computation kernel(s) is/are called, and, finally the results are transferred back to host memory. Using `pvsgain` and `cudapvsgain` as a model, the computational core of the original code, i.e.:

```
if (p->lastframe < p->fa->framecount) {
  for (i = 0; i < framelength; i += 2) {
    fout[i] = fa[i]*gain;
    fout[i+1] = fa[i+1];
  }
  p->fout->framecount = p->fa->framecount;
  p->lastframe = p->fout->framecount;
}
```

is translated into the following lines:

```
if (p->lastframe < p->fa->framecount) {
  // Memory transfer (host to device):
  cudaMemcpy(p->deviceFrame, p->fa>frame.auxp, size,
             cudaMemcpyHostToDevice);
  // Kernel launch:
  applygain<<<p->gridSize, p->blockSize>>>(p->deviceFrame, gain,
                                          framelength);
  // Memory transfer (device to host):
  cudaMemcpy(p->fout->frame.auxp, p->deviceFrame, size,
             cudaMemcpyDeviceToHost);

  p->fout->framecount = p->fa->framecount;
  p->lastframe = p->fout->framecount;
}
```

Here `p` is the data structure that holds all the information related to a particular instance of the *plugin opcode* under analysis (including input and output phase vocoder streams), `fa` indicates the input phase vocoder frame while `fout` indicates the output frame (they are both pointers[39] to floats in host memory). The pointer `p->deviceFrame` points in turn to device memory and is employed as a container for both input and output data[40]. The integer `framelength` specifies the number of elements in a phase vocoder frame, i.e. $N + 2$, where N is the DFT size.
Of course, more complex algorithms may involve more kernel calls and possibly other operations between a kernel call and the next, but this general scheme can be applied to all the nine examined modules. Note that in all *device-only* versions, memory-transfers-related lines need to be erased from the scheme.
In the *CUDA C* implementation, the actual computations to be carried out are defined in the kernel body:

```
__global__ void
applygain (float* deviceFrame, MYFLT gain, int length) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int j = i<<1;
  if (j < length)
    deviceFrame[j] *= gain;
}
```

`inputPVstream` is the input phase vocoder stream (a `PVSDAT` structure).

[39]They are actually shortcuts for “`p->fa>frame.auxp`” and “`p->fout>frame.auxp`”.

[40]Note that in the *device-only* version, two separate pointers will be needed.

CUDA threads are mapped to the index `i` (and `j`, consequently) by means of the *CUDA*-specific syntax[41] used for defining variable `i` itself:

```
int i = threadIdx.x + blockDim.x * blockIdx.x;
```

Index `j` is then used to address the amplitude components of the phase vocoder stream which are multiplied by the gain factor (frequency components are ignored in this specific module).
These operations are performed sequentially in the original CPU codes, by means of a loop (see above). A typical approach in porting sequential algorithms to the GP-GPU scheme is that of re-writing loops in terms of device kernels that spawn a series of parallel threads on the device, in a SIMD fashion.

It is important to stress that the general scheme that has been presented in the previous paragraphs represents a very basic implementation of a *CUDA* application. A lot of optimisation techniques that could be possibly added to this scheme have been intentionally ignored for the sake of simplicity. Some of these optimisations are reported in section 4.5: they could be added to the basic scheme in order to increase GPU performance.

3.5.3 *Sliding* Mode

All phase vocoder *unit generators* in *Csound* feature the possibility of using a *sliding DFT* (SDFT) framework under the surface ([51]), the resulting scheme being callled *sliding phase vocoder* (SPV). The idea behind this technique is that of making the analysis window slide just one sample ahead at each analysis step, thus ultimately using a hop size of just one sample. Instead of actually calculating the DFT at each step, though, in order to update the discrete spectrum this method makes use of the known values of the current DFT frame in addition to the new time sample. The creation of each new DFT frame is effectively an operation of $O(N)$ complexity (all the mathematical insight can be found in [66]).
There are many potential advantages brought by the use of this technique, especially regarding the audible clarity of frequency domain audio effects (i.e, less artefacts) and the possibility for enhanced musical uses, such as mimicking classic FM modulation in the frequency domain, and whole new families of sound transformation, such as *Transformational FM* [51] (to name a few). Not least, a theoretically much lower limit on latency[42] is brought by the SDFT framework but this is only achievable with relatively powerful hardware: the SDFT, and thus the sliding phase vocoder, is a very expensive technique in terms of computational load. Nevertheless, an important feature of the SDFT is that it is totally parallel and, hence, an excellent candidate for SIMD-style parallelism ([50] and [51]). The possibility of exploiting a commodity GPU (using *CUDA*) for sliding phase vocoder analysis and re-synthesis was in fact successfully investigated by Bradford et al. in [49]: up to 8 channels of audio could be analysed and re-synthesised in real time. The sliding scheme was then included in the *CUDA*-based streaming phase vocoder analysis and re-synthesis

[41] See [13] for more details.

[42] The SPV reduces the latency imposed by standard phase vocoder by some 75%, with clear and valuable advantages for real-time performance, provided top-notch hardware is available.

Csound unit generators developed by Lazzarini et al. in [8] but this framework is still under development and it is not completely supported yet.

In the development of this work, a choice was made not to include sliding support for the *CUDA* versions of the spectral processing *plugin opcodes*. This choice is mainly ascribable to the fact that, being sliding operations on a sample-by-sample basis, the question of memory transfers and buffering is more complex and a framework for sliding phase vocoder processing is therefore beyond the scope of this work[43]. As of today, SPV mode is not supported by any of the developed *CUDA*-operating *opcodes*. The development of sliding support for the new *plugin opcodes* is of course one of the main directions for future work.

3.6 *CUDA*-based *Plugin Opcodes*

In this section, the *CUDA* implementation of each algorithm will be analysed and a few considerations will be made in order to discuss each algorithm's congeniality to the GP-GPU scheme.

3.6.1 `cudapvsgain` and `cudapvsgain2`

Effect: scale the amplitude of a phase vocoder stream.

The original `pvsgain` internally multiplies amplitude data from each channel of the input *fsig* by the scaling parameter. This operation is carried out through a **`for`** loop over the phase vocoder channels in the original code but it is actually perfectly suited for the massively parallel computation scheme of GP-GPU processing: the single operations performed inside the original loop are completely independent from one another and the actual order of completion is not relevant at all. Additionally, when a high DFT size is chosen, the opportunity of executing many multiplications simultaneously is very appealing.
This is the main kernel in the *host memory input-output version*:

```
__global__ void
applygain (float* deviceFrame, MYFLT gain, int length) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int j = i<<1;
  if (j < length)
    deviceFrame[j] *= gain;
}
```

It uses in-place computations and thus leaves the frequency components unchanged. The index `i` is used to scan the phase vocoder channels: multiplying by two this index (`j = i<<1`) amplitude values can be accessed.
In-place computations cannot be used in the *device-only* version:

```
__global__ void
applygain(float* output, float* input, MYFLT gain, int length) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int j = i<<1;
```

[43] In addition, including a study of the sliding case in this project would have taken much more time, which was not really available.

```
  if (j < length) {
    output[j] = (float) input[j] * gain;
    output[j+1] = input[j+1];
  }
}
```

Here, the frequency component, which is accessed via `j+1`, is simply copied from input memory locations to output ones.
Both implementations are based on global memory. Memory accesses are coalesced, implying that this algorithm, as many of the others, is congenial to the way memory bursts are managed on the device and it is likely to perform well also on older NVIDIA GPUs, even if they do not provide global memory caching.

The only limitation that is to be expected for a performance gain over CPU implementation is ascribable to memory transfers between host and device, as described in section 3.3.1. Even though `pvsgain` can be smoothly translated into a parallel algorithm, the computational load in each thread is extremely slight (just one multiplication) meaning that a performance gain will be possible only from a certain DFT size upwards, i.e. in those cases where the latency caused by memory transfers is indeed hidden by high computational throughput. This downside should be less apparent in the *device-only* version, `cudapvsgain2`.

3.6.2 `cudapvsfilter` and `cudapvsfilter2`

Effect: filtering in the frequency domain by means of a multiplicative spectral mask.

The original `pvsfilter` internally multiplies amplitude data from each channel of the input *fsig* by the corresponding amplitude value of the filtering *fsig*. The resulting values are weighted by the "wetness" parameter and summed with the properly weighted dry signal. An overall gain is finally applied. Again, in the original code these operations are carried out through a **`for`** loop over the phase vocoder channels and, again, they can be made parallel in a very straightforward way:

```
__global__ void
filter(float* input, float* output, float* mask, MYFLT wet,
       MYFLT dry, float g, int length) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  if (j < length) {
    output[j] = (float) (input[j]*(dry+mask[j]*wet))*g;
    output[j+1] = input[j+1];
  }
}
```

, where `mask` is an array containing the spectral mask used for filtering. Frequency components are just copied from input to output. Only global memory is used and memory accesses are coalesced.
All the concepts that were faced while explaining `cudapvsgain` and `cudapvsfilter` will apply to other *opcodes* as well: this is in fact the basic and most common scheme for GP-GPU spectral processing modules where the very same operation needs to be performed on every single channel.

The per-thread computational complexity is slightly higher than the one displayed in `cudapvsgain` but it is still quite low and, again, a substantial speed-up is expected only for DFT sizes larger than a certain threshold.

3.6.3 cudapvstencil and cudapvstencil2

Effect: selectively apply gain to certain channels according to a spectral mask loaded from a function table.

In the initialisation function, a preliminary step is needed in order to clip to zero potential negative values in the stencil mask, as these would not make sense in an amplitude-frequency phase vocoder format. In the *CUDA* versions, this is accomplished very easily by means of the *Thrust*[44] [67] library:

```
cudaMemcpy(p->devStencil, p->func->ftable, stencilSize,
           cudaMemcpyHostToDevice);
thrust::device_ptr<MYFLT> dev_ptr =
  thrust::device_pointer_cast(p->devStencil);
_is_less_than_zero pred;
thrust::replace_if(thrust::device, dev_ptr, dev_ptr + chans, pred,
                   (MYFLT) 0.0);
```

, where the clipping operation is carried out by the `thrust::replace_if()` method[45] with a predicate defined through the following structure:

```
struct _is_less_than_zero {
  __host__ __device__
  bool operator()(float x) {return x < 0.0f;}
};
```

This operation could have been performed at host side as well, probably with little difference from a performance point of view, but, since the memory transfer of stencil data from the function table to device memory is needed anyway, regardless of the particular *CUDA* version that is being developed, it does make sense to exploit the *Thrust* library this way, instead of clipping phase vocoder data to zero in a sequential way at host side. For that matter, this operation needs to be performed only once, so the argument is not really critical.
The actual computations are performed on the GPU by means of launching the following kernel from the performance function:

```
__global__ void
stencil(float* output, float* input, MYFLT* stencil, float level,
        float gain, int length) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int j = i<<1;
  if (i < length) {
    if (input[j] < (((float) stencil[i])*level)) {
      output[j] = input[j] * gain;
      output[j+1] = input[j+1];
    }
```

[44] *Thrust* is a parallel algorithms library designed to enhance programming productivity while enabling performance portability between GPUs and multicore CPUs. *Thrust* interoperates with *CUDA* and it is actually included in the *CUDA Toolkit*.

[45] See the *Thrust* documentation for more details

```
    else {
      output[j] = input[j];
      output[j+1] = input[j+1];
    }
  }
}
```

, where `stencil` is of course the array containing the spectral mask used for selecting the bins to modify.
From a *CUDA* perspective, this kernel shows a suboptimal implementation of the task to be completed. In fact, it bears the presence of a conditional which, in the worst case scenario, might take a different path in each thread and this is not ideal in a *CUDA* framework. Nevertheless, this behaviour is highly dependent on the nature of the input and stencil frames and, even though there is room for improvement, the above implementation turns out to give good results in most cases. Again, the usual reasoning about low computational complexity inside each thread also applies to `cudapvstencil` and `cudapvstencil2`.

3.6.4 `cudapvscale` and `cudapvscale2`

Effect: pitch scaling with optional formant conservation.

Given the option to choose between two methods for spectral envelope preservation or even not to keep formants at all, this module can be actually seen as three modules in one. Only *mode 0* and *mode 1* will be addressed here (the implementation of *mode 2* can still be found in appendix A[46]).

Mode 0: Basic Pitch Scaling

Even in its simplest form, the GPU version of this algorithm is one of the most convoluted amongst those examined in the development of this work. As a matter of fact, the computations need to be split in a few steps and the device needs to be synchronized in between steps:

```
if (keepform == 0) {
  // Initialise the output PV frame:
  thrust::device_ptr<float> dev_ptr1 =
    thrust::device_pointer_cast(fout);
  thrust::fill(dev_ptr1, dev_ptr1+framelength, -1.0f);

  // Pitch scaling stage:
  freqScaleBasic<<<p->gridSize,p->blockSize>>>(fin, fout, pscal,
                                                Nhalf);

  // Apply gain to all amplitudes:
  fixPVandGain<<<p->gridSize,p->blockSize>>>(fin, fout, g,
                                              framelength);
}
```

As a first step, for reasons that will be clear in a few lines, each output frame element needs to be initialised to a certain reference value (-1 in this case). For convenience,

[46]The GP-GPU versions of *mode 2* could not be tested due to a consistency problem with respect to the original version (see appendix A), this is why it is not included in this chapter and it has been moved to an appendix.

the *Thrust* library is used for this purpose, by means of the `thrust:fill()` method. Then, the actual frequency scaling takes place in the `freqScaleBasic()` kernel and `fixPVandGain` takes care of a few final operations and adds a gain stage. The frequency scaling operation can be described by the following expression:

$$f_{out[k\alpha]} = \alpha f_{in[k]} \tag{3.1}$$

, where f_{in} and f_{out} are the input and output frequency components of the corresponding phase vocoder frames, k is the channel (bin) index and α is the scaling factor.

An important consideration needs to be done: this operation is not symmetric, meaning that scaling upwards or downwards have different implications from a *CUDA* implementation perspective. As a matter of fact, when the scaling is carried out in the upward direction, each thread spawned by the GPU will write to a different output channel, i.e. to a different memory location. This is due to the fact that the frequency range is being stretched and the total number of channels used is not changed. This behaviour is congenial to parallel computation: since there is a one to one relation between input and output bins, each thread takes care of a single channel in the output spectrum. Unfortunately, this is not the case for scaling downwards. In this case, in fact, the frequency range is compressed (rather than stretched) and again the number of bins used is kept constant. This means that more than one input channel will likely fall in the same output bin (the deeper the scaling the more channels will compete to fit in the same output bin). Since the phase vocoder is based on a sinusoidal model that requires only one sinusoid per channel, only one of the potential candidates[47] has to be transferred to the output (with modified frequency, of course, as in equation 3.1). In the original code, each output channel is simply overwritten a few times. However, this is a delicate operation in a parallel computation scenario: it is risky to let more than one thread write a multi-byte result to the same memory location simultaneously as it might be possible to end up with hybrid values made up of bytes from different threads. In order to prevent this, *CUDA* provides *atomic operations*: these are special functions that are designed to reserve the memory locations they are working on, so that no other external action can be done on these, and to carry out their job in one single scheduling step (*read/modify/write*). Conflicting *atomic operations* that try to access the same memory locations are serialised, thus degrading performance.

To sum up, frequency scaling needs *atomic operations* when performed in a *CUDA*-based framework (specifically, `atomicExch()` was used in this work). The `freqScaleBasic` kernel is defined as:

```
__global__ void
freqScaleBasic(float* input, float* output, MYFLT scaleFactor,
               int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = (i<<1) + 2;
  int N = nhalf<<1;
  int newchan;
```

[47]The choice of which channel is the best candidate for being transferred to the output is beyond the scope of this work and was not addressed in the original `pvscale`. After all, choosing an arbitrary candidate has little impact on the perceptual result.

```
  if (i < nhalf-1) {
    newchan = (int)(((i+1)*scaleFactor)+0.5) << 1;
    if (newchan < N && newchan > 0) {
      atomicExch(&output[newchan],input[j]);
      atomicExch(&output[newchan+1],
                 (float)(input[j+1]*scaleFactor));
    }
  }
}
```

, where `nhalf` is half the DFT size ($N/2$).
This kernel is a direct implementation of equation 3.1, taking into account the discrete nature of the problem in the digital domain (note the casting to **`int`** when the `newchan` variable is initialised in order to define the target channel). Output channels are only considered in the range between 0Hz and the Nyquist frequency. Regardless of the direction of scaling, many output bins are not affected by this operation (amplitude and frequency components are thus left at the original value of -1, which will be used as an identification flag).
As a final step in the *mode 0* script, the last kernel, `fixPVandGain()` takes care of setting to zero the amplitude components of each undefined bin (those not affected by the previous kernel) and to apply the desired gain to all the valid bins:

```
__global__ void
fixPVandGain(float* input, float* output, float gain, int length) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  if (j < length) {
    if (isnan(output[j])) {
      output[j] = 0.0f;
    }
    if (output[j+1] == -1.0f) {
      output[j] = 0.0f;
    }
    else
      output[j] *= gain;
    }
  // Keep original DC amplitude:
  if (j == 0) output[0] = input[0];
  // Keep original Nyquist amplitude:
  if (j == length-2) output[length-2] = input[length-2];
}
```

A separate kernel is needed for synchronization reasons: all threads need to get synchronized after the *atomic operations* (even across warps) and this is automatically accomplished in *CUDA* when two kernels are called successively in the same *CUDA stream*[48].

Mode 1: Formant Conservation via Cepstrum Liftering

When *mode 1* is invoked by the user, the amplitude components of the output frame need to be multiplied by the current spectral envelope of the input signal. This multiplication can be achieved through a slight modification of the `freqScaleBasic ()` kernel:

[48]All the *CUDA* implementations presented here are based on a single *CUDA stream*. See section 4.5 for more details about *CUDA streams*.

```
__global__ void
freqScaleFormant(float* input, float* output, float* env,
                 MYFLT scaleFactor, float maxAmp, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = (i<<1) + 2;
  int N = nhalf<<1;
  int newchan;
  if (i < nhalf-1) {
    env[i+1] /= maxAmp; // normalise the spectral envelope
    if (env[i+1] && j < scaleFactor*N) {
      input[j] /= env[i+1]; // 'equalise' the original amplitudes so
                            // that formant shaping is more effective
    }
    newchan = (int)(((i+1)*scaleFactor)+0.5) << 1;
    if (newchan < N && newchan > 0) {
      atomicExch(&output[newchan], input[j]*env[newchan>>1]);
      atomicExch(&output[newchan+1],
                 (float)(input[j+1]*scaleFactor));
    }
  }
}
```

As opposed to `freqScaleBasic()`, this kernel receives an extra array which contains the shape of the input spectral envelope (`env`) and a scalar that keeps track of the maximum amplitude contained in such an envelope (`maxAmp`). The envelope is first normalized and then used to "equalise" the input frame in order to make the formant shaping more effective, which is carried out right after that.
These additional operations are very well suited for parallel operation.

What really differentiates the two formant shaping modes is the way the spectral envelope is defined and hence detected.
The simple *cepstrum liftering* technique employed in *mode 0*, as implemented in the original `pvscale` code, consists of performing the following steps:

1. Take the logarithm of the input amplitude spectrum.
2. Take the DFT of the resulting log-spectrum.
3. Lifter the resulting cepstrum with a low-pass step-shaped mask that keeps only low quefrency coefficients and brings all higher coefficients to zero.
4. Take the inverse DFT of the liftered cepstrum.
5. Exponentiate the resulting log-spectrum in order to go back to linear scale amplitude, obtaining the desired spectral envelope.

The number of cepstrum coefficients to be kept can be specified by the user and is 80 by default.
In a *CUDA* framework, these steps can be carried out in a very straightforward way as they lend themselves well to the parallel processing scheme: all these operations are free from inter-data dependencies and, most importantly, *CUDA* provides a highly efficient parallel FFT implementation in the cuFFT library [21].
The steps enumerated above are performed in the following lines of code:

```
else if (keepform==1) {

  if (coefs<1) coefs = 80;

  // Initialise the output PV frame:
  thrust::device_ptr<float> dev_ptr1 =
    thrust::device_pointer_cast(fout);
  thrust::fill(dev_ptr1, dev_ptr1+framelength, -1.0f);

  // Take the log:
  takeLog<<<p->gridSize,p->blockSize>>>(fin, p->deviceEnv, Nhalf);

  cufftEnv = (cufftComplex*) p->deviceEnv;
  cufftCepstrum = (cufftComplex*) p->deviceCepstrum;

  // Take the fft of the log of the spectral envelope:
  if(cufftExecR2C(p->forwardPlan,(cufftReal*)cufftEnv,cufftCepstrum)!=
     CUFFT_SUCCESS)
    csound->Message(csound, "CUDA FFT error\n");
  if (cudaDeviceSynchronize() != cudaSuccess)
    csound->Message(csound,"CUDA error: Failed to synchronize\n");

  // Liftering stage:
  lifter<<<p->gridSize,p->blockSize>>>(p->deviceCepstrum, coefs,
                                       Nhalf);

  // Take the inverse fft of the liftered cepstrum:
  if(cufftExecC2R(p->inversePlan,cufftCepstrum,(cufftReal*)cufftEnv)!=
     CUFFT_SUCCESS)
    csound->Message(csound, "CUDA FFT error\n");
  if (cudaDeviceSynchronize() != cudaSuccess)
    csound->Message(csound,"CUDA error: Failed to synchronize\n");

  // Exponentiate:
  expon<<<p->gridSize,p->blockSize>>>(p->deviceEnv, Nhalf);

  // Find maximum amp in spectral envelope:
  thrust::device_ptr<float> dev_ptr2 =
    thrust::device_pointer_cast(p->deviceEnv);
  max = *(thrust::max_element(dev_ptr2, dev_ptr2+Nhalf));

  // Pitch scaling stage:
  freqScaleFormant<<<p->gridSize,p->blockSize>>>(fin, fout,
                                                 p->deviceEnv,
                                                 pscal, max, Nhalf);
  // Apply gain to all amplitudes:
  fixPVandGain<<<p->gridSize,p->blockSize>>>(fin, fout, g,
                                             framelength);
}
```

, where `Nhalf` is the number of elements in the amplitude spectral envelope, i.e. $N/2$ (since it is redundant, frequency information is absent in the amplitude spectral envelope). To find the maximum amplitude value in the spectral envelope the *Thrust* function `thrust::max_element()` is used. Finding the maximum of an array is a classic *reduction problem* which can be efficiently performed by parallel processors like GPUs by means of what is called *parallel reduction*. *Thrust* provides a very easy and convenient way[49] of implementing parallel reduction operations in

[49]This might not be the fastest possible solution in the *CUDA* scope but it is sufficient for the purpose of this work.

just one *CUDA C* instruction.
The kernels for taking the logarithm and exponentiating phase vocoder amplitudes are straightforward and congenial to parallel computing, and so is the kernel used for liftering:

```
__global__ void
takeLog(float* input, float* env, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  if (i < nhalf) {
    env[i] = log(input[j] > 0.0 ? input[j] : 1e-20);
  }
}

__global__ void
lifter(float* cepstrum, int nCoefs, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int k = i + nCoefs;
  if (k < nhalf+2-nCoefs) {
    cepstrum[k] = 0.0;   // kill all the cepstrum coefficients
                         // above nCoefs (but keep replicas over Nyquist)
  }
}

__global__ void
expon(float* env, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  if (i < nhalf) {
    env[i] = exp(env[i]/nhalf);
  }
}
```

After `freqScaleFormant()` (i.e., the kernel for frequency scaling with spectral envelope shaping), this algorithm employs the very same kernel presented in *mode 0* in order to complete the phase vocoder frame manipulation (`fixPVandGain()`).

3.6.5 `cudapvshift` and `cudapvshift2`

Effect: pitch shifting with optional formant conservation.

These *plugin opcodes* are basically the same as `cudapvscale` and `cudapvscale2` with only one difference: the actual frequency scaling kernels are replaced by frequency shifting ones. Beside that, the overall structure is the very same and the spectral envelope detection procedures are developed in the same exact way. Therefore, only the shifting kernels will be presented in this section.

This operation can be described by the following expression:

$$f_{out[k+\beta']} = f_{in[k]} + \beta \tag{3.2}$$

, where f_{in} and f_{out} are the input and output frequency components of the corresponding phase vocoder frames, k is the channel (bin) index, β is the frequency offset and β' is the corresponding number of bins (bin offset). The value of β' is an integer obtained by the following *CUDA C* line:

```
int betaprime = abs((int) (beta * N * (1.0/SR)));
```

, where N is the total number of DFT bins and SR is the sampling rate.
Before discussing the shifting kernels, it must be stressed that the intrinsic asymmetry that could be found in a parallel implementation of pitch scaling is no longer present in the shifting scenario. In fact, when pitch shifting is performed (either upwards or downwards) each input channel is transferred to a different output channel, with a one to one relation. As a result, no conflicts arise with threads that need to write to the same memory location and *atomic operations* can be avoided.
The following code snippet shows the kernel used for frequency shifting in the case of no formant conservation:

```
__global__ void
freqShiftBasic(float* input, float* output, MYFLT shift,
               int shiftChan, int lowestIndx, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  int lowestChan = lowestIndx>>1;
  int N = nhalf<<1;
  int newchan;
  if (i < lowestChan) {
    // leave PV data as it is below a certain channel:
    output[j] = input[j];
    output[j+1] = input[j+1];
  }
  if (i >= lowestChan && i < nhalf) {
    newchan = (i + shiftChan) << 1;
    if (newchan < N && newchan >= lowestIndx) {
      output[newchan] = input[j];
      output[newchan+1] = (float) (input[j+1] + shift);
    }
  }
}
```

, while the following snippet shows the kernel used for frequency shifting plus formant conservation:

```
__global__ void
freqShiftFormant(float* input, float* output, float* env, MYFLT shift,
                 int shiftChan, int lowestIndx, float maxAmp, int nhalf) {

  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  int lowestChan = lowestIndx>>1;
  int N = nhalf<<1;
  int newchan;
  if (i < lowestChan) {
    // leave PV data as it is below a certain channel:
    output[j] = input[j];
    output[j+1] = input[j+1];
  }
  else if (i < nhalf) {
    env[i] /= maxAmp;   // normalize the spectral envelope
    input[j] /= env[i];   // equalise the original amplitudes
                          // so that formant shaping is more effective
    newchan = (i + shiftChan) << 1;
    if (newchan < N && newchan >= lowestIndx) {
      output[newchan] = input[j]*env[newchan>>1];
      output[newchan+1] = (float)(input[j+1] + shift);
    }
  }
}
```

In both cases, the shifting operation is performed only by those threads that are related to phase vocoder channels above the specified lowest frequency to be affected (here it is expressed in terms of lowest bin index by means of the variable `lowestChan`). In both cases, the pitch shifting kernel is followed by the very same phase vocoder frame completion kernel that was exploited in `cudapvscale` and `cudapvscale2`, i.e. `fixPVandGain()`.

3.6.6 `cudapvsmooth` and `cudapvsmooth2`

Effect: spectral flux smoothing, affecting both amplitude and frequency components independently.

This is one of a few modules, among those analysed in this work, that act at inter-frame level and need to store a record of past frames in order to work. In this case, since first order parallel IIR filters are employed, only one past phase vocoder frame is needed in combination with the current one.
The two sets of parallel filters used in this module are identical in structure, differing only in the actual coefficients provided by the user:

$$\begin{aligned} y_{a[n,k]} &= x_{a[n,k]}(1+\alpha) - y_{a[n-1,k]}\alpha \quad && for\ the\ amplitude\ flux \\ y_{f[n,k]} &= x_{f[n,k]}(1+\beta) - y_{f[n-1,k]}\beta \quad && for\ the\ frequency\ flux \end{aligned} \tag{3.3}$$

, where k is the usual bin index, n is the phase vocoder frame index (in time) that marks the *fsig* processing rate, α is a coefficient derived from `kacf` and β is a coefficient derived from `kfcf` (α and β are, by design, such that the resulting filters are of the low-pass kind). The application of the two sets of parallel filters is straightforward and is performed in the *CUDA* framework by the following kernel:

```
__global__ void
smooth(float* input, float* output, double alpha, double beta,
          int length) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  if (j < length) {
    output[j] = (float) (input[j] * (1.0 + alpha) - output[j] * alpha);
    output[j+1] = (float) (input[j+1] * (1.0 + beta) -
                           output[j+1] * beta);
  }
}
```

After each step is completed, the resulting output frame is not only used as desired in the processing chain (or re-synthesised if it belongs to the ending module in the chain) but it is also kept into device memory and used in the following processing step as the past output frame, i.e. $y_{a[n-1]}$ and $y_{f[n-1]}$.

This kind of scheme is highly congenial to parallel processing as there is no inter-bin dependency and each filter works in a "blind" way on the data stream from its specific channel. The higher the number of DFT bins, the more parallel filters are spawned and kept busy simultaneously[50]. On the other hand, the usual issue with memory transfer latency not being hidden by computational throughput looms over. If examined at thread level, this kernel does not bring a high computational

[50]Of course, the actual number of truly simultaneous filters depends on the GPU's internal resources (number of *streaming multiprocessors*, mainly) and on warp scheduling criteria.

load (this is usually a virtue of low orders IIR filters) and this means that for a small DFT size, the parallel version might perform worse than the original sequential version. Still, these *plugin opcodes* are expected to give slightly better speed-up factors than `cudapvsgain` and `cudapvsgain2`, for example, because the former involve a slightly higher degree of computational load.

3.6.7 `cudapvsblur` and `cudapvsblur2`

Effect: spectral flux smoothing, affecting both amplitude and frequency components.

As already mentioned in the description of the original `pvsmooth`, the main difference between these two modules lies in the way the spectral flux is filtered: while in `pvsmooth` two sets of first order IIR filters are employed, in `pvsblur` a single set of variable order FIR filters is used, affecting both amplitude and frequency fluxes in the same way. This means that a variable (and perhaps time-varying) number of past phase vocoder frames needs to be recorded in order to compute the output frame. This is done by means of a matrix-like data structure in which phase vocoder frames are stored row-wise and new frames are inserted in successive rows. Parallel filtering is then performed along the vertical direction (which corresponds to time), in each single column (both amplitude and frequency data is processed), over a number of rows that depends on the averaging time provided by the user. The higher the averaging time, the higher the order of the filters, the more frames (rows) are used in the filter computation and the higher the level of blurring. The user also provides the longest averaging time to be considered in the application, which translates into the maximum order of the FIR filters as well as into the total number of rows of the record matrix.
In the *plugin opcodes*'s initialisation function, the record matrix is set up: all frames' amplitudes are set to zero while frequency values are set to each channel's centre frequency. This is done in the `initialise()` kernel:

```
__global__ void
initialise(float* matrix, float sr, int numFrames, int length) {
  int frame = blockIdx.y*blockDim.y + threadIdx.y;
  int chan = (blockIdx.x*blockDim.x + threadIdx.x) << 1;
  if ((frame < numFrames) && (chan < length)) {
    matrix[frame*length+chan] = 0.0f;
    matrix[frame*length+chan+1] = chan * sr / (length-2);
  }
}
```

, where `sr` is the sampling rate as set in the *Csound* environment.
This kernel exploits *CUDA*'s feature of working with two-dimensional thread blocks so that a straightforward mapping can be made between thread indexes and matrix elements. The y dimension is employed for identifying rows (by means of the `frame` index), while the x dimension is employed for selecting the desired column (through the `chan` index).
In the performance function, the new frame is placed in the next available row of the record matrix, wrapping around to the first row when the whole matrix is full. Then, the filters' computations are performed involving a limited number of rows, namely those covered by the user-specified blurring time window. The whole

process is implemented via the following *CUDA C* lines[51]:

```
if (p->lastframe < p->fin->framecount) {

  // Clip the number of frames corresponding to the blurring time
  // between 0 and the maximum blurring time:
  delayframes = delayframes >= 0 ? (delayframes < maxframes ?
                                    delayframes : maxframes - 1) : 0;

  // Insert the new frame in the record matrix:
  cudaMemcpy(p->deviceMatrix+(countr*framelength),fin,size,
             cudaMemcpyDeviceToDevice);

  if (delayframes) {
    if ((first = countr - delayframes) < 0)
      first += maxframes;  // wrap around the initial row

    // Launch parallel FIR filters on the recorded PV frames:
    blur<<<p->gridSize,p->blockSize>>>(p->deviceMatrix, fout, first,
                                       delayframes, countr,
                                       maxframes,framelength);
  }
  else {
    // Bypass blurring:
    cudaMemcpy(fout, fin, size, cudaMemcpyDeviceToDevice);
  }

  p->fout->framecount = p->lastframe = p->fin->framecount;
  countr++;  // this variable keeps track of the row index
             // of the last inserted frame
  p->count = countr < maxframes ? countr : 0;
}
```

The `blur()` kernel implements parallel FIR filtering on each phase vocoder channel. The particular FIR filter used in this module is simply a moving average on a number of frames specified (indirectly) by the user via the `kblurtime` parameter in *Csound*:

```
__global__ void
blur(float* matrix, float* output, int firstFrame, int numFrames,
      int frameCount, int max, int length){
  int chan = (blockIdx.x*blockDim.x+ threadIdx.x)<<1;
  float amp = 0.0f;
  float freq = 0.0f;
  int frame;
  if (chan < length) {
    // Iterate over the involved PV frames
    // (wrapping around the end of the matrix if needed):
    for (frame = firstFrame; frame != frameCount;
         frame = (frame + 1) % max) {
      // Accumulation:
      amp += matrix[frame*length+chan];
      freq += matrix[frame*length+chan+1];
    }
```

[51]This code snippet is taken from the *device-only* version of this module (`cudapvsblur2`). In this version, input and output phase vocoder frames are copied in and out of the record matrix by means of `cudaMemcpy()` API function using the `cudaMemcpyDeviceToDevice` specifier. The *host memory input-output* version is based on the very same scheme, but of course, using the `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost` specifiers.

```
    // Averaging:
    output[chan] = (float) (amp / numFrames);
    output[chan+1] = (float) (freq / numFrames);
  }
}
```

In each channel, amplitude and frequency data is accumulated by means of a `for` loop. This means that computational load at thread level can be high, especially when a substantial level of smoothing is desired and each loop iterates on a substantial number of frames. When this is the case, the *CUDA* versions of this module are expected to perform better than the original CPU-only version, regardless of the DFT size used for the phase vocoder frames (provided that at least a few hundreds of bins are employed).
This kind of parallel filtering is of course congenial to the parallel computing scheme: each channel is filtered independently of the others. Actually, in this scenario there is yet another possibility for an even higher level of parallelism: the accumulation loop carried out in each thread could be performed in a parallel fashion as well. This could be done by means of some kind of parallel reduction in the vertical direction of the matrix in order to obtain, at each step of the process, a single row containing the sums of the amplitude and frequency bins over the considered past frames. However, this approach would possibly entail some overhead and might not be fully justified since the averaging times typically used in musical applications imply a somewhat limited number of frames to be used in the averaging process, probably not enough to gain in performance with parallel reduction. Hence, the sequential approach in the accumulation turns out to be a reasonable solution but it has to be said that there is room for further investigation.
In any case, being `pvsblur` one of the most demanding modules in terms of computational load (among those considered in this work) and, at the same time, very well suited for the parallel scheme, it is safe to say that the *CUDA* implementations of this algorithm are expected to perform very well.

3.6.8 cudapvsmix and cudapvsmix2

Effect: mix two input phase vocoder streams in a seamless way.

This module can be easily implemented in a parallel way by means of the following kernel:

```
__global__ void
mix(float* output, float* frameA, float* frameB, int chans) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int j = i<<1;
  if (i < chans) {
    int test = frameA[j] >= frameB[j];
    if (test) {
      output[j] = frameA[j];
      output[j+1] = frameA[j+1];
    }
    else {
      output[j] = frameB[j];
      output[j+1] = frameB[j+1];
    }
  }
}
```

For each channel, the amplitude of the two input frames are compared and the stronger of the two is sent to the output, together with the corresponding frequency information.
Even though this algorithm is conceptually well suited for parallel processing, it might turn out to be not exactly optimal in a *CUDA* scenario. This is because of the presence of conditionals that might take a different path in each thread and force each warp to make two passes on the same data. Still, this is a minor issue[52] and the main concern with this implementation is the general low level of computational load in each thread which might penalise GP-GPU computing over plain CPU execution when, at each step, the data to be processed is not much.

3.6.9 `cudapvsmorph` and `cudapvsmorph2`

Effect: linearly interpolate two phase vocoder streams.

The *CUDA* implementation for this module is straightforward and is obtained by the means of the following kernel:

```
__global__ void
morph(float* output, float* input1, float* input2, float ampCoeff,
      float freqCoeff, int length) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int j = i<<1;
  if (j  < length) {
    output[j] = input1[j]*(1.0-ampCoeff) + input2[j]*(ampCoeff);
    output[j+1] = input1[j+1]*(1.0-freqCoeff) + input2[j+1]*(freqCoeff);
  }
}
```

In each channel, for both amplitude and frequency components, the input values are linearly interpolated in a way that allows the presence of a bias in favour of one of the two input frames. Each thread is completely independent of the others.

3.7 Conlcusions

Csound and *CUDA* are easily combined to build an efficient framework for audio processing on the GPU. This match is especially favoured by the fact that both tools are based on *C/C++* and the translation of already existing parts of the *Csound* source code into a parallel paradigm is straightforward.

Spectral signal processing is a field of application that lends itself well to the GP-GPU computing scheme: not only its nature is well suited for a parallel computing scenario but it is also a critical framework in terms of computational load, making any kind of performance improvement particularly meaningful.
The likely potentials of this approach were proved by Lazzarini et al. in [8] for phase vocoder analysis and re-synthesis. In this chapter, the same concept is expanded further in order to include in the picture a few basic spectral processing algorithms. These are implemented in the form of nine *CUDA-C*-based *plugin opcodes* for the *Csound* environment. All the implementation details are given, both on a practical

[52]After all, the operations to be carried out in both branches of this conditional are of minimal computational load.

side (the actual code) and on a more theoretical side (concerning performance issues).
The analysis of performance concerns related to memory transfers leads to a slight modification of the original analysis and re-synthesis *CUDA*-based modules of [8]: the output phase vocoder data from the analysis module is kept in device memory after computation and the input phase vocoder data for the re-synthesis module is taken from device memory. In this way, the total number of memory transfers is extremely reduced. This in turn leads to the need of implementing two versions for each algorithm, a *host memory input-output* version and a *device-only* version, in order to assess the expected performance gain of the latter with respect to the former.

Chapter 4

Tests and Experimental Results

This chapter presents the benchmarking stage of the project: the *CUDA*-based *Csound* modules described in 3.6 are tested and their performance is compared to the original versions. The systems and the methodologies used for the tests are also described.

4.1 Testing Environment

The testing phase consists of running a *Csound* script for each developed *plugin opcode* using different settings, and eventually recording the corresponding execution times.

4.1.1 Testing Systems: Hardware

Tests were run on two sample systems. The results that will be shown in the next sections were obtained by running the tests only on one of these two systems[1] (which, from now on, will be referred to as "system 1"), but, in a few occasions, a comparison between the two systems will be provided as well.
As the ultimate objective of this thesis is that of assessing the efficiency of employing the GP-GPU scheme on average computer systems, the testing environment was chosen in such a way that it could be representative for an ordinary desktop computer. From a practical point of view, the aim is that of studying how much gain can be achieved in audio processing applications when a somewhat recent, middle range PCI-express video card is added to a not so recent system. Of course, this work can be also seen from a wider perspective and it can show how, in general, parallel computing can help improving the execution speed of spectral audio processing applications.

Both systems are based on the very same CPU, a 2.93GHz Intel Core 2 Duo (released in 2009), and have very similar characteristics overall, with the exception of the graphics card: the main system is equipped with a Gigabyte GeForce GTX750Ti (Maxwell microarchitecture), while on the second system an ASUS GeForce GT730 (Kepler microarchitecture) was installed (both GPU chips on these cards are from

[1]System 1 was chosen as the main target for the tests because it is the one equipped with the most recent GPU.

NVIDIA). All the details about the two system are reported in table 4.1. Additional specifications for the two GPUs can be found in table 4.2.

Table 4.1: System specifications for the two computers used in the testing stage.

	System 1 (main system)	**System 2 (additional system)**
CPU	Intel Core 2 Duo E7500 @ 2.93 GHz x 2	*idem*
RAM	2 x 2GB DDR3-1333 SDRAM @ 1066 MHz	*idem*
GPU	NVIDIA GeForce GTX750Ti	NVIDIA GeForce GT730

Table 4.2: Specifications of the target GPUs.

	GeForce GTX750Ti	**GeForce GT730**
Number of cores	640	384
Core clock	1085 MHz	902 MHz
Memory clock	5.4 Gbps	5 Gbps
VRAM	2048 MB GDDR5	2048 MB GDDR5
Memory bandwidth	86.4 GB/s 128-bit wide bus	40 GB/s 64-bit wide bus
Processing power (single precision)	1306 GFLOPS	693 GFLOPS
Processing power (double precision)	40.8 GFLOPS	28.9 GFLOPS
Thermal Design Power	60 W	25 W

The video cards were chosen from the middle range slice of the market; in addition, small format and low power models needed to be chosen in order to fit in the available computers. This choice is actually consistent with the setting of this thesis, which focuses on performance improvement of modest computer systems. From a practical point of view, a substantial difference between the two systems is to be found in the *CUDA compute capability* of the corresponding graphics card, implying that different features of the *CUDA* GP-GPU scheme can be exploited, depending on the way the *CUDA C* code is compiled (typically, these features are applied under the surface, in a transparent way). On system 1, which has a GPU featuring *compute capability 5.0*, the `-arch=sm_50` option was used inside the *NVCC* compiler command for building the shared objects. On the other hand, since the GeForce GT730 only has *compute capability 3.5*, the `-arch=sm_30` option was used on system 2 instead.

4.1.2 Testing Systems: Software

Both systems are based on Linux Ubuntu[2] operating system: system 1 runs Ubuntu 15.04 (64-bit), while system 2 runs Ubuntu 14.10 (64-bit).
On both systems, *Csound 6.07* was installed and used for running the tests.
Finally, *CUDA 7.5* was installed on system 1, while *CUDA 7.0* was installed on system 2.

4.2 Testing Procedure

For each of the developed *plugin opcodes*[3], a very basic *Csound* script was written. Each of these scripts was conceptualised as a stand-alone program that is meant to carry out only one signal processing task in the frequency domain.

Each script consists of four *instruments*[4]: each *instrument* implements the very same task, only using different tools. Each time that a script is launched, it allows for only one of the four *instruments* to be instantiated, so that the registered execution time is only related to the particular combination of tools that make up the specific *instrument*. When launching a *Csound* testing script, the desired *instrument* is specified by the user, together with other parameters, by means of additional command line options. Running different *instruments* results in different modules to be employed for the completion of the desired task:

- ***Instrument 1*****:** Computations are performed on the CPU only, in a single-threaded and sequential way. This *instrument* employs `pvsanal` for the phase vocoder analysis stage and `pvsynth` for the re-synthesis stage. In between these two, the original spectral processing *unit generators* are employed in order to carry out the desired manipulations.

- ***Instrument 2*****:** The spectral manipulations are performed on the GPU by means of the new *plugin opcodes*. Here the *host memory input-output* version is employed (see 3.5.1). Analysis and re-synthesis stages are still carried out on the CPU, again via `pvsanal` and `pvsynth`. This is a mixed approach that is designed to assess the efficiency of adding only the new *plugin opcodes* to the programming environment provided by *Csound*, as opposed to the possibility of employing the GPU also for the analysis and re-synthesis stages.

- ***Instrument 3*****:** Both spectral manipulations and analysis/re-synthesis operations are performed on the GPU. Again, the *host memory input-output* version of the new *plugin opcodes* (see 3.5.1) is employed, while, for phase vocoder analysis and re-synthesis, `cudanal` and `cudasynth` are employed, as presented in [8]. This approach involves more memory transfers between host and device than are actually needed, potentially degrading performance.

[2] *Canonical Ltd. Ubuntu*: `http://www.ubuntu.com/`

[3] Actually, in order to test the different operating modes of `cudapvscale` and `cudapvshift`, a separate script for each operating mode was also set up.

[4] In *Csound*, "instruments" are basic code blocks which are comprised of ordinary statements to set values, control the logical flow, or invoke the various signal processing subroutines that lead to audio output.

- ***Instrument 4*:** All signal processing operations are cast to the GPU, as in *instrument 3*. However, `cudanal2` and `cudasynth2` are employed for phase vocoder analysis and re-synthesis and the *device-only* versions of the new *plugin opcodes* (see 3.5.1) is employed for the actual manipulations. With this approach, data is kept in device memory after the analysis stage as well as after the processing stage, minimising expensive data transfers. This is expected to be the best combination of the available modules and it is considered, among the four, the wisest implementation for GPU-operating spectral processing applications.

See the code snippet in the next subsection (4.2.1) for an explicit example of this testing scheme.
The execution times that result from running each *instrument* for a fixed amount of time on the same audio material can be then compared in order to understand which combination of *unit generators* and *plugin opcodes* performs best, and which one results in the worst performance. The same *instrument* is actually run many times, using different parameters, so that numerous combinations of DFT sizes and hop sizes are considered for the analysis/re-synthesis stages. These will be crucial in determining the amount of data that can be processed in a parallel way on the GPU, not only in the *CUDA*-based versions of the analysis/re-synthesis stages, but also in the new *CUDA*-based spectral processing modules.

Note that all the modules considered in this work rely on external, user specified parameters to some extent (for instance, the *gain* parameter in `pvsgain`). The *Csound* scripts used in this benchmarking stage are designed to mimic realistic case studies of musical nature. The specific values used for the involved parameters were chosen in a reasonable range and varied in time, in order to explore potentially different behaviours. In many cases, the actual values used for the input parameters are not supposed to influence the execution time of the single modules, but this is not always the case: as it was shown in section 3.6, these values can have a substantial impact on the way the operations inside of *kernels* are executed on the GPU. The way these parameters are varied is designed to give the most general impression of the performance gain, if any, that can be achieved by exploiting the graphics processor in a *heterogeneous computing* fashion.
As a reference, the following subsection reports the *Csound* script for testing the *CUDA* versions of `pvsgain` against the original CPU version.

4.2.1 Testing Scheme

All the *Csound* scripts that were employed for testing purposes are reported in appendix C. What follows is one of them, reported here in order to give an explicit sample. This is the script for testing `pvsgain` and the related GP-GPU versions:

```
<CsoundSynthesizer>

<CsOptions>
--opcode-lib=libcudapvsgain.so
--opcode-lib=../../libcudapvs2.dylib
--opcode-lib=libcudapvsgain2.so
</CsOptions>
```

```
<CsInstruments>

sr = 44100
ksmps = 128
0dbfs = 1

gifftsize = $FFT
gihopsize = $HOP

instr 1
  kenv   linseg   0, p3/4, 1, p3/4, 0, p3/4, 1.5, p3/4, 0
  asig   soundin   "syrinx.wav"
  fsig   pvsanal   asig, gifftsize, gihopsize, gifftsize, 1
  fsigScaled   pvsgain   fsig, kenv
  asig   pvsynth   fsigScaled
  out   asig
endin

instr 2
  kenv   linseg   0, p3/4, 1, p3/4, 0, p3/4, 1.5, p3/4, 0
  asig   soundin   "syrinx.wav"
  fsig   pvsanal   asig, gifftsize, gihopsize, gifftsize, 1
  fsigScaled   cudapvsgain   fsig, kenv
  asig   pvsynth   fsigScaled
  out   asig
endin

instr 3
  kenv   linseg   0, p3/4, 1, p3/4, 0, p3/4, 1.5, p3/4, 0
  asig   soundin   "syrinx.wav"
  fsig   cudanal   asig, gifftsize, gihopsize, gifftsize, 1
  fsigScaled   cudapvsgain   fsig, kenv
  asig   cudasynth   fsigScaled
  out   asig
endin

instr 4
  kenv   linseg   0, p3/4, 1, p3/4, 0, p3/4, 1.5, p3/4, 0
  asig   soundin   "syrinx.wav"
  fsig   cudanal2   asig, gifftsize, gihopsize, gifftsize, 1
  fsigScaled   cudapvsgain2   fsig, kenv
  asig   cudasynth2   fsigScaled
  out   asig
endin

</CsInstruments>

<CsScore>
i $INSTR 0 60
</CsScore>

</CsoundSynthesizer>
```

Each time this script is run, it uses a triplet of parameters, passed by the user to *Csound* by means of macros: in the *score* section (between `<CsScore>` and `</CsScore>`), `$INSTR` is used to select the *instrument*, i.e. the desired combination of modules, while in the *orchestra* section (between `<CsInstruments>` and `</CsInstruments>`), `$FFT` and `$HOP` are used to select the DFT size and hop size, respectively.

From a higher level of abstraction, all *instruments* perform the very same task: in a real-time fashion, they set a time-varying gain level (between 0 and 1.5) via `linseg`, they read a sound file called "syrinx.wav" from disk via `soundin`, they perform phase vocoder analysis via either `pvsanal`, `cudanal` or `cudanal2`, they perform the desired spectral processing task via either `pvsgain`, `cudapvsgain` or `cudapvsgain2`, they perform the re-synthesis operation via either `pvsynth`, `cudasynth` or `cudasynth2`, and, finally, they send the resulting time domain audio frames to the output buffer via `out`. These operations are carried out for 60 seconds.
In the *options* section (between `<CsOptions>` and `</CsOptions>`), the shared objects related to the custom *plugin opcodes* are loaded.

Once the script has completed execution, *Csound* returns the elapsed time at the end of performance, which has to be under 60 seconds for clean real-time execution (without *dropout* samples). This value is recorded and used as the key parameter for assessing the speed of execution of the analysed application. In fact, speed-up factors are computed via the ratio of pairs of recorded execution times (related to different pairs of *instruments*). For instance, given a specific processing algorithm under analysis and given the execution time results from testing *instrument 1* (t_1) and *instrument 2* (t_2) with a specific DFT size and hop size, the speed-up factor[5] $\sigma_{2,1}$ related to this pair is simply computed as $\sigma_{2,1} = \frac{t_1}{t_2}$. Actually, in order smooth over the effect of the inevitable performance jitter caused by interrupts in a complex operating system, each test[6] is run five times and the resulting execution times are averaged to give the t_i values used above. Timings are taken from the total computation time recorded by *Csound*, which lumps the serial and parallel code, but since the interest here is the feasibility of the system as whole, this is exactly what these tests are designed to measure. Note that in this scenario, the GPU is employed for two discrete tasks, simultaneously: on one side it is called for executing audio signal processing computations, on the other side it still needs to drive the video graphics subsystem. This is actually platform dependent: the computers employed for this project did not allow the dedicated GPU to be released from video processing but this is not a general rule.
The very same scheme is applied to all the other modules, with the obvious changes that are needed in each single case (for instance, some modules need to work on two audio streams).

4.2.2 Audio Specifications

Csound is set to work with an internal sampling rate of 44.1kHz, which is kept consistent in each *instrument*. The audio files used as sound sources are 16-bit mono files of the wav format, saved at a sample rate of 44.1kHz. This is arguably one of the most widespread audio formats in the scope of computer music and it was chosen to maximise the generality of the tests. Of course samples are internaly converted to the floating point format by *Csound* for processing purposes.

For what concerns the phase vocoder framework specifications, a comprehensive set of DFT sizes and hop sizes is examined for these tests. DFT size is varied

[5]This value expresses how much *instrument 2* is better than *instrument 1*.

[6]Here, "one test" refers to one execution of the testing script with a given triplets of parameters (*instrument*, hop size and DFT size).

considering the powers of two between 1024 (2^{10}) and 16384 (2^{14}) while hop size is varied between 128 (2^7) and 2048 (2^{11}), again considering only powers of two. Not all possible pairs of these two parameters are tested, only the most meaningful ones[7] (see table 4.3).

Table 4.3: DFT size and hop size pairs used for the tests.

hop size \ DFT size	**1024**	**2048**	**4096**	**8192**	**16384**
128	✓	✓	✓	✓	✓
256	✓	✓	✓	✓	✓
512	-	✓	✓	✓	✓
1024	-	-	✓	✓	✓
2048	-	-	-	✓	✓

DFT size & hop size

Both DFT size and hop size have a critical impact on performance. The hop size parameter determines the rate at which the phase vocoder processing steps[8] are performed: the lower the hop size, the higher the processing rate, the more demanding the application in terms of computational power (in fact, a higher number of processing steps is needed in a fixed time window). More so, in terms of GP-GPU computing, setting a low hop size results in a higher total number of memory transfers, even for the *device-only* versions of the analysed modules (there must be at least two memory displacements per processing step: one before the analysis stage and one after the re-synthesis stage). At the end of the day, however, if a specific module performs better on the GPU in a single processing step for a given DFT size (meaning that the computational power of the graphics processor succeeds in hiding the latency from memory transfers), decreasing the hop size will necessarily magnify the performance gain measured over 60 seconds. Thus, decreasing the hop size, in general, is expected to magnify the unbalance between the two approaches.
The hop size is somehow related to the concept of time resolution of the phase vocoder scheme: using a lower hop size gives to the user the opportunity of updating processing parameters of a given module at a higher rate, thus increasing the time granularity of any action in the frequency domain. This brings down the choice of this parameter to a trade-off between time granularity and computational burden: the user needs to understand what is the upper threshold for the hop size that guarantees a good level of time granularity for a given application. Decreasing the hop size below this threshold would result in an unjustified increment of the

[7]The main criterion employed here for choosing the allowed pairs of settings was that of having hop sizes that are not bigger than one quarter of the DFT size.

[8]Here the term *processing step* refers to the whole set of operations involved by the chain of an analysis stage, a processing stage and a re-synthesis stage. In one *processing step*, these operations are executed on a number of samples which is equal to the DFT size.

computational burden. Yet, this whole issue about the time granularity of frequency-domain manipulations is only loosely related to the concept of time resolution which is brought about by the very essence of the Fourier transform, together with the ultimate trade-off between time and frequency resolution (see also section 1.2.3). The time granularity for processing parameters and the time resolution of the Fourier transform are thus two different notions that need to be kept conceptually separated.

The DFT size determines the amount of data which is involved in each processing step. A high DFT size implies a high computational burden both in the computation of the DFTs (forward and inverse ones) and in the processing stages. However, a GP-GPU approach to this picture opens up a whole new perspective: the very nature of the GP-GPU computing scheme is that of making, as much as possible, a given processing task independent of the amount of data involved. In fact, the massively parallel computing scheme is precisely based on an abundance of hardware resources (i.e., cores) with the aim of exploiting all of them simultaneously[9], while setting them to work on different data. Thus, increasing the DFT size in a GP-GPU framework usually results in a better usage of the available resources, having little impact on the overall execution time, which is expected to increase of just a small fraction[10]. Most importantly, increasing the DFT size in a GP-GPU model is very likely to result in a much lower increase in the overall execution time with respect to a CPU-based sequential model, hence unbalancing the scale in favour of the GP-GPU model. As a matter of fact, in the CPU-based scheme, an increase in the amount of data to be processed has the inevitable effect of loading the same few cores with more computations: eventually, these will need to be scheduled on a larger time window.
The choice of the DFT size is highly dependant on the nature of the specific application that needs to be developed. This parameter has to be chosen on the basis of the particular specifications regarding the time vs. frequency resolution trade-off discussed in section 1.2.3.

4.3 Results

In this section, the results obtained from each tested algorithm will be presented and commented. At first, this analysis will focus only on the results from the main system (*system 1*). In the last part of this section a comparison between the two systems will be carried out, focusing only on a few algorithms.

The subsection about the *gain* modules will be used as an introductory model for the data analysis: this subsection is wider than the others simply because it is

[9]This scheme is of course applicable only to those processes that can be made parallel (in other words, expressed in a SIMD fashion). Yet, the phase vocoder framework does not raise too many issues, as it has been shown in chapter 3. The FFT algorithm is also perfectly suited for a parallel computing scheme and a high performance FFT library is available in the *CUDA* toolkit, namely *cuFFT* ([21]).

[10]More so, another actor in this scenario is the recurring argument about high computational complexity helping to hide memory transfers' latency, at least when the amount of data involved in the computations is high. See section 3.3.1 for more details.

used to introduce and explain concepts that actually apply to the other modules as well.

4.3.1 *Gain* module: a Preliminary Analysis

Figure 4.1 shows five graphs where the average execution times of the four *gain instruments* are plotted against the DFT size. Each graph is related to a different hop size setting.

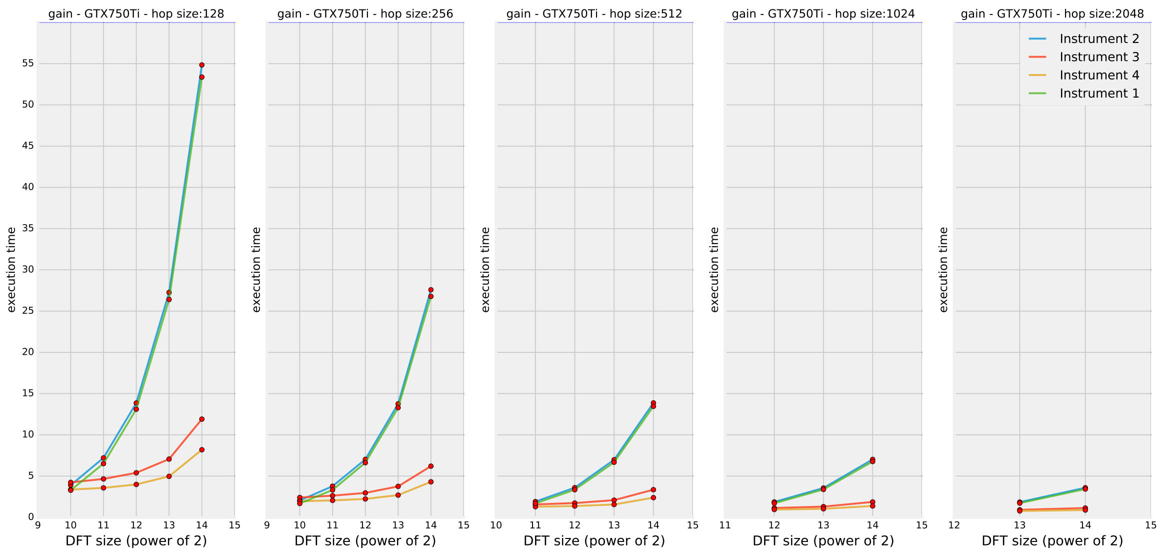

Figure 4.1: Execution time results (in seconds) from testing the *gain* module on system 1 over 60 seconds of audio.

A few comments can be made about these plots. First of all, it is interesting to note that, for highly demanding settings (i.e., low hop size and high DFT size), the basic CPU-based version (green line) gets critically close to the real-time threshold, even for such a simple application. This proves the fact that phase vocoder operations can get very demanding in terms of computational power; after all, this program is only applying some gain to a signal in the frequency domain: this is just a basic block (yet a very useful one) for more articulated processing chains. As soon as this block is included in a more complex project, real-time operation is very unlikely to be guaranteed. It has to be stressed that these highly demanding settings are often essential for high quality sound processing, especially in those situations where frequency resolution is more important than time resolution and a considerable time granularity is needed for the processing actions; thus, if it was possible to decrease the execution times in some way (e.g., via GP-GPU computing), a better quality for real-time spectral processing would be achievable on a given machine. The five plots of figure 4.1 show that, for the *gain* algorithm, substituting `pvsgain` with `cudapvsgain` does not change much the overall execution time. Actually, it slightly degrades the performance in all cases (blue line). This can be explained by means of the *low computational complexity argument* that was described in section 3.3.1: the *gain* module, in fact, shows a very low level of time complexity if analysed from a parallel computing perspective.

Fortunately, this not-so-promising scenario drastically changes in favour of GP-GPU computing as soon as *instrument 3* is examined (red line): the introduction of GPU-based phase vocoder analysis and re-synthesis stages (via `cudanal` and `cudasynth`) cuts down execution times remarkably for almost all settings. The actual performance gain depends on both the DFT size and the hop size settings, and these relations will be analysed more thoroughly in a few paragraphs, while commenting figure 4.2 and 4.3. The general performance gain (which is more than four times in some cases) is mainly ascribable to the use of cuFFT ([21]) in the analysis and re-synthesis modules, which, apparently, benefits the most when it is applied to chunks of data that are wider than (or equal to) 2048 samples. For a DFT size of 1024 samples, it is difficult to infer the nature of the swing in the performance (it depends on the hop size as well), but the change is not much in any case.
Finally, by inspecting figure 4.1, it is clear that the implementation of a *device-only* version (*instrument 4*) for the *gain* module helps improving the performance with respect to the implementation of *instrument 3*, as anticipated in section 3.5.1. The improvement is manifested for each pair of settings and the resulting execution times are the lowest that have been recorded in almost all cases[11](yellow line), arguably making *instrument 4* the overall best performing solution.

All in all, it is evident that a GP-GPU computing implementation brings a general performance improvement to the *gain* module, especially for high DFT sizes and low hop sizes. This is a very promising result for GPU-based phase vocoder processing in general, especially when considering the low computational complexity (per thread) that characterises this algorithm (which is expected to result in a bias towards the plain CPU approach). To some extent, these conclusions could be predicted on the basis of the previous work by Lazzarini et al. on this topic ([8]): what really makes the difference, in fact, seems to be the introduction of *CUDA*-based analysis and re-synthesis stages in the phase vocoder framework. However, these results also show that the further introduction of *CUDA*-based processing stages seems to be not only viable but also desirable for an additional performance improvement.

In order to better visualise the degree of improvement achieved by the GP-GPU implementations of the *gain* module, the speed-up factors are plotted in figure 4.2 and 4.3. Putting aside for a moment the critical case of 1024-points DFTs and focusing on *instrument 4* (which results to be the best implementation in all other cases), the speed-up factors that were recorded in these tests span the range between $\sigma_{4,1} = 1.62$ and $\sigma_{4,1} = 6.51$, depending on the employed settings. If the attention is focused on an intermediate DFT size of 4096 points, the recorded speed-up factors vary between $\sigma_{4,1} = 1.80$ and $\sigma_{4,1} = 3.30$, depending on the employed hop size.
What makes these results particularly relevant is that the speed-up factors get to their highest values exactly for those settings that are more critical for the plain CPU implementation. In a way, this has the effect of making the execution time of a well-tuned GP-GPU implementation less dependant on DFT size (this is clearer in picture 4.1).

Two final considerations can be made about figure 4.2 and 4.3. First, as it has

[11]Again, the only exceptions are the two cases where the DFT size is set to 1024 samples.

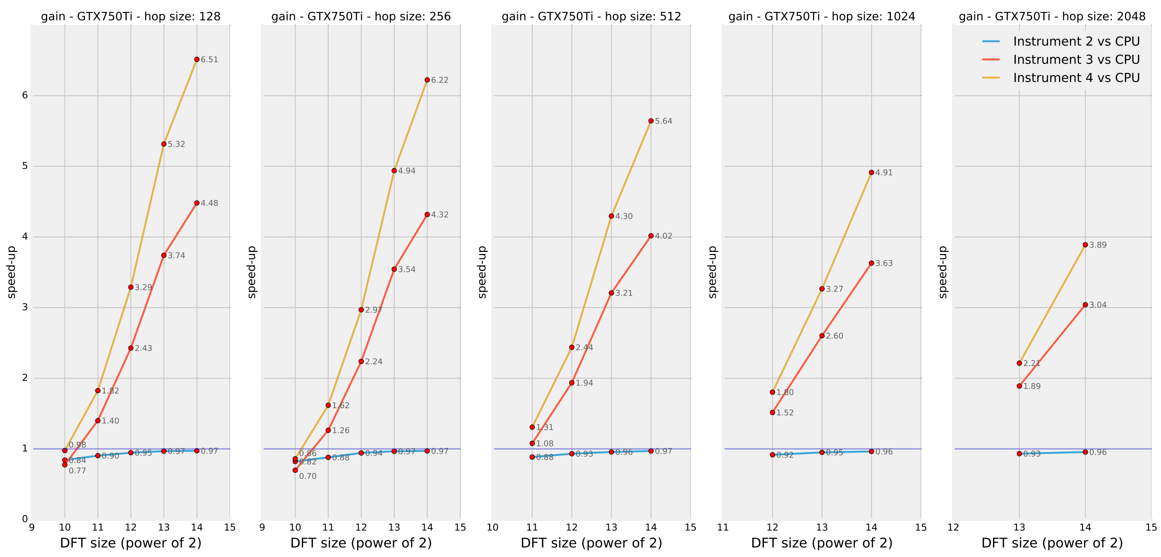

Figure 4.2: Speed-up factors related to different implementations of the *gain* module on system 1.

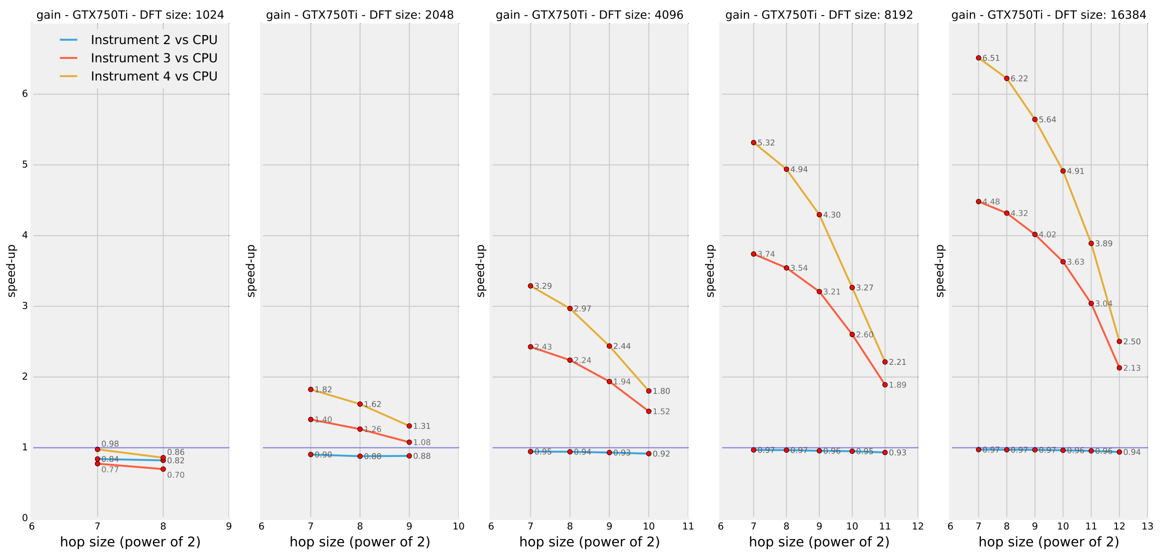

Figure 4.3: Speed-up factors related to different implementations of the *gain* module on system 1.

been pointed out already, the most naive GP-GPU implementation of *instrument 2* (blue line) fails to give any improvement over the original CPU code, but, at the same time, it does not degrade the performance too much (the lower it gets is $\sigma_{2,1} = 0.8$). It is important to stress that, even if these were the results of the overall best performing GP-GPU implementation, it could still make sense to consider a heterogeneous computing solution for the phase vocoder *gain* module. In fact, while it is true that delegating this processing task to the GPU slightly slows down the computations, at the same time the CPU load is diminished, hence opening up room for utilisation by other concurring processes. Whether this can be useful or not depends on the level of utilisation of the computer system (as well as that of the GPU) at the time of processing, but this is an attractive option to be considered in general. More so, under this perspective, the speed-up results achieved by *instrument 4* appear to be even more valuable: not only this implementation brings down execution times, it also frees up CPU resources.

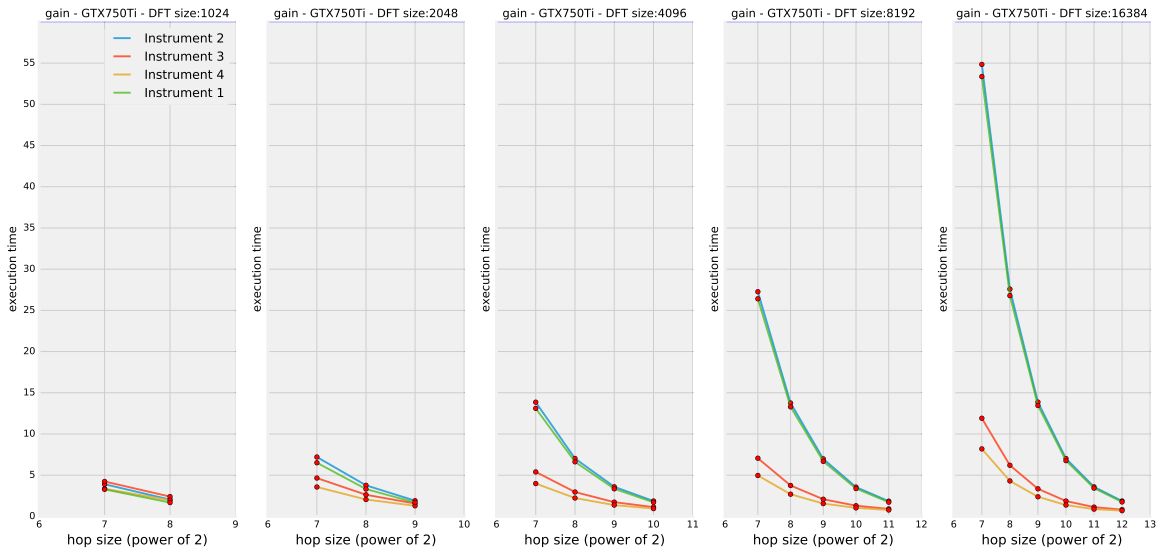

Figure 4.4: Execution time results (in seconds) from testing the *gain* module on system 1 over 60 seconds of audio.

Finally, figure 4.3 clearly shows that all implementations fail to give any improvement for a DFT size of 1024 points. The most reasonable explanation for this behaviour is that the size of the data to pe processed at each step is not enough to justify a parallel computing scheme, even for quite compute-intensive tasks like the FFTs included in the analysis and re-synthesis stages. The overheads introduced by a *CUDA*-based scheme (memory transfers, kernel scheduling, etc.) are not hidden by massively parallel processing: basing the computation on 1024 elements seems to be too little for vindicating the use of the term “massive”. Fortunately, this weakness of *CUDA*-based 1024-points spectral processing is not really problematic since this is also the case that exhibits the lower execution times in general, as shown in figure 4.4, and it is therefore the least needy for a performance improvement.

4.3.2 *Filter* module

Figure 4.5 shows the average execution times of the four *filter instruments* plotted against the DFT size. Each graph is related to a different hop size setting. Not surprisingly, the trends are very similar to those found in figure 4.1. Being the filtering algorithm slightly more demanding in terms of computational complexity with respect to the gain one (after all, it involves managing two phase vocoder frames instead of one, and it involves a little more arithmetic operations), all execution times are shifted up with respect to the *gain* case. This has the effect of bringing *instrument 1* and *instrument 2* out of the real-time zone of the plots (execution times under 60 seconds), for a large DFT size of 16384 points and a small

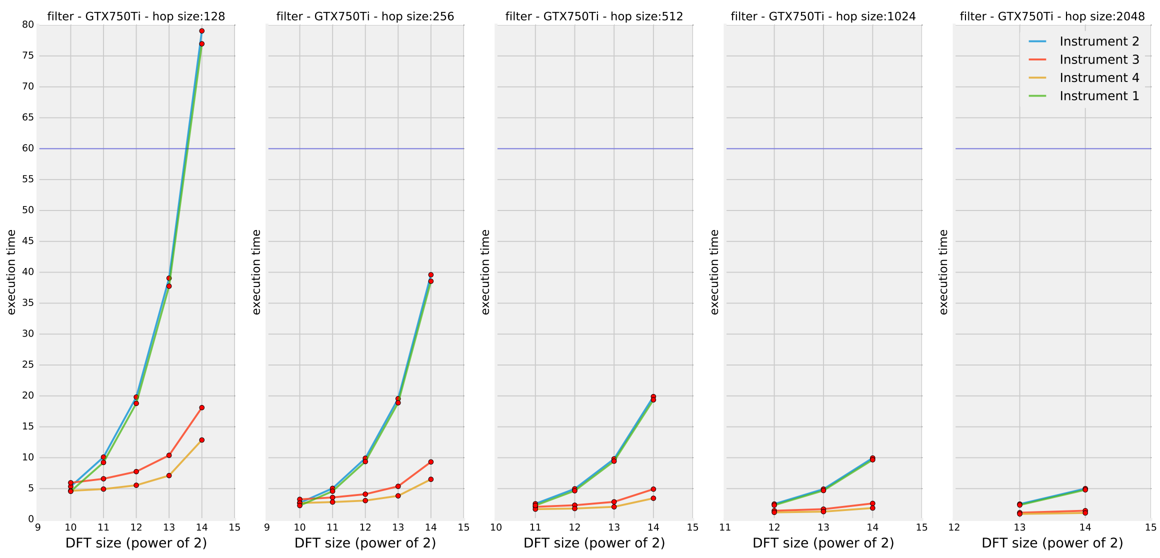

Figure 4.5: Execution time results (in seconds) from testing the *filter* module on system 1 over 60 seconds of audio.

hop size of 128 samples. The more advanced GP-GPU solutions of *instrument 3* and *instrument 4*, however, provide enough performance gain to guarantee real-time execution even in this extreme case.

The speed-up factors are plotted in figure 4.6 and 4.7. A very similar behaviour with respect to the *gain* module is recorded and similar considerations apply.

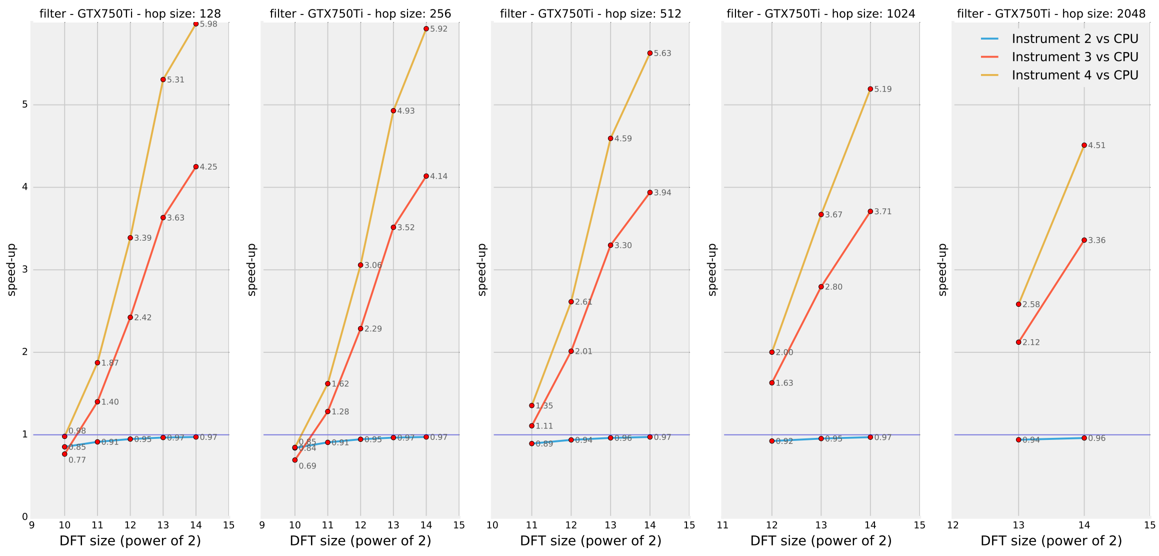

Figure 4.6: Speed-up factors related to different implementations of the *filter* module on system 1.

Instrument 4 is again the best performing overall. Putting aside the critical case of processing 1024 channels, the speed-up factors related to this implementation range from $\sigma_{4,1} = 1.35$ to $\sigma_{4,1} = 5.98$, depending on the settings employed. If the attention is focused on an intermediate DFT size of 4096 points, the recorded speed-up factors vary between $\sigma_{4,1} = 2.00$ and $\sigma_{4,1} = 3.39$, depending on the employed hop size.

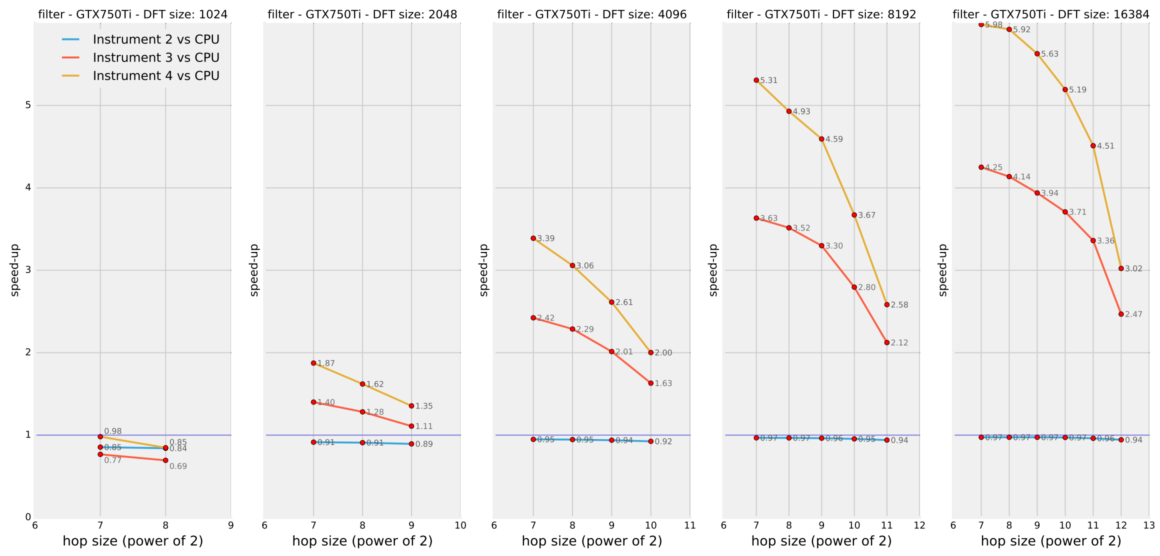

Figure 4.7: Speed-up factors related to different implementations of the *filter* module on system 1.

4.3.3 *Stencil* module

The *stencil* algorithm is characterised by a computational load that is highly dependant on the input signal and on the desired shape for the spectral mask. Thus, the design of a test that is meant to return general results is quite problematic. In order to try and achieve a decent level of generality, a random spectral mask was generated for each instance of the tests[12]. In addition, a spectrally rich and dynamic input audio was used for the tests.

The speed-up factors resulted from testing the *stencil* module are shown in figure 4.8 and 4.9. Again, the trends of the three lines seem to match what have been shown until now and even the speed-up factors are similar to those recorded for the *gain* and *filter* modules. With a DFT size of 1024 points, all GP-GPU versions perform worse than the original CPU code; yet *instrument 4* gets very close to *instrument 1* even in this unfavourable case. For the other DFT sizes, the speed-up factors achieved by *instrument 4* range from $\sigma_{4,1} = 1.31$ and $\sigma_{4,1} = 6.51$, depending on the settings employed. For an intermediate DFT size of 4096 points, the speed-up factors related to *instrument 4* vary between $\sigma_{4,1} = 1.80$ and $\sigma_{4,1} = 3.29$, depending on the hop size settings.
All the execution times recorded for this module stay under the real-time threshold of 60 seconds.

[12]See appendix C for the complete *Csound* script used for these tests. Note that each test is repeated five times and the results are averaged.

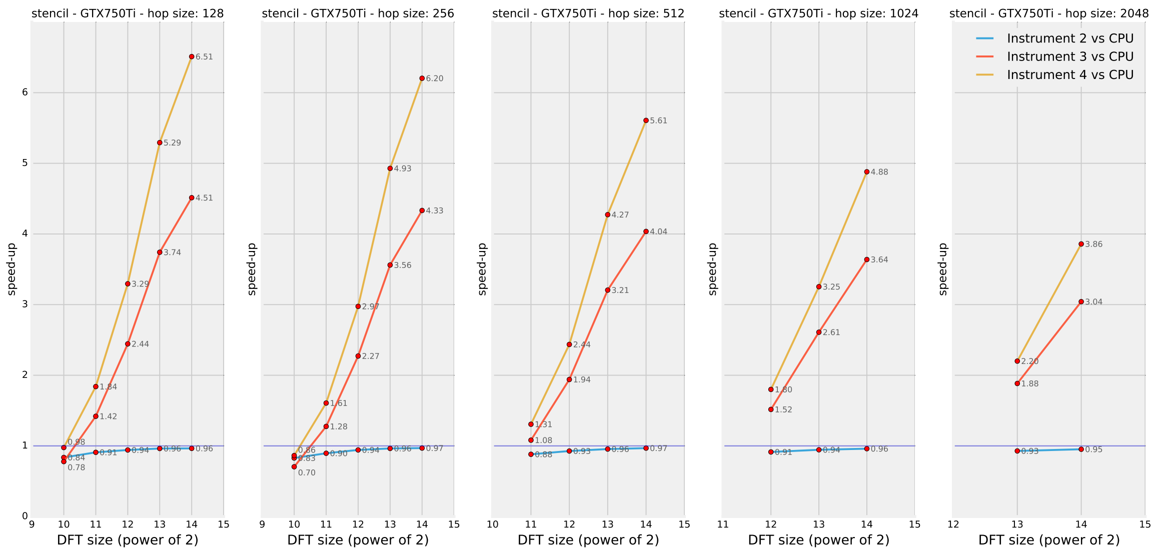

Figure 4.8: Speed-up factors related to different implementations of the *stencil* module on system 1.

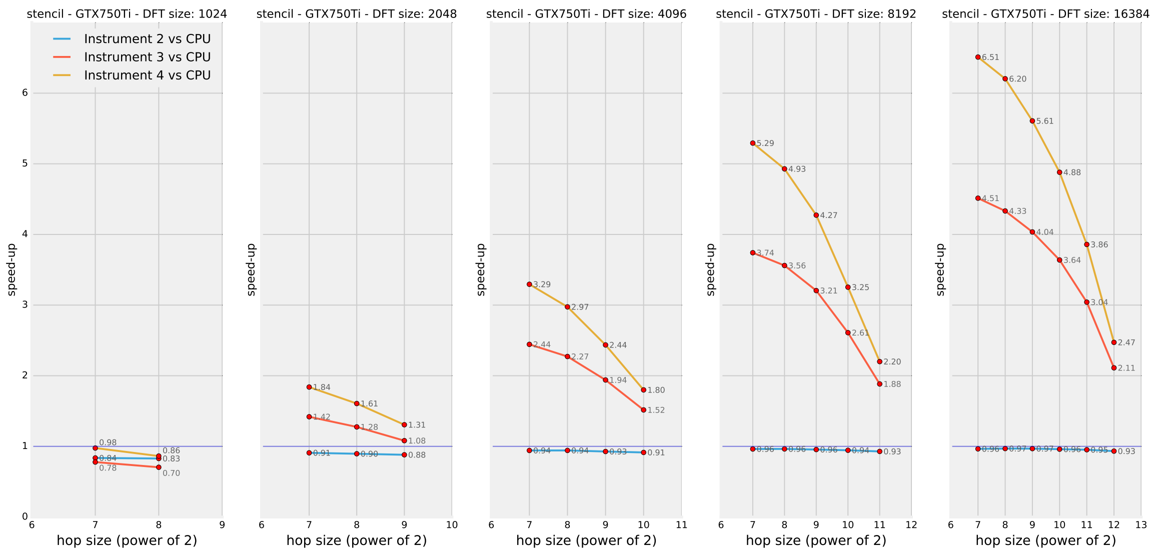

Figure 4.9: Speed-up factors related to different implementations of the *stencil* module on system 1.

4.3.4 *Scale* module

Mode 0: Basic Pitch Scaling

As it can be learned from figure 4.10 and 4.11, when testing their basic mode of operation (*mode 0*), the GP-GPU versions of the *scale* module achieve similar results to what has been shown until now.

As usual, using a DFT size of 1024 samples on the GPU does not bring any speed-up with respect to the original CPU version (yet, at the same time, the speed ratio does not get under 0.7). In all other cases, the *device-only* version of this module (*instrument 4*) succeeds in boosting the execution speed with speed-up factors that

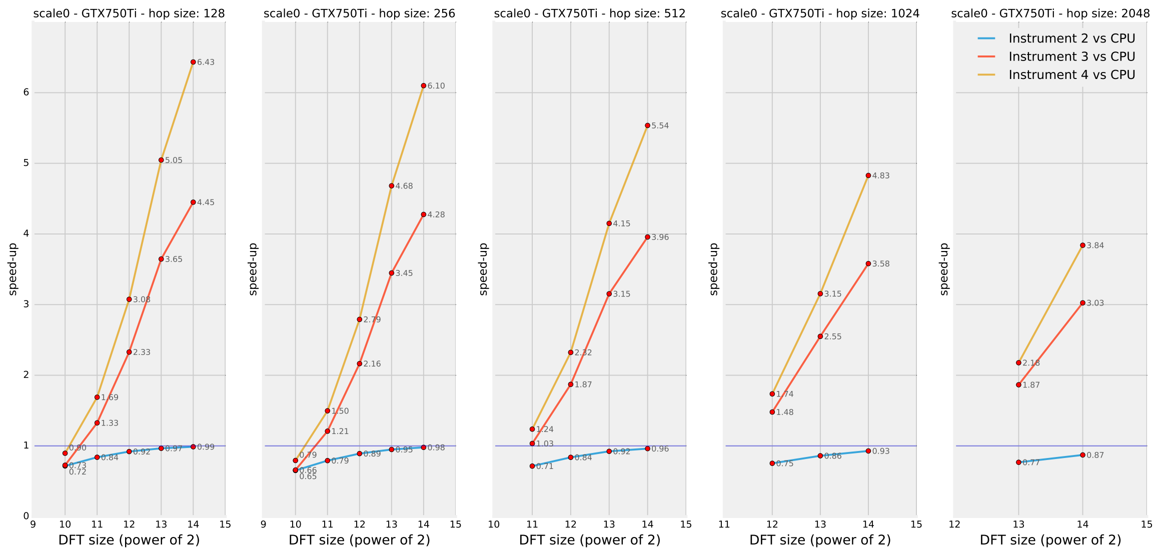

Figure 4.10: Speed-up factors related to different implementations of the *scale* module (mode 0) on system 1.

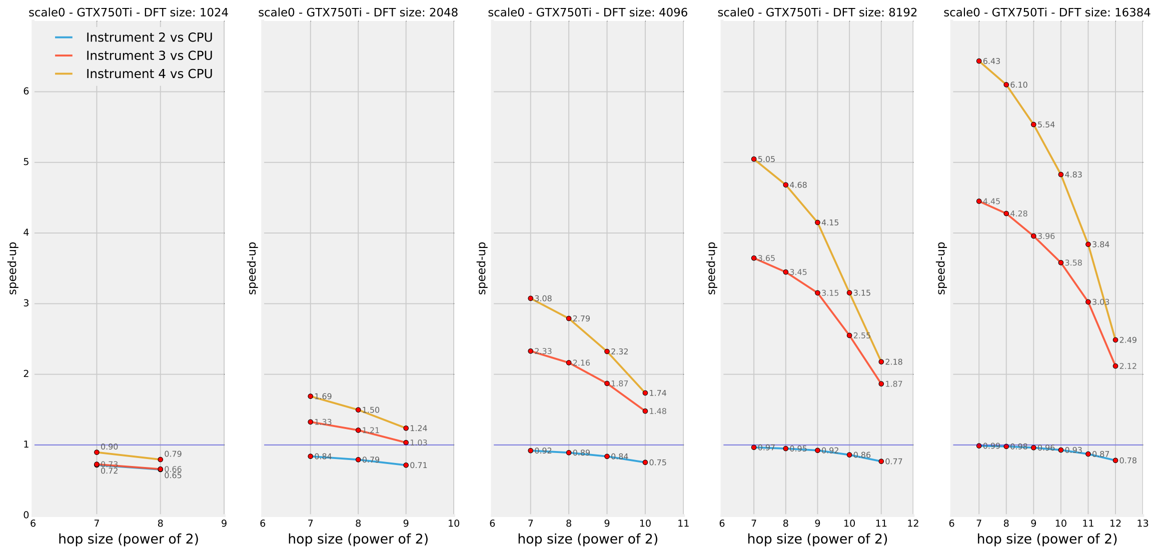

Figure 4.11: Speed-up factors related to different implementations of the *scale* module (mode 0) on system 1.

range from $\sigma_{4,1} = 1.24$ to $\sigma_{4,1} = 6.44$. When focusing on an intermediate DFT size of 4096 points, the speed-up factors range from $\sigma_{4,1} = 1.74$ and $\sigma_{4,1} = 3.08$.

As it was shown in subsection 3.6.4, this algorithm is more convoluted than those that have been analysed so far. In general terms, it has been argued that the more computationally demanding an algorithm is, the higher the potential performance gain that can be achieved with a GP-GPU approach, provided that the algorithm is well suited for parallel processing. This seems to be in contrast with a comparison between the results of the tests on the *scale* module and, for instance, those on the *gain* module: they are in fact very similar, regardless of

the different level of complexity[13] (whereas one could expect the former to achieve higher speed-ups). This can actually be explained by means of two arguments. First, it is not completely true that the pitch scaling algorithm is perfectly congenial to parallel processing (as it has been discussed in subsection 3.6.4, there is a need for atomic operations when scaling downwards); the presence of inter-bin dependencies also raise the need for some kind of synchronisation among all threads (not only at *block* level): to achieve this, separate kernels (and *Thrust* functions) are launched one after the other[14], and this introduces a significant level of overhead. Second, both the employed kernels contain some level of intra-*warp* branching, and this is potentially degrading for the overall performance in a *CUDA* environment. All in all, these factors could limit the potentially better achievements of a more compute-intensive algorithm that is run on the GPU.
All the execution times recorded for this module stay under the real-time threshold of 60 seconds.

Mode 1: Formant Conservation (Cepstrum Liftering)

While the basic pitch scaling algorithm is more computationally demanding than the gain one (for instance), the level of complexity remains quite low. When a cepstrum-based formant preservation method is introduced in the pitch scaling module, however, the complexity of the whole scheme is very much increased, especially because of the additional two FFT passes that are needed for this.
Figure 4.12 and 4.13 show the execution times recorded while testing this technique.

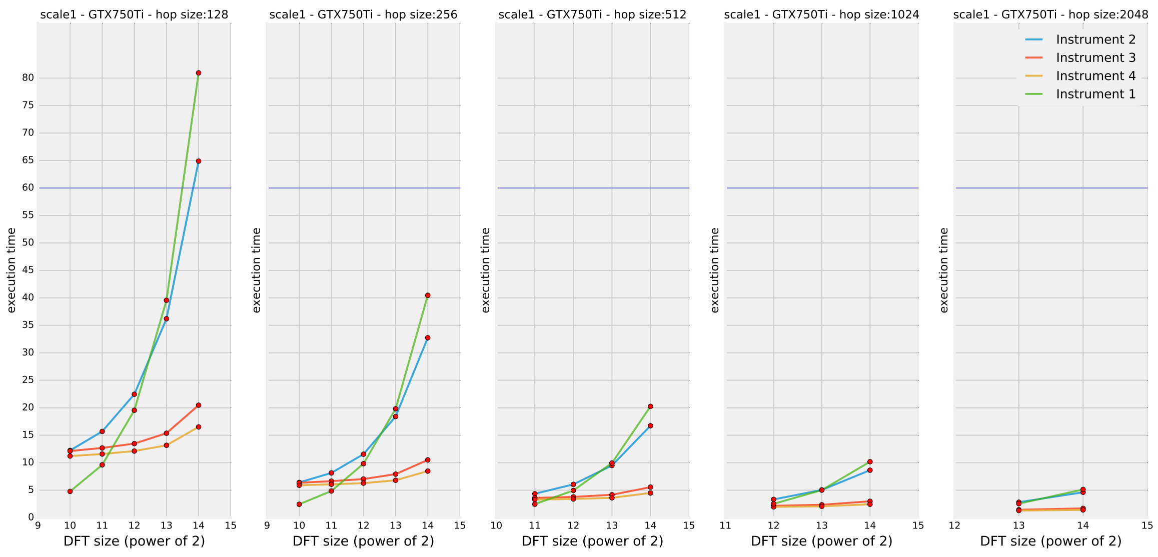

Figure 4.12: Execution time results (in seconds) from testing the *scale* module (mode 1) on system 1 over 60 seconds of audio.

[13]Anyway, the level of computational complexity of the pitch scaling algorithm is not extremely higher than that of the gain one.
[14]See subsection 3.6.4.

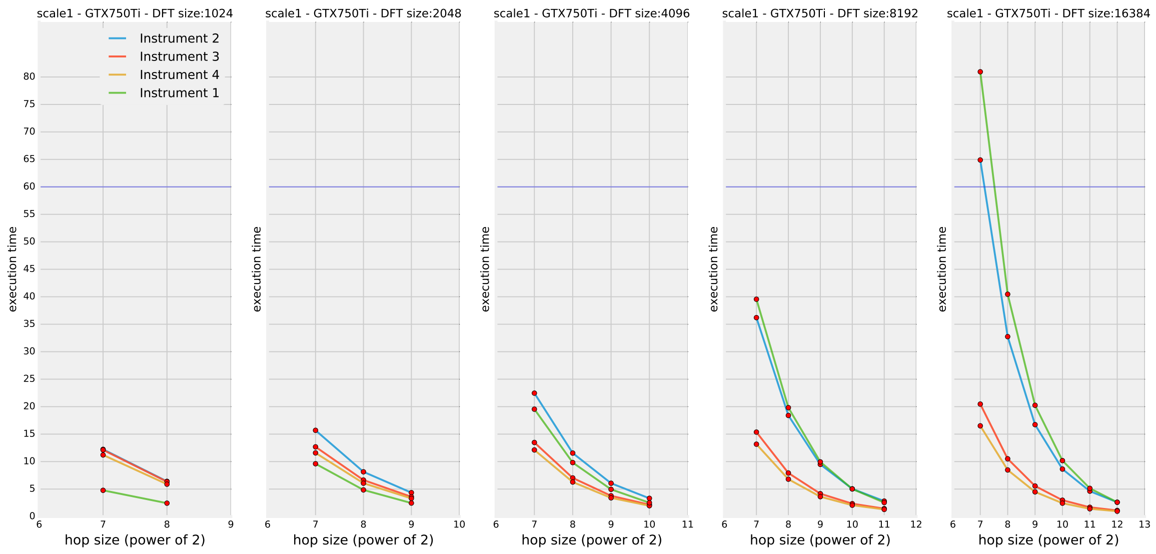

Figure 4.13: Execution time results (in seconds) from testing the *scale* module (mode 1) on system 1 over 60 seconds of audio.

From these graphs it can be learned that, indeed, the execution times related to the plain CPU version are higher with respect to all other analysed cases (so far). As in the *filter* case, extreme settings bring the execution time above the real-time threshold but the more refined GP-GPU versions succeed in bringing execution times back to a real-time scenario. Interestingly enough, *instrument 2* does not always perform worse than *instrument 1* and its performance strongly depends on the DFT size: for low DFT sizes it is strongly worsened, while for 8192-points DFTs and above it is even improved. This behaviour is a clear consequence of the introduction of a computationally expensive process in the GPU code (forward and inverse FFT to get to the cepstral domain and back).

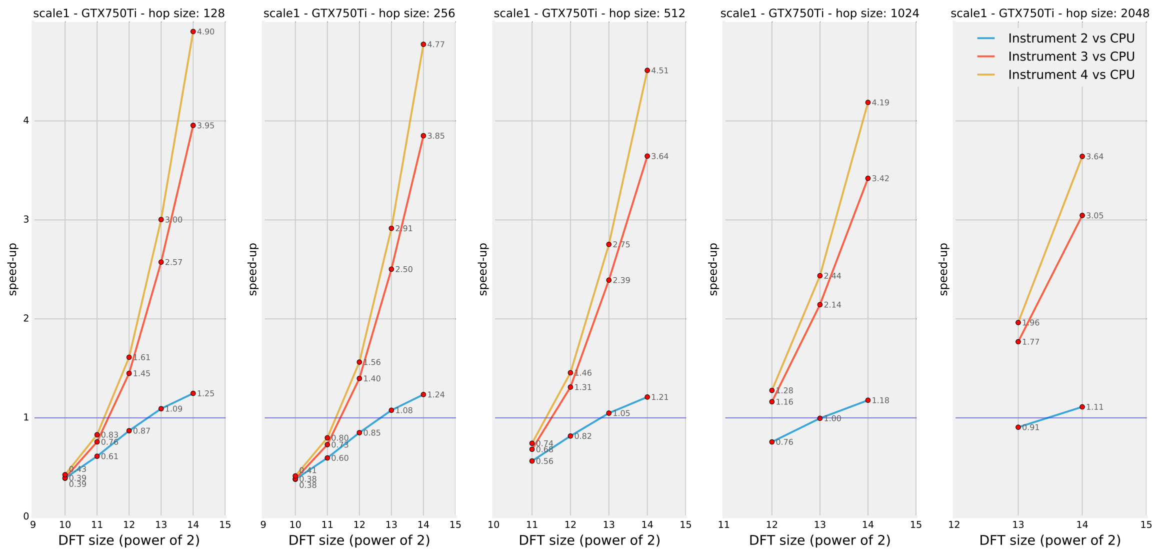

Figure 4.14: Speed-up factors related to different implementations of the *scale* module (mode 1) on system 1.

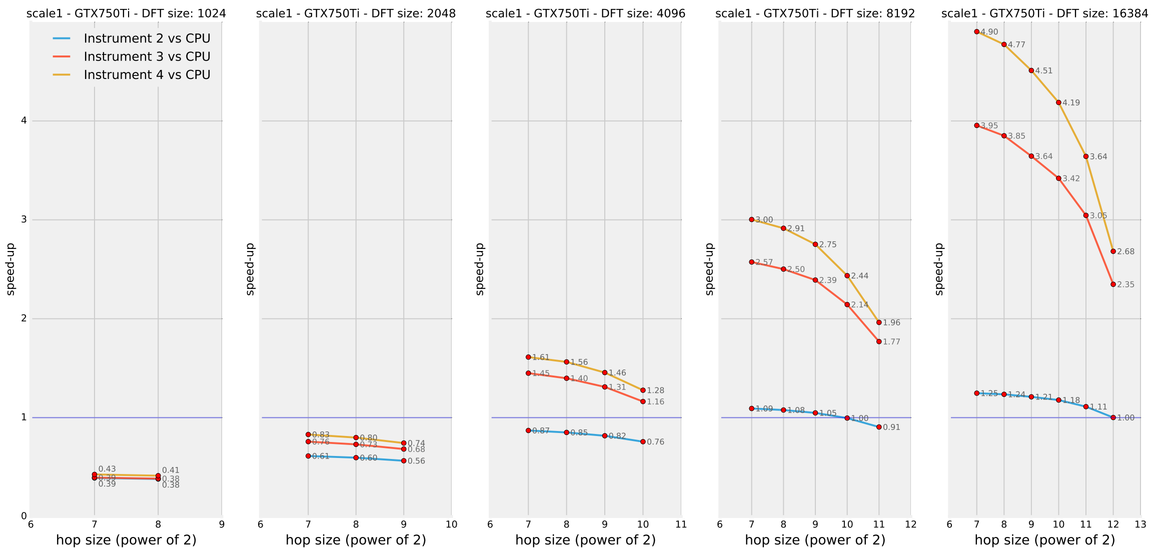

Figure 4.15: Speed-up factors related to different implementations of the *scale* module (mode 1) on system 1.

Now, turning the attention to the speed-up factors, the plots in figure 4.14 and 4.15 can be analysed. Two main differences can be noticed in these trends. First, the highest speed-up factors (those resulting from *instrument 4*) are not as good as in the other cases (they stay below a ratio of 5). Second, the speed-up increase rate with respect to an increasing DFT size seems to be higher than that seen in the previous cases: while in the *gain* module, for instance, the speed-up factor increases of a somewhat fixed amount[15] for each doubling of the DFT size (second derivative close to zero in figure 4.14), in this case the speed-up factor grows of a greater and greater amount as the DFT size is doubled (positive second derivative). As a consequence, as it has been mentioned already, *instrument 2* actually achieves positive speed-ups (up to around 1.2) for high DFT sizes, even if it goes down to a very bad 0.4 for the lowest DFT size of 1024 points. Note that also 2048-points-based processing gets critical for GPU-based spectral manipulations (figure 4.15, second graph).
While the latter phenomenon (second derivative increase) can be easily explained by the introduction of *CUDA*-based FFTs in the processing module, the general decrease of the speed-up factors related to *instrument 3* and *instrument 4* is quite unexpected. Perhaps it has to do with the fact that these algorithms comprise of many different kernels. As it is well known, in fact, kernel launches introduce a certain delay from the time the host executes the kernel on the device until the device begins execution of the kernel itself. *Kernel launch overhead* can be a limiting factor, especially when an algorithm, such as the one under analysis, needs to be split into more kernels, thus resulting in more overhead. Usually, GPU instructions are split into separate kernels for synchronisation reasons[16] (before and after FFTs,

[15]This mainly concerns *instrument 3* and *instrument 4*. *Instrument 2* actually shows a different trend (negative second derivative in the *gain* case and zero second derivative in the *scale* case) but a similar reasoning also applies to this case.

[16]It could be interesting to implement and test another version of this algorithm such that, when

for instance). Of course, synchronisation barriers themselves represent another limiting factor for the performance of this module. Yet, *instrument 4* is again the best performing overall: putting aside the critical case of a 1024-points DFT, the speed-up factors related to this implementation range from $\sigma_{4,1} = 0.74$ (this is actually a performance degradation) to $\sigma_{4,1} = 4.90$, depending on the settings employed. If the attention is focused on an intermediate DFT size of 4096 points, the recorded speed-up factors vary between $\sigma_{4,1} = 1.28$ and $\sigma_{4,1} = 1.61$, depending on the employed hop size.

4.3.5 *Shift* module

Mode 0: Basic Pitch Shifting

As it was pointed out in chapter 3, the *shift* module is almost identical to the *scale* one, with a very subtle difference which has a strong effect on the perceived result but does not change much the computational complexity.

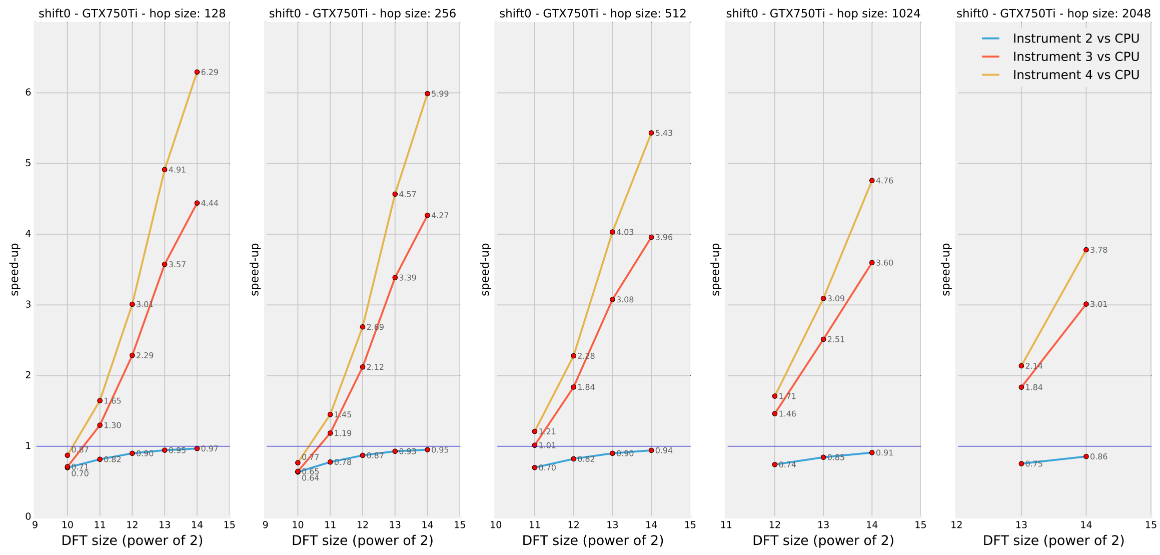

Figure 4.16: Speed-up factors related to different implementations of the *shift* module (mode 0) on system 1.

In the *CUDA* versions, the atomic operations needed for the *scale* module can be removed from the *shift* module and this could potentially lead to a further improvement. By analysing figure 4.16 and 4.17, though, this hypothetical improvement is not really measured[17]: the speed-up factors achieved by both modules are comparable in all cases (see also figure 4.10 and 4.11 for a comparison).

Instrument 4 is as usual the best performing overall: putting aside the critical case of a 1024-points DFT, the speed-up factors related to this implementation

possible, `cudaDeviceSynchronize()` API function [13] is used inside one single kernel instead of breaking up GPU computations in multiple kernels. This solution was not investigated because of a lack of time.

[17]There are other minor differences between the two algorithms that could influence their relative performances.

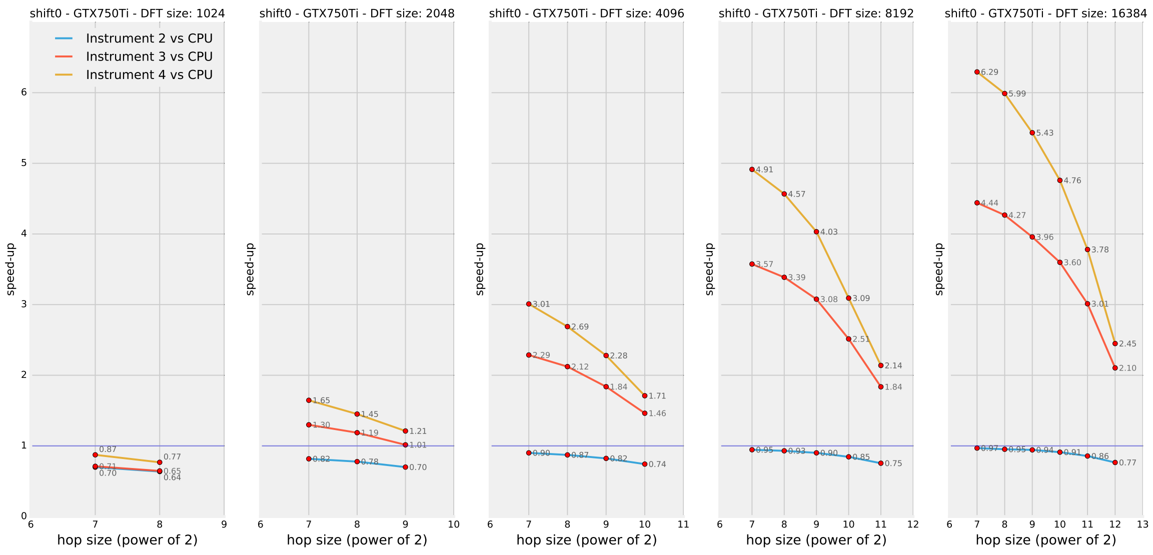

Figure 4.17: Speed-up factors related to different implementations of the *shift* module (mode 0) on system 1.

range from $\sigma_{4,1} = 1.21$ to $\sigma_{4,1} = 6.29$, depending on the settings employed. If the attention is focused on an intermediate DFT size of 4096 points, the recorded speed-up factors vary between $\sigma_{4,1} = 1.71$ and $\sigma_{4,1} = 3.01$. All the execution times recorded for this module stay under the real-time threshold of 60 seconds.

Mode 1: Formant Conservation (Cepstrum Liftering)

Again, the *shift* module shows very similar results with respect to what has been displayed for the *scale* module, also when *mode 1* is activated in order to preserve formants in the output spectrum.

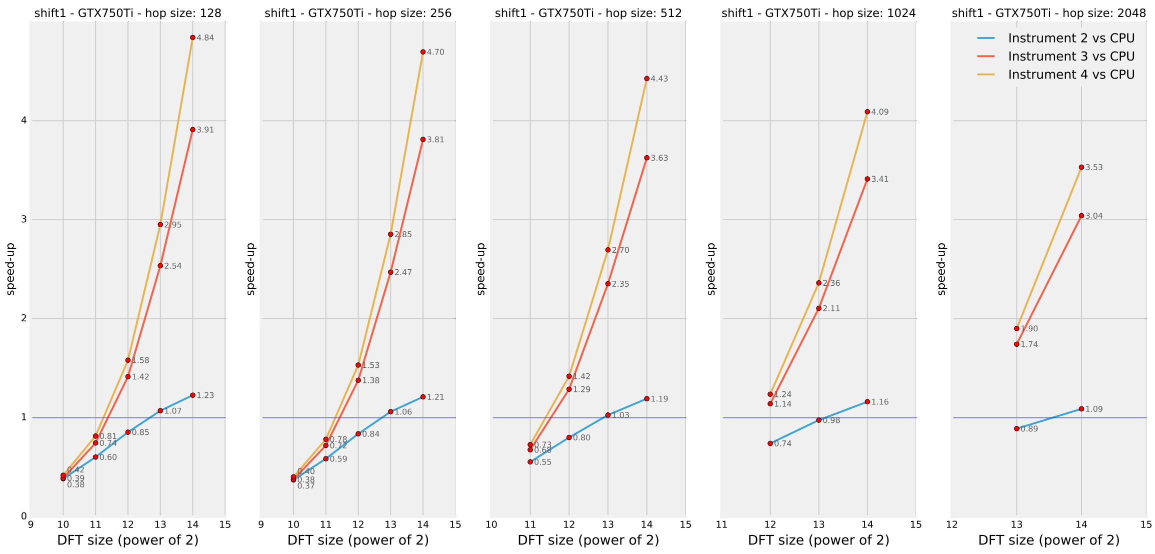

Figure 4.18: Speed-up factors related to different implementations of the *shift* module (mode 1) on system 1.

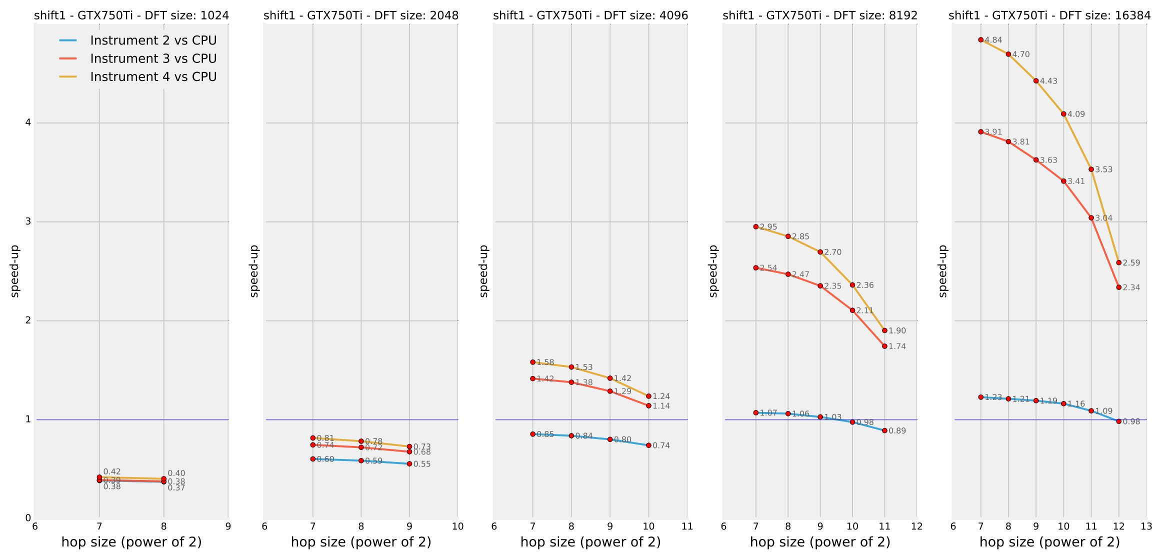

Figure 4.19: Speed-up factors related to different implementations of the *shift* module (mode 1) on system 1.

The speed-up factors related to this module can be found in figure 4.18 and 4.19.
Instrument 4 is as usual the best performing overall: putting aside the critical case of a 1024-points DFT, the speed-up factors related to this implementation range from $\sigma_{4,1} = 0.73$ to $\sigma_{4,1} = 4.84$, depending on the settings employed. If the attention is focused on an intermediate DFT size of 4096 points, the recorded speed-up factors vary between $\sigma_{4,1} = 1.24$ and $\sigma_{4,1} = 1.58$, depending on the employed hop size.
The CPU version of this module fails to perform in real-time for a DFT size of 16384 points and a hop size of 128 samples (while the GP-GPU versions succeed).

4.3.6 *Smooth* module

This module exhibits very similar results to those of the *gain*, *filter* and *stencil* modules. The speed-up factors achieved by the three *CUDA*-based versions of this algorithm are reported in figure 4.20 and 4.21.

As always, *instrument 4* achieves the best results overall: its speed-up ratios range from $\sigma_{4,1} = 1.32$ to $\sigma_{4,1} = 6.56$, depending on the specific settings (not considering the unfavourable case of a 1204-points DFT). Focusing on the intermediate DFT size of 4096 points, the recorded speed-up factors are between $\sigma_{4,1} = 1.82$ and $\sigma_{4,1} = 3.32$, depending on the specific hop size.
All the execution times recorded for this module stay under the real-time threshold of 60 seconds.

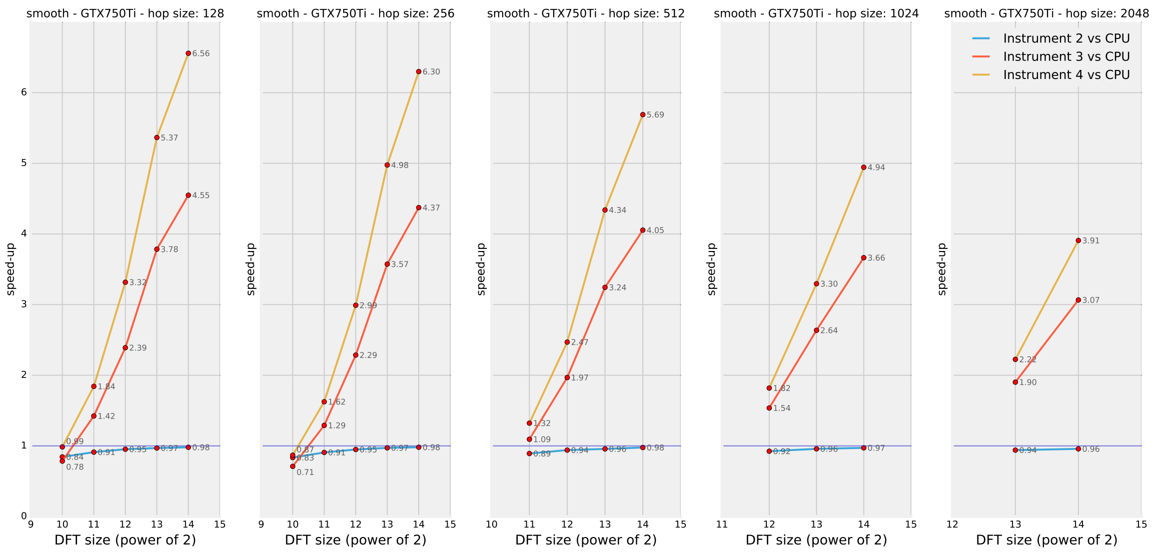

Figure 4.20: Speed-up factors related to different implementations of the *smooth* module on system 1.

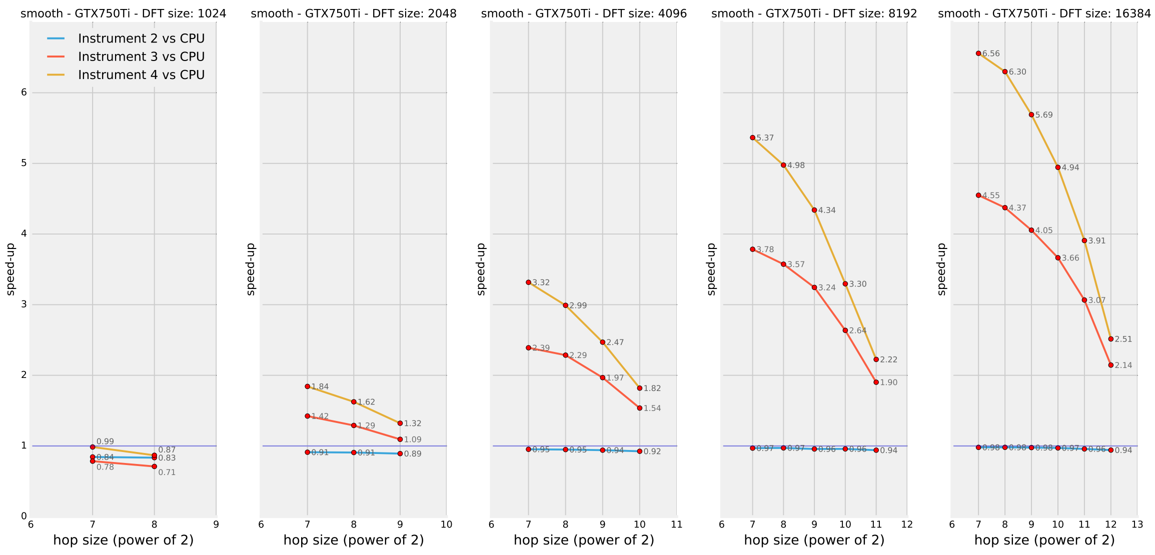

Figure 4.21: Speed-up factors related to different implementations of the *smooth* module on system 1.

4.3.7 *Blur* module

As it has been pointed out in section 3.6.7, the *blur* module is quite demanding in terms of computational load and, being very well suited for parallel processing, it is expected to perform very well on the GPU.
It is important to stress that the computational load of this algorithm is very much dependant on the specific averaging time employed in the application. To try and simulate a generic situation, the tests were designed in such a way that this parameter is changed over time, linearly, from a null averaging window (which has no effect on the output) to a blurring period of one second (which, in turn, strongly affects the output stream). The `imaxdel` parameter is thus set to one second.

Figure 4.22 shows the execution times recorded after these tests.

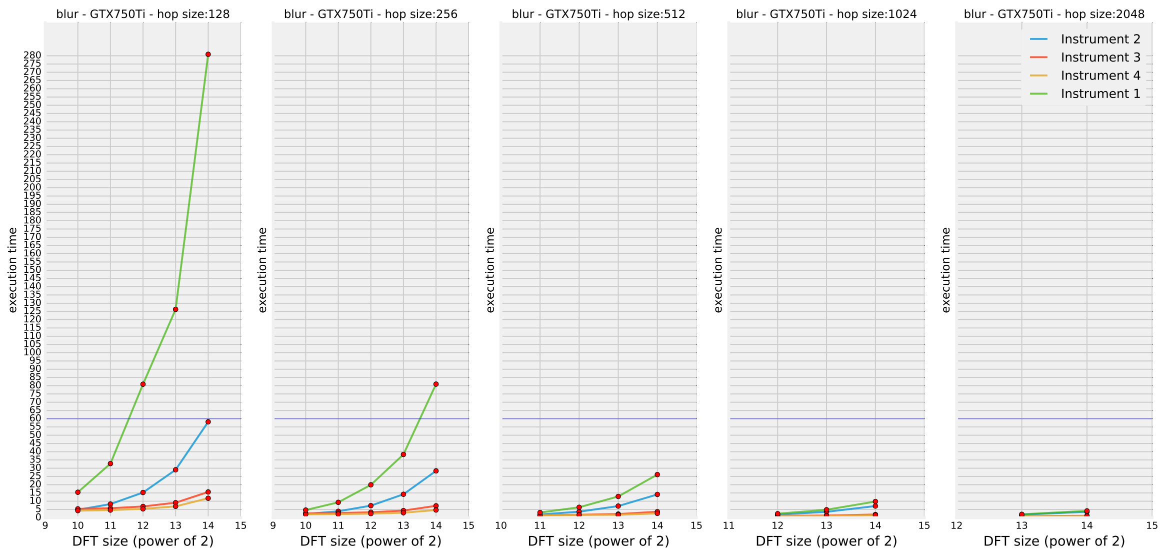

Figure 4.22: Execution time results (in seconds) from testing the *blur* module on system 1 over 60 seconds of audio.

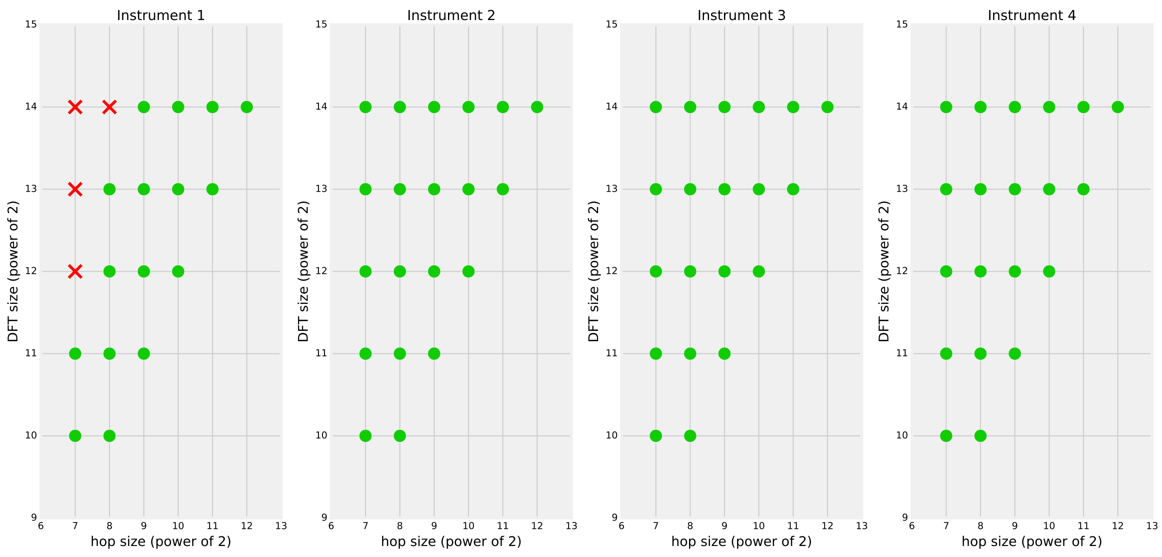

Figure 4.23: Real-time analysis for the four different versions of the *blur* module. Red crosses reveal real-time operation failure while green dots indicate success.

It is evident that, for what concerns the original CPU version (green line) the resulting execution times grow very fast as the hop size is decreased. They rapidly get out of scale, crossing the real-time threshold in more than one case and by some very large amounts (in the extreme case, the CPU version is more than four times slower than the real-time limit).
Interestingly, this is not the case for the *CUDA*-based implementations: in fact, they all turn out to be capable of real-time operation, under all settings. Figure 4.23 shows that any GP-GPU implementation succeeds in cutting the execution

time enough, making real-time operation possible in four critical cases (red crosses). Turning the attention to the speed-up factors, these are reported in figure 4.24 and 4.25.

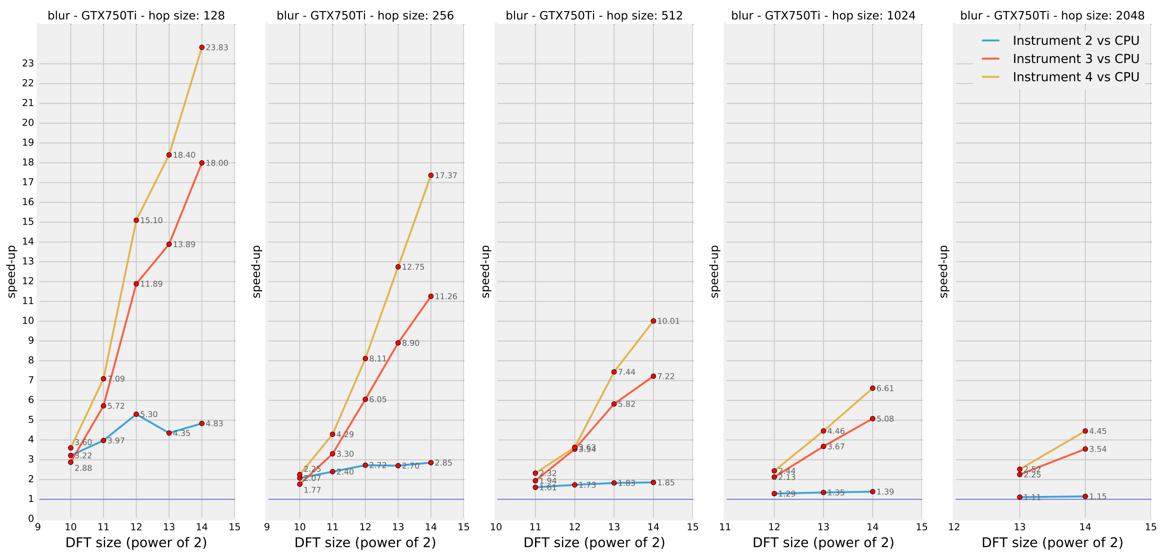

Figure 4.24: Speed-up factors related to different implementations of the *blur* module on system 1.

These plots exhibit a few differences with respect to those related to all previous modules. First of all, the speed-up factors are greatly improved in many cases. *Instrument 4* even achieves a speed-up factor as high as almost two dozens. It is safe to speculate that these exceptional improvements could be stretched even further if a higher degree of blurring was employed in the testing application for somewhat extreme effects (one second of maximum averaging time is not an extreme value, after all). In fact, these extraordinary speed-up ratios are certainly ascribable to the presence of the accumulation process which is performed inside of each GPU thread: the more frames are involved in this accumulation process, the more time GPU resources are kept busy for, simultaneously. This has the double effect of hiding the latencies related to memory transfers and boosting the execution speed even further, especially for high DFT sizes. In the original CPU implementation all the accumulation processes (each one is related to a different phase vocoder channel) need to be spread out over time, rather than over hardware resources, as in the *CUDA* versions. Quite surprisingly, even the least refined *CUDA* version, *instrument 2*, happens to perform better (even much better, in some cases) than the original CPU implementation.
Also, while it has been recognised as a critical case throughout the analysis of all the modules so far, the 1024-points spectral processing scenario actually brings good results for this kind of manipulation. As a consequence, *instrument 4* turns out to be the best performing version under all conditions. It achieves speed-up factors that range from $\sigma_{4,1} = 2.25$ to $\sigma_{4,1} = 23.83$, depending on the employed settings. If the DFT size is kept to an intermediate level of 4096, the resulting speed-up ratios range between $\sigma_{4,1} = 2.44$ and $\sigma_{4,1} = 15.10$, depending on the specific hop size.

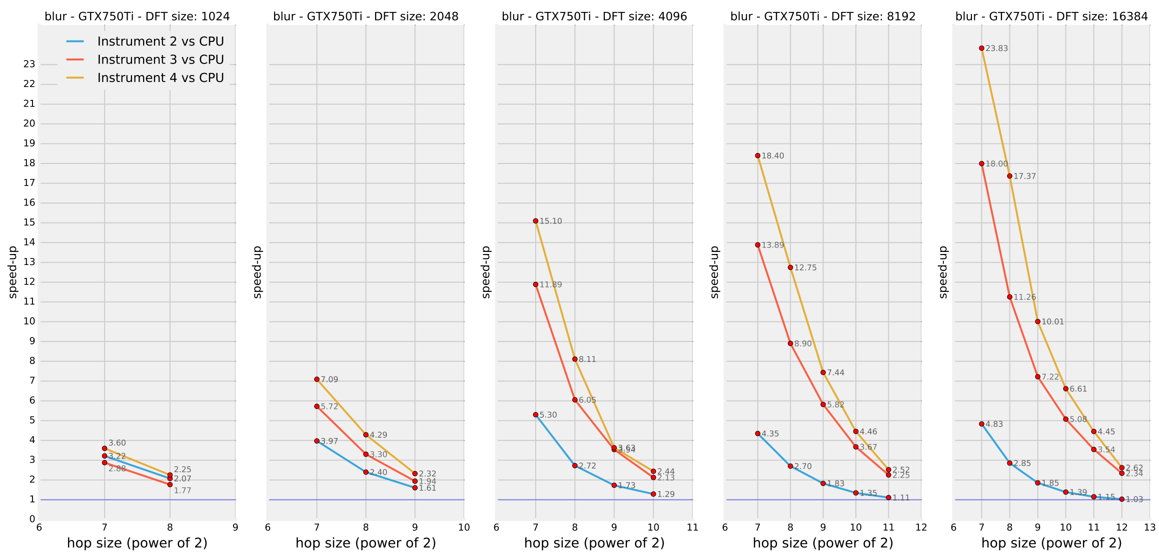

Figure 4.25: Speed-up factors related to different implementations of the *blur* module on system 1.

4.3.8 *Mix* Module

The speed-up factors resulted from testing the different versions of the *mix* module are shown in figure 4.26 and 4.27.

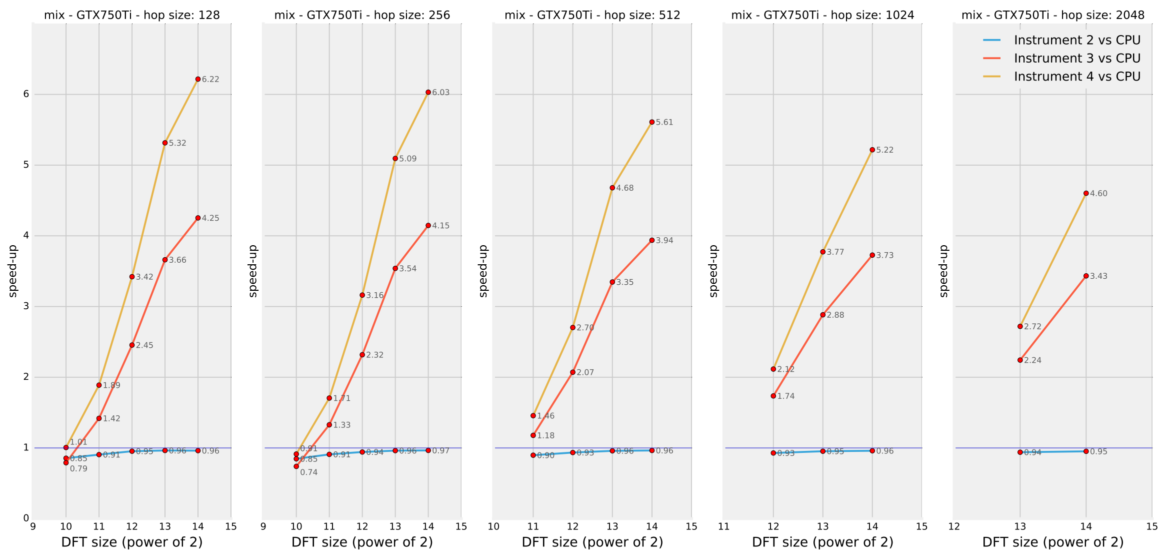

Figure 4.26: Speed-up factors related to different implementations of the *mix* module on system 1.

These values and their trends are very similar to those analysed for the *gain* module, for instance. The same comments apply to this scenario as well.
The concerns that were raised in section 3.6.8 about the potentially undermining presence of conditionals in the *CUDA C* implementations seem to be negligible, indeed, as the speed-up factors are comparable to those resulted from other modules

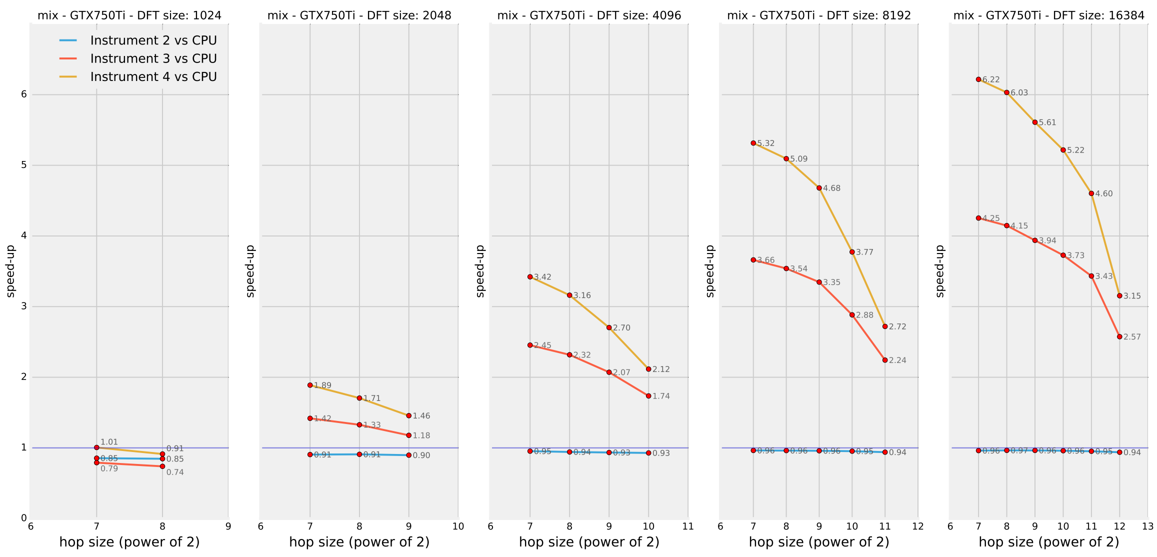

Figure 4.27: Speed-up factors related to different implementations of the *mix* module on system 1.

that are not characterised by this issue.

Instrument 4 is again the best performing overall. Putting aside, as usual, the critical case of a 1024-points DFT, the speed-up factors related to this implementation range from $\sigma_{4,1} = 1.46$ to $\sigma_{4,1} = 6.22$, depending on the settings employed. If the attention is focused on an intermediate DFT size of 4096 points, the recorded speed-up factors vary between $\sigma_{4,1} = 2.12$ and $\sigma_{4,1} = 3.42$.
Both *instrument 1* and *instrument 2* fail to achieve real-time performance under critical settings (16384-points DFT and 128-samples hop size), but *instrument 3* and *instrument 4* actually succeed.

4.3.9 *Morph* Module

The speed-up factors resulted from testing the different versions of the *morph* module are shown in figure 4.28 and 4.29.
These values and their trends are very similar to those analysed for the *gain* module, for instance. The same comments apply to this scenario as well.

Instrument 4 is again the best performing overall. Putting aside, as usual, the critical case of a 1024-points DFT, the speed-up factors related to this implementation range from $\sigma_{4,1} = 1.45$ to $\sigma_{4,1} = 6.18$, depending on the settings employed. If the attention is focused on an intermediate DFT size of 4096 points, the recorded speed-up factors vary between $\sigma_{4,1} = 2.12$ and $\sigma_{4,1} = 3.38$, depending on the employed hop size.
Both *instrument 1* and *instrument 2* fail to achieve real-time performance under critical settings (16384-points DFT and 128-samples hop size), but *instrument 3* and *instrument 4* actually succeed.

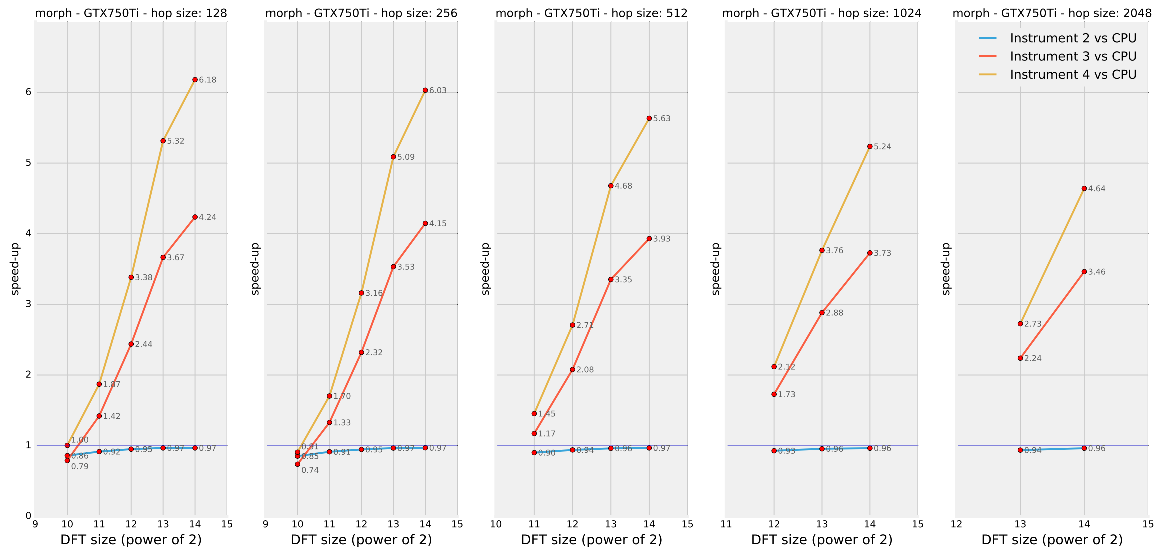

Figure 4.28: Speed-up factors related to different implementations of the *morph* module on system 1.

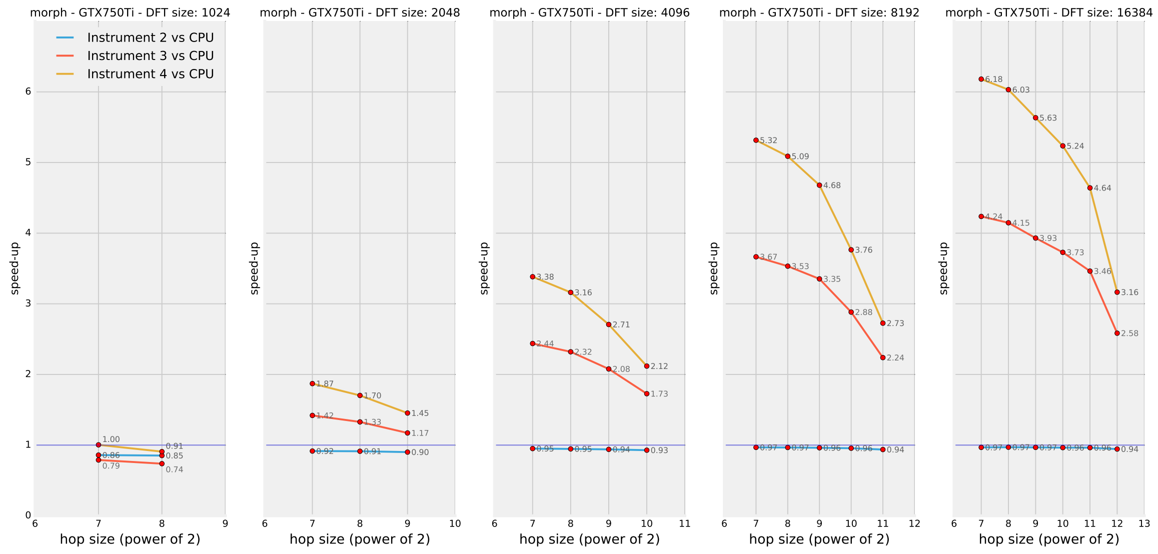

Figure 4.29: Speed-up factors related to different implementations of the *morph* module on system 1.

4.4 GPU Comparison

All the tests that have been shown in the previous section were actually run on system 2 as well[18]. In this section, some of the results from the two systems will be compared in order to draw a more complete picture of *CUDA*-based spectral audio processing.
Not all the results from system 2 will be reported here: the performance of the two systems will be compared only for what concerns three modules that have shown distinctive behaviours, and can thus be chosen as representative for different classes

[18]Please refer to section 4.1 for both systems' detailed specifications.

of algorithms. These are the *gain* module (the simplest of all), the *blur* module (the one characterised by the highest density of operations per thread) and the *scale* module with formant conservation (mode 1, which is the one that has the most articulated internal structure and achieves the lowest speed-up factors).
Also, in order to keep data analysis simple, only the performance of *instrument 4* will be addressed. The meter chosen for this comparison is thus the ratio between the execution time of *instrument 4* on system 2 over the execution time of the same *instrument* on system 1: $\sigma_{s_1,s_2} = \frac{t_{s_2}}{t_{s_1}}$, where t_{s_1} and t_{s_2} are the execution times related to *instrument 4* on system 1 and 2 respectively. This parameter will be addressed as a "speed-up factor" even though it turns out that system 2 is not always slower than system 1 (see below).

4.4.1 *Gain* Module

Figure 4.30 and 4.31 show how much of a difference exists between the execution speeds resulting from running the tests on the two different GPUs, while varying the DFT size and the hop size settings.

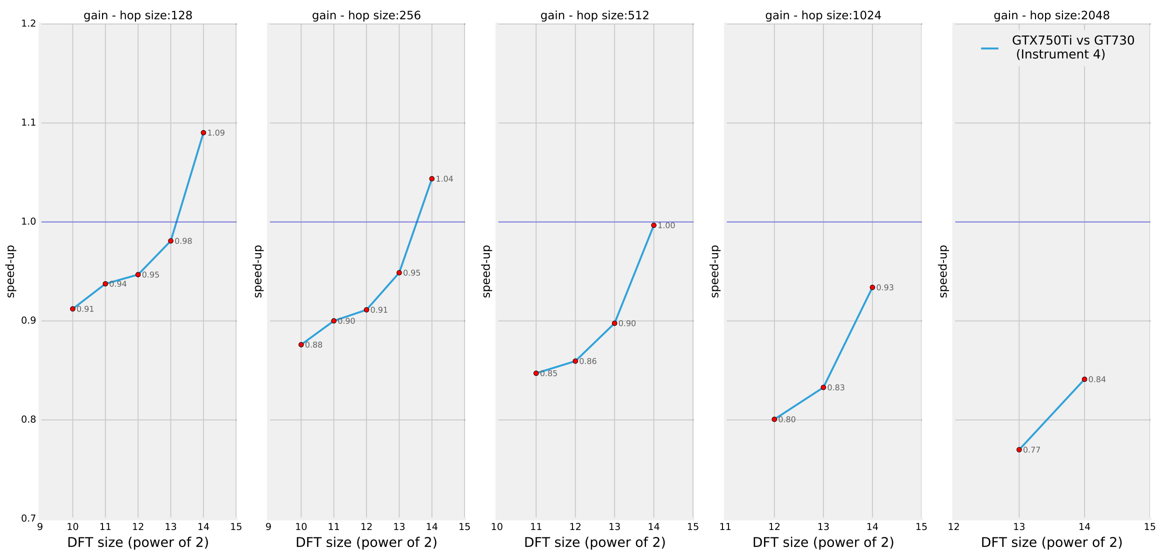

Figure 4.30: Speed-up ratios obtained from the two different systems used for running the *gain* module (*instrument 4* only).

The horizontal dark blue line highlights the points of the graphs characterised by an equal performance from the two systems. Above this line system 1 (with a GeForce GTX750Ti) performs better than system 2 (GeForce GT730). Vice versa, under this line system 1 performs worse than system 2.
These plots show that, for this module in particular, there is not a substantial difference between the two systems, as the recorded ratios turn out to be close to 1 in all cases. Quite surprisingly, system 1 seems to be a little slower than system 2 in most cases. However, if the attention is focused on trends rather than on single speed-up factors, it seems that the GeForce GTX750Ti of system 1 is a potentially better candidate for massively parallel tasks. In fact, figure 4.30 clearly shows that

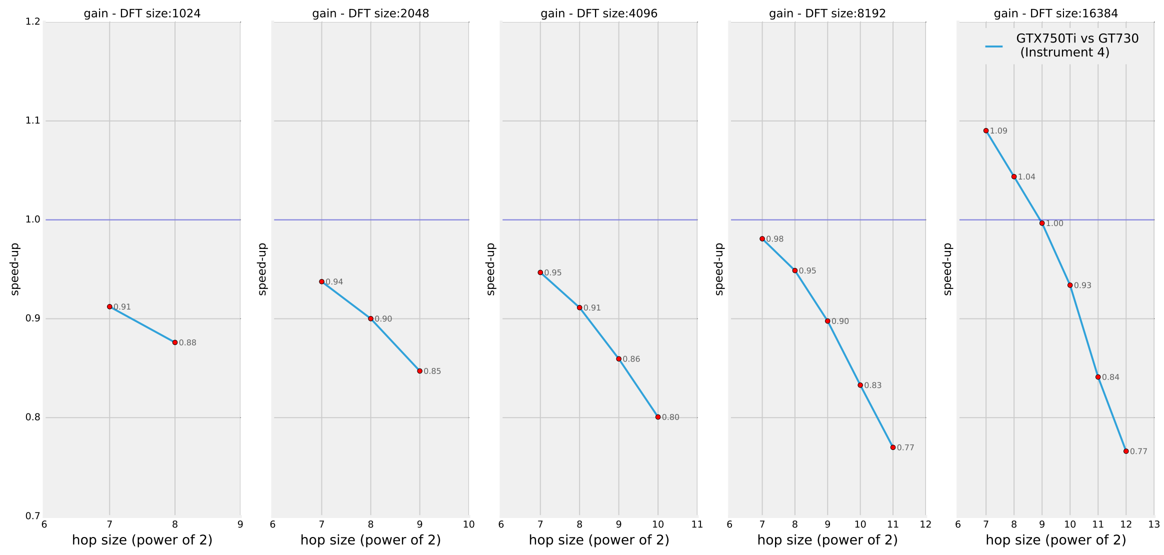

Figure 4.31: Speed-up ratios obtained from the two different systems used for running the *gain* module (*instrument 4* only).

system 1 performs better and better when the DFT size is increased. It is safe to speculate that, if a set of higher DFT sizes was considered, system 1 would have performed better than system 2 in most cases. Still, it has to be said that a DFT size higher than 16384 points is rarely useful in the audio scope, especially when real-time applications for live musical performances are considered. Also, when analysing 4.31, it can be learned that the difference between the two systems changes quite rapidly as the hop size is changed: system 1 improves its performance, with respect to system 2, when the hop size is reduced. Reducing the hop size eventually translates into a higher level of utilisation of the GPU, as it is required to operate more often. This means that the more the GeForce GTX750Ti is utilised, the better its performance gets if compared to the older GeForce GT730. Bringing together this observation and the previous one, it can be inferred that system 1 performs better than system 2 when the GPU is called into action frequently and with a lot of data to be processed in parallel. In other words, it seems that for this module the GeForce GTX750Ti is under-utilised for the employed settings.

4.4.2 *Scale* Module (Mode 1)

Figure 4.32 and 4.33 show a performance comparison between the two computer systems used for testing the *scale* module (*mode 1*).

Comparing these plots with those obtained for the *gain* module, it can be observed that there is not much difference: the trends are in fact very similar. The same observations can be made as well: the GeForce GTX750Ti seems to be under-utilised for all the employed settings but it is in theory a better candidate for data-intensive and high throughput parallel computation.

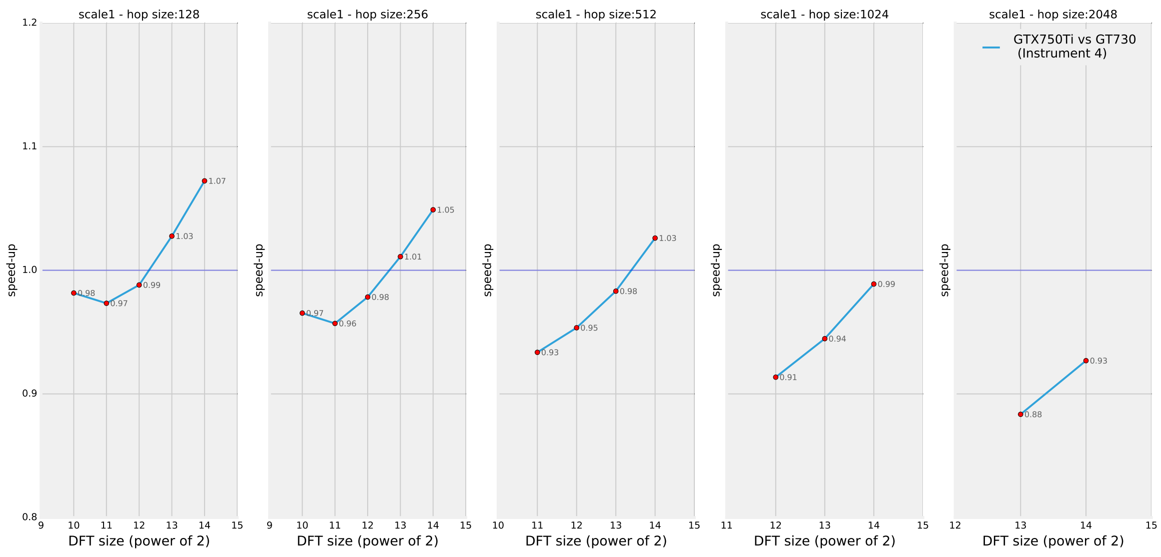

Figure 4.32: Speed-up ratios obtained from the two different systems used for running the *scale* module (mode 1, *instrument 4* only).

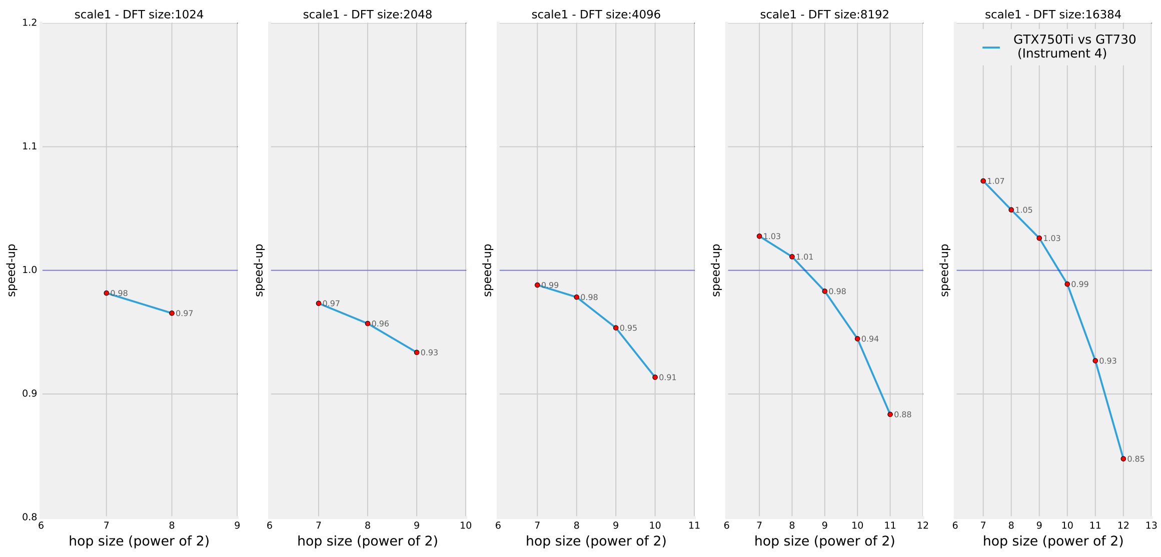

Figure 4.33: Speed-up ratios obtained from the two different systems used for running the *scale* module (mode 1, *instrument 4* only).

4.4.3 *Blur* Module

Figure 4.34 and 4.35 show a performance comparison between the two computer systems used for testing the *blur* module.

As it can be learned from these plots, as well as by comparison to those obtained in the *gain* and *scale* case, this module shows a clear advantage of system 1 over system 2, on average. As a matter of fact, the *blur* module seems to be computationally demanding enough to exploit the computing power of the GeForce GTX750Ti even for non-extreme DFT size and hop size settings. For example, when the hop size is kept under 512 samples, system 1 performs better than system 2 regardless of the DFT size employed (figure 4.34, first two graphs). In order to make the GeForce

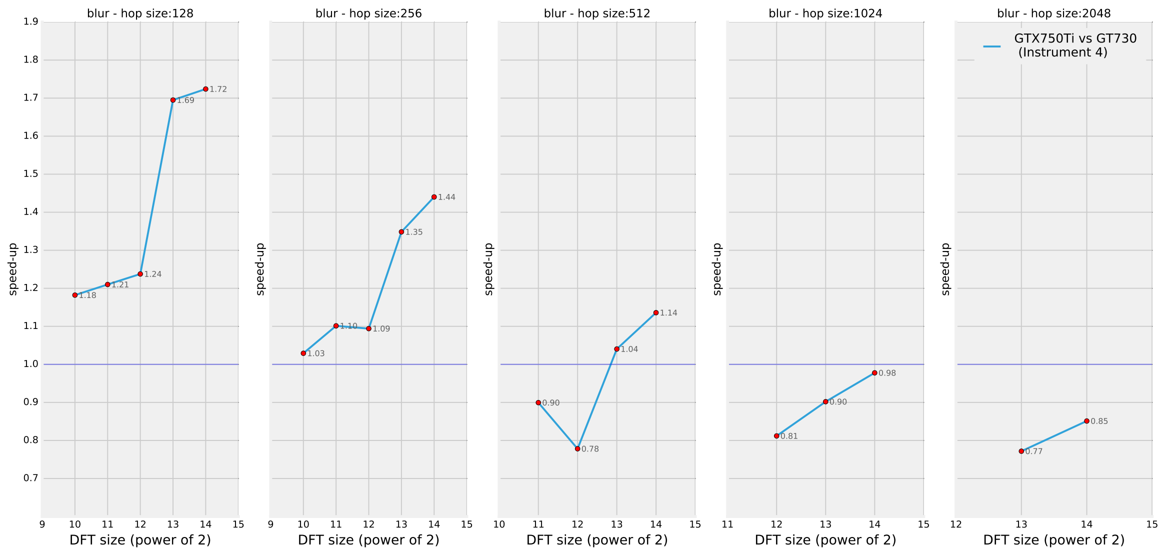

Figure 4.34: Speed-up ratios obtained from the two different systems used for running the *blur* module (*instrument 4* only).

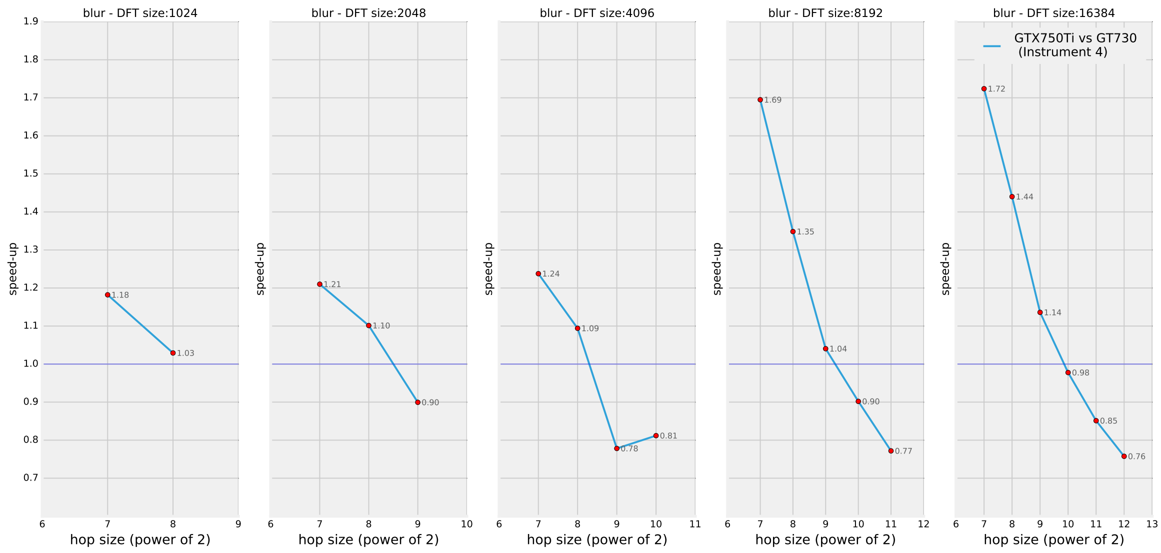

Figure 4.35: Speed-up ratios obtained from the two different systems used for running the *blur* module (*instrument 4* only).

GTX750Ti actually stand out, however, the hop size needs to be kept small, even when high DFT sizes are employed (figure 4.35, last two graphs).

4.5 Possible Improvements: Code Optimisation

It is very important to stress that the generally positive results obtained by testing the GP-GPU versions of phase vocoder *unit generators* have been achieved while developing quite basic implementations of *CUDA*-based shared objects. *CUDA* is a programming environment that can be approached very easily: it allows any beginner to program the GPU by means of a few simple steps and does not require

much preliminary knowledge about GPU architecture. At the same time, *CUDA* offers to the more experienced programmer a wide range of tools that can be exploited in order to optimise GP-GPU programs and achieve much greater speed-up factors with respect to a basic implementation. For what concerns some kinds of algorithms, it could be the case that only a properly optimised GP-GPU code has a chance to compete in terms of execution speed with a normal sequential code running on a high-end CPU.
The shared objects presented in chapter 3 (and tested in this chapter) were written using only the very basic concepts of *CUDA* programming. Unfortunately, there was no time to try and develop more sophisticated versions of the same algorithms so that they could be optimised for *CUDA*-enabled GPUs[19]. Besides, the experimental results obtained via the most basic versions are very promising in general and they were enough to validate the benefits of GP-GPU computing in the framework of spectral audio manipulations.

For completeness, this section presents a few techniques that can be possibly employed in order to try and further optimise the codes that have been shown in chapter 3. These are classic strategies that are provided to the *CUDA* programmer and that usually lead to a performance improvement. The concepts that are going to be presented here are mostly taken from [9].

4.5.1 *Task Parallelism & CUDA Streams*:

As it has been shown in section 3.5.2, any kind of GP-GPU application involves a computation scheme that is made of three separate stages: host-to-device data transfer, parallel computation on the GPU and device-to-host data transfer. These stages are usually called *tasks* in GP-GPU parlance. The *CUDA* environment provides ways to execute different kind of tasks simultaneously[20]. Not only it is possible to overlap computations with memory transfers in general[21], but it is also possible to overlap in-going and out-going data displacements. In fact, the PCIe interconnection that usually links host and device in a computer system actually supports the simultaneous transmission of data in both directions. As a result, *CUDA* allows for some level of *task parallelism*: specifically, up to three tasks (one of each kind) can be executed simultaneously (see figure 4.36). This feature is obviously very useful when the GPU is employed for more than one process at the same time (and each process is based on different data), but it can also be convenient even when dealing with one single process at a time: in some cases, the data to be processed can be partitioned in subsets that are sent to the device and handled sequentially; in this way, portions of data are transferred to the device as other subsets are processed and results are moved back to the host. Task parallelism is achievable in *CUDA* by means of so-called *CUDA streams*, a software abstraction

[19]In this whole argument, the development of *device-only* versions of the modules is being given for granted, even though it could be classified as some sort of code optimisation (see section 3.5.1).

[20]Actually, this is a feature that was not available in older *CUDA*-enabled GPUs: it is only available in models that support *device overlap*.

[21](Of course, computations must be performed on data that has already been transferred to the device in a previous step and results can be moved back to the host only after the computations are completed.

that groups together the tasks that make up a processing chain. The programmer is given the possibility of assigning each process to a particular *CUDA stream* or even partition a single process into two or three of these. *Streams* are then automatically handled and scheduled by means of two functional units (a *copy engine* for memory transfers and a *kernel engine* for GPU-based parallel computations) and a queuing system called *hyper queues* in order to maximise *stream* overlap while assuring a correct data flow.

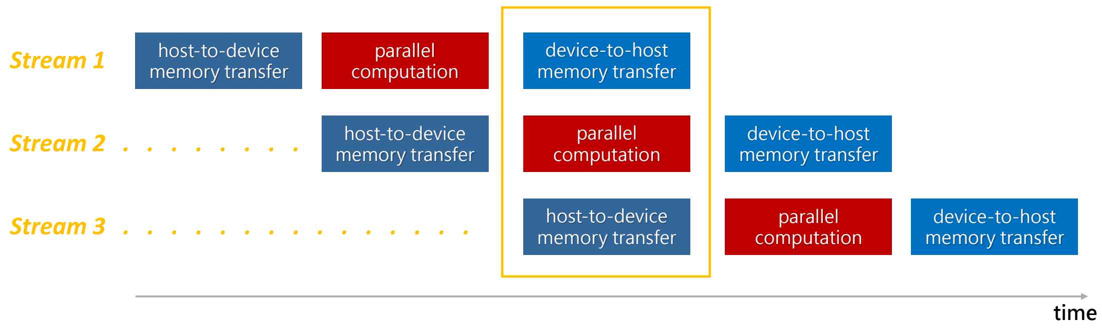
Stream 1
Stream 2
Stream 3
host-to-device memory transfer
parallel computation
device-to-host memory transfer
host-to-device memory transfer
parallel computation
device-to-host memory transfer
host-to-device memory transfer
parallel computation
device-to-host memory transfer
time

Figure 4.36: A graphical representation of *task parallelism* and *CUDA streams*. Each box represents a *task*. The yellow rectangle in the middle highlights a situation for which *task parallelism* is optimised.

4.5.2 *Shared Memory*

In addition to *global memory*, *CUDA*-enabled GPUs feature at least one extra level of on-chip memory inside the *Streaming Multiprocessors*. This memory level is much faster than *global memory*, both in terms of latency and throughput (it is highly parallel) but it is also much smaller. This storage area is divided into two conceptually different kinds of memory: an automatically managed L1 cache between the *CUDA cores* and the *global memory* (or the L2 cache when present), and a so-called *shared memory*, which is meant to be managed by the programmer. It is called *shared memory* because the scope of its content is actually shared between threads inside the same *block*. In many cases, exploiting the *shared memory* in a clever way can help achieving much better performances, given its lower latency and higher throughput. The idea is that of loading from *global memory* into *shared memory* data that is meant to be accessed more than once (most of the times by more than one thread) so to reduce the total number of *global memory* accesses. The fact that *shared memory* is quite limited, though, raises the need to design “tiled” versions of the desired algorithms. In a tiled algorithm, operations are carried out, step by step, only on a limited subset of input data in order to give partial output results that will become definitive only after the last iteration. A typical example of an application that can easily exploit *shared memory* when it is performed using a tiled algorithm is matrix multiplication. See the *CUDA Programming Guide* ([13]) for more details about *shared memory* management.

4.5.3 *Shared Memory Privatisation* and *Atomic Operations*

Shared memory can be also used in combination with *atomic operations* in order to decrease the level of serialisation that they might cause. This is done by creating different copies of the output data, so that each of these is *private* to each specific thread block. Once each needed operation has been performed, all the partial (and private) results are combined into one single definitive result in the *global memory*. In this way, many output conflicts (those that actually raised the need for *atomic operations*) are avoided and there is much less need for serialisation. At the same time, *atomic operations* are performed faster, thanks to *shared memory*. Of course, the operations involved in this process need to be associative and commutative. Also, the size of the output result needs to be small, so that all the private copies can fit in *shared memory*.

4.5.4 *Pinned Host Memory*

When memory transfers between host and device are performed, pinned host memory has to be employed in order to avoid that the involved data is paged out by the operating system while the displacement is taking place. The use of pinned memory is provided automatically by the *CUDA* API memory transfer functions and it is transparent to the programmer. However, this introduces some overhead: for instance, when data is copied to device memory from host memory, it actually needs to be copied to a pinned memory area before the displacement is actually started, otherwise the same data could be potentially (and accidentally) paged out by the operating system during the course of the slow transfer. This extra step can be avoided if data is allocated into pinned memory in the first place. *CUDA* provides special API functions to do just that. See the *CUDA Programming Guide* ([13]) for more details about pinned memory allocation.

4.6 Conclusions

The tests presented in section 4.3 confirm that a GP-GPU approach to phase-vocoder-based sound manipulations generally leads to a good level of speed-up with respect to plain CPU processing, especially when data transfers between host and device are kept limited[22]. In some cases, this speed-up factor can be higher than one or two dozens. This can lead to a dramatic shift in the execution speed of phase-vocoder-based applications, allowing highly demanding processes to run in real time on less powerful machines, provided that they are equipped with a graphics processor. At the same time, CPU resources are freed and made available to other possible concurring tasks.

[22]Thus, when considering the *device-only* version of the developed *plugin opcodes*.

Table 4.4: A recap of the results presented in section 4.3: GPU vs CPU speed-up factors when considering DFT sizes between 2048 points and 16384 points and varying hop size.

Module	Lowest speed-up factor	Highest speed-up factor
Gain	1.62	6.51
Filter	1.35	5.98
Stencil	1.31	6.51
Scale (0)	1.24	6.44
Scale (1)	0.74	4.90
Shift (0)	1.21	6.29
Shift (1)	0.73	4.84
Smooth	1.32	6.56
Blur	2.25	23.83
Mix	1.46	6.22
Morph	1.45	6.18

As it can be quickly learned from table 4.4 and 4.5, most of the modules considered in this project have shown a homogeneous set of results (the *blur* module being the only substantial exception). Eventually, these algorithms are characterised by very short chains of simple operations to be carried out on each phase vocoder channel (sometimes as short as one single operation). This model has proven to benefit from a parallel computing scheme, especially when the total number of channels is high, but it has also proven not to be optimal. In fact, much better results can be achieved when longer chains of operations are required on the data related to each channel. This can be learned from analysing the results obtained by the *blur* module in particular, which stands out for its impressive speed-up ratios. The *scale* and *shift* modules with formant conservation seem to be an exception to this: even though they contain forward and inverse FFT stages (which were proven to be well suited for GPU-based parallel processing), they seem to suffer from synchronisation issues between different stages of these algorithms. Still, these two modules also show the fastest rate of speed-up increase with respect to doubling the DFT size: this leads to speculate that if larger DFT sizes had been tested[23] these modules could have shown speed-up ratios in between those from *gain* (i.e., from the main group of modules) and those from *blur*.
The three different versions of a *CUDA*-based spectral manipulation framework (i.e., the three GP-GPU *Csound instruments*) that were analysed in this chapter have shown the predicted behaviour. *Instrument 2* is the slowest of them: it seldom performs better than the original CPU version (still, it never performs critically worse). *Instrument 3* shows much better speed-up factors in general,

[23]However, DFT sizes higher than 16834 are seldom useful in an audio processing framework.

especially for higher DFT sizes. *Instrument 4* increases these speed-up factors even more: it is in fact the best combination of GP-GPU tools for phase-vocoder-based processing in *Csound*. There is no reason not to choose this solution in real life applications. The significant gap between the performance of *instrument 2* and that of *instrument 3* suggests that what really makes the difference in the performance boost gained when employing *CUDA* is actually the introduction of the GPU-based phase vocoder analysis and re-synthesis stages provided by `cudanal` and `cudasynth` (or, even better, `cudanal2` and `cudasynth2`). Still, the results obtained by *instrument 4* show that *CUDA*-based spectral processing can provide for an extra performance boost. Also, these results show how the whole phase vocoder chain (analysis, processing and re-synthesis) can be cast to the GPU as a single block, avoiding extra memory transfers, which are very expensive in terms of latency. It is safe to speculate that if we had considered longer spectral processing chains, *intrument 4* would have performed even better if compared to *intrument 3*. This is because *instrument 3* needs to bounce data between host and device at every single processing step, while the *device-only* version only needs two memory transfers in total.

Table 4.5: A recap of the results presented in section 4.3: GPU vs CPU speed-up factors when considering a DFT size of 4096 points and varying hop size.

Module	Lowest speed-up factor	Highest speed-up factor
Gain	1.80	3.30
Filter	2.00	3.39
Stencil	1.80	3.29
Scale (0)	1.74	3.08
Scale (1)	1.28	1.61
Shift (0)	1.71	3.01
Shift (1)	1.24	1.58
Smooth	1.82	3.32
Blur	2.44	15.10
Mix	2.12	3.42
Morph	2.12	3.38

Considering *instrument 4* in particular, the performance of *CUDA*-based spectral processing *Csound instruments* ranges from being more than six times faster[24] than the original CPU-based *opcodes* to being slightly slower, depending on the employed DFT size and hop size. The only cases for which there is a loss of execution speed are typically those based on a 1024-points-wide spectral representation, which turns

[24]In the case of the *blur* module: much more than six times faster (see table 4.4).

out to be too narrow, on average[25], to actually exploit the parallel computation horsepower of modern GPUs. Nevertheless, the performance degradation experienced with narrow spectral frames is usually a minor one: even in these cases it does make sense to use the GP-GPU version of these algorithms so to free up computational resources for other tasks on the host side. The user might not get a direct benefit from the computations related to the specific GP-GPU module, but an overall performance gain can be experienced in more complex and involved applications that require a higher utilisation level of the CPU[26].

Overall, these results validate that the use of *CUDA*-based *plugin opcodes* for spectral manipulations is indeed very attractive for *Csound* users[27]. From a wider perspective, they are also very promising in showing that the whole audio spectral processing framework can be successfully ported to parallel computing architectures.

[25]The *blur* module is an exception to this: it actually gains in execution speed even when the DFT size is set to 1024 points.

[26]Also note that 1024-points-based spectral processing is in general less problematic than manipulations based on wider spectral frames. Thus, it is less needy for a performance improvement.

[27]Of course, they will need to own *CUDA*-enabled NVIDIA GPUs.

Conclusions

This thesis provides eloquent information about the practice of harnessing graphics processors in the wide scope of audio programming and computing. A particular focus is set on the investigation of the GP-GPU computing model in relation to a set of widespread spectral manipulation processes in a *Csound* environment.

Audio Computing in the GP-GPU Framework

From this thesis, it can be learned that using GPUs for audio computing can have different outcomes in terms of performance, depending on the specific process under analysis and its congeniality to the massively parallel scheme implied by any modern GPU architecture. Research studies have been carried out mainly in those areas of audio computing involving processes that are particularly well suited for a parallel implementation. In these areas the GP-GPU computing framework often proves to be much more efficient than normal CPU-based execution, even when the SSE model is exploited on multi-core CPUs. Speed-up factors[28] in the order of a few dozens have been recorded in particularly favourable situations.

A key aspect in the field of GP-GPU computing is in fact the need for operations to fit the GPU architecture well, meaning that, in order to be good candidates for being cast to the GPU, they should be suitable for the SIMD computation model, with a null or minimal level of data interdependency. More specifically, the best results are achieved when a *massively parallel* SIMD model can be applied, meaning that the data involved by one single instruction is at least in the order of hundreds or, better, thousands of elements. This way, the huge amount of processing cores featured on modern GPUs can be fully utilised.
Some simple algorithms are immediate candidates for a massively parallel scheme but often there is not a sharp separation line between processes that are congenial to the parallel scheme and those that are not. This is particularly true for more involved algorithms that can be conceptually split in multiple steps: some steps could be easily ported to the parallel scheme, while others might not. Some studies have been carried out also focusing on operations characterised by a sub-optimal level of potential parallelism or on operations which cannot be easily expressed in an SIMD way. In some cases the aim is that of finding alternative computational schemes for these operations in order to exploit a massively parallel architecture nonetheless (the most obvious example is represented by recursive operations, like

[28]In favour of the GP-GPU model against the CPU-only scheme.

the computation of IIR filters[29]). In other cases, the aim is that of testing the performance of these sub-optimal parallel algorithms[30] on the GPU in order to assess the degree of the hypothetical loss (if any). Indeed, another key aspect of GP-GPU computing is represented by the possibility of relieving the CPU from excessive computational load when GPU resources are available, thus allowing more tasks to be carried out concurrently. This can be very appealing, especially in general-purpose, highly multi-tasking systems that could possibly experience situations of CPU overload while GPU resources are mostly idle. This is often the case, indeed, in the scenario of pure audio processing[31] on general-purpose devices such as computers, tablets and smartphones. In this scenario, it is worth considering the GP-GPU computing approach even for sub-optimal algorithms, provided that they do not slow down the execution too much. As a matter of fact, if the GPU-based performance of a specific process is comparable with the one achieved with a standard CPU-based approach, it could be worth exploiting the heterogeneous computing solution for resource economy reasons. Under this perspective, highly efficient GP-GPU processes are even more valuable: not only they can significantly improve the execution speed of computationally demanding algorithms, but they also free up CPU resources and they allow more tasks to be run simultaneously.

The scope of audio computing can be generally divided into two broad categories: off-line/batch processing and real-time processing. While the batch framework is particularly congenial to GPU-based implementations, when a GP-GPU computing approach to real-time applications is considered, a few implications must be faced. The real-time context is in fact one of the key aspects that needs to be considered in the scope of *High-Performance Audio Computing* (HiPAC): unlike standard *High Performance Computing* (HPC), HiPAC demands exceptional awareness of latency and the many demands of real-time operation (see [68], [2] and [3]). The issue with this context is related to the very nature of the *throughput-oriented* computation model on which modern GPUs are based: in order to be effective, a massively parallel GP-GPU algorithm needs to involve a large amount of data, which is typically available in batch applications but not in low-latency real-time situations, no matter the nature of the process (being it for sound analysis, manipulation or synthesis). As a matter of fact, real-time applications need to split their operation in successive processing steps that are meant to deal with limited portions of audio signals, hence a limited amount of information is involved in each step and only a limited number of processing elements can be considered. This is of course in contrast with the ideal utilisation scheme of modern GPUs, and a trade-off between low-latency[32] and high-throughput GPU-operating implementations is met. Nevertheless, even though

[29]See section 2.8.

[30]Sometimes the number of elements involved by one instruction is very limited or perhaps the algorithm itself needs a substantial degree of barrier synchronisation due to the presence of data interdependency.

[31]In contrast, this might not be the case for applications that also involve a high degree of graphics or video processing and rendering. GPU-based audio processing and synthesis in the scope of videogams, for example, needs some extra care in terms of the availability of GPU resources.

[32]Low-latency applications imply the use of small chunks of data per processing step.

real-time applications in the GP-GPU computing framework are not ideal, this thesis provides many examples[33] of such a scheme being successfully applied, and in many cases being superior, in terms of performance, to a standard CPU-based implementation, even in low-latency applications.

Another key concept that emerges from this study about audio-related GP-GPU computing is the impact of the intrinsic time complexity of a specific algorithm on the observed efficiency of a GPU-based implementation. Quite intuitively, if the computation of one single operation on multiple data (in an SIMD fashion) brings some kind of benefit to the processing speed, this improvement is expected to build up significantly when many consecutive operations are needed in a more complex algorithm. As a result, computation-bound algorithms are usually more likely to benefit from the GP-GPU model, as opposed to memory bandwidth-bound algorithms, which, on the GPU, tend to suffer from the presence of a more articulated memory structure[34].
This concept gets even more decisive when considering the well-known, main limitation of *heterogeneous computing*: data transfer latency between host and device. Those algorithms characterised by a low level of computational complexity on each data element might not experience a sufficient improvement build-up, making the latency caused by memory transfers prevail on the speed-up obtained by the parallel computation scheme, hence undermining the potential benefits of this approach. This might happen even though a particular algorithm is theoretically well suited for parallel processing but, of course, the more data is involved by SIMD instructions, the less critical this problem gets, as memory transfer latency is eventually hidden by a high computation throughput, anyway. Conversely, those algorithms characterised by a sufficiently high level of computational complexity on each data element might succeed in hiding memory transfer latency even when they do not involve a huge amount of data in a parallel fashion.

Knowing that memory transfers are very expensive in terms of latency and, in general, in terms of performance, it is very important to minimise the total number of these. It is worth noting that, in the audio computing scope, some applications do not need data to be sent to the device, as, in some cases, it could be generated on the GPU directly. In this case, the generated data could be either needed by the host for further use or it could be ready for playback. In the former scenario, at least one memory displacement is inevitably needed (from device to host), while in the latter scenario, sending out the audio signal to the HDMI port of the graphics card might be an option, hence calling off the need for any memory transfer in either direction[35]. In the majority of cases, however, data originates at host side and needs to me moved to the device for manipulation purposes and then back to host (for recording purposes, for instance). This is indeed the case for what concerns

[33]Many examples can be found in chapter 2. Still, also the empirical part of this work is based on the assessment of real-time audio processes: in chapter 4 a performance improvement of GPU-based spectral processing algorithms over CPU-based implementations of the same processes is proven.

[34]However, the relatively recent introduction of cache memory on modern GPUs and the possibility of exploiting shared memory (much faster than global memory) significantly reduce this issue.

[35]It has to be said that this is actually a very unlikely and rare scenario.

all the applications addressed in the empirical part of this project (described in chapter 3), and, in general, for what concerns the area of digital audio effects.

In any case, care must be taken so that the implementation of the desired algorithm does not involve more memory transfers than actually needed. As already mentioned, each memory transfer between host and device is very expensive in terms of latency and strongly limits the potential benefits of a GP-GPU approach, especially in a real-time scenario. In most cases it is possible to reduce the number of transfers to only two. For instance, all the spectral processing applications that have been developed in this work could be implemented in such a way that only two memory transfers are needed. Also, the processing chain can be extended at will without increasing the number of memory transfers beyond two. The longer the processing chain within any two displacements, the finer the utilisation of GPU resources (as discussed above), the better the performance improvement that can be expected over the sequential model.

Spectral Processing of Audio in the GP-GPU Framework

In the particular case of phase vocoder processing, it has been shown that merging the analysis stage with the manipulation and re-synthesis stages in a compact GPU-based processing chain brings a remarkable improvement over a more naive implementation that needs to move data back and forth at each processing step (as well as for the analysis and re-synthesis stages). With the *compact processing chain* solution developed in this project, spectral processing in a *Csound* environment becomes a much lighter task for the CPU, and it can also be computed potentially faster (sometimes much faster) than it would be if the original, CPU-based *unit generators* were employed (at least this is what was measured on the two computer systems used in this project).

It has been shown that performance varies quite significantly depending on the spectral resolution considered for the target manipulations and on the rate at which the processing steps are carried out (i.e., on the analysis hop size). Interestingly, the higher the desired frequency resolution, the better the performance of the GPU if compared to the original CPU-based implementation. This is a very significant result: higher frequency resolution applications are in fact characterised by higher computational complexity and, thus, by higher execution times on a standard *latency-orineted* CPU. With a GP-GPU approach, however, execution time is cut more severely exactly in the more critical situations. Not to mention, a higher frequency resolution often translates in better quality audio effects. In the computer systems employed for this project, the separation line between performance gain and loss was found to sit amidst 1024-channels-wide and 2048-channels-wide phase vocoder frames: on average, using a DFT size of 1024 points (or lower) for GPU-based spectral manipulation processes did not provide any gain in performance[36], while using a DFT size of 2048 points or higher provided for an ever growing speed-up over CPU-based execution. Nevertheless, the performance degradation experienced with narrow spectral frames is usually a minor one: even in these cases

[36]The *blur* module is an exception to this: it actually gains in execution speed even when the DFT size is set to 1024 points.

it does make sense to use the GP-GPU version of these algorithms so to free up computational resources for other tasks on the host side. The user might not get a direct benefit from the computations related to the specific GP-GPU module, but an overall performance gain can be experienced in more complex and involved applications that require a higher utilisation level of the CPU.

For what concerns the rate at which the processing steps are carried out (i.e., the analysis hop size), this parameter influences the performance of GPU-based phase vocoder processing applications in a very predictable way: if processing a single frame on the GPU brings about some level of speed-up over CPU execution, then the repetition of this process at a high rate (i.e., low hop-size) has the effect of magnifying the beneficial effect.

Actually, the performance of GPU-operating spectral processing algorithms does not depend only on parameters such as DFT size and hop size, but it also depends on the very nature of the specific algorithm under analysis. Among the nine *Csound* modules for spectral manipulation addressed in this project, it was possible to identify three main classes, depending on their performance in a parallel computing framework. On the target systems[37], the majority of the tested algorithms showed an intermediate behaviour characterised by a performance which is, on average, roughly 3 times faster (in somewhat standard conditions) than that obtained by the original CPU-based implementation (see chapter 4 for more details). A second class of algorithms was found to perform a little worse, with speed-ups around 1.5, in standard conditions. One module in particular (i.e., the application of spectral flux blurring[38]) was instead found to be particularly well suited for parallel processing on GPUs and it showed speed-up factors between 3 and 15 when an intermediate DFT size of 4096 was employed (varying significantly depending on the hop size setting). The same module was found to perform almost 24 times faster than its CPU-based counterpart in particularly favourable conditions, namely a high DFT size of 16384 points and a low hop size of 128 samples.

To summarise, the results obtained in this thesis not only validate the possibility of using mid-range dedicated GPUs (instead of the CPU) for real-time streaming phase vocoder processing in the *Csound* environment, but also prove that a substantial (sometimes outstanding) performance gain can be obtained in most situations, on the target systems. This means that more and better-sounding digital audio effects can be applied in real-time on these systems and on similar computers. These results also suggest that phase vocoder processing applications are good candidates for parallel computing to the greatest extent, not only on GPUs but also on other kinds of parallel processors characterised by a wide vector architecture, which have increasingly gained popularity in the last few years (see for instance AMD's APUs and NVIDIA Tegra systems-on-a-chip). Thus, massively parallel spectral audio processing should not be confined to dedicated hardware and desktop computers but it could also be efficiently carried out on other platforms such as gaming consoles, notebooks, tablets, smartphones and systems-on-a-chip.

[37]The system used for the tests were based on a quite old CPU (released in 2009) flanked by a quite recent, mid-range dedicated GPU. See subsection 4.1.1 for detailed specifications.

[38]Specifically, the spectral flux blurring effect is achieved by means of applying a moving average on the time evolution functions of each phase vocoder channel (both amplitude and frequency components).

Future Work

A few directions for future work can be outlined. First of all, in addition to the nine modules considered in this project, many more streaming spectral processing *Csound unit generators* already exist, and it would be proper to transpose these as well to the massively parallel computing model, as they are all likely to benefit form it. Note that the GPU-based *scale* and *shift* modules are not really complete yet, as they lack a verified implementation for their alternative mode of operation which uses the *true envelope* estimation algorithm for formant conservation (see appendix A).
All phase-vocoder-based *unit generators* in *Csound* actually feature a *sliding mode* that can be employed for enhanced quality and reduced latency. As discussed in subsection 3.5.3, a GPU-based *sliding mode* was not implemented in this project for any of the addressed modules, even though it is expected to be a very good candidate for the parallel computing model. This needs to be investigated further. When the previously outlined shortcomings will be fixed, it would be nice to include the resulting source code in a future *Csound* official release. For what concerns the *CUDA* implementation, as it was already discussed in section 4.5, the developed applications might need some level of code optimisation. It would be interesting to implement and test the optimisation techniques outlined in section 4.5, in order to see if further improvement can be achieved.

A huge limitation of this whole project is the fact that the developed applications are only available to users that own NVIDIA GPUs, specifically those NVIDIA GPUs that are *CUDA*-enabled[39]. In order to make these applications more general and widely applicable on a broader set of devices, the *OpenCL*[40] standard must be considered. In fact, *OpenCL* is an open, royalty-free, and, most importantly, cross-platform standard for parallel programming of diverse processors found in personal computers, servers, mobile devices and embedded platforms. Note that, if an *OpenCL* version of the GPU-based *Csound* modules was desired, a complete rework of the actual implementations would be needed. In this case, it would be interesting to compare the performance results obtained by *CUDA*-based and *OpenCL*-based implementations of the same algorithm, running on the same NVIDIA device and/or on comparable GPUs from NVIDIA and AMD.

Finally, a further investigation of an even more massively parallel computing scheme applied to spectral processing could be carried out by means of running the addressed applications on multiple GPUs, simultaneously, in the same system. This is a parallel computing architecture that is quickly and easily constructed from readily available hardware, and it is well supported by both *CUDA* and *OpenCL*.

[39] All recent releases by NVIDIA are actually compatible with *CUDA* functionalities.
[40] *OpenCL*: Open Computing Language

Appendix A

True Envelope Estimation Algorithm for Formant Conservation in *scale* and *shift* modules (*Mode 2*)

This appendix is thought as a continuation to 3.6.4 and 3.6.5 subsections. It was included in this work in order to report a possible *CUDA*-based implementation for *mode 2* in the *scale* and *shift* modules. In fact, for the development of these two *plugin opcodes*, *mode 2* was not addressed in chapter 3. The reason for this is that *mode 2* could not be tested due to a discrepancy between the *CUDA* versions and the original CPU-only version, which caused them to run a different number of iterations with a given input signal and a given number of cepstral coefficients to be preserved. Unfortunately there was not enough time to properly analyse and fix this problem, but the developed code is not buggy *per se* and it returns the expected results from a perceptual point of view.

The *true envelope estimation* technique ([64] and [65]) is more accurate than simple *cepstrum liftering* (i.e., it produces smoother envelopes), but it is also computationally more demanding.
This method is again based on the concept of cepstral smoothing and consists of an iterative process that progressively adjusts successive estimations of the spectral envelope, until a particular exit condition is met. The algorithm iteratively updates the smoothed input spectrum $A_{i[k]}$ with the maximum between the original spectrum and the current cepstral representation:

$$A_{i[k]} = max(\log(|X_{[k]}|), C_{i[k]}) \tag{A.1}$$

and applies the cepstral smoothing to $A_{i[k]}$ to obtain $C_{i[k]}$, in the exact same way as it is carried out in *mode 1*. The algorithm stops if for all k the relation

$$A_{i[k]} < C_{i[k]} + \vartheta \tag{A.2}$$

is true with ϑ being a user supplied threshold.

In the performance function of `cudapvscale` and `cudapvscale2` this whole process is implemented as follows:

```
else if (keepform==2) {

  int cond = 1;
  if (coefs<1) coefs = 80;
  cufftEnv = (cufftComplex*) p->deviceEnv;
  cufftCepstrum = (cufftComplex*) p->deviceCepstrum;
  cufftTrueEnv = (cufftComplex*) p->deviceTrueEnv;
  cufftSmoothTrueEnv = (cufftComplex*) p->deviceSmoothTrueEnv;
  thrust::device_ptr<float> dev_ptr1 = thrust::device_pointer_cast(fout);
  thrust::fill(dev_ptr1, dev_ptr1+framelength, -1.0f);

  takeLog<<<p->gridSize,p->blockSize>>>(fin, p->deviceEnv, Nhalf);

  // loop initialization stage: smooth the original log spectral envelope
  // first, take the fft of the log of the spectral envelope...
  if (cufftExecR2C(p->forwardPlan,(cufftReal*)cufftEnv,cufftCepstrum)!=
    CUFFT_SUCCESS)
    csound->Message(csound, "CUDA FFT error\n");
  if (cudaDeviceSynchronize() != cudaSuccess)
    csound->Message(csound,"CUDA error: Failed to synchronize\n");

  // liftering stage
  lifter<<<p->gridSize,p->blockSize>>>(p->deviceCepstrum, coefs, Nhalf);

  // take the inverse fft of the liftered cepstrum...
  if (cufftExecC2R(p->inversePlan,cufftCepstrum,
    (cufftReal*)cufftSmoothTrueEnv)!= CUFFT_SUCCESS)
    csound->Message(csound, "CUDA FFT error\n");
  if (cudaDeviceSynchronize() != cudaSuccess)
    csound->Message(csound,"CUDA error: Failed to synchronize\n");

  while (cond) {

    // update the smoothed input spectrum:
    update<<<p->gridSize,p->blockSize>>>(p->deviceEnv, p->deviceTrueEnv,
                                         p->deviceSmoothTrueEnv, Nhalf);

    // take the fft of the true envelope...
    if (cufftExecR2C(p->forwardPlan,(cufftReal*)cufftTrueEnv,
      cufftCepstrum)!= CUFFT_SUCCESS)
      csound->Message(csound, "CUDA FFT error\n");
    if (cudaDeviceSynchronize() != cudaSuccess)
      csound->Message(csound,"CUDA error: Failed to synchronize\n");

    // liftering stage
    lifter<<<p->gridSize,p->blockSize>>>(p->deviceCepstrum, coefs, Nhalf);

    // take the inverse fft of the liftered cepstrum...
    if (cufftExecC2R(p->inversePlan,cufftCepstrum,
      (cufftReal*)cufftSmoothTrueEnv)!= CUFFT_SUCCESS)
      csound->Message(csound, "CUDA FFT error\n");
    if (cudaDeviceSynchronize() != cudaSuccess)
      csound->Message(csound,"CUDA error: Failed to synchronize\n");

    // test the exit condition:
    test<<<p->gridSize,p->blockSize>>>(p->deviceTrueEnv,
                                       p->deviceSmoothTrueEnv,
                                       p->deviceMask, Nhalf);
    thrust::device_ptr<int> dev_ptr4 =
       thrust::device_pointer_cast(p->deviceMask);
    if ((thrust::reduce(dev_ptr4, dev_ptr4+Nhalf)) == 0)
      cond = 0;
  }

  expon<<<p->gridSize,p->blockSize>>>(p->deviceSmoothTrueEnv, Nhalf);

  thrust::device_ptr<float> dev_ptr3 =
    thrust::device_pointer_cast(p->deviceSmoothTrueEnv);
  max = *(thrust::max_element(dev_ptr3, dev_ptr3+Nhalf));

  freqScaleFormant<<<p->gridSize,p->blockSize>>>(fin, fout,
                                                p->deviceSmoothTrueEnv,
                                                pscal, max, Nhalf);
  fixPVandGain<<<p->gridSize,p->blockSize>>>(fin, fout, g, framelength);
}
```

The framework here is very similar to that on which *mode 1* is based, the only difference being the presence of a **while** loop in which spectral smoothing is iterated and the spectral envelope is updated by means of the `update()` kernel:

```
__global__ void
update(float* original, float* newTE, float* current, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  if (i < nhalf) {
    current[i] /= nhalf;
    newTE[i] = (original[i] < current[i]) ? current[i] : original[i];
  }
}
```

Finally, the exit condition A.2 is tested by means of the `test()` kernel:

```
__global__ void
test(float* nonSmoothed, float* smoothed, int* mask, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int diff;
  if (i < nhalf) {
    diff = fabs(nonSmoothed[i] - smoothed[i]/nhalf);
    mask[i] = (diff > 0.23) ? 1 : 0;  // here 0.23 is the threshold
  }
}
```

Here each thread fills the corresponding element of a boolean mask with 1 or 0 depending on the result of comparing the difference between smoothed and non-smoothed envelope values to the threshold (equation A.2). After the boolean mask is built, all its elements are summed (using `thrust::reduce()`) and the result of this summation is compared with zero in order to assess the exit condition[1].
The kernel for the actual frequency scaling and the one for completing the output phase vocoder frame are not different from those employed in *mode 1*.
Of course, `cudapvshift` and `cudapvshift2` are based on the very same structure. The only difference is in the kernel used for frequency shifting, instead of frequency scaling: `freqScaleFormant()` needs to be swapped with `freqShiftFormant()` (see section 3.6.5).

[1]There is definitely room for improvement in the way the whole process of checking the exit condition is implemented.

Appendix B

Plugin Opcodes: *CUDA C* Scripts

In this appendix, the nine *CUDA C* shared objects for streaming phase vocoder processing that were presented and described in chapter 3 are reported in their entirety. They can be easily compared by the reader to the original *C/C++* versions, which are included in the *Csound* source code[1]. For each module, both the *host memory input-out* version and the *device-only* version are reported for completeness.

Common Functions

A few common functions are employed in many *plugin opcodes*. These are all reported in this section for convenience.

`handleCudaError()`

This function is used to inform the user of a device memory management error by printing an error message to the screen:

```
static void handleCudaError(CSOUND *csound, cudaError_t error) {
  if (error!= cudaSuccess) {
    csound->Message(csound, "%s in %s at line %d\n",
                    cudaGetErrorString(error),
                    __FILE__, __LINE__);
    return csound->InitError(csound, "Oops, something went wrong"
                             "while managing the GPU memory \n");
  }
}
```

`AuxCudaAlloc()`

This function is used to allocate memory in the device and set it to zero:

```
static void AuxCudaAlloc(int size, AUXCH *p){
  float *mem;
  cudaMalloc(&mem, size);
  cudaMemset(mem, 0, size);
  p->auxp = mem;
  p->size = size;
}
```

[1] *Csound*'s source code is available at `http://csound.github.io`

Gain Module

cudapvsgain

```
#include <csdl.h>
#include <pstream.h>

typedef struct _cudapvsgain {
  OPDS    h;
  PVSDAT  *fout;
  PVSDAT  *fa;
  MYFLT   *kgain;
  float* deviceFrame;   // pointer to device memory
  int gridSize;   // number of blocks in the grid (1D)
  int blockSize;   // number of threads in one block (1D)
  uint32  lastframe;
} CUDAPVSGAIN;

// kernel for scaling PV amplitudes
__global__
void devicepvsgain (float* deviceFrame, MYFLT gain, int framesize) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if(i < framesize>>1)
    deviceFrame[i<<1] *= gain;
}

static int free_device(CSOUND* csound, void* pp){
  CUDAPVSGAIN* p = (CUDAPVSGAIN*) pp;
  cudaFree(p->deviceFrame);
  return OK;
}

static int cudapvsgainset(CSOUND *csound, CUDAPVSGAIN *p) {

  int32 N = p->fa->N;
  int size = (N+2) * sizeof(float);
  int maxBlockDim;
  int SMcount;
  int totNumThreads = (N+2)/2;
  cudaError_t error;

  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound, "cudapvsgain running on device %s
                  (capability %d.%d)\n",
                  deviceProp.name, deviceProp.major,
                  deviceProp.minor);

  error = cudaMalloc(&p->deviceFrame, size);
  handleCudaError(csound, error);

  p->fout->sliding = 0;

  if (p->fout->frame.auxp == NULL ||
      p->fout->frame.size < sizeof(float) * (N + 2))
    csound->AuxAlloc(csound, (N + 2) * sizeof(float), &p->fout->frame);

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N = N;
  p->fout->overlap = p->fa->overlap;
  p->fout->winsize = p->fa->winsize;
  p->fout->wintype = p->fa->wintype;
  p->fout->format = p->fa->format;
  p->fout->framecount = 1;
  p->lastframe = 0;
  if (UNLIKELY(!(p->fout->format == PVS_AMP_FREQ) ||
      (p->fout->format == PVS_AMP_PHASE)))
    return csound->InitError(csound, Str("cudapvsgain: signal format "
                             "must be amp-phase or amp-freq."));
  csound->RegisterDeinitCallback(csound, p, free_device);
  return OK;
}

static int cudapvsgain(CSOUND *csound, CUDAPVSGAIN *p)
{
  int32   framelength = p->fa->N + 2;
  int size = (int) framelength * sizeof(float);
  MYFLT gain = *p->kgain;

  if (p->lastframe < p->fa->framecount) {
    cudaMemcpy(p->deviceFrame, p->fa->frame.auxp, size,
               cudaMemcpyHostToDevice);
    devicepvsgain<<<p->gridSize,p->blockSize>>>(p->deviceFrame, gain,
                                                framelength);
    cudaMemcpy(p->fout->frame.auxp, p->deviceFrame, size,
               cudaMemcpyDeviceToHost);
    p->fout->framecount = p->fa->framecount;
    p->lastframe = p->fout->framecount;
  }

  return OK;
}

static OENTRY localops[] = {
  {"cudapvsgain", sizeof(CUDAPVSGAIN), 0, 3, "f", "fk",
                  (SUBR) cudapvsgainset, (SUBR) cudapvsgain, NULL}
};

extern "C" {
  LINKAGE
}
```

cudapvsgain2

```
#include <csdl.h>
#include <pstream.h>

typedef struct _cudapvsgain2 {
  OPDS    h;
  PVSDAT  *fout;
  PVSDAT  *fa;
  MYFLT   *kgain;
  int gridSize;   // number of blocks in the grid (1D)
  int blockSize;   // number of threads in one block (1D)
  uint32  lastframe;
} CUDAPVSGAIN2;
```

```
// kernel for scaling PV amplitudes
__global__
void applygain(float* output, float* input, MYFLT g, int length) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int j = i<<1;
  if(j < length) {}
    output[j] = (float) input[j] * g;
  output[j+1] = input[j+1];
  }
}

static int free_device(CSOUND* csound, void* pp){
  CUDAPVSGAIN2* p = (CUDAPVSGAIN2*) pp;
  cudaFree(p->fout->frame.auxp);
  return OK;
}

static int cudapvsgain2set(CSOUND *csound, CUDAPVSGAIN2 *p){

  int32 N = p->fa->N;
  int size = (N+2) * sizeof(float);
  int maxBlockDim;
  int SMcount;
  int totNumThreads = (N+2)/2;

  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound,
                "cudapvsgain2 running on device %s (capability %d.%d)\n"
    ,
                deviceProp.name, deviceProp.major,
                deviceProp.minor);

  p->fout->sliding = 0;

  if (p->fout->frame.auxp == NULL || p->fout->frame.size < size)
  AuxCudaAlloc(size, &p->fout->frame);

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N = N;
  p->fout->overlap = p->fa->overlap;
  p->fout->winsize = p->fa->winsize;
  p->fout->wintype = p->fa->wintype;
  p->fout->format = p->fa->format;
  p->fout->framecount = 1;
  p->lastframe = 0;

  csound->RegisterDeinitCallback(csound, p, free_device);
  return OK;
}
```

```
static int cudapvsgain2(CSOUND *csound, CUDAPVSGAIN2 *p)
{
  int32   framelength = p->fa->N + 2;
  MYFLT gain = *p->kgain;
  float* fo = (float*) p->fout->frame.auxp;
  float* fi = (float*) p->fa->frame.auxp;

  if (p->lastframe < p->fa->framecount) {
    applygain<<<p->gridSize,p->blockSize>>>(fo, fi, gain, framelength);
    p->fout->framecount = p->fa->framecount;
    p->lastframe = p->fout->framecount;
  }

  return OK;
}

static OENTRY localops[] = {
  {"cudapvsgain2", sizeof(CUDAPVSGAIN2), 0, 3, "f", "fk",
    (SUBR) cudapvsgain2set, (SUBR) cudapvsgain2, NULL}
};

extern "C" {
  LINKAGE
}
```

Filter Module

cudapvsfilter

```
#include <csdl.h>
#include <pstream.h>

typedef struct _cudapvsfilter {
  OPDS    h;
  PVSDAT  *fout;
  PVSDAT  *fin;
  PVSDAT  *fmask;
  MYFLT   *kdepth;
  MYFLT   *igain;
  float*  deviceInput;   // pointer to device memory (input frame)
  float*  deviceOutput;   // pointer to device memory (output frame)
  float*  deviceMask;   // pointer to device memory (mask frame)
  int  gridSize;   // number of blocks in the grid (1D)
  int blockSize;   // number of threads in one block (1D)
  uint32  lastframe;
} CUDAPVSFILTER;

// kernel for filtering in the frequency domain, given a spectral mask
__global__
void filter(float* input, float* output, float* mask, MYFLT wet,
            MYFLT dry, float g, int length) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  if (j < length) {
    output[j] = (float) (input[j]*(dry+mask[j]*wet))*g;
    output[j+1] = input[j+1];
  }
}

static int fsigs_equal(const PVSDAT *f1, const PVSDAT *f2)
{
  if (
    (f1->sliding == f2->sliding) &&
    (f1->overlap == f2->overlap) &&
    (f1->winsize == f2->winsize) &&
    (f1->wintype == f2->wintype) &&     /* harsh, maybe... */
    (f1->N == f2->N) &&
    (f1->format == f2->format)
  ) return 1;
  return 0;
}

static int free_device(CSOUND* csound, void* pp) {
  CUDAPVSFILTER* p = (CUDAPVSFILTER*) pp;
  cudaFree(p->deviceInput);
  cudaFree(p->deviceOutput);
  cudaFree(p->deviceMask);
  return OK;
}
```

```
static int cudapvsfilterset(CSOUND *csound, CUDAPVSFILTER *p)
{
  int N = p->fin->N;
  int size = (N+2) * sizeof(float);
  int maxBlockDim;
  int SMcount;
  int totNumThreads = (N+2)>>1;
  cudaError_t error;

  // get info about device
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound, "cudapvsfilter running on device %s
                  (capability %d.%d)\n", deviceProp.name,
                  deviceProp.major, deviceProp.minor);

  // device memory allocation (input data)
  error = cudaMalloc(&p->deviceInput, size);
  handleCudaError(csound, error);

  // device memory allocation (input mask)
  error = cudaMalloc(&p->deviceMask, size);
  handleCudaError(csound, error);

  // device memory allocation and set to zero (output data to be computed)
  error = cudaMalloc(&p->deviceOutput, size);
  handleCudaError(csound, error);
  cudaMemset(p->deviceOutput,0,size);

  if (UNLIKELY(p->fin == p->fout || p->fmask == p->fout))
  csound->Warning(csound, Str("Unsafe to have same
                  fsig as in (or filter) and out"));
  if (UNLIKELY(!(p->fout->format == PVS_AMP_FREQ) ||
                (p->fout->format == PVS_AMP_PHASE)))
  return csound->InitError(csound, Str("cudapvsfilter: signal format "
                           "must be amp-phase or amp-freq."));
  p->fout->sliding = 0;

  if (p->fout->frame.auxp == NULL ||
      p->fout->frame.size < sizeof(float) * (N + 2))
    csound->AuxAlloc(csound, sizeof(float) * (N + 2), &p->fout->frame);

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N = N;
  p->fout->overlap = p->fin->overlap;
  p->fout->winsize = p->fin->winsize;
  p->fout->wintype = p->fin->wintype;
  p->fout->format = p->fin->format;
  p->fout->framecount = 1;
  *p->igain = 1.0;
  p->lastframe = 0;

  csound->RegisterDeinitCallback(csound, p, free_device);

  return OK;
}
```

```
static int cudapvsfilter(CSOUND *csound, CUDAPVSFILTER *p) {
  int N = p->fout->N;
  int framelength = N+2;
  int size = framelength * sizeof(float);
  float* fout = (float*) p->fout->frame.auxp;
  float* fin = (float*) p->fin->frame.auxp;
  float* fmask = (float*) p->fmask->frame.auxp;
  MYFLT depth = *p->kdepth;
  float gain = (float) *p->igain;
  MYFLT dirgain;

  if (UNLIKELY(fout == NULL)) goto err1;
  if (UNLIKELY(!fsigs_equal(p->fin,p->fmask))) goto err2;

  if (p->lastframe < p->fin->framecount) {

    cudaMemcpy(p->deviceInput,fin,size,cudaMemcpyHostToDevice);
    cudaMemcpy(p->deviceMask,fmask,size,cudaMemcpyHostToDevice);

    // clip depth between zero and one...
    depth = depth >= 0 ? (depth <= 1 ? depth : 1) : FL(0.0);
    dirgain = (1 - depth);

    // filtering on the GPU:
    filter<<<p->gridSize,p->blockSize>>>(p->deviceInput,
                                   p->deviceOutput, p->deviceMask,
   depth, dirgain, gain, framelength);

    cudaMemcpy(fout,p->deviceOutput,size,cudaMemcpyDeviceToHost);

    p->fout->framecount = p->lastframe = p->fin->framecount;
  }

  return OK;

  err1: return csound->PerfError(csound, p->h.insdshead,
                                 Str("cudapvsfilter: not initialised"));
  err2: return csound->PerfError(csound, p->h.insdshead,
                                 Str("cudapvsfilter: formats are
                                 different."));
}

static OENTRY localops[] = {
  {"cudapvsfilter", sizeof(CUDAPVSFILTER),0, 3, "f", "ffxp",
    (SUBR) cudapvsfilterset,(SUBR) cudapvsfilter},
};

extern "C" {
  LINKAGE
}
```

cudapvsfilter2

```
#include <csdl.h>
#include <pstream.h>

typedef struct _cudapvsfilter2 {
  OPDS    h;
  PVSDAT  *fout;
  PVSDAT  *fin;
  PVSDAT  *fmask;
  MYFLT   *kdepth;
  MYFLT   *igain;
  int  gridSize;   // number of blocks in the grid (1D)
  int blockSize;   // number of threads in one block (1D)
  uint32  lastframe;
} CUDAPVSFILTER2;

// kernel for filtering in the frequency domain, given a spectral mask
__global__
void filter(float* input, float* output, float* mask, MYFLT wet,
            MYFLT dry, float g, int length) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  if (j < length) {
    output[j] = (float) (input[j]*(dry+mask[j]*wet))*g;
    output[j+1] = input[j+1];
  }
}

static int fsigs_equal(const PVSDAT *f1, const PVSDAT *f2)
{
  if (
    (f1->sliding == f2->sliding) &&
    (f1->overlap == f2->overlap) &&
    (f1->winsize == f2->winsize) &&
    (f1->wintype == f2->wintype) &&     /* harsh, maybe... */
    (f1->N == f2->N) &&
    (f1->format == f2->format)
  )  return 1;
  return 0;
}

static int free_device(CSOUND* csound, void* pp){
  CUDAPVSFILTER2* p = (CUDAPVSFILTER2*) pp;
  cudaFree(p->fout->frame.auxp);
  return OK;
}

static int cudapvsfilter2set(CSOUND *csound, CUDAPVSFILTER2 *p)
{
  int N = p->fin->N;
  int size = (N+2) * sizeof(float);
  int maxBlockDim;
  int SMcount;
  int totNumThreads = (N+2)>>1;

  // get info about device
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound, "cudapvsfilter2 running on device %s
                  (capability %d.%d)\n", deviceProp.name,
                  deviceProp.major, deviceProp.minor);

  if (UNLIKELY(p->fin == p->fout || p->fmask == p->fout))
    csound->Warning(csound, Str("Unsafe to have same fsig as in
                                (or filter) and out"));

  p->fout->sliding = 0;

  if (p->fout->frame.auxp == NULL ||
      p->fout->frame.size < sizeof(float) * (N + 2))
    AuxCudaAlloc(size, &p->fout->frame);

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N = N;
  p->fout->overlap = p->fin->overlap;
  p->fout->winsize = p->fin->winsize;
  p->fout->wintype = p->fin->wintype;
  p->fout->format = p->fin->format;
  p->fout->framecount = 1;
  *p->igain = 1.0;
  p->lastframe = 0;

  csound->RegisterDeinitCallback(csound, p, free_device);

  return OK;
}

static int cudapvsfilter2(CSOUND *csound, CUDAPVSFILTER2 *p) {
  int N = p->fout->N;
  int framelength = N+2;
  float* fout = (float*) p->fout->frame.auxp;
  float* fin = (float*) p->fin->frame.auxp;
  float* fmask = (float*) p->fmask->frame.auxp;
  MYFLT depth = *p->kdepth;
  float gain = (float) *p->igain;
  MYFLT dirgain;

  if (UNLIKELY(fout == NULL)) goto err1;
  if (UNLIKELY(!fsigs_equal(p->fin,p->fmask))) goto err2;

  if (p->lastframe < p->fin->framecount) {

    // clip depth between zero and one...
    depth = depth >= 0 ? (depth <= 1 ? depth : 1) : FL(0.0);
    dirgain = (1 - depth);

    // filtering on the GPU:
    filter<<<p->gridSize,p->blockSize>>>(fin, fout, fmask,
                                         depth, dirgain, gain,
                                         framelength);

    p->fout->framecount = p->lastframe = p->fin->framecount;
  }

  return OK;
  err1: return csound->PerfError(csound, p->h.insdshead,
    Str("cudapvsfilter2: not initialised"));
  err2: return csound->PerfError(csound, p->h.insdshead,
    Str("cudapvsfilter2: formats are different."));
}

static OENTRY localops[] = {
  {"cudapvsfilter2", sizeof(CUDAPVSFILTER2),0, 3, "f", "ffxp",
  (SUBR) cudapvsfilter2set, (SUBR) cudapvsfilter2},
};

extern "C" {
  LINKAGE
}
```

Stencil Module

cudapvstencil

```
#include <csdl.h>
#include <pstream.h>
#include <thrust/replace.h>
#include <thrust/transform.h>
#include <thrust/device_ptr.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>

typedef struct _cudapvstencil {
  OPDS    h;
  PVSDAT  *fout;
  PVSDAT  *fin;
  MYFLT   *kgain;
  MYFLT   *klevel;
  MYFLT   *ifn;
  FUNC    *func;
  float*  devFrame;    // device memory pointer (input/output frame)
  MYFLT*  devStencil;  // device memory pointer
                       // (function table to use as stencil)
  int     gridSize;    // number of blocks in the grid (1D)
  int     blockSize;   // number of threads in one block (1D)
  uint32  lastframe;
} CUDAPVSTENCIL;

struct _is_less_than_zero {
  __host__ __device__
  bool operator()(float x) {return x < 0.0f;}
};

__global__
void stencilkernel(float* frame, MYFLT* stencil, float level,
                  float gain, int length) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int j = i<<1;
  if (i < length) {
    if (frame[j] < (((float) stencil[i])*level)) {
      frame[j] *= gain;
    }
  }
}

static int free_device(CSOUND* csound, void* pp) {
  CUDAPVSTENCIL* p = (CUDAPVSTENCIL*) pp;
  cudaFree(p->devFrame);
  cudaFree(p->devStencil);
  return OK;
}

static int cudapvstencilset(CSOUND *csound, CUDAPVSTENCIL *p)
{
  int32    N = p->fin->N;
  int framelength = N+2;
  int chans = framelength>>1;
  int size = framelength*sizeof(float);
  int stencilSize = chans*sizeof(MYFLT);
  int maxBlockDim;
  int SMcount;
  int totNumThreads = chans;
  cudaError_t error;

  // get info about device
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound, "cudapvstencil running on device %s
                  (capability %d.%d)\n", deviceProp.name,
                  deviceProp.major, deviceProp.minor);

  // device memory allocation (PV frame)
  error = cudaMalloc(&p->devFrame, size);
  handleCudaError(csound, error);

  // device memory allocation (stencil)
  error = cudaMalloc(&p->devStencil, stencilSize);
  handleCudaError(csound, error);

  p->fout->sliding = 0;

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N = N;
  p->fout->overlap = p->fin->overlap;
  p->fout->winsize = p->fin->winsize;
  p->fout->wintype = p->fin->wintype;
  p->fout->format = p->fin->format;
  p->fout->framecount = 1;
  p->lastframe = 0;

  p->fout->NB = chans;

  if (p->fout->frame.auxp == NULL ||
      p->fout->frame.size < sizeof(float) * (N + 2))
    csound->AuxAlloc(csound, (N + 2) * sizeof(float), &p->fout->frame);

  if (UNLIKELY(!(p->fout->format == PVS_AMP_FREQ) ||
                 (p->fout->format == PVS_AMP_PHASE)))
    return csound->InitError(csound, Str("cudapvstencil: signal format "
                                     "must be amp-phase or amp-freq."));

  p->func = csound->FTnp2Find(csound, p->ifn);
  if (p->func == NULL)
    return OK;

  if (UNLIKELY(p->func->flen + 1 < (unsigned int)chans))
    return csound->InitError(csound, Str("cudapvstencil: ftable needs"
                                     "to equal the number of bins"));

  cudaMemcpy(p->devStencil, p->func->ftable,
             stencilSize, cudaMemcpyHostToDevice);

  thrust::device_ptr<MYFLT> dev_ptr =
    thrust::device_pointer_cast(p->devStencil);

  _is_less_than_zero pred;
```

```
  thrust::replace_if(thrust::device, dev_ptr,
                     dev_ptr + chans, pred, (MYFLT) 0.0);

  cudaMemcpy(p->func->ftable, p->devStencil,
             stencilSize, cudaMemcpyDeviceToHost);

  csound->RegisterDeinitCallback(csound, p, free_device);

  return OK;
}

static int cudapvstencil(CSOUND *csound, CUDAPVSTENCIL *p)
{
  int framelength = p->fin->N + 2;
  int chans = framelength>>1;
  int size = framelength * sizeof(float);
  float* fout = (float *) p->fout->frame.auxp;
  float* fin = (float *) p->fin->frame.auxp;
  float   g = fabsf((float)*p->kgain);
  float   level = fabsf((float)*p->klevel);

  if (UNLIKELY(fout == NULL)) goto err1;

  if (p->lastframe < p->fin->framecount) {

    cudaMemcpy(p->devFrame, fin, size, cudaMemcpyHostToDevice);

    stencilkernel<<<p->gridSize,p->blockSize>>>(p->devFrame,
                                            p->devStencil, level,
                                            g, chans);

    cudaMemcpy(fout, p->devFrame, size, cudaMemcpyDeviceToHost);

    p->fout->framecount = p->lastframe = p->fin->framecount;
  }

  return OK;
  err1:
    return csound->PerfError(csound, p->h.insdshead,
                             Str("cudapvstencil: not initialised"));
}

static OENTRY localops[] = {
  {"cudapvstencil", sizeof(CUDAPVSTENCIL), TR, 3, "f", "fkki",
  (SUBR) cudapvstencilset, (SUBR) cudapvstencil}
};

extern "C" {
  LINKAGE
}
```

cudapvstencil2

```
#include <csdl.h>
#include <pstream.h>
#include <thrust/replace.h>
#include <thrust/transform.h>
#include <thrust/device_ptr.h>
#include <thrust/functional.h>
#include <thrust/execution_policy.h>
```

```
typedef struct _cudapvstencil2 {
  OPDS    h;
  PVSDAT  *fout;
  PVSDAT  *fin;
  MYFLT   *kgain;
  MYFLT   *klevel;
  MYFLT   *ifn;
  FUNC    *func;
  MYFLT*  devStencil;  // device memory pointer
                       // (function table to use as stencil)
  int     gridSize;    // number of blocks in the grid (1D)
  int     blockSize;   // number of threads in one block (1D)
  uint32  lastframe;
} CUDAPVSTENCIL2;

struct _is_less_than_zero {
  __host__ __device__
  bool operator()(float x) {return x < 0.0f;}
};

__global__
void stencilkernel(float* output, float* input, MYFLT* stencil,
                   float level, float gain, int length) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int j = i<<1;
  if (i < length) {
    if (input[j] < (((float) stencil[i])*level)) {
      output[j] = input[j] * gain;
      output[j+1] = input[j+1];
    }
    else {
      output[j] = input[j];
      output[j+1] = input[j+1];
    }
  }
}

static int free_device(CSOUND* csound, void* pp){
  CUDAPVSTENCIL2* p = (CUDAPVSTENCIL2*) pp;
  cudaFree(p->fout->frame.auxp);
  cudaFree(p->devStencil);
  return OK;
}

static int cudapvstencil2set(CSOUND *csound, CUDAPVSTENCIL2 *p)
{
  int32    N = p->fin->N;
  int framelength = N+2;
  int chans = framelength>>1;
  int size = framelength*sizeof(float);
  int stencilSize = chans*sizeof(MYFLT);

  int maxBlockDim;
  int SMcount;
  int totNumThreads = chans;
  cudaError_t error;

  // get info about device
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound, "cudapvstencil2 running on device %s
                  (capability %d.%d)\n", deviceProp.name,
                  deviceProp.major, deviceProp.minor);

  // device memory allocation (stencil)
  error = cudaMalloc(&p->devStencil, stencilSize);
  handleCudaError(csound, error);

  p->fout->sliding = 0;

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N = N;
  p->fout->overlap = p->fin->overlap;
  p->fout->winsize = p->fin->winsize;
  p->fout->wintype = p->fin->wintype;
  p->fout->format = p->fin->format;
  p->fout->framecount = 1;
  p->lastframe = 0;

  p->fout->NB = chans;

  if (p->fout->frame.auxp == NULL ||
      p->fout->frame.size < sizeof(float) * (N + 2))
    AuxCudaAlloc(size, &p->fout->frame);

  p->func = csound->FTnp2Find(csound, p->ifn);
  if (p->func == NULL)
    return OK;

  if (UNLIKELY(p->func->flen + 1 < (unsigned int)chans))
    return csound->InitError(csound, Str("cudapvstencil2:
                                      ftable needs to equal "
                                      "the number of bins"));

  cudaMemcpy(p->devStencil, p->func->ftable,
             stencilSize, cudaMemcpyHostToDevice);

  thrust::device_ptr<MYFLT> dev_ptr =
    thrust::device_pointer_cast(p->devStencil);

  _is_less_than_zero pred;
  thrust::replace_if(thrust::device, dev_ptr, dev_ptr + chans,
                     pred, (MYFLT) 0.0);

  cudaMemcpy(p->func->ftable, p->devStencil,
             stencilSize, cudaMemcpyDeviceToHost);

  csound->RegisterDeinitCallback(csound, p, free_device);

  return OK;
}

static int cudapvstencil2(CSOUND *csound, CUDAPVSTENCIL2 *p)
{
  int framelength = p->fin->N + 2;
  int chans = framelength>>1;
  float* fout = (float *) p->fout->frame.auxp;
  float* fin = (float *) p->fin->frame.auxp;
  float   g = fabsf((float)*p->kgain);
  float   level = fabsf((float)*p->klevel);

  if (UNLIKELY(fout == NULL)) goto err1;

  if (p->lastframe < p->fin->framecount) {

    stencilkernel<<<p->gridSize,p->blockSize>>>(fout, fin,
                                                p->devStencil, level,
                                                g, chans);

    p->fout->framecount = p->lastframe = p->fin->framecount;
  }

  return OK;
  err1:
    return csound->PerfError(csound, p->h.insdshead,
                             Str("cudapvstencil2: not initialised"));
}

static OENTRY localops[] = {
  {"cudapvstencil2", sizeof(CUDAPVSTENCIL2), TR, 3, "f", "fkki",
  (SUBR) cudapvstencil2set, (SUBR) cudapvstencil2}
};

extern "C" {
  LINKAGE
}
```

Scale Module

cudapvscale

```
// Consider writing another "freqScaleBasic" kernel for scaling
// upwards only (without atomic operations, as they are not needed)

#include <csdl.h>
#include <pstream.h>
#include <cufft.h>
#include <thrust/extrema.h>
#include <thrust/device_ptr.h>
#include <thrust/fill.h>
#include <thrust/reduce.h>

typedef struct _cudapvscale {
  OPDS    h;
  PVSDAT  *fout;
  PVSDAT  *fin;
  MYFLT   *kscal;
  MYFLT   *keepform;
  MYFLT   *gain;
  MYFLT   *coefs;
  AUXCH   fenv, ceps;
  float*  deviceInput;   // pointer to device memory (input frame)
  float*  deviceOutput;   // pointer to device memory (output frame)
  float*  deviceEnv;   // pointer to device memory
                       // (amplitude spectral envelope frame)
  float*  deviceCepstrum;   // pointer to device memory (cepstrum frame)
  float*  deviceTrueEnv;   // pointer to device memory  (true envelope)
  float*  deviceSmoothTrueEnv;   // pointer to device memory
                                 // (true envelope, smoothed)
  int*    deviceMask;   // pointer to device memory
                        // (boolean mask for condition checking)
  int  gridSize;   // number of blocks in the grid (1D)
  int blockSize;   // number of threads in one block (1D)
  cufftHandle forwardPlan;   // forward cuFFT plan
  cufftHandle inversePlan;   // inverse cuFFT plan
  uint32  lastframe;
} CUDAPVSCALE;

// kernel for frequency scaling without formant keeping (part one)
__global__
void freqScaleBasic(float* input, float* output,
                    MYFLT scaleFactor, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = (i<<1) + 2;
  int N = nhalf<<1;
  int newchan;
  if (i < nhalf-1) {
    newchan = (int)(((i+1)*scaleFactor)+0.5) << 1;
    if (newchan < N && newchan > 0) {
      atomicExch(&output[newchan],input[j]);  // move amplitude data
                                              // to new positions
      atomicExch(&output[newchan+1],
                 (float)(input[j+1]*scaleFactor));  // change bin
                                                    // frequencies
    }
  }
}
```

```
// kernel for frequency scaling
// (part two, with or without formant keeping)
__global__
void fixPVandGain(float* output, float gain, int length) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  if (j < length) {
    if (isnan(output[j]))
      output[j] = 0.0f;  // set to zero any invalid amplitude
    if (output[j+1] == -1.0f) {
      output[j] = 0.0f;   // set to zero the amp related to any
                          // undefined frequency
    }
    else
      output[j] *= gain;   // scale all amplitudes
                           // by the gain factor
  }
}

__global__
void takeLog(float* input, float* env, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  if (i < nhalf) {
    env[i] = log(input[j] > 0.0 ? input[j] : 1e-20);  // take the log
                                                      // of the amplitudes
  }
}

__global__
void lifter(float* cepstrum, int nCoefs, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int k = i + nCoefs;
  if (k < nhalf+2-nCoefs) {
    cepstrum[k] = 0.0;  // kill all the cepstrum coefficients
                        // above nCoefs
  }
}

__global__
void expon(float* env, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  if (i < nhalf) {
    env[i] = exp(env[i]/nhalf);   // exponentiate
  }
}

// kernel for frequency scaling with formant keeping (part one)
__global__
void freqScaleFormant(float* input, float* output, float* env,
                      MYFLT scaleFactor, float maxAmp, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = (i<<1) + 2;
  int N = nhalf<<1;
  int newchan;
  if (i < nhalf-1) {
    env[i+1] /= maxAmp;   // normalize the spectral envelope
    input[j] /= env[i+1];   // equalize the original amplitudes so that
                            // formant shaping is more effective
    newchan = (int)(((i+1)*scaleFactor)+0.5) << 1;
    if (newchan < N && newchan > 0) {
      // move amp data to new positions and
      // weight by normalized env:
      atomicExch(&output[newchan], input[j]*env[newchan>>1]*0.9);
      // change bin frequencies:
      atomicExch(&output[newchan+1],
                 (float)(input[j+1]*scaleFactor));
    }
  }
}

// after completing the inverse fft,
// this kernel updates the true envelope:
// for each bin, the max of input and smoothed spectral envelopes is taken
__global__
void update(float* original, float* newTE, float* current, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  if (i < nhalf) {
    current[i] /= nhalf;
    newTE[i] = (original[i] < current[i]) ? current[i] : original[i];
  }
}

// kernel for testing the true envelope condition
__global__
void test(float* nonSmoothed, float* smoothed, int* mask, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int diff;
  if (i < nhalf) {
    diff = fabs(nonSmoothed[i] - smoothed[i]/nhalf);
    mask[i] = (diff > 0.23) ? 1 : 0;   // WHAT THRESHOLD TO USE?
  }
}

static int free_device(CSOUND* csound, void* pp){
  CUDAPVSCALE* p = (CUDAPVSCALE*) pp;
  cudaFree(p->deviceInput);
  cudaFree(p->deviceOutput);
  cudaFree(p->deviceEnv);
  cudaFree(p->deviceCepstrum);
  cudaFree(p->deviceTrueEnv);
  cudaFree(p->deviceSmoothTrueEnv);
  cudaFree(p->deviceMask);
  cufftDestroy(p->forwardPlan);
  cufftDestroy(p->inversePlan);
  return OK;
}

static int cudapvscaleset(CSOUND *csound, CUDAPVSCALE *p)
{
  int N = p->fin->N;
  int Nhalf = N>>1;
  int size = (N+2) * sizeof(float);
  int smallSize = ((Nhalf>>1)+1) * sizeof(cufftComplex);
  int maxBlockDim;
  int SMcount;
  int totNumThreads = Nhalf;
  cudaError_t error;

  // get info about device
  cudaDeviceProp deviceProp;
```

```
cudaGetDeviceProperties(&deviceProp,0);
maxBlockDim = deviceProp.maxThreadsPerBlock;
SMcount = deviceProp.multiProcessorCount;
csound->Message(csound, "cudapvscale running on device
                %s (capability %d.%d)\n", deviceProp.name,
                deviceProp.major, deviceProp.minor);

// create a cuFFT plan to use later
cufftPlan1d(&p->forwardPlan, Nhalf, CUFFT_R2C, 1);
cufftPlan1d(&p->inversePlan, Nhalf, CUFFT_C2R, 1);
cufftSetCompatibilityMode(p->forwardPlan, CUFFT_COMPATIBILITY_NATIVE);
cufftSetCompatibilityMode(p->inversePlan, CUFFT_COMPATIBILITY_NATIVE);

// device memory allocation and copy from host (input data)
error = cudaMalloc(&p->deviceInput, size);
handleCudaError(csound, error);

// device memory allocation and set to zero (output data to be computed)
error = cudaMalloc(&p->deviceOutput, size);
handleCudaError(csound, error);
cudaMemset(p->deviceOutput,0,size);

// device memory allocation and set to zero
// (amplitude spectral envelope, approx half the length)
error = cudaMalloc(&p->deviceEnv, smallSize);
handleCudaError(csound, error);
cudaMemset(p->deviceEnv,0,smallSize);

// device memory allocation and set to zero
// (cepstrum, approx half the length)
error = cudaMalloc(&p->deviceCepstrum, smallSize);
handleCudaError(csound, error);
cudaMemset(p->deviceCepstrum,0,smallSize);

// device memory allocation and set to zero
// (true envelope, approx half the length)
error = cudaMalloc(&p->deviceTrueEnv, smallSize);
handleCudaError(csound, error);
cudaMemset(p->deviceTrueEnv,0,smallSize);

// device memory allocation and set to zero
// (smoothed true envelope, approx half the length)
error = cudaMalloc(&p->deviceSmoothTrueEnv, smallSize);
handleCudaError(csound, error);
cudaMemset(p->deviceSmoothTrueEnv,0,smallSize);

// device memory allocation and set to one (mask)
error = cudaMalloc(&p->deviceMask, Nhalf*sizeof(int));
handleCudaError(csound, error);
cudaMemset(p->deviceMask,1,Nhalf*sizeof(int));

if (UNLIKELY(p->fin == p->fout))
  csound->Warning(csound, Str("Unsafe to have same
                              fsig as in and out"));
p->fout->NB = p->fin->NB;
p->fout->sliding = p->fin->sliding;

if (p->fout->frame.auxp == NULL ||
    p->fout->frame.size < sizeof(float) * (N + 2))  /* RWD MUST
                                                       be 32bit */
  csound->AuxAlloc(csound, sizeof(float) * (N + 2), &p->fout->frame);
  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N = N;
  p->fout->overlap = p->fin->overlap;
  p->fout->winsize = p->fin->winsize;
  p->fout->wintype = p->fin->wintype;
  p->fout->format = p->fin->format;
  p->fout->framecount = 1;
  p->lastframe = 0;

  csound->RegisterDeinitCallback(csound, p, free_device);
  return OK;
}

static int cudapvscale(CSOUND *csound, CUDAPVSCALE *p)
{
  int     i, N = p->fout->N;
  int Nhalf = N>>1;
  int framelength = N+2;
  int size = (framelength) * sizeof(float);
  float   max = 0.0f;
  MYFLT   pscal = (MYFLT) *p->kscal;
  int     keepform = (int) *p->keepform;
  float   g = (float) *p->gain;
  float   *fin = (float *) p->fin->frame.auxp;
  float   *fout = (float *) p->fout->frame.auxp;
  int coefs = (int) *p->coefs;

  cufftComplex* cufftEnv;
  cufftComplex* cufftCepstrum;
  cufftComplex* cufftTrueEnv;
  cufftComplex* cufftSmoothTrueEnv;

  thrust::device_ptr<float> dev_ptr1 =
    thrust::device_pointer_cast(p->deviceOutput);
  thrust::device_ptr<float> dev_ptr2 =
    thrust::device_pointer_cast(p->deviceEnv);
  thrust::device_ptr<float> dev_ptr3 =
    thrust::device_pointer_cast(p->deviceSmoothTrueEnv);
  thrust::device_ptr<int> dev_ptr4 =
    thrust::device_pointer_cast(p->deviceMask);

  if (UNLIKELY(fout == NULL)) goto err1;

  if (p->lastframe < p->fin->framecount) {

    cudaMemcpy(p->deviceInput,fin,size,cudaMemcpyHostToDevice);

    if (keepform == 0) {
      // resets the output:
      thrust::fill(dev_ptr1, dev_ptr1+framelength, -1.0f);
      // freq scaling:
      freqScaleBasic<<<p->gridSize,p->blockSize>>>(p->deviceInput,
                                                   p->deviceOutput,
                                                   pscal, Nhalf);
      // apply gain to all amplitudes:
      fixPVandGain<<<p->gridSize,p->blockSize>>>(p->deviceOutput,
                                                 g, framelength);
    }
```

```
else if (keepform==1) {
  if (coefs<1) coefs = 80;

  // resets the output:
  thrust::fill(dev_ptr1, dev_ptr1+framelength, -1.0f);
  // take log:
  takeLog<<<p->gridSize,p->blockSize>>>(p->deviceInput,
                                        p->deviceEnv, Nhalf);

  cufftEnv = (cufftComplex*) p->deviceEnv;
  cufftCepstrum = (cufftComplex*) p->deviceCepstrum;

  // take the fft of the log of the spectral envelope:
  if(cufftExecR2C(p->forwardPlan,(cufftReal*)cufftEnv,
                  cufftCepstrum)!= CUFFT_SUCCESS)
    csound->Message(csound, "CUDA FFT error\n");
  if (cudaDeviceSynchronize() != cudaSuccess)
    csound->Message(csound,"CUDA error: Failed to synchronize\n");

  // liftering stage: keep only low quefrency coefficients:
  lifter<<<p->gridSize,p->blockSize>>>(p->deviceCepstrum, coefs,
                                       Nhalf);

  // take the inverse fft of the liftered cepstrum:
  if(cufftExecC2R(p->inversePlan,cufftCepstrum,
                  (cufftReal*)cufftEnv)!= CUFFT_SUCCESS)
    csound->Message(csound, "CUDA FFT error\n");
  if (cudaDeviceSynchronize() != cudaSuccess)
    csound->Message(csound,"CUDA error: Failed to synchronize\n");

  // scale the result of the inverse transform and exponentiate
  // to go back to true amplitudes:
  expon<<<p->gridSize,p->blockSize>>>(p->deviceEnv, Nhalf);

  // find maximum amp in spectral envelope:
  max = *(thrust::max_element(dev_ptr2, dev_ptr2+Nhalf));

  // ormalize spectral env and freq scale the input:
  freqScaleFormant<<<p->gridSize,p->blockSize>>>(p->deviceInput,
                                                 p->deviceOutput,
                                                 p->deviceEnv,
                                                 pscal, max,
                                                 Nhalf);
  // apply gain to all amplitudes:
  fixPVandGain<<<p->gridSize,p->blockSize>>>(p->deviceOutput, g,
                                             framelength);
}

else if (keepform==2) {
  int cond = 1;
  if (coefs<1) coefs = 80;
  cufftEnv = (cufftComplex*) p->deviceEnv;
  cufftCepstrum = (cufftComplex*) p->deviceCepstrum;
  cufftTrueEnv = (cufftComplex*) p->deviceTrueEnv;
  cufftSmoothTrueEnv = (cufftComplex*) p->deviceSmoothTrueEnv;

  // resets the output:
  thrust::fill(dev_ptr1, dev_ptr1+framelength, -1.0f);
  takeLog<<<p->gridSize,p->blockSize>>>(p->deviceInput,
                                        p->deviceEnv, Nhalf);
```

```
// loop initialization stage:
// smooth the original log spectral envelope...
// take the fft of the log of the spectral envelope:
if(cufftExecR2C(p->forwardPlan,(cufftReal*)cufftEnv,
                cufftCepstrum)!= CUFFT_SUCCESS)
  csound->Message(csound, "CUDA FFT error\n");
if (cudaDeviceSynchronize() != cudaSuccess)
  csound->Message(csound,"CUDA error: Failed to synchronize\n");

// liftering: keep only low quefrency coefficients:
lifter<<<p->gridSize,p->blockSize>>>(p->deviceCepstrum,
                                     coefs, Nhalf);

// take the inverse fft of the liftered cepstrum:
if(cufftExecC2R(p->inversePlan,cufftCepstrum,
                (cufftReal*)cufftSmoothTrueEnv)!= CUFFT_SUCCESS)
  csound->Message(csound, "CUDA FFT error\n");
if (cudaDeviceSynchronize() != cudaSuccess)
  csound->Message(csound,"CUDA error: Failed to synchronize\n");

// main loop:
while(cond) {
  update<<<p->gridSize,p->blockSize>>>(p->deviceEnv,
                                       p->deviceTrueEnv,
                                       p->deviceSmoothTrueEnv,
                                       Nhalf);

  // take the fft of the true envelope:
  if(cufftExecR2C(p->forwardPlan,(cufftReal*)cufftTrueEnv,
                  cufftCepstrum)!= CUFFT_SUCCESS)
    csound->Message(csound, "CUDA FFT error\n");
  if (cudaDeviceSynchronize() != cudaSuccess)
    csound->Message(csound,"CUDA error:
                          Failed to synchronize\n");

  // liftering: keep only low quefrency coefficients:
  lifter<<<p->gridSize,p->blockSize>>>(p->deviceCepstrum,
                                       coefs, Nhalf);

  // take the inverse fft of the liftered cepstrum:
  if(cufftExecC2R(p->inversePlan,cufftCepstrum,
                  (cufftReal*)cufftSmoothTrueEnv)!= CUFFT_SUCCESS)
    csound->Message(csound, "CUDA FFT error\n");
  if (cudaDeviceSynchronize() != cudaSuccess)
    csound->Message(csound,"CUDA error:
                          Failed to synchronize\n");

  test<<<p->gridSize,p->blockSize>>>(p->deviceTrueEnv,
                                     p->deviceSmoothTrueEnv,
                                     p->deviceMask, Nhalf);
  if((thrust::reduce(dev_ptr4, dev_ptr4+Nhalf)) == 0)
    cond = 0;
}

// scale the result of the inverse transform
// and exponentiate to go back to true amplitudes:
expon<<<p->gridSize,p->blockSize>>>(p->deviceSmoothTrueEnv,
                                    Nhalf);
```

```
      // find maximum amp in spectral envelope:
      max = *(thrust::max_element(dev_ptr3, dev_ptr3+Nhalf));

      // normalize spectral env and freq scale the input:
      freqScaleFormant<<<p->gridSize,p->blockSize>>>(p->deviceInput,
                                                     p->deviceOutput,
                                                     p->deviceSmoothTrueEnv,
                                                     pscal, max, Nhalf);
      // apply gain to all amplitudes:
      fixPVandGain<<<p->gridSize,p->blockSize>>>(p->deviceOutput, g,
                                                   framelength);
    }

    cudaMemcpy(fout, p->deviceOutput, size, cudaMemcpyDeviceToHost);
    fout[0] = fin[0];   // keep original DC amplitude
    fout[N] = fin[N];   // keep original Nyquist amplitude
    p->fout->framecount = p->lastframe = p->fin->framecount;
  }

  return OK;

  err1:
    return csound->PerfError(csound, p->h.insdshead,
                             Str("cudapvscale: not initialised"));
}

static OENTRY localops[] = {
  {"cudapvscale", sizeof(CUDAPVSCALE),0, 3, "f", "fxOPO",
    (SUBR) cudapvscaleset, (SUBR) cudapvscale}
};


extern "C" {
  LINKAGE
}
```

cudapvscale2

```
// Consider writing another "freqScaleBasic" kernel for scaling
// upwards only (without atomic operations, as they are not needed)

#include <csdl.h>
#include <pstream.h>
#include <cufft.h>
#include <thrust/extrema.h>
#include <thrust/device_ptr.h>
#include <thrust/fill.h>
#include <thrust/reduce.h>

typedef struct _cudapvscale2 {
  OPDS    h;
  PVSDAT  *fout;
  PVSDAT  *fin;
  MYFLT   *kscal;
  MYFLT   *keepform;
  MYFLT   *gain;
  MYFLT   *coefs;
  AUXCH   fenv, ceps;
  float*  deviceEnv;   // pointer to device memory
                       // (amplitude spectral envelope frame)
  float*  deviceCepstrum;   // pointer to device memory (cepstrum frame)
  float*  deviceTrueEnv;   // pointer to device memory  (true envelope)
  float*  deviceSmoothTrueEnv;   // pointer to device memory
                                 // (true envelope, smoothed)
  int*    deviceMask;   // pointer to device memory
                        // (boolean mask for condition checking)
  int  gridSize;   // number of blocks in the grid (1D)
  int blockSize;   // number of threads in one block (1D)
  cufftHandle forwardPlan;   // forward cuFFT plan
  cufftHandle inversePlan;   // inverse cuFFT plan
  uint32  lastframe;
} CUDAPVSCALE2;

// kernel for frequency scaling without formant keeping (part one)
__global__
void freqScaleBasic(float* input, float* output,
                    MYFLT scaleFactor, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = (i<<1) + 2;
  int N = nhalf<<1;
  int newchan;
  if (i < nhalf-1) {
    newchan = (int)(((i+1)*scaleFactor)+0.5) << 1;
    if (newchan < N && newchan > 0) {
      // move amplitude data to new positions:
      atomicExch(&output[newchan],input[j]);
      // change bin frequencies:
      atomicExch(&output[newchan+1],
                 (float)(input[j+1]*scaleFactor));
    }
  }
}

// kernel for frequency scaling
// (part two, with or without formant keeping)
__global__
void fixPVandGain(float* input, float* output, float gain, int length) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  if (j < length) {
    if (isnan(output[j]))
      output[j] = 0.0f;  // set to zero any invalid amplitude
    if (output[j+1] == -1.0f) {
      output[j] = 0.0f;   // set to zero the amplitude
                          // related to any undefined frequency
    }
    else
      output[j] *= gain;   // scale all amplitudes
                           // by the gain factor
  }
  // keep original DC amplitude:
  if (j == 0) output[0] = input[0];
  // keep original Nyquist amplitude:
  if (j == length-2) output[length-2] = input[length-2];
}
```

```
__global__
void takeLog(float* input, float* env, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  if (i < nhalf) {
    // take the log of the amplitudes:
    env[i] = log(input[j] > 0.0 ? input[j] : 1e-20);
  }
}

__global__
void lifter(float* cepstrum, int nCoefs, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int k = i + nCoefs;
  if (k < nhalf+2-nCoefs) {
    cepstrum[k] = 0.0;   // kill all the cepstrum
                         // coefficients above nCoefs
  }
}

__global__
void expon(float* env, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  if (i < nhalf) {
    env[i] = exp(env[i]/nhalf);   // exponentiate
  }
}

// kernel for frequency scaling with formant keeping (part one)
__global__
void freqScaleFormant(float* input, float* output, float* env,
                      MYFLT scaleFactor, float maxAmp, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = (i<<1) + 2;
  int N = nhalf<<1;
  int newchan;
  if (i < nhalf-1) {
    env[i+1] /= maxAmp;   // normalize the spectral envelope
    input[j] /= env[i+1];   // equalize the original amplitudes so that
                            // formant shaping is more effective
    newchan = (int)(((i+1)*scaleFactor)+0.5) << 1;
    if (newchan < N && newchan > 0) {
      // move amp data to new positions and weight by normalized env:
      atomicExch(&output[newchan], input[j]*env[newchan>>1]*0.9);
      // change bin frequencies:
      atomicExch(&output[newchan+1],
                 (float)(input[j+1]*scaleFactor));
    }
  }
}

// after completing the inverse fft this kernel updates the true envelope:
// for each bin, the max of input and smoothed spectral envelopes is taken
__global__
void update(float* original, float* newTE, float* current, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  if (i < nhalf) {
    current[i] /= nhalf;
    newTE[i] = (original[i] < current[i]) ? current[i] : original[i];
  }
}
```

```
// kernel for testing the true envelope condition
__global__
void test(float* nonSmoothed, float* smoothed, int* mask, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int diff;
  if (i < nhalf) {
    diff = fabs(nonSmoothed[i] - smoothed[i]/nhalf);
    mask[i] = (diff > 0.23) ? 1 : 0;   // WHAT THRESHOLD TO USE?
  }
}

static int free_device(CSOUND* csound, void* pp){
  CUDAPVSCALE2* p = (CUDAPVSCALE2*) pp;
  cudaFree(p->fout->frame.auxp);
  cudaFree(p->deviceEnv);
  cudaFree(p->deviceCepstrum);
  cudaFree(p->deviceTrueEnv);
  cudaFree(p->deviceSmoothTrueEnv);
  cudaFree(p->deviceMask);
  cufftDestroy(p->forwardPlan);
  cufftDestroy(p->inversePlan);
  return OK;
}

static int cudapvscale2set(CSOUND *csound, CUDAPVSCALE2 *p)
{
  int N = p->fin->N;
  int Nhalf = N>>1;
  int size = (N+2) * sizeof(float);
  int smallSize = ((Nhalf>>1)+1) * sizeof(cufftComplex);
  int maxBlockDim;
  int SMcount;
  int totNumThreads = Nhalf;
  cudaError_t error;

  // get info about device
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound, "cudapvscale2 running on device %s
                  (capability %d.%d)\n", deviceProp.name,
                  deviceProp.major, deviceProp.minor);

  // create a cuFFT plan to use later
  cufftPlan1d(&p->forwardPlan, Nhalf, CUFFT_R2C, 1);
  cufftPlan1d(&p->inversePlan, Nhalf, CUFFT_C2R, 1);
  cufftSetCompatibilityMode(p->forwardPlan, CUFFT_COMPATIBILITY_NATIVE);
  cufftSetCompatibilityMode(p->inversePlan, CUFFT_COMPATIBILITY_NATIVE);

  // device memory allocation and set to zero
  // (amplitude spectral envelope, approx half the length)
  error = cudaMalloc(&p->deviceEnv, smallSize);
  handleCudaError(csound, error);
  cudaMemset(p->deviceEnv,0,smallSize);

  // device memory allocation and set to zero
  // (cepstrum, approx half the length)
  error = cudaMalloc(&p->deviceCepstrum, smallSize);
  handleCudaError(csound, error);
  cudaMemset(p->deviceCepstrum,0,smallSize);
  // device memory allocation and set to zero
  // (true envelope, approx half the length)
  error = cudaMalloc(&p->deviceTrueEnv, smallSize);
  handleCudaError(csound, error);
  cudaMemset(p->deviceTrueEnv,0,smallSize);

  // device memory allocation and set to zero
  // (smoothed true envelope, approx half the length)
  error = cudaMalloc(&p->deviceSmoothTrueEnv, smallSize);
  handleCudaError(csound, error);
  cudaMemset(p->deviceSmoothTrueEnv,0,smallSize);

  // device memory allocation and set to one (mask)
  error = cudaMalloc(&p->deviceMask, Nhalf*sizeof(int));
  handleCudaError(csound, error);
  cudaMemset(p->deviceMask,1,Nhalf*sizeof(int));

  if (UNLIKELY(p->fin == p->fout))
    csound->Warning(csound, Str("Unsafe to have same fsig
                              as in and out"));
  p->fout->NB = p->fin->NB;

  p->fout->sliding = 0;

  if (p->fout->frame.auxp == NULL ||
      p->fout->frame.size < sizeof(float) * (N + 2))  /* RWD MUST be
                                                         32bit */
  AuxCudaAlloc(size, &p->fout->frame);

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N = N;
  p->fout->overlap = p->fin->overlap;
  p->fout->winsize = p->fin->winsize;
  p->fout->wintype = p->fin->wintype;
  p->fout->format = p->fin->format;
  p->fout->framecount = 1;
  p->lastframe = 0;

  csound->RegisterDeinitCallback(csound, p, free_device);

  return OK;
}

static int cudapvscale2(CSOUND *csound, CUDAPVSCALE2 *p)
{
  int     i, N = p->fout->N;
  int Nhalf = N>>1;
  int framelength = N+2;
  float   max = 0.0f;
  MYFLT   pscal = (MYFLT) *p->kscal;
  int     keepform = (int) *p->keepform;
  float   g = (float) *p->gain;
  float   *fin = (float *) p->fin->frame.auxp;
  float   *fout = (float *) p->fout->frame.auxp;
  int coefs = (int) *p->coefs;

  cufftComplex* cufftEnv;
  cufftComplex* cufftCepstrum;
```

```
cufftComplex* cufftTrueEnv;
cufftComplex* cufftSmoothTrueEnv;

thrust::device_ptr<float> dev_ptr1 =
  thrust::device_pointer_cast(fout);
thrust::device_ptr<float> dev_ptr2 =
  thrust::device_pointer_cast(p->deviceEnv);
thrust::device_ptr<float> dev_ptr3 =
  thrust::device_pointer_cast(p->deviceSmoothTrueEnv);
thrust::device_ptr<int> dev_ptr4 =
  thrust::device_pointer_cast(p->deviceMask);

if (UNLIKELY(fout == NULL)) goto err1;

if (p->lastframe < p->fin->framecount) {

  if (keepform == 0) {
    // resets the output:
    thrust::fill(dev_ptr1, dev_ptr1+framelength, -1.0f);
    // freq scaling:
    freqScaleBasic<<<p->gridSize,p->blockSize>>>(fin, fout,
                                                  pscal, Nhalf);
    // apply gain to all amplitudes:
    fixPVandGain<<<p->gridSize,p->blockSize>>>(fin, fout, g,
                                                framelength);
  }

  else if (keepform==1) {
    if (coefs<1) coefs = 80;

    // resets the output:
    thrust::fill(dev_ptr1, dev_ptr1+framelength, -1.0f);
    takeLog<<<p->gridSize,p->blockSize>>>(fin, p->deviceEnv, Nhalf);

    cufftEnv = (cufftComplex*) p->deviceEnv;
    cufftCepstrum = (cufftComplex*) p->deviceCepstrum;

    // take the fft of the log of the spectral envelope:
    if(cufftExecR2C(p->forwardPlan,(cufftReal*)cufftEnv,
                    cufftCepstrum)!= CUFFT_SUCCESS)
      csound->Message(csound, "CUDA FFT error\n");
    if (cudaDeviceSynchronize() != cudaSuccess)
      csound->Message(csound,"CUDA error: Failed to synchronize\n");

    // liftering stage: keep only low quefrency coefficients:
    lifter<<<p->gridSize,p->blockSize>>>(p->deviceCepstrum,
                                          coefs, Nhalf);

    // take the inverse fft of the liftered cepstrum:
    if(cufftExecC2R(p->inversePlan,cufftCepstrum,
                    (cufftReal*)cufftEnv)!= CUFFT_SUCCESS)
      csound->Message(csound, "CUDA FFT error\n");
    if (cudaDeviceSynchronize() != cudaSuccess)
      csound->Message(csound,"CUDA error: Failed to synchronize\n");

    // scale the result of the inverse transform
    // and exponentiate to go back to true amplitudes:
    expon<<<p->gridSize,p->blockSize>>>(p->deviceEnv, Nhalf);

    // find maximum amp in spectral envelope:
    max = *(thrust::max_element(dev_ptr2, dev_ptr2+Nhalf));
    // normalize spectral env and freq scale the input:
    freqScaleFormant<<<p->gridSize,p->blockSize>>>(fin, fout,
                                                    p->deviceEnv,
                                                    pscal, max,
                                                    Nhalf);
    // apply gain to all amplitudes:
    fixPVandGain<<<p->gridSize,p->blockSize>>>(fin, fout, g,
                                                framelength);
  }

  else if (keepform==2) {
    int cond = 1;
    if (coefs<1) coefs = 80;
    cufftEnv = (cufftComplex*) p->deviceEnv;
    cufftCepstrum = (cufftComplex*) p->deviceCepstrum;
    cufftTrueEnv = (cufftComplex*) p->deviceTrueEnv;
    cufftSmoothTrueEnv = (cufftComplex*) p->deviceSmoothTrueEnv;

    // resets the output:
    thrust::fill(dev_ptr1, dev_ptr1+framelength, -1.0f);
    takeLog<<<p->gridSize,p->blockSize>>>(fin, p->deviceEnv, Nhalf);

    // loop initialization stage:
    // smooth the original log spectral envelope:
    // take the fft of the log of the spectral envelope:
    if(cufftExecR2C(p->forwardPlan,(cufftReal*)cufftEnv,
                    cufftCepstrum)!= CUFFT_SUCCESS)
      csound->Message(csound, "CUDA FFT error\n");
    if (cudaDeviceSynchronize() != cudaSuccess)
      csound->Message(csound,"CUDA error: Failed to synchronize\n");

    // liftering: keep only low quefrency coefficients:
    lifter<<<p->gridSize,p->blockSize>>>(p->deviceCepstrum,
                                          coefs, Nhalf);

    // take the inverse fft of the liftered cepstrum:
    if(cufftExecC2R(p->inversePlan,cufftCepstrum,
                    (cufftReal*)cufftSmoothTrueEnv)!= CUFFT_SUCCESS)
      csound->Message(csound, "CUDA FFT error\n");
    if (cudaDeviceSynchronize() != cudaSuccess)
      csound->Message(csound,"CUDA error: Failed to synchronize\n");

    while(cond) {
      update<<<p->gridSize,p->blockSize>>>(p->deviceEnv,
                                            p->deviceTrueEnv,
                                            p->deviceSmoothTrueEnv,
                                            Nhalf);

      // take the fft of the true envelope:
      if(cufftExecR2C(p->forwardPlan,(cufftReal*)cufftTrueEnv,
                      cufftCepstrum)!= CUFFT_SUCCESS)
        csound->Message(csound, "CUDA FFT error\n");
      if (cudaDeviceSynchronize() != cudaSuccess)
        csound->Message(csound,"CUDA error: Failed to
                        synchronize\n");

      // liftering: keep only low quefrency coefficients:
      lifter<<<p->gridSize,p->blockSize>>>(p->deviceCepstrum,
                                            coefs, Nhalf);
        // take the inverse fft of the liftered cepstrum:
        if(cufftExecC2R(p->inversePlan,cufftCepstrum,
                        (cufftReal*)cufftSmoothTrueEnv)!= CUFFT_SUCCESS)
          csound->Message(csound, "CUDA FFT error\n");
        if (cudaDeviceSynchronize() != cudaSuccess)
          csound->Message(csound,"CUDA error:
                                  Failed to synchronize\n");

        test<<<p->gridSize,p->blockSize>>>(p->deviceTrueEnv,
                                           p->deviceSmoothTrueEnv,
                                           p->deviceMask, Nhalf);
        if((thrust::reduce(dev_ptr4, dev_ptr4+Nhalf)) == 0)
          cond = 0;
      }

      // scale the result of the inverse transform
      // and exponentiate to go back to true amplitudes:
      expon<<<p->gridSize,p->blockSize>>>(p->deviceSmoothTrueEnv,
                                          Nhalf);

      // find maximum amp in spectral envelope:
      max = *(thrust::max_element(dev_ptr3, dev_ptr3+Nhalf));

      // normalize spectral env and freq scale the input:
      freqScaleFormant<<<p->gridSize,p->blockSize>>>(fin, fout,
                                               p->deviceSmoothTrueEnv,
                                               pscal, max, Nhalf);
      // apply gain to all amplitudes:
      fixPVandGain<<<p->gridSize,p->blockSize>>>(fin, fout,
                                                 g, framelength);
    }

    p->fout->framecount = p->lastframe = p->fin->framecount;
  }

  return OK;

  err1:
    return csound->PerfError(csound, p->h.insdshead,
                             Str("cudapvscale2: not initialised"));
}

static OENTRY localops[] = {
  {"cudapvscale2", sizeof(CUDAPVSCALE2),0, 3, "f", "fxOPO",
  (SUBR) cudapvscale2set, (SUBR) cudapvscale2}
};

extern "C" {
  LINKAGE
}
```

Shift Module

cudapvshift

```
#include <csdl.h>
#include <pstream.h>
#include <cufft.h>
#include <thrust/extrema.h>
#include <thrust/device_ptr.h>
#include <thrust/fill.h>
#include <thrust/reduce.h>

typedef struct _cudapvshift {
  OPDS    h;
  PVSDAT  *fout;
  PVSDAT  *fin;
  MYFLT   *kshift;
  MYFLT   *lowest;
  MYFLT   *keepform;
  MYFLT   *gain;
  MYFLT   *coefs;
  AUXCH   fenv, ceps;
  float*  deviceInput;   // pointer to device memory (input frame)
  float*  deviceOutput;   // pointer to device memory (output frame)
  float*  deviceEnv;   // pointer to device memory
                       // (amplitude spectral envelope frame)
  float*  deviceCepstrum;   // pointer to device memory (cepstrum frame)
  float*  deviceTrueEnv;   // pointer to device memory  (true envelope)
  float*  deviceSmoothTrueEnv;   // pointer to device memory
                                 // (true envelope, smoothed)
  int*    deviceMask;   // pointer to device memory
                        // (boolean mask for condition checking)
  int  gridSize;   // number of blocks in the grid (1D)
  int  blockSize;   // number of threads in one block (1D)
  cufftHandle forwardPlan;   // forward cuFFT plan
  cufftHandle inversePlan;   // inverse cuFFT plan
  uint32  lastframe;
} CUDAPVSHIFT;

// kernel for frequency shifting without formant keeping (part one)
__global__
void freqShiftBasic(float* input, float* output, MYFLT shift,
                    int shiftChan, int lowestIndx, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  int lowestChan = lowestIndx>>1;
  int N = nhalf<<1;
  int newchan;
  if (i < lowestChan) {
    // leave PV data as it is below a certain channel:
    output[j] = input[j];
    output[j+1] = input[j+1];
  }
  if (i >= lowestChan && i < nhalf) {
    newchan = (i + shiftChan) << 1;
    if (newchan < N && newchan >= lowestIndx) {
      // move amplitude data to new positions:
      output[newchan] = input[j];
      // change bin frequencies:
      output[newchan+1] = (float) (input[j+1] + shift);
    }
  }
}

// kernel for frequency shifting
// (part two, with or without formant keeping)
__global__
void fixPVandGain(float* output, float gain, int lowestIndx,
                  int length) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  if (j >= lowestIndx && j < length) {
    if (isnan(output[j]))
    output[j] = 0.0f;  // set to zero any invalid amplitude
    if (output[j+1] == -1.0f) {
      output[j] = 0.0f;   // set to zero the amplitude
                          // related to any undefined frequency
    }
    else {
      output[j] *= gain;   // scale all amplitudes
                           // by the gain factor
    }
  }
}

__global__
void takeLog(float* input, float* env, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  if (i < nhalf) {
    // take the log of the amplitudes:
    env[i] = log(input[j] > 0.0 ? input[j] : 1e-20);
  }
}

__global__
void lifter(float* cepstrum, int nCoefs, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int k = i + nCoefs;
  if (k < nhalf+2-nCoefs) {
    cepstrum[k] = 0.0;   // kill all the cepstrum coefficients
                         // above nCoefs
  }
}

__global__
void expon(float* env, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  if (i < nhalf) {
    env[i] = exp(env[i]/nhalf);   // exponentiate
  }
}
```

```
// kernel for frequency shifting with formant keeping (part one)
__global__
void freqShiftFormant(float* input, float* output, float* env,
                      MYFLT shift, int shiftChan, int lowestIndx,
                      float maxAmp, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  int lowestChan = lowestIndx>>1;
  int N = nhalf<<1;
  int newchan;
  if (i < lowestChan) {
    // leave PV data as it is below a certain channel:
    output[j] = input[j];
    output[j+1] = input[j+1];
  }
  else if (i < nhalf) {
    env[i] /= maxAmp;   // normalize the spectral envelope
    input[j] /= env[i];   // equalize the original amplitudes
                          // so that formant shaping is more effective
    newchan = (i + shiftChan) << 1;
    if (newchan < N && newchan >= lowestIndx) {
      // move amp data to new positions and weight by normalized env:
      output[newchan] = input[j]*env[newchan>>1]*0.9;
      // change bin frequencies:
      output[newchan+1] = (float)(input[j+1] + shift);
    }
  }
}

// after completing the inverse fft,
// this kernel updates the true envelope:
// for each bin, the max of input and smoothed spectral envelopes is taken
__global__
void update(float* original, float* newTE, float* current, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  if (i < nhalf) {
    current[i] /= nhalf;
    newTE[i] = (original[i] < current[i]) ? current[i] : original[i];
  }
}

// kernel for testing the true envelope condition
__global__
void test(float* nonSmoothed, float* smoothed, int* mask, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int diff;
  if (i < nhalf) {
    diff = fabs(nonSmoothed[i] - smoothed[i]/nhalf);
    mask[i] = (diff > 0.23) ? 1 : 0;   // WHAT THRESHOLD TO USE??
  }
}
```

```
static int free_device(CSOUND* csound, void* pp){
  CUDAPVSHIFT* p = (CUDAPVSHIFT*) pp;
  cudaFree(p->deviceInput);
  cudaFree(p->deviceOutput);
  cudaFree(p->deviceEnv);
  cudaFree(p->deviceCepstrum);
  cudaFree(p->deviceTrueEnv);
  cudaFree(p->deviceSmoothTrueEnv);
  cudaFree(p->deviceMask);
  cufftDestroy(p->forwardPlan);
  cufftDestroy(p->inversePlan);
  return OK;
}

static int cudapvshiftset(CSOUND *csound, CUDAPVSHIFT *p)
{
  int N = p->fin->N;
  int Nhalf = N>>1;
  int size = (N+2) * sizeof(float);
  int smallSize = ((Nhalf>>1)+1) * sizeof(cufftComplex);
  int maxBlockDim;
  int SMcount;
  int totNumThreads = Nhalf;
  cudaError_t error;

  // get info about device
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound, "cudapvshift running on device %s
                  (capability %d.%d)\n", deviceProp.name,
                  deviceProp.major, deviceProp.minor);

  // create a cuFFT plan to use later
  cufftPlan1d(&p->forwardPlan, Nhalf, CUFFT_R2C, 1);
  cufftPlan1d(&p->inversePlan, Nhalf, CUFFT_C2R, 1);
  cufftSetCompatibilityMode(p->forwardPlan, CUFFT_COMPATIBILITY_NATIVE);
  cufftSetCompatibilityMode(p->inversePlan, CUFFT_COMPATIBILITY_NATIVE);

  // device memory allocation and copy from host (input data)
  error = cudaMalloc(&p->deviceInput, size);
  handleCudaError(csound, error);

  // device memory allocation and set to zero (output data to be computed)
  error = cudaMalloc(&p->deviceOutput, size);
  handleCudaError(csound, error);
  cudaMemset(p->deviceOutput,0,size);

  // device memory allocation and set to zero
  // (amplitude spectral envelope, approx half the length)
  error = cudaMalloc(&p->deviceEnv, smallSize);
  handleCudaError(csound, error);
  cudaMemset(p->deviceEnv,0,smallSize);

  // device memory allocation and set to zero
  // (cepstrum, approx half the length)
  error = cudaMalloc(&p->deviceCepstrum, smallSize);
  handleCudaError(csound, error);
  cudaMemset(p->deviceCepstrum,0,smallSize);
  // device memory allocation and set to zero
  // (true envelope, approx half the length)
  error = cudaMalloc(&p->deviceTrueEnv, smallSize);
  handleCudaError(csound, error);
  cudaMemset(p->deviceTrueEnv,0,smallSize);

  // device memory allocation and set to zero
  // (smoothed true envelope, approx half the length)
  error = cudaMalloc(&p->deviceSmoothTrueEnv, smallSize);
  handleCudaError(csound, error);
  cudaMemset(p->deviceSmoothTrueEnv,0,smallSize);

  // device memory allocation and set to one (mask)
  error = cudaMalloc(&p->deviceMask, Nhalf*sizeof(int));
  handleCudaError(csound, error);
  cudaMemset(p->deviceMask,1,Nhalf*sizeof(int));

  if (UNLIKELY(p->fin == p->fout))
    csound->Warning(csound, Str("Unsafe to have same fsig
                                 as in and out"));

  if (p->fout->frame.auxp == NULL ||
      p->fout->frame.size < sizeof(float) * (N + 2))  /* RWD MUST be
                                                         32bit */
    csound->AuxAlloc(csound, (N + 2) * sizeof(float), &p->fout->frame);
  else memset(p->fout->frame.auxp, 0, (N+2)*sizeof(float));

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N = N;
  p->fout->overlap = p->fin->overlap;
  p->fout->winsize = p->fin->winsize;
  p->fout->wintype = p->fin->wintype;
  p->fout->format = p->fin->format;
  p->fout->framecount = 1;
  p->lastframe = 0;
  p->fout->sliding = p->fin->sliding;
  p->fout->NB = p->fin->NB;

  csound->RegisterDeinitCallback(csound, p, free_device);

  return OK;
}

static int cudapvshift(CSOUND *csound, CUDAPVSHIFT *p)
{
  int     i, N = p->fout->N;
  int Nhalf = N>>1;
  int framelength = N+2;
  int size = framelength * sizeof(float);
  float   max = 0.0f;
  MYFLT   pshift = (MYFLT) *p->kshift;
  int     cshift = (int) (pshift * N * (1.0/csound->GetSr(csound)));
  int     lowest = abs((int) (*p->lowest * N *
                              (1.0/csound->GetSr(csound))));
  int     keepform = (int) *p->keepform;
  float   g = (float) *p->gain;
  float   *fin = (float *) p->fin->frame.auxp;
  float   *fout = (float *) p->fout->frame.auxp;
  int coefs = (int) *p->coefs;
```

```
cufftComplex* cufftEnv;
cufftComplex* cufftCepstrum;
cufftComplex* cufftTrueEnv;
cufftComplex* cufftSmoothTrueEnv;

thrust::device_ptr<float> dev_ptr1 =
  thrust::device_pointer_cast(p->deviceOutput);
thrust::device_ptr<float> dev_ptr2 =
  thrust::device_pointer_cast(p->deviceEnv);
thrust::device_ptr<float> dev_ptr3 =
  thrust::device_pointer_cast(p->deviceSmoothTrueEnv);
thrust::device_ptr<int> dev_ptr4 =
  thrust::device_pointer_cast(p->deviceMask);

if (UNLIKELY(fout == NULL)) goto err1;

if (p->lastframe < p->fin->framecount) {

  lowest = lowest ? (lowest > Nhalf ? Nhalf : lowest<<1) : 2;

  cudaMemcpy(p->deviceInput,fin,size,cudaMemcpyHostToDevice);

  if (keepform == 0) {
    // resets the output:
    thrust::fill(dev_ptr1, dev_ptr1+framelength, -1.0f);
    // freq shifting:
    freqShiftBasic<<<p->gridSize,p->blockSize>>>(p->deviceInput,
                                                  p->deviceOutput,
                                                  pshift, cshift,
                                                  lowest, Nhalf);
    // apply gain to all amplitudes:
    fixPVandGain<<<p->gridSize,p->blockSize>>>(p->deviceOutput, g,
                                                lowest,
                                                framelength);
  }

  else if (keepform==1) {
    if (coefs<1) coefs = 80;

    // resets the output:
    thrust::fill(dev_ptr1, dev_ptr1+framelength, -1.0f);
    takeLog<<<p->gridSize,p->blockSize>>>(p->deviceInput,
                                           p->deviceEnv, Nhalf);

    cufftEnv = (cufftComplex*) p->deviceEnv;
    cufftCepstrum = (cufftComplex*) p->deviceCepstrum;

    // take the fft of the log of the spectral envelope:
    if(cufftExecR2C(p->forwardPlan,(cufftReal*)cufftEnv,
                    cufftCepstrum)!= CUFFT_SUCCESS)
      csound->Message(csound, "CUDA FFT error\n");
    if (cudaDeviceSynchronize() != cudaSuccess)
      csound->Message(csound,"CUDA error: Failed to synchronize\n");

    // liftering stage: keep only low quefrency coefficients:
    lifter<<<p->gridSize,p->blockSize>>>(p->deviceCepstrum, coefs,
                                          Nhalf);
    // take the inverse fft of the liftered cepstrum...
    if(cufftExecC2R(p->inversePlan,cufftCepstrum,
                    (cufftReal*)cufftEnv)!= CUFFT_SUCCESS)
      csound->Message(csound, "CUDA FFT error\n");
    if (cudaDeviceSynchronize() != cudaSuccess)
      csound->Message(csound,"CUDA error: Failed to synchronize\n");

    // scale the result of the inverse transform
    // and exponentiate to go back to true amplitudes:
    expon<<<p->gridSize,p->blockSize>>>(p->deviceEnv, Nhalf);

    // find maximum amp in spectral envelope:
    max = *(thrust::max_element(dev_ptr2, dev_ptr2+Nhalf));

    // normalize spectral env and freq scale the input:
    freqShiftFormant<<<p->gridSize,p->blockSize>>>(p->deviceInput,
                                                   p->deviceOutput,
                                                   p->deviceEnv,
                                                   pshift, cshift,
                                                   lowest, max,
                                                   Nhalf);
    // apply gain to all amplitudes:
    fixPVandGain<<<p->gridSize,p->blockSize>>>(p->deviceOutput,
                                               g, lowest,
                                               framelength);
  }

  else if (keepform==2) {
    int cond = 1;
    if (coefs<1) coefs = 80;
    cufftEnv = (cufftComplex*) p->deviceEnv;
    cufftCepstrum = (cufftComplex*) p->deviceCepstrum;
    cufftTrueEnv = (cufftComplex*) p->deviceTrueEnv;
    cufftSmoothTrueEnv = (cufftComplex*) p->deviceSmoothTrueEnv;

    // resets the output:
    thrust::fill(dev_ptr1, dev_ptr1+framelength, -1.0f);
    takeLog<<<p->gridSize,p->blockSize>>>(p->deviceInput,
                                          p->deviceEnv, Nhalf);

    // loop initialization stage:
    // smooth the original log spectral envelope:
    // take the fft of the log of the spectral envelope:
    if(cufftExecR2C(p->forwardPlan,(cufftReal*)cufftEnv,
                    cufftCepstrum)!= CUFFT_SUCCESS)
      csound->Message(csound, "CUDA FFT error\n");
    if (cudaDeviceSynchronize() != cudaSuccess)
      csound->Message(csound,"CUDA error: Failed to synchronize\n");

    // liftering: keep only low quefrency coefficients:
    lifter<<<p->gridSize,p->blockSize>>>(p->deviceCepstrum,
                                         coefs, Nhalf);

    // take the inverse fft of the liftered cepstrum:
    if(cufftExecC2R(p->inversePlan,cufftCepstrum,
                    (cufftReal*)cufftSmoothTrueEnv)!= CUFFT_SUCCESS)
      csound->Message(csound, "CUDA FFT error\n");
    if (cudaDeviceSynchronize() != cudaSuccess)
      csound->Message(csound,"CUDA error: Failed to synchronize\n");
    while(cond) {
      update<<<p->gridSize,p->blockSize>>>(p->deviceEnv,
                                           p->deviceTrueEnv,
                                           p->deviceSmoothTrueEnv,
                                           Nhalf);

      // take the fft of the true envelope:
      if(cufftExecR2C(p->forwardPlan,(cufftReal*)cufftTrueEnv,
                      cufftCepstrum)!= CUFFT_SUCCESS)
        csound->Message(csound, "CUDA FFT error\n");
      if (cudaDeviceSynchronize() != cudaSuccess)
        csound->Message(csound,"CUDA error: Failed to
                                synchronize\n");

      // liftering: keep only low quefrency coefficients:
      lifter<<<p->gridSize,p->blockSize>>>(p->deviceCepstrum,
                                           coefs, Nhalf);

      // take the inverse fft of the liftered cepstrum:
      if(cufftExecC2R(p->inversePlan,cufftCepstrum,
                     (cufftReal*)cufftSmoothTrueEnv)!=
                     CUFFT_SUCCESS)
        csound->Message(csound, "CUDA FFT error\n");
      if (cudaDeviceSynchronize() != cudaSuccess)
        csound->Message(csound,"CUDA error: Failed to
                                synchronize\n");

      test<<<p->gridSize,p->blockSize>>>(p->deviceTrueEnv,
                                         p->deviceSmoothTrueEnv,
                                         p->deviceMask, Nhalf);
      if((thrust::reduce(dev_ptr4, dev_ptr4+Nhalf)) == 0)
        cond = 0;
    }

    // scale the result of the inverse transform
    // and exponentiate to go back to true amplitudes:
    expon<<<p->gridSize,p->blockSize>>>(p->deviceSmoothTrueEnv,
                                        Nhalf);

    // find maximum amp in spectral envelope:
    max = *(thrust::max_element(dev_ptr3, dev_ptr3+Nhalf));

    // normalize spectral env and freq scale the input:
    freqShiftFormant<<<p->gridSize,p->blockSize>>>(p->deviceInput,
                                              p->deviceOutput,
                                              p->deviceSmoothTrueEnv,
                                              pshift, cshift,
                                              lowest, max, Nhalf);
    // apply gain to all amplitudes:
    fixPVandGain<<<p->gridSize,p->blockSize>>>(p->deviceOutput,
                                               g, lowest, framelength);
  }
  cudaMemcpy(fout, p->deviceOutput, size, cudaMemcpyDeviceToHost);
  fout[0] = fin[0];   // keep original DC amplitude
  fout[N] = fin[N];   // keep original Nyquist amplitude
  p->fout->framecount = p->lastframe = p->fin->framecount;
}

return OK;
err1:
  return csound->PerfError(csound, p->h.insdshead,
                           Str("cudapvshift: not initialised"));
}

static OENTRY localops[] = {
{"cudapvshift", sizeof(CUDAPVSHIFT),0, 3, "f", "fxkOPO",
(SUBR) cudapvshiftset, (SUBR) cudapvshift}
};

extern "C" {
LINKAGE
}
```

cudapvscale2

```
#include <csdl.h>
#include <pstream.h>
#include <cufft.h>
#include <thrust/extrema.h>
#include <thrust/device_ptr.h>
#include <thrust/fill.h>
#include <thrust/reduce.h>

typedef struct _cudapvshift2 {
  OPDS    h;
  PVSDAT  *fout;
  PVSDAT  *fin;
  MYFLT   *kshift;
  MYFLT   *lowest;
  MYFLT   *keepform;
  MYFLT   *gain;
  MYFLT   *coefs;
  AUXCH   fenv, ceps;
  float*  deviceEnv;   // pointer to device memory
                       // (amplitude spectral envelope frame)
  float*  deviceCepstrum;   // pointer to device memory (cepstrum frame)
  float*  deviceTrueEnv;   // pointer to device memory  (true envelope)
  float*  deviceSmoothTrueEnv;   // pointer to device memory
                                 // (true envelope, smoothed)
  int*    deviceMask;   // pointer to device memory
                        // (boolean mask for condition checking)
  int  gridSize;   // number of blocks in the grid (1D)
  int  blockSize;   // number of threads in one block (1D)
  cufftHandle forwardPlan;   // forward cuFFT plan
  cufftHandle inversePlan;   // inverse cuFFT plan
  uint32  lastframe;
} CUDAPVSHIFT2;

// kernel for frequency shifting
// without formant keeping (part one)
__global__
void freqShiftBasic(float* input, float* output, MYFLT shift,
                    int shiftChan, int lowestIndx, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  int lowestChan = lowestIndx>>1;
  int N = nhalf<<1;
  int newchan;
  if (i < lowestChan) {
    // leave PV data as it is below a certain channel:
    output[j] = input[j];
    output[j+1] = input[j+1];
  }
  if (i >= lowestChan && i < nhalf) {
    newchan = (i + shiftChan) << 1;
    if (newchan < N && newchan >= lowestIndx) {
      // move amplitude data to new positions:
      output[newchan] = input[j];
      // change bin frequencies:
      output[newchan+1] = (float) (input[j+1] + shift);
    }
  }
}

// kernel for frequency shifting
// (part two, with or without formant keeping)
__global__
void fixPVandGain(float* input, float* output, float gain,
                  int lowestIndx, int length) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  if (j >= lowestIndx && j < length) {
    if (isnan(output[j]))
    output[j] = 0.0f;  // set to zero any invalid amplitude
    if (output[j+1] == -1.0f) {
      output[j] = 0.0f;   // set to zero the amplitude
                          // related to any undefined frequency
    }
    else
    output[j] *= gain;   // scale all amplitudes
                         // by the gain factor
  }
  if (j == 0) output[0] = input[0];   // keep original DC amplitude
  if (j == length-2) output[length-2] = input[length-2]; // keep original
                                                          // Nyquist
    amplitude
}

__global__
void takeLog(float* input, float* env, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  if (i < nhalf) {
    env[i] = log(input[j] > 0.0 ? input[j] : 1e-20); // take the log
                                                      // of the amplitudes
  }
}

__global__
void lifter(float* cepstrum, int nCoefs, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int k = i + nCoefs;
  if (k < nhalf+2-nCoefs) {
    cepstrum[k] = 0.0;   // kill all the cepstrum coefficients
                         // above nCoefs
  }
}
```

```
__global__
void expon(float* env, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  if (i < nhalf) {
    env[i] = exp(env[i]/nhalf);   // exponentiate
  }
}

// kernel for frequency shifting with formant keeping (part one)
__global__
void freqShiftFormant(float* input, float* output, float* env,
                      MYFLT shift, int shiftChan, int lowestIndx,
                      float maxAmp, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  int lowestChan = lowestIndx>>1;
  int N = nhalf<<1;
  int newchan;
  if (i < lowestChan) {
    // leave PV data as it is below a certain channel:
    output[j] = input[j];
    output[j+1] = input[j+1];
  }
  else if (i < nhalf) {
    env[i] /= maxAmp;   // normalize the spectral envelope
    input[j] /= env[i];   // equalize the original amplitudes so that
   formant shaping is more effective
    newchan = (i + shiftChan) << 1;
    if (newchan < N && newchan >= lowestIndx) {
      // move amp data to new positions and weight by normalized env:
      output[newchan] = input[j]*env[newchan>>1]*0.9;
      // change bin frequencies:
      output[newchan+1] = (float)(input[j+1] + shift);
    }
  }
}

// after completing the inverse fft,
// this kernel updates the true envelope:
// for each bin, the max of input and
// smoothed spectral envelopes is taken
__global__
void update(float* original, float* newTE, float* current, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  if (i < nhalf) {
    current[i] /= nhalf;
    newTE[i] = (original[i] < current[i]) ? current[i] : original[i];
  }
}

// kernel for testing the true envelope condition
__global__
void test(float* nonSmoothed, float* smoothed, int* mask, int nhalf) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int diff;
  if (i < nhalf) {
    diff = fabs(nonSmoothed[i] - smoothed[i]/nhalf);
    mask[i] = (diff > 0.23) ? 1 : 0;   // WHAT THRESHOLD TO USE??
  }
}
```

```
static int free_device(CSOUND* csound, void* pp){
  CUDAPVSHIFT2* p = (CUDAPVSHIFT2*) pp;
  cudaFree(p->fout->frame.auxp);
  cudaFree(p->deviceEnv);
  cudaFree(p->deviceCepstrum);
  cudaFree(p->deviceTrueEnv);
  cudaFree(p->deviceSmoothTrueEnv);
  cudaFree(p->deviceMask);
  cufftDestroy(p->forwardPlan);
  cufftDestroy(p->inversePlan);
  return OK;
}

static int cudapvshift2set(CSOUND *csound, CUDAPVSHIFT2 *p)
{
  int N = p->fin->N;
  int Nhalf = N>>1;
  int size = (N+2) * sizeof(float);
  int smallSize = ((Nhalf>>1)+1) * sizeof(cufftComplex);
  int maxBlockDim;
  int SMcount;
  int totNumThreads = Nhalf;
  cudaError_t error;

  // get info about device
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound, "cudapvshift2 running on device %s
                  (capability %d.%d)\n", deviceProp.name,
                  deviceProp.major, deviceProp.minor);

  // create a cuFFT plan to use later
  cufftPlan1d(&p->forwardPlan, Nhalf, CUFFT_R2C, 1);
  cufftPlan1d(&p->inversePlan, Nhalf, CUFFT_C2R, 1);
  cufftSetCompatibilityMode(p->forwardPlan, CUFFT_COMPATIBILITY_NATIVE);
  cufftSetCompatibilityMode(p->inversePlan, CUFFT_COMPATIBILITY_NATIVE);

  // device memory allocation and set to zero
  // (amplitude spectral envelope, approx half the length)
  error = cudaMalloc(&p->deviceEnv, smallSize);
  handleCudaError(csound, error);
  cudaMemset(p->deviceEnv,0,smallSize);

  // device memory allocation and set to zero
  // (cepstrum, approx half the length)
  error = cudaMalloc(&p->deviceCepstrum, smallSize);
  handleCudaError(csound, error);
  cudaMemset(p->deviceCepstrum,0,smallSize);

  // device memory allocation and set to zero
  // (true envelope, approx half the length)
  error = cudaMalloc(&p->deviceTrueEnv, smallSize);
  handleCudaError(csound, error);
  cudaMemset(p->deviceTrueEnv,0,smallSize);
  // device memory allocation and set to zero
  // (smoothed true envelope, approx half the length)
  error = cudaMalloc(&p->deviceSmoothTrueEnv, smallSize);
  handleCudaError(csound, error);
  cudaMemset(p->deviceSmoothTrueEnv,0,smallSize);

  // device memory allocation and set to one (mask)
  error = cudaMalloc(&p->deviceMask, Nhalf*sizeof(int));
  handleCudaError(csound, error);
  cudaMemset(p->deviceMask,1,Nhalf*sizeof(int));

  if (UNLIKELY(p->fin == p->fout))
    csound->Warning(csound, Str("Unsafe to have same fsig
                             as in and out"));

  if (p->fout->frame.auxp == NULL ||
      p->fout->frame.size < sizeof(float) * (N + 2))  /* RWD MUST
                                                          be 32bit */
    AuxCudaAlloc(size, &p->fout->frame);
  else cudaMemset(p->fout->frame.auxp, 0, size);

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N = N;
  p->fout->overlap = p->fin->overlap;
  p->fout->winsize = p->fin->winsize;
  p->fout->wintype = p->fin->wintype;
  p->fout->format = p->fin->format;
  p->fout->framecount = 1;
  p->lastframe = 0;
  p->fout->sliding = 0;
  p->fout->NB = p->fin->NB;

  csound->RegisterDeinitCallback(csound, p, free_device);

  return OK;
}

static int cudapvshift2(CSOUND *csound, CUDAPVSHIFT2 *p)
{
  int     i, N = p->fout->N;
  int Nhalf = N>>1;
  int framelength = N+2;
  float   max = 0.0f;
  MYFLT   pshift = (MYFLT) *p->kshift;
  int     cshift = (int) (pshift * N * (1.0/csound->GetSr(csound)));
  int     lowest = abs((int) (*p->lowest * N *
                           (1.0/csound->GetSr(csound))));
  int     keepform = (int) *p->keepform;
  float   g = (float) *p->gain;
  float   *fin = (float *) p->fin->frame.auxp;
  float   *fout = (float *) p->fout->frame.auxp;
  int coefs = (int) *p->coefs;

  cufftComplex* cufftEnv;
  cufftComplex* cufftCepstrum;
  cufftComplex* cufftTrueEnv;
  cufftComplex* cufftSmoothTrueEnv;
```

```
thrust::device_ptr<float> dev_ptr1 =
  thrust::device_pointer_cast(fout);
thrust::device_ptr<float> dev_ptr2 =
  thrust::device_pointer_cast(p->deviceEnv);
thrust::device_ptr<float> dev_ptr3 =
  thrust::device_pointer_cast(p->deviceSmoothTrueEnv);
thrust::device_ptr<int> dev_ptr4 =
  thrust::device_pointer_cast(p->deviceMask);

if (UNLIKELY(fout == NULL)) goto err1;

if (p->lastframe < p->fin->framecount) {

  lowest = lowest ? (lowest > Nhalf ? Nhalf : lowest<<1) : 2;

  if (keepform == 0) {
    // resets the output:
    thrust::fill(dev_ptr1, dev_ptr1+framelength, -1.0f);
    // freq shifting
    freqShiftBasic<<<p->gridSize,p->blockSize>>>(fin, fout, pshift,
                                             cshift, lowest,
                                             Nhalf);
    // apply gain to all amplitudes:
    fixPVandGain<<<p->gridSize,p->blockSize>>>(fin, fout,
                                            g, lowest, framelength);
  }

  else if (keepform==1) {
    if (coefs<1) coefs = 80;

    // resets the output:
    thrust::fill(dev_ptr1, dev_ptr1+framelength, -1.0f);
    takeLog<<<p->gridSize,p->blockSize>>>(fin, p->deviceEnv, Nhalf);

    cufftEnv = (cufftComplex*) p->deviceEnv;
    cufftCepstrum = (cufftComplex*) p->deviceCepstrum;

    // take the fft of the log of the spectral envelope:
    if(cufftExecR2C(p->forwardPlan,(cufftReal*)cufftEnv,
       cufftCepstrum)!= CUFFT_SUCCESS)
      csound->Message(csound, "CUDA FFT error\n");
    if (cudaDeviceSynchronize() != cudaSuccess)
      csound->Message(csound,"CUDA error: Failed to synchronize\n");

    // liftering stage: keep only low quefrency coefficients:
    lifter<<<p->gridSize,p->blockSize>>>(p->deviceCepstrum,
                                     coefs, Nhalf);

    // take the inverse fft of the liftered cepstrum:
    if(cufftExecC2R(p->inversePlan,cufftCepstrum,
       (cufftReal*)cufftEnv)!= CUFFT_SUCCESS)
      csound->Message(csound, "CUDA FFT error\n");
    if (cudaDeviceSynchronize() != cudaSuccess)
      csound->Message(csound,"CUDA error: Failed to synchronize\n");

    // scale the result of the inverse transform
    // and exponentiate to go back to true amplitudes:
    expon<<<p->gridSize,p->blockSize>>>(p->deviceEnv, Nhalf);

    // find maximum amp in spectral envelope:
    max = *(thrust::max_element(dev_ptr2, dev_ptr2+Nhalf));
    // normalize spectral env and freq scale the input:
    freqShiftFormant<<<p->gridSize,p->blockSize>>>(fin, fout,
                                            p->deviceEnv,
                                            pshift, cshift,
                                            lowest, max,
                                            Nhalf);
    // apply gain to all amplitudes:
    fixPVandGain<<<p->gridSize,p->blockSize>>>(fin, fout, g,
                                        lowest, framelength);
  }

  else if (keepform==2) {
    int cond = 1;
    if (coefs<1) coefs = 80;
    cufftEnv = (cufftComplex*) p->deviceEnv;
    cufftCepstrum = (cufftComplex*) p->deviceCepstrum;
    cufftTrueEnv = (cufftComplex*) p->deviceTrueEnv;
    cufftSmoothTrueEnv = (cufftComplex*) p->deviceSmoothTrueEnv;

    // resets the output:
    thrust::fill(dev_ptr1, dev_ptr1+framelength, -1.0f);
    takeLog<<<p->gridSize,p->blockSize>>>(fin, p->deviceEnv, Nhalf);

    // loop initialization stage:
    // smooth the original log spectral envelope:
    // take the fft of the log of the spectral envelope:
    if(cufftExecR2C(p->forwardPlan,(cufftReal*)cufftEnv,
                    cufftCepstrum)!= CUFFT_SUCCESS)
      csound->Message(csound, "CUDA FFT error\n");
    if (cudaDeviceSynchronize() != cudaSuccess)
      csound->Message(csound,"CUDA error: Failed to synchronize\n");

    // liftering: keep only low quefrency coefficients:
    lifter<<<p->gridSize,p->blockSize>>>(p->deviceCepstrum, coefs,
                                     Nhalf);

    // take the inverse fft of the liftered cepstrum:
    if(cufftExecC2R(p->inversePlan,cufftCepstrum,
                    (cufftReal*)cufftSmoothTrueEnv)!= CUFFT_SUCCESS)
      csound->Message(csound, "CUDA FFT error\n");
    if (cudaDeviceSynchronize() != cudaSuccess)
      csound->Message(csound,"CUDA error: Failed to synchronize\n");

    while(cond) {
      update<<<p->gridSize,p->blockSize>>>(p->deviceEnv,
                                     p->deviceTrueEnv,
                                     p->deviceSmoothTrueEnv,
                                     Nhalf);

      // take the fft of the true envelope:
      if(cufftExecR2C(p->forwardPlan,(cufftReal*)cufftTrueEnv,
                      cufftCepstrum)!= CUFFT_SUCCESS)
        csound->Message(csound, "CUDA FFT error\n");
      if (cudaDeviceSynchronize() != cudaSuccess)
        csound->Message(csound,"CUDA error:
                        Failed to synchronize\n");

      // liftering: keep only low quefrency coefficients:
      lifter<<<p->gridSize,p->blockSize>>>(p->deviceCepstrum,
                                       coefs, Nhalf);
        // take the inverse fft of the liftered cepstrum:
        if(cufftExecC2R(p->inversePlan,cufftCepstrum,
                        (cufftReal*)cufftSmoothTrueEnv)!=
                        CUFFT_SUCCESS)
          csound->Message(csound, "CUDA FFT error\n");
        if (cudaDeviceSynchronize() != cudaSuccess)
          csound->Message(csound,"CUDA error:
                                  Failed to synchronize\n");

        test<<<p->gridSize,p->blockSize>>>(p->deviceTrueEnv,
                                           p->deviceSmoothTrueEnv,
                                           p->deviceMask, Nhalf);
        if((thrust::reduce(dev_ptr4, dev_ptr4+Nhalf)) == 0)
          cond = 0;
      }

      // scale the result of the inverse transform
      // and exponentiate to go back to true amplitudes:
      expon<<<p->gridSize,p->blockSize>>>(p->deviceSmoothTrueEnv,
                                          Nhalf);

      // find maximum amp in spectral envelope:
      max = *(thrust::max_element(dev_ptr3, dev_ptr3+Nhalf));

      // normalize spectral env and freq scale the input:
      freqShiftFormant<<<p->gridSize,p->blockSize>>>(fin, fout,
                                                     p->deviceSmoothTrueEnv,
                                                     pshift, cshift,
                                                     lowest, max,
                                                     Nhalf);
      // apply gain to all amplitudes:
      fixPVandGain<<<p->gridSize,p->blockSize>>>(fin, fout, g,
                                                 lowest, framelength);
    }

    p->fout->framecount = p->lastframe = p->fin->framecount;
  }

  return OK;

  err1:
    return csound->PerfError(csound, p->h.insdshead,
                             Str("cudapvshift2: not initialised"));
}

static OENTRY localops[] = {
  {"cudapvshift2", sizeof(CUDAPVSHIFT2),0, 3, "f", "fxkOPO",
   (SUBR) cudapvshift2set, (SUBR) cudapvshift2}
};

extern "C" {
  LINKAGE
}
```

Smooth Module

cudapvsmooth

```
#include <csdl.h>
#include <pstream.h>

typedef struct _cudapvsmooth {
  OPDS    h;
  PVSDAT  *fout;
  PVSDAT  *fin;
  MYFLT   *kfra;
  MYFLT   *kfrf;
  // AUXCH   del;  // not needed anymore
  float*  deviceInput;   // pointer to device memory (input frame)
  float*  deviceOutput;   // pointer to device memory (output frame)
  int  gridSize;   // number of blocks in the grid (1D)
  int  blockSize;   // number of threads in one block (1D)
  uint32  lastframe;
} CUDAPVSMOOTH;

// kernel for smoothing the time evolution functions of each channel
// (both amp and freq)
__global__
void smoothing(float* input, float* output, double alpha,
               double beta, int length) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  if (j < length) {
    output[j] = (float) (input[j] * (1.0 + alpha) - output[j] * alpha);
    output[j+1] = (float) (input[j+1] * (1.0 + beta) -
                           output[j+1] * beta);
  }
}

static int free_device(CSOUND* csound, void* pp){
  CUDAPVSMOOTH* p = (CUDAPVSMOOTH*) pp;
  cudaFree(p->deviceInput);
  cudaFree(p->deviceOutput);
  return OK;
}

static int cudapvsmoothset(CSOUND *csound, CUDAPVSMOOTH *p)
{
  int N = p->fin->N;
  int size = (N+2) * sizeof(float);
  int maxBlockDim;
  int SMcount;
  int totNumThreads = (N+2)>>1;
  cudaError_t error;

  // get info about device
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound, "cudapvsmooth running on device %s
                         (capability %d.%d)\n", deviceProp.name,
                         deviceProp.major, deviceProp.minor);
  // device memory allocation (input data)
  error = cudaMalloc(&p->deviceInput, size);
  handleCudaError(csound, error);

  // device memory allocation and set to zero (output data to be computed)
  error = cudaMalloc(&p->deviceOutput, size);
  handleCudaError(csound, error);
  cudaMemset(p->deviceOutput,0,size);

  if (UNLIKELY(p->fin == p->fout))
    csound->Warning(csound, Str("Unsafe to have same fsig
                                as in and out"));
  p->fout->NB = (N/2)+1;
  p->fout->sliding = 0;
  if (p->fout->frame.auxp == NULL ||
      p->fout->frame.size < sizeof(float) * (N + 2))
    csound->AuxAlloc(csound, (N + 2) * sizeof(float), &p->fout->frame);

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N = N;
  p->fout->overlap = p->fin->overlap;
  p->fout->winsize = p->fin->winsize;
  p->fout->wintype = p->fin->wintype;
  p->fout->format = p->fin->format;
  p->fout->framecount = 1;
  p->lastframe = 0;

  if (UNLIKELY(!(p->fout->format == PVS_AMP_FREQ) ||
             (p->fout->format == PVS_AMP_PHASE)))
    return csound->InitError(csound, Str("cudapvsmooth: signal format
                                     must be amp-phase or amp-freq."));

  csound->RegisterDeinitCallback(csound, p, free_device);

  return OK;
}

static int cudapvsmooth(CSOUND *csound, CUDAPVSMOOTH *p) {
  int N = p->fout->N;
  int framelength = N+2;
  int size = framelength * sizeof(float);
  double ffa = (double) *p->kfra;  // cutoff frequency in fractions of PI
                                   // (for the amplitude stream)
  double ffr = (double) *p->kfrf;  // cutoff frequency in fractions of PI
                                   // (for the frequency stream)

  if (p->lastframe < p->fin->framecount) {
    float   *fout, *fin;
    double  costh1, costh2, coef1, coef2;
    fout = (float *) p->fout->frame.auxp;
    fin = (float *) p->fin->frame.auxp;

    cudaMemcpy(p->deviceInput,fin,size,cudaMemcpyHostToDevice);

    ffa = ffa < FL(0.0) ? FL(0.0) : (ffa > FL(1.0) ? FL(1.0) : ffa);
    ffr = ffr < FL(0.0) ? FL(0.0) : (ffr > FL(1.0) ? FL(1.0) : ffr);
    costh1 = 2.0 - cos(PI * ffa);
    costh2 = 2.0 - cos(PI * ffr);
    coef1 = sqrt(costh1 * costh1 - 1.0) - costh1;
    coef2 = sqrt(costh2 * costh2 - 1.0) - costh2;

    // channel by channel parallel filtering on the GPU
    // (both amp and freq):
    smoothing<<<p->gridSize,p->blockSize>>>(p->deviceInput,
                                            p->deviceOutput, coef1,
                                            coef2, framelength);

    cudaMemcpy(fout,p->deviceOutput,size,cudaMemcpyDeviceToHost);

    p->fout->framecount = p->lastframe = p->fin->framecount;
  }

  return OK;
}


static OENTRY localops[] = {
  {"cudapvsmooth", sizeof(CUDAPVSMOOTH),0, 3, "f", "fxx",
   (SUBR) cudapvsmoothset, (SUBR) cudapvsmooth, NULL}
};

extern "C" {
  LINKAGE
}
```

cudapvsmooth2

```
#include <csdl.h>
#include <pstream.h>

typedef struct _cudapvsmooth2 {
  OPDS    h;
  PVSDAT  *fout;
  PVSDAT  *fin;
  MYFLT   *kfra;
  MYFLT   *kfrf;
  int  gridSize;   // number of blocks in the grid (1D)
  int  blockSize;   // number of threads in one block (1D)
  uint32  lastframe;
} CUDAPVSMOOTH2;

// kernel for smoothing the time evolution functions of each channel
// (both amp and freq)
__global__
void smoothing(float* input, float* output, double alpha,
               double beta, int length) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int j = i<<1;
  if (j < length) {
    output[j] = (float) (input[j] * (1.0+alpha) - output[j] * alpha);
    output[j+1] = (float) (input[j+1] * (1.0+beta) - output[j+1] * beta);
  }
}
```

```
static int free_device(CSOUND* csound, void* pp){
  CUDAPVSMOOTH2* p = (CUDAPVSMOOTH2*) pp;
  cudaFree(p->fout->frame.auxp);
  return OK;
}

static int cudapvsmooth2set(CSOUND *csound, CUDAPVSMOOTH2 *p)
{
  int N = p->fin->N;
  int size = (N+2) * sizeof(float);
  int maxBlockDim;
  int SMcount;
  int totNumThreads = (N+2)>>1;

  // get info about device
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound, "cudapvsmooth2 running on device %s
                (capability %d.%d)\n", deviceProp.name,
                deviceProp.major, deviceProp.minor);

  if (UNLIKELY(p->fin == p->fout))
    csound->Warning(csound, Str("Unsafe to have same fsig
                               as in and out"));
  p->fout->NB = (N/2)+1;
  p->fout->sliding = 0;
  if (p->fout->frame.auxp == NULL ||
      p->fout->frame.size < sizeof(float) * (N + 2))
    AuxCudaAlloc(size, &p->fout->frame);

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N = N;
  p->fout->overlap = p->fin->overlap;
  p->fout->winsize = p->fin->winsize;
  p->fout->wintype = p->fin->wintype;
  p->fout->format = p->fin->format;
  p->fout->framecount = 1;
  p->lastframe = 0;

  csound->RegisterDeinitCallback(csound, p, free_device);

  return OK;
}

static int cudapvsmooth2(CSOUND *csound, CUDAPVSMOOTH2 *p) {
  int N = p->fout->N;
  int framelength = N+2;
  double ffa = (double) *p->kfra;  // cutoff frequency in fractions of PI
                                   // (for the amplitude stream)
  double ffr = (double) *p->kfrf;  // cutoff frequency in fractions of PI
                                   // (for the frequency stream)

  if (p->lastframe < p->fin->framecount) {
    float   *fout, *fin;
    double  costh1, costh2, coef1, coef2;
    fout = (float *) p->fout->frame.auxp;
    fin = (float *) p->fin->frame.auxp;
    ffa = ffa < FL(0.0) ? FL(0.0) : (ffa > FL(1.0) ? FL(1.0) : ffa);
    ffr = ffr < FL(0.0) ? FL(0.0) : (ffr > FL(1.0) ? FL(1.0) : ffr);
    costh1 = 2.0 - cos(PI * ffa);
    costh2 = 2.0 - cos(PI * ffr);
    coef1 = sqrt(costh1 * costh1 - 1.0) - costh1;
    coef2 = sqrt(costh2 * costh2 - 1.0) - costh2;

    // channel by channel parallel filtering on the GPU
    // (both amp and freq):
    smoothing<<<p->gridSize,p->blockSize>>>(fin, fout, coef1, coef2,
                                            framelength);

    p->fout->framecount = p->lastframe = p->fin->framecount;
  }

  return OK;
}

static OENTRY localops[] = {
  {"cudapvsmooth2", sizeof(CUDAPVSMOOTH2),0, 3, "f", "fxx",
    (SUBR) cudapvsmooth2set, (SUBR) cudapvsmooth2, NULL}
};

extern "C" {
  LINKAGE
}
```

Blur Module

cudapvsblur

```
#include <csdl.h>
#include <pstream.h>

#define SR (csound->GetSr(csound))

typedef struct _cudapvsblur{
  OPDS h;
  PVSDAT* fout;
  PVSDAT* fin;
  MYFLT* kdel;          // averaging window length (in sec)
  MYFLT* maxdel;        // maximum expected time
                        // for the averaging window (in sec)
  float* deviceOutput; // pointer to device memory (result frame)
  float* deviceMatrix; // pointer to decive memory
                        // (matrix to store current and past frames)
  int gridSize;         // number of blocks in the grid (1D)
  int blockSize;        // number of threads in one block (1D)
  MYFLT frpsec;         // frames per second
  int32 count;
  uint32 lastframe;
} CUDAPVSBLUR;

__global__
void initialize(float* matrix, float sr, int numFrames, int length) {
  int frame = blockIdx.y*blockDim.y + threadIdx.y;
  int chan = (blockIdx.x*blockDim.x + threadIdx.x) << 1;
  if ((frame < numFrames) && (chan < length)) {
    matrix[frame*length+chan] = 0.0f;
    matrix[frame*length+chan+1] = chan * sr / (length-2);
  }
}

// blur kernel
__global__
void blur(float* matrix, float* output, int firstFrame, int numFrames,
          int frameCount, int max, int length){
  int chan = (blockIdx.x*blockDim.x+ threadIdx.x)<<1;
  float amp = 0.0f;
  float freq = 0.0f;
  int frame;
  if (chan < length) {
    for (frame = firstFrame; frame != frameCount; frame = (frame+1)%max) {
      amp += matrix[frame*length+chan];
      freq += matrix[frame*length+chan+1];
    }
    output[chan] = (float) (amp / numFrames);
    output[chan+1] = (float) (freq / numFrames);
  }
}

static int free_device(CSOUND* csound, void* pp){
  CUDAPVSBLUR* p = (CUDAPVSBLUR*) pp;
  cudaFree(p->deviceOutput);
  cudaFree(p->deviceMatrix);
  return OK;
}
```

```
static int cudapvsblurset(CSOUND *csound, CUDAPVSBLUR *p)
{
  int32   N = p->fin->N;
  int     olap = p->fin->overlap;
  int     maxframes, framelength = N + 2;
  int size = (N+2) * sizeof(float);
  int bigSize;
  int maxBlockDim;
  int SMcount;
  int totNumThreads = (N+2)>>1;
  cudaError_t error;

  // get info about device
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound, "cudapvsblur running on device %s
                  (capability %d.%d)\n", deviceProp.name,
                  deviceProp.major, deviceProp.minor);

  // device memory allocation
  error = cudaMalloc(&p->deviceOutput, size);
  handleCudaError(csound, error);
  cudaMemset(p->deviceOutput,0,size);

  if (UNLIKELY(p->fin == p->fout))
    csound->Warning(csound, Str("Unsafe to have same fsig
                                 as in and out"));

  p->frpsec = SR / olap;

  maxframes = (int) (*p->maxdel * p->frpsec);  // max number of frames
                                               // considered, i.e. number
                                               // of rows

  bigSize = size * maxframes;

  // device memory allocation
  error = cudaMalloc(&p->deviceMatrix, bigSize);
  handleCudaError(csound, error);

  // REVISE THIS...(this might prevent memory coalescence)
  // temporary gridSize and blockSize, just for "initialize" kernel
  dim3 initBlockSize((int) maxBlockDim / maxframes, maxframes, 1);
  dim3 initGridSize((framelength*maxframes>>1) / maxBlockDim + 1, 1, 1);

  /*
  // REVISE THIS...(this should be a little better, NOT SURE)
  // NOTE: IF THESE SIZES ARE USED,
  //       THE "INITIALIZE" KERNEL NEEDS TO BE CHANGED!
  // temporary gridSize and blockSize, just for "initialize" kernel:
  dim3 initBlockSize(maxBlockDim, 1, 1);
  dim3 initGridSize((framelength*maxframes>>1) / maxBlockDim + 1, 1, 1);
  */

  initialize<<<initGridSize,initBlockSize>>>(p->deviceMatrix, SR,
                                             maxframes, framelength);
  if (p->fout->frame.auxp == NULL ||
      p->fout->frame.size < sizeof(float) * (N + 2))
    csound->AuxAlloc(csound, (N + 2) * sizeof(float), &p->fout->frame);

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N = N;
  p->fout->overlap = olap;
  p->fout->winsize = p->fin->winsize;
  p->fout->wintype = p->fin->wintype;
  p->fout->format = p->fin->format;
  p->fout->framecount = 1;
  p->lastframe = 0;
  p->count = 0;
  p->fout->sliding = p->fin->sliding;
  p->fout->NB = p->fin->NB;

  csound->RegisterDeinitCallback(csound, p, free_device);

  return OK;
}

static int cudapvsblur(CSOUND *csound, CUDAPVSBLUR *p)
{
  int32    N = p->fout->N, first, framelength = N + 2;
  int32    countr = p->count;
  int size = (N+2) * sizeof(float);
  int     delayframes = (int) (*p->kdel * p->frpsec);
  int     maxframes = (int) (*p->maxdel * p->frpsec);
  float   *fin = (float *) p->fin->frame.auxp;
  float   *fout = (float *) p->fout->frame.auxp;
  float   *delay = (float *) p->delframes.auxp;

  if (UNLIKELY(fout == NULL || delay == NULL)) goto err1;

  if (p->lastframe < p->fin->framecount) {

    delayframes = delayframes >= 0 ? (delayframes < maxframes ?
                                 delayframes : maxframes - 1) : 0;

    cudaMemcpy(p->deviceMatrix+(countr*framelength), fin, size,
               cudaMemcpyHostToDevice);

    if (delayframes) {
      if ((first = countr - delayframes) < 0)
        first += maxframes;

      blur<<<p->gridSize,p->blockSize>>>(p->deviceMatrix,
                                     p->deviceOutput, first,
                                     delayframes, countr,
                                     maxframes, framelength);

      cudaMemcpy(fout, p->deviceOutput, size, cudaMemcpyDeviceToHost);
    }
    else {
      memcpy(fout, fin, size);   // bypass blurring
    }

    p->fout->framecount = p->lastframe = p->fin->framecount;
    countr++;
    p->count = countr < maxframes ? countr : 0;
  }

  return OK;
  err1: return csound->PerfError(csound, p->h.insdshead,
                                Str("cudapvsblur: not initialised"));
}

static OENTRY localops[] = {
  {"cudapvsblur", sizeof(CUDAPVSBLUR),0, 3, "f", "fki",
   (SUBR) cudapvsblurset, (SUBR) cudapvsblur, NULL}
};

extern "C" {
  LINKAGE
}
```

cudapvsblur2

```
// To enhance performance, it might be a good idea not to use cudaMemcpy
// with DeviceToDevice specifier: instead, write a kernel just to copy and
// paste data from one location to the other. (it seems that cudaMemcpy is
// quite slow even when transferring data internally)

#include <csdl.h>
#include <pstream.h>

#define SR (csound->GetSr(csound))

typedef struct _cudapvsblur2{
  OPDS h;
  PVSDAT* fout;
  PVSDAT* fin;
  MYFLT* kdel;          // averaging window length (in sec)
  MYFLT* maxdel;        // maximum expected time for the averaging window (
   in sec)
  float* deviceMatrix; // pointer to decive memory
                       // (matrix to store current and
  // past frames)
  int gridSize;         // number of blocks in the grid (1D)
  int blockSize;        // number of threads in one block (1D)
  MYFLT frpsec;         // frames per second
  int32 count;
  uint32 lastframe;
} CUDAPVSBLUR2;

__global__
void initialize(float* matrix, float sr, int numFrames, int length) {
  int frame = blockIdx.y*blockDim.y + threadIdx.y;
  int chan = (blockIdx.x*blockDim.x + threadIdx.x) << 1;
  if ((frame < numFrames) && (chan < length)) {
    matrix[frame*length+chan] = 0.0f;
    matrix[frame*length+chan+1] = chan * sr / (length-2);
  }
}
```

```
// blurring kernel
__global__
void blur(float* matrix, float* output, int firstFrame, int numFrames,
          int frameCount, int max, int length){
  int chan = (blockIdx.x*blockDim.x+ threadIdx.x)<<1;
  float amp = 0.0f;
  float freq = 0.0f;
  int frame;
  if (chan < length) {
    for (frame = firstFrame; frame != frameCount; frame = (frame+1)%max) {
      amp += matrix[frame*length+chan];
      freq += matrix[frame*length+chan+1];
  }
    output[chan] = (float) (amp / numFrames);
    output[chan+1] = (float) (freq / numFrames);
  }
}

static int free_device(CSOUND* csound, void* pp){
  CUDAPVSBLUR2* p = (CUDAPVSBLUR2*) pp;
  cudaFree(p->fout->frame.auxp);
  cudaFree(p->deviceMatrix);
  return OK;
}

static int cudapvsblur2set(CSOUND *csound, CUDAPVSBLUR2 *p)
{
  int32   N = p->fin->N;
  int     olap = p->fin->overlap;
  int     maxframes, framelength = N + 2;
  int size = (N+2) * sizeof(float);
  int bigSize;
  int maxBlockDim;
  int SMcount;
  int totNumThreads = (N+2)>>1;
  cudaError_t error;

  // get info about device
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound, "cudapvsblur2 running on device %s
                  (capability %d.%d)\n", deviceProp.name,
                  deviceProp.major, deviceProp.minor);

  // device memory allocation
  //error = cudaMalloc(&p->deviceOutput, size);
  //handleCudaError(csound, error);
  //cudaMemset(p->deviceOutput,0,size);

  if (UNLIKELY(p->fin == p->fout))
    csound->Warning(csound, Str("Unsafe to have same fsig
                               as in and out"));

  p->frpsec = SR / olap;

  maxframes = (int) (*p->maxdel * p->frpsec);  // max number of frames
                                               // considered, i.e. number
                                               // of rows
  bigSize = size * maxframes;

  // device memory allocation
  error = cudaMalloc(&p->deviceMatrix, bigSize);
  handleCudaError(csound, error);

  // REVISE THIS...(this might prevent memory coalescence)
  // temporary gridSize and blockSize, just for "initialize" kernel
  dim3 initBlockSize((int) maxBlockDim / maxframes, maxframes, 1);
  dim3 initGridSize((framelength*maxframes>>1) / maxBlockDim + 1, 1, 1);

  /*
  // REVISE THIS...(this should be a little better, NOT SURE)
  // NOTE: IF THESE SIZES ARE USED,
  // THE "INITIALIZE" KERNEL NEEDS TO BE CHANGED!
  // temporary gridSize and blockSize, just for "initialize" kernel:
  dim3 initBlockSize(maxBlockDim, 1, 1);
  dim3 initGridSize((framelength*maxframes>>1) / maxBlockDim + 1, 1, 1);
  */

  initialize<<<initGridSize,initBlockSize>>>(p->deviceMatrix, SR,
                                             maxframes, framelength);

  if (p->fout->frame.auxp == NULL ||
      p->fout->frame.size < sizeof(float) * (N + 2))
    AuxCudaAlloc(size, &p->fout->frame);

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N = N;
  p->fout->overlap = olap;
  p->fout->winsize = p->fin->winsize;
  p->fout->wintype = p->fin->wintype;
  p->fout->format = p->fin->format;
  p->fout->framecount = 1;
  p->lastframe = 0;
  p->count = 0;
  p->fout->sliding = 0;
  p->fout->NB = p->fin->NB;

  csound->RegisterDeinitCallback(csound, p, free_device);

  return OK;
}

static int cudapvsblur2(CSOUND *csound, CUDAPVSBLUR2 *p)
{
  int32    N = p->fout->N, first, framelength = N + 2;
  int32    countr = p->count;
  int size = (N+2) * sizeof(float);
  int     delayframes = (int) (*p->kdel * p->frpsec);
  int     maxframes = (int) (*p->maxdel * p->frpsec);
  float   *fin = (float *) p->fin->frame.auxp;
  float   *fout = (float *) p->fout->frame.auxp;

  if (UNLIKELY(fout == NULL)) goto err1;
  if (p->lastframe < p->fin->framecount) {

    delayframes = delayframes >= 0 ? (delayframes < maxframes ?
                                      delayframes : maxframes - 1) : 0;

    cudaMemcpy(p->deviceMatrix+(countr*framelength), fin, size,
               cudaMemcpyDeviceToDevice);

    if (delayframes) {
      if ((first = countr - delayframes) < 0)
        first += maxframes;

      blur<<<p->gridSize,p->blockSize>>>(p->deviceMatrix, fout,
                                          first, delayframes, countr,
                                          maxframes, framelength);
    }
    else {
      // bypass blurring:
      cudaMemcpy(fout, fin, size, cudaMemcpyDeviceToDevice);
    }

    p->fout->framecount = p->lastframe = p->fin->framecount;
    countr++;
    p->count = countr < maxframes ? countr : 0;
  }

  return OK;
  err1: return csound->PerfError(csound, p->h.insdshead,
                                 Str("cudapvsblur2: not initialised"));
}

static OENTRY localops[] = {
  {"cudapvsblur2", sizeof(CUDAPVSBLUR2),0, 3, "f", "fki",
  (SUBR) cudapvsblur2set, (SUBR) cudapvsblur2, NULL}
};

extern "C" {
  LINKAGE
}
```

Mix Module

cudapvsmix

```
#include <csdl.h>
#include <pstream.h>

typedef struct _cudapvsmix {
  OPDS    h;
  PVSDAT  *fout;
  PVSDAT  *fa;
  PVSDAT  *fb;
  float*  devOutput;    // device memory pointer (output PV frame)
  float*  devFrameA;    // device memory pointer (PV frame #1)
  float*  devFrameB;    // device memory pointer (PV frame #2)
  int     gridSize;     // number of blocks in the grid (1D)
  int     blockSize;    // number of threads in one block (1D)
  uint32  lastframe;
} CUDAPVSMIX;

__global__
void mix(float* output, float* frameA, float* frameB, int chans) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int j = i<<1;
  if (i < chans) {
    int test = frameA[j] >= frameB[j];
    if (test) {
      output[j] = frameA[j];
      output[j+1] = frameA[j+1];
    }
    else {
      output[j] = frameB[j];
      output[j+1] = frameB[j+1];
    }
  }
}

static int fsigs_equal(const PVSDAT *f1, const PVSDAT *f2)
{
  if (
  (f1->sliding == f2->sliding) &&
  (f1->overlap == f2->overlap) &&
  (f1->winsize == f2->winsize) &&
  (f1->wintype == f2->wintype) &&     /* harsh, maybe... */
  (f1->N == f2->N) &&
  (f1->format == f2->format))

  return 1;
  return 0;
}

static int free_device(CSOUND* csound, void* pp){
  CUDAPVSMIX* p = (CUDAPVSMIX*) pp;
  cudaFree(p->devOutput);
  cudaFree(p->devFrameA);
  cudaFree(p->devFrameB);
  return OK;
}
```

```
static int cudapvsmixset(CSOUND *csound, CUDAPVSMIX *p)
{
  int32    N = p->fa->N;
  int framelength = N+2;
  int chans = framelength>>1;
  int size = framelength*sizeof(float);

  int maxBlockDim;
  int SMcount;
  int totNumThreads = chans;
  cudaError_t error;

  // get info about device
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound, "cudapvsmix running on device %s
                  (capability %d.%d)\n", deviceProp.name,
                  deviceProp.major, deviceProp.minor);

  // device memory allocation (PV frame A)
  error = cudaMalloc(&p->devFrameA, size);
  handleCudaError(csound, error);

  // device memory allocation (PV frame B)
  error = cudaMalloc(&p->devFrameB, size);
  handleCudaError(csound, error);

  // device memory allocation (PV output frame)
  error = cudaMalloc(&p->devOutput, size);
  handleCudaError(csound, error);
  cudaMemset(p->devOutput,0,size);

  p->fout->sliding = 0;

  if (p->fout->frame.auxp == NULL ||
      p->fout->frame.size < sizeof(float) * (N + 2))
    csound->AuxAlloc(csound, (N + 2) * sizeof(float), &p->fout->frame);

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N = N;
  p->fout->overlap = p->fa->overlap;
  p->fout->winsize = p->fa->winsize;
  p->fout->wintype = p->fa->wintype;
  p->fout->format = p->fa->format;
  p->fout->framecount = 1;
  p->lastframe = 0;
  if (UNLIKELY(!(p->fout->format == PVS_AMP_FREQ) ||
      (p->fout->format == PVS_AMP_PHASE)))
    return csound->InitError(csound, Str("cudapvsmix: signal format "
                                         "must be amp-phase or amp-freq."));

  csound->RegisterDeinitCallback(csound, p, free_device);

  return OK;
}
```

```
static int cudapvsmix(CSOUND *csound, CUDAPVSMIX *p)
{
  int32    N = p->fa->N;
  int framelength = N+2;
  int chans = framelength>>1;
  int size = framelength*sizeof(float);
  float* fout = (float *) p->fout->frame.auxp;
  float* fa = (float *) p->fa->frame.auxp;
  float* fb = (float *) p->fb->frame.auxp;

  if (UNLIKELY(!fsigs_equal(p->fa, p->fb))) goto err1;

  if (p->lastframe < p->fa->framecount) {

    cudaMemcpy(p->devFrameA, fa, size, cudaMemcpyHostToDevice);
    cudaMemcpy(p->devFrameB, fb, size, cudaMemcpyHostToDevice);

    mix<<<p->gridSize,p->blockSize>>>(p->devOutput, p->devFrameA,
                                      p->devFrameB, chans);

    cudaMemcpy(fout, p->devOutput, size, cudaMemcpyDeviceToHost);

    p->fout->framecount =  p->fa->framecount;
    p->lastframe = p->fout->framecount;
  }

  return OK;
  err1:
    return csound->PerfError(csound, p->h.insdshead,
                             Str("cudapvsmix: formats are different."));
}

static OENTRY localops[] = {
  {"cudapvsmix", sizeof(CUDAPVSMIX),0, 3, "f", "ff",
  (SUBR) cudapvsmixset, (SUBR)cudapvsmix, NULL}
};

extern "C" {
  LINKAGE
}
```

cudapvsmix2

```
#include <csdl.h>
#include <pstream.h>

typedef struct _cudapvsmix2 {
  OPDS    h;
  PVSDAT  *fout;
  PVSDAT  *fa;
  PVSDAT  *fb;
  int     gridSize;     // number of blocks in the grid (1D)
  int     blockSize;    // number of threads in one block (1D)
  uint32  lastframe;
} CUDAPVSMIX2;
```

```
__global__
void mix(float* output, float* frameA, float* frameB, int chans) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int j = i<<1;
  if (i < chans) {
    int test = frameA[j] >= frameB[j];
    if (test) {
      output[j] = frameA[j];
      output[j+1] = frameA[j+1];
    }
    else {
      output[j] = frameB[j];
      output[j+1] = frameB[j+1];
    }
  }
}

static int fsigs_equal(const PVSDAT *f1, const PVSDAT *f2)
{
  if (
  (f1->sliding == f2->sliding) &&
  (f1->overlap == f2->overlap) &&
  (f1->winsize == f2->winsize) &&
  (f1->wintype == f2->wintype) &&     /* harsh, maybe... */
  (f1->N == f2->N) &&
  (f1->format == f2->format))

  return 1;
  return 0;
}

static int free_device(CSOUND* csound, void* pp){
  CUDAPVSMIX2* p = (CUDAPVSMIX2*) pp;
  cudaFree(p->fout->frame.auxp);
  return OK;
}

static int cudapvsmix2set(CSOUND *csound, CUDAPVSMIX2 *p)
{
  int32    N = p->fa->N;
  int framelength = N+2;
  int chans = framelength>>1;
  int size = framelength*sizeof(float);

  int maxBlockDim;
  int SMcount;
  int totNumThreads = chans;

  // get info about device
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound, "cudapvsmix2 running on device %s
                (capability %d.%d)\n", deviceProp.name,
                deviceProp.major, deviceProp.minor);

  p->fout->sliding = 0;
  if (p->fout->frame.auxp == NULL ||
      p->fout->frame.size < sizeof(float) * (N + 2))
    AuxCudaAlloc(size, &p->fout->frame);

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N = N;
  p->fout->overlap = p->fa->overlap;
  p->fout->winsize = p->fa->winsize;
  p->fout->wintype = p->fa->wintype;
  p->fout->format = p->fa->format;
  p->fout->framecount = 1;
  p->lastframe = 0;

  csound->RegisterDeinitCallback(csound, p, free_device);

  return OK;
}

static int cudapvsmix2(CSOUND *csound, CUDAPVSMIX2 *p)
{
  int32    N = p->fa->N;
  int framelength = N+2;
  int chans = framelength>>1;
  float* fout = (float *) p->fout->frame.auxp;
  float* fa = (float *) p->fa->frame.auxp;
  float* fb = (float *) p->fb->frame.auxp;

  if (UNLIKELY(!fsigs_equal(p->fa, p->fb))) goto err1;

  if (p->lastframe < p->fa->framecount) {

    mix<<<p->gridSize,p->blockSize>>>(fout, fa, fb, chans);

    p->fout->framecount =  p->fa->framecount;
    p->lastframe = p->fout->framecount;
  }

  return OK;
  err1:
    return csound->PerfError(csound, p->h.insdshead,
                             Str("cudapvsmix2: formats are different."));
}

static OENTRY localops[] = {
  {"cudapvsmix2", sizeof(CUDAPVSMIX2),0, 3, "f", "ff",
  (SUBR) cudapvsmix2set, (SUBR)cudapvsmix2, NULL}
};

extern "C" {
  LINKAGE
}
```

Morph Module

cudapvsmorph

```
#include <csdl.h>
#include <pstream.h>

typedef struct _cudapvsmorph {
  OPDS h;
  PVSDAT  *fout;
  PVSDAT  *fin;
  PVSDAT  *ffr;
  MYFLT   *kampDepth;
  MYFLT   *kfreqDepth;
  float*  devOutput;   // pointer to device memory (output frame)
  float*  devInput1;   // pointer to device memory (input frame #1)
  float*  devInput2;   // pointer to device memory (input frame #2)
  int gridSize;        // number of blocks in the grid (1D)
  int blockSize;       // number of threads in one block (1D)
  uint32  lastframe;
} CUDAPVSMORPH;

__global__
void morph(float* output, float* input1, float* input2, float ampCoeff,
           float freqCoeff, int length) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int j = i<<1;
  if (j  < length) {
    output[j] = input1[j]*(1.0-ampCoeff) + input2[j]*(ampCoeff);
    output[j+1] = input1[j+1]*(1.0-freqCoeff) + input2[j+1]*(freqCoeff);
  }
}

static int free_device(CSOUND* csound, void* pp){
  CUDAPVSMORPH* p = (CUDAPVSMORPH*) pp;
  cudaFree(p->devOutput);
  cudaFree(p->devInput1);
  cudaFree(p->devInput2);
  return OK;
}

static int cudapvsmorphset(CSOUND *csound, CUDAPVSMORPH *p)
{
  int32 N = p->fin->N;
  int framelength = N+2;
  int size = framelength * sizeof(float);
  int maxBlockDim;
  int SMcount;
  int totNumThreads = framelength>>1;
  cudaError_t error;

  // get info about device
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound, "cudapvsmorph running on device %s
                  (capability %d.%d)\n", deviceProp.name,
                  deviceProp.major, deviceProp.minor);
  // device memory allocation (output frame)
  error = cudaMalloc(&p->devOutput, size);
  handleCudaError(csound, error);
  cudaMemset(p->devOutput,0,size);

  // device memory allocation (input 1)
  error = cudaMalloc(&p->devInput1, size);
  handleCudaError(csound, error);

  // device memory allocation (input 2)
  error = cudaMalloc(&p->devInput2, size);
  handleCudaError(csound, error);

  if (p->fout->frame.auxp==NULL ||
      p->fout->frame.size<(N+2)*sizeof(float))
    csound->AuxAlloc(csound,(N+2)*sizeof(float),&p->fout->frame);

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N =  N;
  p->fout->overlap = p->fin->overlap;
  p->fout->winsize = p->fin->winsize;
  p->fout->wintype = p->fin->wintype;
  p->fout->format = p->fin->format;
  p->fout->framecount = 1;
  p->lastframe = 0;

  if (UNLIKELY(!(p->fout->format==PVS_AMP_FREQ) ||
               (p->fout->format==PVS_AMP_PHASE))) {
    return csound->InitError(csound, Str("signal format must be
                                  amp-phase ""or amp-freq.""\n"));
  }

  csound->RegisterDeinitCallback(csound, p, free_device);

  return OK;
}

static int cudapvsmorph(CSOUND *csound, CUDAPVSMORPH *p)
{
  int32 N = p->fout->N;
  int framelength = N + 2;
  int size = framelength * sizeof(float);
  float ampDepth = (float) *p->kampDepth;
  float freqDepth = (float) *p->kfreqDepth;
  float *fi1 = (float *) p->fin->frame.auxp;
  float *fi2 = (float *) p->ffr->frame.auxp;
  float *fout = (float *) p->fout->frame.auxp;

  if (UNLIKELY(fout==NULL)) goto err1;

  if (p->lastframe < p->fin->framecount) {

    cudaMemcpy(p->devInput1,fi1,size,cudaMemcpyHostToDevice);
    cudaMemcpy(p->devInput2,fi2,size,cudaMemcpyHostToDevice);

    ampDepth = ampDepth > 0 ? (ampDepth <= 1 ?
               ampDepth : FL(1.0)): FL(0.0);
    freqDepth = freqDepth > 0 ?
                (freqDepth <= 1 ? freqDepth : FL(1.0)): FL(0.0);
    morph<<<p->gridSize,p->blockSize>>>(p->devOutput, p->devInput1,
                                        p->devInput2, ampDepth,
                                        freqDepth, framelength);

    cudaMemcpy(fout,p->devOutput,size,cudaMemcpyDeviceToHost);

    p->fout->framecount = p->lastframe = p->fin->framecount;
  }

  return OK;
  err1:
    return csound->PerfError(csound, p->h.insdshead,
                             Str("cudapvsmorph: not initialised\n"));
}

static OENTRY localops[] = {
  {"cudapvsmorph", sizeof(CUDAPVSMORPH), 0,3, "f", "ffkk",
    (SUBR) cudapvsmorphset, (SUBR) cudapvsmorph}
};

extern "C" {
  LINKAGE
}
```

cudapvsmorph2

```
#include <csdl.h>
#include <pstream.h>

typedef struct _cudapvsmorph2 {
  OPDS h;
  PVSDAT  *fout;
  PVSDAT  *fin;
  PVSDAT  *ffr;
  MYFLT   *kampDepth;
  MYFLT   *kfreqDepth;
  int gridSize;         // number of blocks in the grid (1D)
  int blockSize;        // number of threads in one block (1D)
  uint32  lastframe;
} CUDAPVSMORPH2;

__global__
void morph(float* output, float* input1, float* input2, float ampCoeff,
           float freqCoeff, int length) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int j = i<<1;
  if (j  < length) {
    output[j]  = input1[j]*(1.0-ampCoeff) + input2[j]*(ampCoeff);
    output[j+1]  = input1[j+1]*(1.0-freqCoeff) + input2[j+1]*(freqCoeff);
  }
}

static int free_device(CSOUND* csound, void* pp){
  CUDAPVSMORPH2* p = (CUDAPVSMORPH2*) pp;
  cudaFree(p->fout->frame.auxp);
  return OK;
}
```

```
static int cudapvsmorph2set(CSOUND *csound, CUDAPVSMORPH2 *p)
{
  int32 N = p->fin->N;
  int framelength = N+2;
  int size = framelength * sizeof(float);
  int maxBlockDim;
  int SMcount;
  int totNumThreads = framelength>>1;

  // get info about device
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp,0);
  maxBlockDim = deviceProp.maxThreadsPerBlock;
  SMcount = deviceProp.multiProcessorCount;
  csound->Message(csound, "cudapvsmorph2 running on device %s
                  (capability %d.%d)\n", deviceProp.name,
                  deviceProp.major, deviceProp.minor);

  if (p->fout->frame.auxp==NULL ||
      p->fout->frame.size<(N+2)*sizeof(float))
    AuxCudaAlloc(size, &p->fout->frame);

  p->blockSize = (((totNumThreads/SMcount)/32)+1)*32;
  if (p->blockSize > maxBlockDim) p->blockSize = maxBlockDim;
  p->gridSize = totNumThreads / p->blockSize + 1;
  p->fout->N =  N;
  p->fout->overlap = p->fin->overlap;
  p->fout->winsize = p->fin->winsize;
  p->fout->wintype = p->fin->wintype;
  p->fout->format = p->fin->format;
  p->fout->sliding = 0;
  p->fout->framecount = 1;
  p->lastframe = 0;

  csound->RegisterDeinitCallback(csound, p, free_device);

  return OK;
}

static int cudapvsmorph2(CSOUND *csound, CUDAPVSMORPH2 *p)
{
  int32 N = p->fout->N;
  int framelength = N + 2;
  float ampDepth = (float) *p->kampDepth;
  float freqDepth = (float) *p->kfreqDepth;
  float *fi1 = (float *) p->fin->frame.auxp;
  float *fi2 = (float *) p->ffr->frame.auxp;
  float *fout = (float *) p->fout->frame.auxp;

  if (UNLIKELY(fout==NULL)) goto err1;

  if (p->lastframe < p->fin->framecount) {

    ampDepth = ampDepth > 0 ? (ampDepth <= 1 ?
                ampDepth : FL(1.0)): FL(0.0);
    freqDepth = freqDepth > 0 ?
                 (freqDepth <= 1 ? freqDepth : FL(1.0)): FL(0.0);

    morph<<<p->gridSize,p->blockSize>>>(fout, fi1, fi2, ampDepth,
                                         freqDepth, framelength);
    p->fout->framecount = p->lastframe = p->fin->framecount;
  }

  return OK;

  err1:
    return csound->PerfError(csound, p->h.insdshead,
                             Str("cudapvsmorph2: not initialised\n"));
}

static OENTRY localops[] = {
  {"cudapvsmorph2", sizeof(CUDAPVSMORPH2), 0,3, "f", "ffkk",
  (SUBR) cudapvsmorph2set, (SUBR) cudapvsmorph2}
};

extern "C" {
  LINKAGE
}
```

Appendix C

Csound Scripts for Testing Purposes

This appendix reports the *Csound* scripts used for testing the performance of the *plugin opcodes* described in chapter 3. These are presented here in order to show the tests' structure, as the way tests are designed may affect the actual performance of the *Csound* modules under analysis. Thus, the information contained in this appendix makes the results reported in chapter 4 more objective.
In the *Csound* scripts of the *scale* and *shift* modules, the mode of operation of `pvscale`, `cudapvscale`, `cudapvscale2`, `pvshift`, `cudapvshift`, and `cudapvshift2` needs to be selected appropriately. Here, only *mode 0* is reported for convenience.

Gain Module

```
<CsoundSynthesizer>

<CsOptions>
--opcode-lib=libcudapvsgain.so
--opcode-lib=../../libcudapvs2.dylib
--opcode-lib=libcudapvsgain2.so
</CsOptions>

<CsInstruments>

sr = 44100
ksmps = 128
0dbfs = 1

gifftsize = $FFT
gihopsize = $HOP

instr 1
   kenv   linseg   0, p3/4, 1, p3/4, 0, p3/4, 1.5, p3/4, 0
   asig   soundin   "syrinx.wav"
   fsig   pvsanal   asig, gifftsize, gihopsize, gifftsize, 1
   fsigScaled   pvsgain   fsig, kenv
   asig   pvsynth   fsigScaled
   out   asig
endin
```

```
instr 2
   kenv   linseg   0, p3/4, 1, p3/4, 0, p3/4, 1.5, p3/4, 0
   asig   soundin   "syrinx.wav"
   fsig   pvsanal   asig, gifftsize, gihopsize, gifftsize, 1
   fsigScaled   cudapvsgain   fsig, kenv
   asig   pvsynth   fsigScaled
   out   asig
endin

instr 3
   kenv   linseg   0, p3/4, 1, p3/4, 0, p3/4, 1.5, p3/4, 0
   asig   soundin   "syrinx.wav"
   fsig   cudanal   asig, gifftsize, gihopsize, gifftsize, 1
   fsigScaled   cudapvsgain   fsig, kenv
   asig   cudasynth   fsigScaled
   out   asig
endin

instr 4
   kenv   linseg   0, p3/4, 1, p3/4, 0, p3/4, 1.5, p3/4, 0
   asig   soundin   "syrinx.wav"
   fsig   cudanal2   asig, gifftsize, gihopsize, gifftsize, 1
   fsigScaled   cudapvsgain2   fsig, kenv
   asig   cudasynth2   fsigScaled
   out   asig
endin

</CsInstruments>

<CsScore>
i $INSTR 0 60
</CsScore>

</CsoundSynthesizer>
```

Filter Module

```
<CsoundSynthesizer>

<CsOptions>
--opcode-lib=libcudapvsfilter.so
--opcode-lib=libcudapvsfilter2.so
--opcode-lib=../../libcudapvs2.dylib
</CsOptions>

<CsInstruments>

ksmps = 128
0dbfs = 1

giSine   ftgen   0, 0, 4096, 10, 1

gifftsize = $FFT
gihopsize = $HOP
```

```
instr 1
   kfreq   expseg   500, p3/3, 4000, p3/3, 500, p3/3, 4000   ; 3-octave
                                                             ; sweep
   kdepth   linseg   1, p3/2, 0.5, p3/2, 1   ; varying filter depth

   asig   soundin   "syrinx.wav"   ; input
   afil   oscili   1, kfreq, giSine   ; filter t-domain signal

   fim   pvsanal   asig,gifftsize,gihopsize,gifftsize,0   ; pvoc analysis
   fil   pvsanal   afil,gifftsize,gihopsize,gifftsize,0
   fou   pvsfilter   fim, fil, kdepth
   aout   pvsynth   fou   ; pvoc synthesis
   out(aout)
endin

instr 2
   kfreq   expseg   500, p3/3, 4000, p3/3, 500, p3/3, 4000   ; 3-octave
                                                             ; sweep
   kdepth   linseg   1, p3/2, 0.5, p3/2, 1   ; varying filter depth

   asig   soundin   "syrinx.wav"   ; input
   afil  oscili   1, kfreq, giSine   ; filter t-domain signal

   fim   pvsanal   asig,gifftsize,gihopsize,gifftsize,0   ; pvoc analysis
   fil   pvsanal   afil,gifftsize,gihopsize,gifftsize,0
   fou   cudapvsfilter   fim, fil, kdepth
   aout   pvsynth   fou   ; pvoc synthesis
   out(aout)
endin

instr 3
   kfreq   expseg 500, p3/3, 4000, p3/3, 500, p3/3, 4000   ; 3-octave
                                                           ; sweep
   kdepth   linseg 1, p3/2, 0.5, p3/2, 1   ; varying filter depth

   asig   soundin   "syrinx.wav"   ; input
   afil   oscili   1, kfreq, giSine   ; filter t-domain signal

   fim   cudanal   asig,gifftsize,gihopsize,gifftsize,0   ; pvoc analysis
   fil   cudanal   afil,gifftsize,gihopsize,gifftsize,0
   fou   cudapvsfilter   fim, fil, kdepth
   aout   cudasynth   fou   ; pvoc synthesis
   out(aout)
endin

instr 4
   kfreq   expseg 500, p3/3, 4000, p3/3, 500, p3/3, 4000   ; 3-octave
                                                           ; sweep
   kdepth   linseg 1, p3/2, 0.5, p3/2, 1   ; varying filter depth

   asig   soundin   "syrinx.wav"   ; input
   afil   oscili   1, kfreq, giSine   ; filter t-domain signal

   fim   cudanal2   asig,gifftsize,gihopsize,gifftsize,0   ; pvoc analysis
   fil   cudanal2   afil,gifftsize,gihopsize,gifftsize,0
   fou   cudapvsfilter2   fim, fil, kdepth
   aout   cudasynth2   fou   ; pvoc synthesis
   out(aout)
endin

</CsInstruments>
```

```
<CsScore>
i $INSTR 0 60
</CsScore>

</CsoundSynthesizer>
```

Stencil Module

```
<CsoundSynthesizer>

<CsOptions>
--opcode-lib=libcudapvstencil.so
--opcode-lib=../../libcudapvs2.dylib
--opcode-lib=libcudapvstencil2.so
</CsOptions>

<CsInstruments>

sr = 44100
ksmps = 128
0dbfs = 1

gifftsize = $FFT
gihopsize = $HOP

instr 1
   kgain   linseg   0, p3, 1.5
   klevel   linseg   0, p3, .5
   asig1   soundin   "ends.wav"
   fsig1 = pvsanal(asig1, gifftsize, gihopsize, gifftsize, 1)
   ftps   pvstencil   fsig1, kgain, klevel, 1
   atps   pvsynth   ftps
   out   atps
endin

instr 2
   kgain   linseg   0, p3, 1.5
   klevel   linseg   0, p3, .5
   asig1   soundin   "ends.wav"
   fsig1 = pvsanal(asig1, gifftsize, gihopsize, gifftsize, 1)
   ftps   cudapvstencil   fsig1, kgain, klevel, 1
   atps   pvsynth   ftps
   out   atps
endin

instr 3
   kgain   linseg   0, p3, 1.5
   klevel   linseg   0, p3, .5
   asig1   soundin   "ends.wav"
   fsig1 = cudanal(asig1, gifftsize, gihopsize, gifftsize, 1)
   ftps   cudapvstencil   fsig1, kgain, klevel, 1
   atps   cudasynth   ftps
   out   atps
endin
```

```
instr 4
   kgain   linseg   0, p3, 1.5
   klevel   linseg   0, p3, .5
   asig1   soundin   "ends.wav"
   fsig1   cudanal2   asig1, gifftsize, gihopsize, gifftsize, 1
   ftps   cudapvstencil2   fsig1, kgain, klevel, 1
   atps   cudasynth2   ftps
   out   atps
endin

</CsInstruments>

<CsScore>
f1 0 -[$FFT+1] 21 1
i $INSTR 0 60
</CsScore>

</CsoundSynthesizer>
```

Scale Module

```
<CsoundSynthesizer>

<CsOptions>
--opcode-lib=libcudapvscale.so
--opcode-lib=../../libcudapvs2.dylib
--opcode-lib=libcudapvscale2.so
</CsOptions>

<CsInstruments>

sr = 44100
ksmps = 128
0dbfs = 1

gifftsize = $FFT
gihopsize = $HOP

instr 1
   kscale   linseg   .3, p3, 3
   kgain   linseg   1, p3/3, 0, p3/3, 1.5, p3/3, 1
   asig   soundin   "syrinx.wav"
   fsig = pvsanal(asig, gifftsize, gihopsize, gifftsize, 1)
   fsigScaled   pvscale   fsig, kscale, 0, kgain
   asig   pvsynth   fsigScaled
   out   asig
endin

instr 2
   kscale   linseg   .3, p3, 3
   kgain   linseg   1, p3/3, 0, p3/3, 1.5, p3/3, 1
   asig   soundin   "syrinx.wav"
   fsig = pvsanal(asig, gifftsize, gihopsize, gifftsize, 1)
   fsigScaled   cudapvscale   fsig, kscale, 0, kgain
   asig   pvsynth   fsigScaled
   out    asig
endin
```

```
instr 3
   kscale   linseg   .3, p3, 3
   kgain   linseg   1, p3/3, 0, p3/3, 1.5, p3/3, 1
   asig   soundin   "syrinx.wav"
   fsig = cudanal(asig, gifftsize, gihopsize, gifftsize, 1)
   fsigScaled   cudapvscale   fsig, kscale, 0, kgain
   asig   cudasynth   fsigScaled
   out   asig
endin

instr 4
   kscale   linseg   .3, p3, 3
   kgain   linseg   1, p3/3, 0, p3/3, 1.5, p3/3, 1
   asig   soundin   "syrinx.wav"
   fsig   cudanal2   asig, gifftsize, gihopsize, gifftsize, 1
   fsigScaled   cudapvscale2   fsig, kscale, 0, kgain
   asig   cudasynth2   fsigScaled
   out   asig
endin

</CsInstruments>

<CsScore>
i $INSTR 0 60
</CsScore>

</CsoundSynthesizer>
```

Shift Module

```
<CsoundSynthesizer>

<CsOptions>
--opcode-lib=libcudapvshift.so
--opcode-lib=../../libcudapvs2.dylib
--opcode-lib=libcudapvshift2.so
</CsOptions>

<CsInstruments>

sr = 44100
ksmps = 128
0dbfs = 1

gifftsize = $FFT
gihopsize = $HOP

instr 1
   kgain   linseg   .4, p3, 2
   kshift   linseg   -2000, p3, 2000
   klowest   linseg   20, p3, 400
   asig   soundin   "syrinx.wav"
   fsig = pvsanal(asig, gifftsize, gihopsize, gifftsize, 1)
   fsigShifted   pvshift   fsig, kshift, klowest, 0, kgain
   asig   pvsynth   fsigShifted
   out asig
endin
```

```
instr 2
   kgain   linseg   .4, p3, 2
   kshift   linseg   -2000, p3, 2000
   klowest   linseg   20, p3, 400
   asig   soundin   "syrinx.wav"
   fsig = pvsanal(asig, gifftsize, gihopsize, gifftsize, 1)
   fsigShifted cudapvshift fsig, kshift, klowest, 0, kgain
   asig pvsynth fsigShifted
  out asig
endin

instr 3
   kgain   linseg   .4, p3, 2
   kshift   linseg   -2000, p3, 2000
   klowest   linseg   20, p3, 400
   asig   soundin   "syrinx.wav"
   fsig = cudanal(asig, gifftsize, gihopsize, gifftsize, 1)
   fsigShifted   cudapvshift   fsig, kshift, klowest, 0, kgain
   asig   cudasynth   fsigShifted
   out   asig
endin

instr 4
   kgain   linseg   .4, p3, 2
   kshift   linseg   -2000, p3, 2000
   klowest   linseg   20, p3, 400
   asig   soundin   "syrinx.wav"
   fsig   cudanal2   asig, gifftsize, gihopsize, gifftsize, 1
   fsigShifted   cudapvshift2   fsig, kshift, klowest, 0, kgain
   asig   cudasynth2   fsigShifted
   out   asig
endin

</CsInstruments>

<CsScore>
i $INSTR 0 60
</CsScore>

</CsoundSynthesizer>
```

Smooth Module

```
<CsoundSynthesizer>

<CsOptions>
--opcode-lib=libcudapvsmooth.so
--opcode-lib=../../libcudapvs2.dylib
--opcode-lib=libcudapvsmooth2.so
</CsOptions>

<CsInstruments>

sr = 44100
ksmps = 128
0dbfs = 1
```

```
gifftsize = $FFT
gihopsize = $HOP

instr 1
   kacf   linseg   0.001, p3, 1
   kfcf   linseg   1, p3, 0.001
   asig   soundin   "syrinx.wav"
   fsig = pvsanal(asig, gifftsize, gihopsize, gifftsize, 1)
   ftps   pvsmooth   fsig, kacf, kfcf
   atps   pvsynth   ftps
   out   atps
endin

instr 2
   kacf   linseg   0.001, p3, 1
   kfcf   linseg   1, p3, 0.001
   asig   soundin   "syrinx.wav"
   fsig = pvsanal(asig, gifftsize, gihopsize, gifftsize, 1)
   ftps   cudapvsmooth   fsig, kacf, kfcf
   atps   pvsynth   ftps
   out   atps
endin

instr 3
   kacf   linseg   0.001, p3, 1
   kfcf   linseg   1, p3, 0.001
   asig   soundin   "syrinx.wav"
   fsig = cudanal(asig, gifftsize, gihopsize, gifftsize, 1)
   ftps   cudapvsmooth   fsig, kacf, kfcf
   atps   cudasynth   ftps
   out   atps
endin

instr 4
   kacf   linseg   0.001, p3, 1
   kfcf   linseg   1, p3, 0.001
   asig   soundin   "syrinx.wav"
   fsig   cudanal2   asig, gifftsize, gihopsize, gifftsize, 1
   ftps   cudapvsmooth2   fsig, kacf, kfcf
   atps   cudasynth2   ftps
   out   atps
endin

</CsInstruments>

<CsScore>
i $INSTR 0 60
</CsScore>

</CsoundSynthesizer>
```

Blur Module

```
<CsoundSynthesizer>

<CsOptions>
--opcode-lib=libcudapvsblur.so
--opcode-lib=../../libcudapvs2.dylib
--opcode-lib=libcudapvsblur2.so
</CsOptions>

<CsInstruments>

sr = 44100
ksmps = 128
0dbfs = 1

gifftsize = $FFT
gihopsize = $HOP

instr 1
   asig   soundin   "syrinx.wav"
   kblurtime   line   0, 60, .99
   fsig = pvsanal(asig, gifftsize, gihopsize, gifftsize, 1)
   ftps   pvsblur   fsig, kblurtime, 1
   atps   pvsynth   ftps
   out   atps
endin

instr 2
   asig   soundin   "syrinx.wav"
   kblurtime   line   0, 60, .99
   fsig = pvsanal(asig, gifftsize, gihopsize, gifftsize, 1)
   ftps   cudapvsblur   fsig, kblurtime, 1
   atps   pvsynth   ftps
   out   atps
endin

instr 3
   asig   soundin   "syrinx.wav"
   kblurtime   line   0, 60, .99
   fsig = cudanal(asig, gifftsize, gihopsize, gifftsize, 1)
   ftps   cudapvsblur   fsig, kblurtime, 1
   atps   cudasynth   ftps
   out   atps
endin

instr 4
   asig   soundin   "syrinx.wav"
   kblurtime   line   0, 60, .99
   fsig   cudanal2   asig, gifftsize, gihopsize, gifftsize, 1
   ftps   cudapvsblur2   fsig, kblurtime, 1
   atps   cudasynth2   ftps
   out   atps
endin

</CsInstruments>
```

```
<CsScore>
i $INSTR 0 60
</CsScore>

</CsoundSynthesizer>
```

Mix Module

```
<CsoundSynthesizer>

<CsOptions>
--opcode-lib=libcudapvsmix.so
--opcode-lib=../../libcudapvs2.dylib
--opcode-lib=libcudapvsmix2.so
</CsOptions>

<CsInstruments>

sr = 44100
ksmps = 128
0dbfs = 1

gifftsize = $FFT
gihopsize = $HOP

instr 1
   asig1   soundin   "syrinx.wav"
   asig2   soundin   "ends.wav"
   fsig1 = pvsanal(asig1, gifftsize, gihopsize, gifftsize, 1)
   fsig2 = pvsanal(asig2, gifftsize, gihopsize, gifftsize, 1)
   ftps   pvsmix   fsig1, fsig2
   atps   pvsynth   ftps
   out   atps
endin

instr 2
   asig1   soundin   "syrinx.wav"
   asig2   soundin   "ends.wav"
   fsig1 = pvsanal(asig1, gifftsize, gihopsize, gifftsize, 1)
   fsig2 = pvsanal(asig2, gifftsize, gihopsize, gifftsize, 1)
   ftps   cudapvsmix   fsig1, fsig2
   atps   pvsynth   ftps
   out   atps
endin

instr 3
   asig1   soundin   "syrinx.wav"
   asig2   soundin   "ends.wav"
   fsig1 = cudanal(asig1, gifftsize, gihopsize, gifftsize, 1)
   fsig2 = cudanal(asig2, gifftsize, gihopsize, gifftsize, 1)
   ftps   cudapvsmix fsig1, fsig2
   atps   cudasynth ftps
   out   atps
endin
```

```
instr 4
   asig1   soundin   "syrinx.wav"
   asig2   soundin   "ends.wav"
   fsig1   cudanal2   asig1, gifftsize, gihopsize, gifftsize, 1
   fsig2   cudanal2   asig2, gifftsize, gihopsize, gifftsize, 1
   ftps   cudapvsmix2   fsig1, fsig2
   atps   cudasynth2   ftps
   out   atps
endin

</CsInstruments>

<CsScore>
i $INSTR 0 60
</CsScore>

</CsoundSynthesizer>
```

Morph Module

```
<CsoundSynthesizer>

<CsOptions>
--opcode-lib=libcudapvsmorph.so
--opcode-lib=../../libcudapvs2.dylib
--opcode-lib=libcudapvsmorph2.so
</CsOptions>

<CsInstruments>

sr = 44100
ksmps = 128
0dbfs = 1

gifftsize = $FFT
gihopsize = $HOP

instr 1
   asig1   soundin   "syrinx.wav"
   asig2   soundin   "ends.wav"
   kampint   linseg   0, p3, 1
   kfrqint   linseg   0, p3, 1
   fsig1 = pvsanal(asig1, gifftsize, gihopsize, gifftsize, 1)
   fsig2 = pvsanal(asig2, gifftsize, gihopsize, gifftsize, 1)
   ftps   pvsmorph   fsig1, fsig2, kampint, kfrqint
   atps   pvsynth   ftps
   out   atps
endin
```

```
instr 2
   asig1   soundin   "syrinx.wav"
   asig2   soundin   "ends.wav"
   kampint   linseg   0, p3, 1
   kfrqint   linseg   0, p3, 1
   fsig1 = pvsanal(asig1, gifftsize, gihopsize, gifftsize, 1)
   fsig2 = pvsanal(asig2, gifftsize, gihopsize, gifftsize, 1)
   ftps   cudapvsmorph   fsig1, fsig2, kampint, kfrqint
   atps   pvsynth   ftps
   out   atps
endin

instr 3
   asig1   soundin   "syrinx.wav"
   asig2   soundin   "ends.wav"
   kampint   linseg   0, p3, 1
   kfrqint   linseg   0, p3, 1
   fsig1 = cudanal(asig1, gifftsize, gihopsize, gifftsize, 1)
   fsig2 = cudanal(asig2, gifftsize, gihopsize, gifftsize, 1)
   ftps   cudapvsmorph   fsig1, fsig2, kampint, kfrqint
   atps   cudasynth   ftps
   out   atps
endin

instr 4
   asig1   soundin   "syrinx.wav"
   asig2   soundin   "ends.wav"
   kampint   linseg   0, p3, 1
   kfrqint   linseg   0, p3, 1
   fsig1   cudanal2   asig1, gifftsize, gihopsize, gifftsize, 1
   fsig2   cudanal2   asig2, gifftsize, gihopsize, gifftsize, 1
   ftps   cudapvsmorph2   fsig1, fsig2, kampint, kfrqint
   atps   cudasynth2   ftps
   out   atps
endin

</CsInstruments>

<CsScore>
i $INSTR 0 60
</CsScore>

</CsoundSynthesizer>
```

Acronyms

ALU	Arithmetic Logic Unit An arithmetic logic unit is a combinational digital electronic circuit that performs arithmetic and bitwise logical operations on integer binary numbers. www.en.wikipedia.org
API	Application Programming Interface In computer programming, an application programming interface is a set of routine definitions, protocols, and tools for building software and applications. An *API* expresses a software component in terms of its operations, inputs, outputs, and underlying types, defining functionalities that are independent of their respective implementations, which allows definitions and implementations to vary without compromising the interface. A good *API* makes it easier to develop a program by providing all the building blocks, which are then put together by the programmer. www.en.wikipedia.org
CPU	Central Processing Unit A central processing unit is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions. The term has been used in the computer industry at least since the early 1960s. Traditionally, the term *CPU* refers to a processor, more specifically to its processing unit and control unit, distinguishing these core elements of a computer from external components such as main memory and I/O circuitry. www.en.wikipedia.org
CUDA	Compute Unified Device Architecture *CUDA* is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use a *CUDA*-enabled graphics processing unit (GPU) for general purpose processing - an approach known as GPGPU. The *CUDA* platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels. www.en.wikipedia.org
DFT	Discrete Fourier Transform In mathematics, the discrete Fourier transform converts a finite sequence of equally-spaced samples of a function into an equivalent-length sequence of equally-spaced samples of the discrete-time Fourier transform (DTFT), which is a complex-valued function of frequency. The interval at which the DTFT is sampled is the reciprocal of the duration of the input sequence. www.en.wikipedia.org

DSP Digital Signal Processing/Processor

Digital signal processing is the numerical manipulation of signals, usually with the intention to measure, filter, produce or compress continuous analog signals. It is characterized by the use of digital signals to represent these signals as discrete time, discrete frequency, or other discrete domain signals in the form of a sequence of numbers or symbols to permit the digital processing of these signals.
www.en.wikipedia.org

A digital signal processor is a specialized microprocessor, with its architecture optimized for the operational needs of digital signal processing. The goal of *DSP*s is usually to measure, filter and/or compress continuous real-world analog signals. Most general-purpose microprocessors can also execute digital signal processing algorithms successfully, but dedicated *DSP*s usually have better power efficiency thus they are more suitable in portable devices.
www.en.wikipedia.org

FFT Fast Fourier Transform

A fast Fourier transform algorithm computes the discrete Fourier transform (DFT) of a sequence, or its inverse. An *FFT* rapidly computes such transformations by factorizing the DFT matrix into a product of sparse (mostly zero) factors. As a result, it manages to reduce the complexity of computing the DFT from $O(n^2)$, which arises if one simply applies the definition of DFT, to $O(n \log n)$, where n is the data size.
www.en.wikipedia.org

FIR Finite Impulse Response

In signal processing, a finite impulse response filter is a filter whose impulse response (or response to any finite length input) is of finite duration, because it settles to zero in finite time.
www.en.wikipedia.org

GP-GPU General-Purpose Graphics Processing Unit

General-purpose computing on graphics processing units is the use of a graphics processing unit, which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit. The use of multiple graphics cards in one computer, or large numbers of graphics chips, further parallelizes the already parallel nature of graphics processing. In addition, even a single GPU-CPU framework provides advantages that multiple CPUs on their own do not offer due to the specialization in each chip.
www.en.wikipedia.org

GPU Graphics Processing Unit

A graphics processing unit is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display. *GPUs* are used in embedded systems, mobile phones, personal computers, workstations, and game consoles. Modern *GPUs* are very efficient at manipulating computer graphics and image processing, and their highly parallel structure makes them more efficient than general-purpose CPUs for algorithms where the processing of large blocks of data is done in parallel. In a personal computer, a *GPU* can be present on a video card, or it can be embedded on the motherboard or - in certain CPUs - on the CPU die.
www.en.wikipedia.org

HDMI High-Definition Multimedia Interface

HDMI is a proprietary audio/video interface for transferring uncompressed video data and compressed or uncompressed digital audio data from an *HDMI*-compliant source device, such as a display controller, to a compatible computer monitor, video projector, digital television, or digital audio device.
www.en.wikipedia.org

HiPAC High-Performance Audio Computing

HiPAC is a domain of study that explores the potential for new advanced processor architectures to transform the current landscape of audio synthesis, processing and music composition. Taking its name from the well established domain of general High-Performance Computing (HPC), it addresses the emergence of new and forthcoming generations of highly parallel floating-point processors, and of large-scale multi-core platforms offering TeraFlop-scale computation power. This offers the possibility of running processes previously disregarded as too computationally expensive, in real-time.
http://quod.lib.umich.edu/

HPC High-Performance Computing

High-Performance Computing most generally refers to the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business.
http://insidehpc.com/

HRTF Head-Related Transfer Function

A head-related transfer function is a response that characterizes how an ear receives a sound from a point in space; a pair of *HRTF*s for two ears can be used to synthesize a binaural sound that seems to come from a particular point in space. It is a transfer function, describing how a sound from a specific point will arrive at the ear (generally at the outer end of the auditory canal).
www.en.wikipedia.org

HSA Heterogeneous System Architecture

Heterogeneous System Architecture is a cross-vendor set of specifications that allow for the integration of central processing units and graphics processors on the same bus, with shared memory and tasks. The *HSA* is being developed by the *HSA* Foundation, which includes (among many others) AMD and ARM. The platform's stated aim is to reduce communication latency between CPUs, GPUs and other compute devices, and make these various devices more compatible from a programmer's perspective.
www.en.wikipedia.org

IIR Infinite Impulse Response

Infinite impulse response is a property applying to many linear time-invariant systems. Systems with this property are known as *IIR* systems or *IIR* filters, and are distinguished by having an impulse response which does not become exactly zero past a certain point, but continues indefinitely.
www.en.wikipedia.org

PCIe Peripheral Component Interconnect Express

PCI Express is a high-speed serial computer expansion bus standard, designed to replace the older *PCI*, *PCI-X*, and AGP bus standards. *PCIe* has numerous improvements over the older standards, including higher maximum system bus

throughput, lower I/O pin count and smaller physical footprint, better performance scaling for bus devices, a more detailed error detection and reporting mechanism, and native hot-plug functionality.
www.en.wikipedia.org

PCM Pulse-Code Modulation

Pulse-code modulation is a method used to digitally represent sampled analog signals. It is the standard form of digital audio in computers, Compact Discs, digital telephony and other digital audio applications. In a *PCM* stream, the amplitude of the analog signal is sampled regularly at uniform intervals, and each sample is quantized to the nearest value within a range of digital steps.
www.en.wikipedia.org

SIMD Single Instruction Multiple Data

Single instruction, multiple data is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Thus, such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment.
www.en.wikipedia.org

SIMT Single Instruction Multiple Thread

Single instruction, multiple thread is an execution model used in parallel computing where single instruction, multiple data (SIMD) is combined with multi threading.
www.en.wikipedia.org

SSE Streaming SIMD Extension

In computing, Streaming SIMD Extensions is an SIMD instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in their Pentium III series processors as a reply to AMD's 3DNow!. *SSE* contains 70 new instructions, most of which work on single precision floating point data. SIMD instructions can greatly increase performance when exactly the same operations are to be performed on multiple data objects. Typical applications are digital signal processing and graphics processing.
www.en.wikipedia.org

STFT Short-Time Fourier Transform

The short-time Fourier transform is a Fourier-related transform used to determine the sinusoidal frequency and phase content of local sections of a signal as it changes over time. In practice, the procedure for computing *STFT*s is to divide a longer time signal into shorter segments of equal length and then compute the Fourier transform separately on each shorter segment. This reveals the Fourier spectrum on each shorter segment.
www.en.wikipedia.org

Bibliography

[1] Zhiwei Xu et al. "Four styles of parallel and net programming". In: *Frontiers of Computer Science in China* 3.3 (2009), pp. 290–301 (cit. on p. 2).

[2] Richard Dobson, Russell Bradford, et al. "High performance audio computing: a position paper". In: (2008) (cit. on pp. 2, 144).

[3] Richard Dobson, Russell Bradford, et al. "The Imperative for High-Performance Audio Computing". In: (2009) (cit. on pp. 2, 144).

[4] Niklas Röber, Ulrich Kaminski, and Maic Masuch. "Ray acoustics using computer graphics technology". In: *10th International Conference on Digital Audio Effects (DAFx-07), S.* Citeseer. 2007, pp. 117–124 (cit. on pp. 3, 46).

[5] Lauri Savioja, Dinesh Manocha, and M Lin. "Use of GPUs in room acoustic modeling and auralization". In: *Proc. Int. Symposium on Room Acoustics.* 2010 (cit. on pp. 3, 46).

[6] Pei-Yin Tsai, Tien-Ming Wang, and Alvin Su. "GPU-based spectral model synthesis for real-time sound rendering". In: *13th International Conference on Digital Audio Effects.* 2010, pp. 1–5 (cit. on pp. 3, 40, 41).

[7] Brian Hamilton and Craig J Webb. "Room acoustics modelling using GPU-accelerated finite difference and finite volume methods on a face-centered cubic grid". In: *Proc. Digital Audio Effects (DAFx), Maynooth, Ireland* (2013), pp. 336–343 (cit. on pp. 3, 46).

[8] Victor Lazzarini, Joseph Timoney, Russell Bradford, et al. "Streaming Spectral Processing with Consumer-level Graphics Processing Units". In: (2014) (cit. on pp. 3, 4, 25, 67–69, 76, 79, 83, 88, 102, 103, 107, 114).

[9] Wen mei W. Hwu ; University of Illinois at Urbana-Champaign ; Coursera. *Heterogeneous Parallel Programming.* URL: https://www.coursera.org/course/hetero (cit. on pp. 7, 137).

[10] Chris McClanahan. "History and evolution of GPU architecture". In: *A Survey Paper* (2010) (cit. on p. 9).

[11] Tomas Akenine-Möller and Jacob Ström. "Graphics processing units for handhelds". In: *Proceedings of the IEEE* 96.5 (2008), pp. 779–789 (cit. on p. 11).

[12] W1zzard. *NVIDIA GeForce GTX 750 Ti 2 GB.* URL: https://www.techpowerup.com/reviews/NVIDIA/GeForce_GTX_750_Ti/1.html (cit. on pp. 12–14).

[13] CUDA Nvidia. *Programming Guide 7.5.* 2015 (cit. on pp. 16, 87, 124, 138, 139).

[14] Mark Harris. *Unified Memory in CUDA 6.* URL: https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/ (cit. on p. 15).

[15] James L Flanagan and RM Golden. "Phase vocoder". In: *Bell System Technical Journal* 45.9 (1966), pp. 1493–1509 (cit. on pp. 17, 18).

[16] Mark Dolson. "The phase vocoder: A tutorial". In: *Computer Music Journal* 10.4 (1986), pp. 14–27 (cit. on pp. 18–20, 24, 76).

[17] Richard Boulanger and Victor Lazzarini. *The Audio Programming Book.* the MIT Press, 2010 (cit. on p. 19).

[18] Thomas F Quatieri and Robert J McAulay. "Audio signal processing based on sinusoidal analysis/synthesis". In: *Applications of digital signal processing to audio and acoustics.* Springer, 2002, pp. 343–416 (cit. on pp. 20, 24).

[19] Jean Laroche. "Time and pitch scale modification of audio signals". In: *Applications of digital signal processing to audio and acoustics.* Springer, 2002, pp. 279–309 (cit. on p. 21).

[20] Toshihiko Abe, Takao Kobayashi, and Satoshi Imai. "The IF spectrogram: a new spectral representation". In: *Proc. ASVA* 97 (1997), pp. 423–430 (cit. on p. 22).

[21] NVIDIA. *cuFFT.* URL: https://developer.nvidia.com/cufft (cit. on pp. 25, 48, 54, 55, 59, 62, 63, 94, 112, 114).

[22] Naga K Govindaraju et al. "High performance discrete Fourier transforms on graphics processors". In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing.* IEEE Press. 2008, p. 2 (cit. on p. 25).

[23] Ramesh C Agarwal, Fred G Gustavson, and Mohammad Zubair. "A high performance parallel algorithm for 1-D FFT". In: *Proceedings of the 1994 ACM/IEEE conference on Supercomputing.* IEEE Computer Society Press. 1994, pp. 34–40 (cit. on p. 25).

[24] ELEAnoR Chu and ALAn GEoRGE. "FFT algorithms and their adaptation to parallel processing". In: *Linear Algebra and its Applications* 284.1 (1998), pp. 95–124 (cit. on p. 25).

[25] Franz Franchetti et al. "Discrete Fourier transform on multicore". In: *Signal Processing Magazine, IEEE* 26.6 (2009), pp. 90–102 (cit. on p. 25).

[26] Li-Yi Wei. "A Crash Course on Programmable Graphics Hardware". In: *Microsoft Research Asia, Tsinghua University, Beijing* (2005) (cit. on p. 28).

[27] Sean Whalen. "Audio and the graphics processing unit". In: *Author report, University of California Davis* 47 (2005), p. 51 (cit. on pp. 29, 30).

[28] Frederik Fabritius. "Audio processing algorithms on the GPU". PhD thesis. Technical University of Denmark, DTU, DK-2800 Kgs. Lyngby, Denmark, 2009 (cit. on pp. 30, 32, 51, 52).

[29] Alexey Smirnov and Tzi-cker Chiueh. "An Implementation of a FIR Filter on a GPU". In: *Experimental Computer Systems Lab, Stony Brook University, Tech. Rep* (2005) (cit. on pp. 30–32).

[30] CURSO DE CIÊNCIA DA COMPUTAÇÃO. “A GPU-based Real-Time Modular Audio Processing System”. PhD thesis. UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL, 2006 (cit. on p. 33).

[31] Qiong Zhang, Lu Ye, and Zhigeng Pan. “Physically-based sound synthesis on GPUs”. In: *Entertainment Computing-ICEC 2005*. Springer, 2005, pp. 328–333 (cit. on pp. 34–36, 61).

[32] Lauri Savioja, Vesa Välimäki, and Julius O Smith III. “Real-time additive synthesis with one million sinusoids using a GPU”. In: *Audio Engineering Society Convention 128*. Audio Engineering Society. 2010 (cit. on pp. 37–39, 55, 67).

[33] Marc Sosnick and William Hsu. “Efficient finite difference-based sound synthesis using GPUs”. In: *Proceedings of the Sound and Music Computing Conference*. 2010 (cit. on pp. 42, 44).

[34] Bill Hsu and Marc Sosnick-Pérez. “Realtime GPU Audio”. In: *Queue* 11.4 (Apr. 2013), 40:40–40:55. ISSN: 1542-7730. DOI: 10.1145/2466486.2484010. URL: http://doi.acm.org/10.1145/2466486.2484010 (cit. on p. 45).

[35] Marcin Jedrzejewski and Krzysztof Marasek. “Computation of room acoustics using programmable video hardware”. In: *Computer Vision and Graphics*. Springer, 2006, pp. 587–592 (cit. on p. 45).

[36] Nicolas Tsingos. “Using programmable graphics hardware for auralization”. In: *Proc. EAA Symposium on Auralization, Espoo, Finland*. 2009 (cit. on p. 46).

[37] Niklas Röber, Martin Spindler, and Maic Masuch. “Waveguide-based room acoustics through graphics hardware”. In: *Proceedings of ICMC*. 2006 (cit. on p. 46).

[38] Jose A Belloch et al. “Headphone-based spatial sound with a GPU accelerator”. In: *Procedia Computer Science* 9 (2012), pp. 116–125 (cit. on pp. 48, 49).

[39] Thomas G Stockham Jr. “High-speed convolution and correlation”. In: *Proceedings of the April 26-28, 1966, Spring joint computer conference*. ACM. 1966, pp. 229–233 (cit. on p. 52).

[40] Jia-sien Soo and Khee K Pang. “A new structure for block FIR adaptive digital filters”. In: *Proceedings IRECOON, volume 38*. 1987, pp. 364–367 (cit. on p. 52).

[41] Jia-sien Soo and Khee K Pang. “Multidelay block frequency domain adaptive filter”. In: *Acoustics, Speech and Signal Processing, IEEE Transactions on* 38.2 (1990), pp. 373–376 (cit. on p. 52).

[42] Avery Wang and Julius O Smith III. “On fast FIR filters implemented as tail-canceling IIR filters”. In: *Signal Processing, IEEE Transactions on* 45.6 (1997), pp. 1415–1427 (cit. on p. 52).

[43] Matteo Frigo and Steven G. Johnson (MIT). *FFTW*. URL: http://www.fftw.org/ (cit. on pp. 54, 55, 59, 62).

[44] Lauri Savioja, Vesa Välimäki, and Julius O Smith. “Audio signal processing using graphics processing units”. In: *Journal of the Audio Engineering Society* 59.1/2 (2011), pp. 3–19 (cit. on pp. 55, 56).

[45] Fernando Trebien. “An efficient GPU-based implementation of recursive linear filters and its application to realistic real-time re-synthesis for interactive virtual worlds”. In: (2009) (cit. on pp. 58, 60).

[46] Fernando Trebien and Manuel M Oliveira. “Realistic real-time sound re-synthesis and processing for interactive virtual worlds”. In: *The Visual Computer* 25.5-7 (2009), pp. 469–477 (cit. on p. 58).

[47] Russel Bradford. “A short note on long recursion”. Unpublished. 2015 (cit. on pp. 61, 62).

[48] André J Bianchi and Marcelo Queiroz. *Measuring the Performance of Realtime DSP Using Pure Data and GPU*. Ann Arbor, MI: Michigan Publishing, University of Michigan Library, 2012 (cit. on pp. 63–65, 67).

[49] Russell Bradford, Richard Dobson, et al. *Real-time sliding phase vocoder using a commodity GPU*. University of Huddersfield and ICMA, 2011 (cit. on pp. 65–67, 87).

[50] Russell Bradford, Richard Dobson, et al. “The sliding phase vocoder”. In: (2007) (cit. on pp. 65, 87).

[51] R Dobson, R Bradford, et al. “Sliding DFT for fun and musical profit”. In: (2008) (cit. on pp. 65, 87).

[52] Iryna Tsimashenka. “The Development, Implementation and Analysis of a Real-Time Parallel Algorithm of Sliding Discrete Fourier Transform”. PhD thesis. University of Bath, 2011 (cit. on p. 66).

[53] Dimitris Theodoropoulos, Georgi Kuzmanov, and Georgi Gaydadjiev. “Multicore platforms for beamforming and wave field synthesis”. In: *IEEE Transactions on multimedia* 13.2 (2011), pp. 235–245 (cit. on p. 71).

[54] Jorge Lorente et al. “Parallel implementations of beamforming design and filtering for microphone array applications”. In: *Signal Processing Conference, 2011 19th European*. IEEE. 2011, pp. 501–505 (cit. on p. 71).

[55] Paul R Dixon, Tasuku Oonishi, and Sadaoki Furui. “Harnessing graphics processors for the fast computation of acoustic likelihoods in speech recognition”. In: *Computer Speech & Language* 23.4 (2009), pp. 510–526 (cit. on p. 71).

[56] ffitch J. et al. “The New Developments in Csound 6”. In: *ICMC 2015 – Sept. 25 - Oct. 1, 2015 – CEMI, University of North Texas*. ICMC. 2015 (cit. on p. 73).

[57] Victor Lazzarini. “Extensions to the Csound language: from user-defined to plugin opcodes and beyond”. In: *Proc. of the 3rd Linux Audio Conf.* Citeseer. 2005, pp. 13–20 (cit. on pp. 74, 85).

[58] V Lazzarini, J Timoney, and T Lysaght. “Spectral Signal Processing in Csound 5”. In: *Proc. Intl. Computer Music Conf., New Orleans, USA*. 2006 (cit. on pp. 75, 76).

[59] Richard Dobson. *PVOC-EX, File format for Phase Vocoder data*. 2000. URL: `http://www.cs.bath.ac.uk/~jpff/NOS-DREAM/researchdev/pvocex/pvocex.html` (cit. on p. 76).

[60] Victor Lazzarini, Joseph Timoney, and Thomas Lysaght. "Time-stretching using the instantaneous frequency distribution and partial tracking". In: (2005) (cit. on pp. 76, 77).

[61] Collaborative Documentation. *The Csound Floss Manual*. 2016. URL: `http://write.flossmanuals.net/csound/_draft/_v/1.1/preface/` (cit. on p. 77).

[62] Barry Vercoe et al. *The Canonical Csound Reference Manual*. URL: `http://www.csounds.com/manual/html/` (cit. on p. 81).

[63] Bruce P Bogert, Michael JR Healy, and John W Tukey. "The quefrency alanysis of time series for echoes: Cepstrum, pseudo-autocovariance, cross-cepstrum and saphe cracking". In: *Proceedings of the symposium on time series analysis*. Vol. 15. chapter. 1963, pp. 209–243 (cit. on p. 81).

[64] S. Imai and Y. Abe. "Spectral envelope extraction by improved cepstral method". In: *Electron. Comm. (in Japan) 62 (4), 10–17* (1979) (cit. on pp. 81, 149).

[65] Axel Röbel, Fernando Villavicencio, and Xavier Rodet. "On cepstral and all-pole based spectral envelope modeling with unknown model order". In: *Pattern Recognition Letters* 28.11 (2007), pp. 1343–1350 (cit. on pp. 81, 149).

[66] Russell Bradford, Richard Dobson, et al. "Sliding is Smoother than Jumping". In: *International Computer Music Conference 2005 (ICMC 2005)*. University of Bath. 2005, pp. 287–290 (cit. on p. 87).

[67] Jared Hoberock and Nathan Bell. *Thrust*. URL: `http://thrust.github.io/` (cit. on p. 90).

[68] Roger Dannenberg et al. "Reinventing Audio and Music Computation for Many-Core Processors". In: () (cit. on p. 144).