NTNU
Norwegian University of
Science and Technology

CPU and GPU Co-processing for Sound

Aleksander Gjermundsen

Master of Science in Computer Science

Submission date:	July 2010
Supervisor:	Anne Cathrine Elster, IDI
Co-supervisor:	Thorvald Natvig, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

GPUs are becoming very attractive devices for supporting HPC applications. This project will look at the GPU as an accelerator that can off-load the CPU for real-time and/or computationally intensive tasks.

Preprocessing of human speech will be used as a case study for how to process an application on either the CPU or the GPU or both and the potential for dynamically allocating computational tasks between the two at run-time.

Potential parallelism will include utilizing OpenCL, a new framework for enabling improved performance on both CPUs and GPUs.

Assignment given: 17. February 2010
Supervisor: Anne Cathrine Elster, IDI

Abstract

When using voice communications, one of the problematic phenomena that can occur, is participants hearing an echo of their own voice. Acoustic echo cancellation (AEC) is used to remove this echo, but can be computationally demanding.

The recent OpenCL standard allows high-level programs to be run on both multicore CPUs, as well as Graphics Processing Units (GPUs) and custom accelerators. This opens up new possibilities for offloading computations, which is especially important for real-time applications. Although many algorithms for image- and video-processing have been studied on the GPU, audio processing algorithms have not similarly been well researched. This can be due to these algorithms not being viewed as computationally heavy and thus as suitable for GPU-offloading as, for instance, dense linear algebra.

This thesis studies the AEC filter from the open-source library Speex for speech compression and audio preprocessing. We translate the original code into an optimized OpenCL program that can run on both CPUs and GPUs. Since the overhead of the OpenCL vendor implementations dominate running times, our results show that the existing reference implementation is faster for single channel input/output, due to its simplicity and low computational intensity. However, by increasing the number of channels processed by the filter and the length of the echo tail, a speed-up of up to 5 on CPU+GPU over CPU only, was achieved.

Although these cases may not be the most common, the techniques developed in this thesis are expected to be of increasing importance as GPUs and CPUs become more integrated, especially on embedded devices. This makes latencies less of an issue and hence the value of our results stronger. An outline for future work in this area is thus also included.

Acknowledgements

This master thesis was written at the High Performance Computing group at the Department of Computer and Information Science, at the Norwegian University of Science and Technology.

I extend my gratitude to several people for making this project possible. First of all, I would like to thank my advisor on this project, Dr. Anne C. Elster, for being of invaluable assistance throughout the project. Through her widespread network of contacts, she has obtained industry collaboration and sponsoring for the HPC lab, without which this project would not have been possible. In particular, we would like to thank NVIDIA and AMD for donating cutting edge hardware to the lab.
I would also like to thank my co-advisor on the project, Thorvald Natvig for coming up with the initial idea and for invaluable technical insight, and HPC-lab Post Doc Dr. John P. Ryan for detailed feedback on this thesis.
Last, but not least, I would like to thank all my fellow students at the HPC lab: Holger Ludvigsen, Ahmed Adnan Aqrawi, Øystein Eklund Krog, Andreas Dreyer Hysing and Runar Heggelien Refsnæs for being very helpful throughout the semester, both with technical topics and motivation.

Aleksander Gjermundsen, Trondheim, Norway, July 2010

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Signal processing is a wide field with many applications, each with a variety of possible implementations. Currently, it is often applied to multimedia data: Images, video or audio. Each of these types of data have their own specific computational demands, but some of the methods are common among them. Some discrete transforms for instance, can be run with satisfactory performance on the simple processors found in embedded devices such as mobile phones for audio compression or filtering. To run the same transform on high-resolution video signals in real-time, the increased data throughput may require a multi-core CPU in modern PCs in order to achieve the same effect in software.

In recent years, Graphics Processing Units (GPUs) have become a quite common processing platform for solving numerous computing tasks. Although traditionally used for decoding/encoding video, visualizations and games, one signal processing area not very widely researched is audio processing on GPUs, even though many of the algorithms used lend themselves to highly parallel execution in the same way as image and video processing. Discrete transforms used in audio processing, like the Discrete Fourier Transform (DFT), have already been thoroughly researched on GPUs for other applications[1].

In previous years, developing programs for GPUs often involved very low-level and vendor specific code. But as the use of General-Purpose computing on GPUs (GPGPU) became more mainstream, a number of high-level and vendor-independent frameworks have appeared. One such framework is OpenCL[2], which allows the same program to be run on GPUs from different vendors, dedicated accelerators and even multi-core CPUs with vector instructions. This opens up new possibilities for offloading computations between devices on demand, which in turn can lead to increased performance and energy efficiency.

The specific focus within audio processing in this thesis is on acoustic echo cancellation, which is often used in systems for real-time voice communication. It is the process of removing the audio from a remote user that is being picked up by the recording device in the room of the local user, resulting in the remote user hearing the echo of their own transmission. This typically occurs when a loudspeaker is used to play the sound received from the remote end, such that the sound is played into the room and is not isolated from the microphone.

1.1 Motivations

In the case study in this thesis, we develop an acoustic echo cancellation filter implementation for GPUs for OpenCL and load-balancing functionality to offload computations from the CPU. Our implementation is based on functionality in the open-source library Speex[3], which is used for speech compression and general audio preprocessing on standard CPUs. Following are some of the motivations for implementing this in OpenCL.

1.1.1 Offloading Computations on PCs

Performing pre-precessing on a high quality audio signal can put a strain on the CPU during real-time audio communications. Even on modern PCs with very powerful CPUs with several processor cores, performing extensive filtering of audio when other CPU-intensive applications are also running, can lead to a substantial decrease in computing power available on the system. A few examples of this is during execution of video conferencing, games and other virtual worlds where voice communication is very common, the audio equipment used to record the sound could be of variable quality and a large amount of the processing power of the CPU is often required for other tasks. Audio preprocessing is then needed to extract a clear signal a as possible of the voice of the person speaking, this can involve methods like noise removal, acoustic echo cancellation and only transmitting sound when someone is actually speaking.

To be able to do all this filtering with high accuracy without affecting overall system performance, the computations can be offloaded to a co-processor. One such co-processor, that has become fairly common in PCs in recent years, is the GPU. It is a massively parallel device that often has spare capacity, and with normal desktop usage is often running completely idle. The parallel nature of the GPU should also make it possible to perform the audio processing in even faster than on the CPU, but this is not an absolute requirement if it can free processing time for other tasks.

1.1.2 Offloading Computations on Embedded Devices

Embedded devices, especially mobile phones, are rapidly becoming equipped with more processing power than what was common in PCs a few years ago. GPUs powerful enough for 3D gaming are now being equipped into mobile phones, which is a development seen earlier for PCs and in the long term enabled them to be used as co-processors in addition to be used for visualization purposes. While becoming increasingly powerful, embedded devices are also very power constrained because they usually run on a form of battery power and have limited cooling, this constrains the speed that their processors can run at.

To achieve the performance needed by some applications they have to utilize all the processing power inside the device, including the GPU. Other applications might run more power efficient on a parallel architecture such as the GPU. In general, offloading computations to the GPU can be essential for future embedded applications. When performing audio filtering on an embedded device for voice over IP applications, offloading the main processor or achieving better audio quality by utilizing both, is an appealing prospect. OpenCL is a technology that will be used for this purpose in the near future, and is the focus of our implementation in this thesis. An advantage with GPU computing on embedded devices is that the CPU and GPU share main memory, in contrast to (most[1]) PCs where the graphics card has a separate memory that is connected over a relatively slow bus. This means that there is little to no overhead of performing operations on the GPU instead of the CPU.

1.1.3 Massively Parallel Audio

Other applications might involve massively parallel processing of separate audio streams, such as in a larger telephony systems. Such a system can be configured to perform audio processing for handsets with limited built-in computing power. The audio quality might have to be restrained here if filtering the audio from all the endpoints on a traditional CPU, or might require multiple CPUs to handle the load. Co-processing with even more parallel architectures than modern CPUs, might significantly decrease the cost of such a system, or make it easier to expand the capabilities of an existing system.

[1]NVIDIA ION shares memory between CPU and GPU with an integrated GPU for PCs (retrieved 20.July 2010): `http://www.nvidia.com/object/sff_ion.html`

1.2 Goals and Contributions

The main goal of this thesis is to explore the possibilities for audio processing on GPUs so that GPUs can be used to offload processing in co-operation with the CPU. A case study will be conducted based on the existing echo cancelling filter of the open-source Speex library for compression of voice data. An OpenCL version is developed, implemented and tested on both AMD Cypress and NVIDIA Fermi GPUs. Our results were also validated against the Speex CPU version. Using OpenCL, it is possible to switch between running the same filter code on either the CPU or GPU.

The performance of the OpenCL CPU and GPU versions will be evaluated, and strategies for load-balancing between the CPU and GPU will be discussed and evaluated in the implementation where they are relevant in practice.

1.3 Outline

The rest of this thesis is structured as follows:

Chapter 2 (Parallel computing) contains background information on parallel computing and GPGPU. The chapter begins with introduction to why GPUs are important to parallel computing, describes the new OpenCL standard and gives an overview of the issues associated with load balancing of applications.

Chapter 3 (Sound preprocessing with Speex) contains background information on sound preprocessing in general, and more specifically on the methods investigated in our case study. It starts by investigating previous work with audio processing on GPUs, gives an introduction to the Speex library and continues on to describe details of the MDF algorithm for echo cancellation and finally gives an introduction to the FFT.

Chapter 4 (Optimizing Speex Echo Cancellation for OpenCL) describes the steps performed to adapt the sound processing library to support GPGPU. First, the testing application implementation with timing and verification functionality is described. The adaptions made to the third party FFT library to work with the filter is then documented. The methods used to parallelize the filter are listed, and the resulting kernel functions described. After the implementation was finished, some initial performance testing was done and the findings from this and how to it affected further development is reported. The chapter finishes with documentation of how co-processing can utilized with the implementation.

Chapter 5 (Benchmarking and Results) contains the results of performance testing and information about the hardware platforms and methods used for benchmarking. The benchmarks that were run on the OpenCL version of the filter when it was finished, both on GPU and CPU, primarily tested the scalability of the filter with respect to echo tail length and the number of input and output channels. To create context for the application performance, latency on the OpenCL implementations used is tested, as well as the performance of the FFT library used against a well-known reference implementation. The run times of the individual kernel functions are analyzed, and the chapter finished with a discussion of the results and the impact this had on co-processing.

Chapter 6 (Conclusions and further work) summarizes the findings in earlier chapters and draws some conclusions about the case study. A section describing future possibilities with both the heterogeneous computing technology and the sound processing library specifically.

Chapter 2

Parallel Computing

This chapter contains contains technical background information on GPU and heterogeneous computing relevant to this thesis. Section 2.1 describes the issues facing high-performance computing today and why GPUs are a possible solution to some of them. This is further explored in Section 2.2 which goes into detail on how programs are written for GPUs and some technical considerations. Finally, Section 2.3 contains an overview of the concept of load balancing, how it has been applied in the past and how it applies to a heterogeneous system composed of CPUs and GPUs.

2.1 Parallel Computing with GPUs

In recent years, one of the main ways computing performance has increased has been due to higher clockspeed in processors, leading to faster execution of a single process or thread. For a considerable period of time, this development did not hit any apparent limits in performance. Large multi-processor systems were obsolete after few years because single processors with a hugely increased clockspeed could outperform them. But the clockspeed and single process performance of processors has reached a plateau in recent years. This can be traced back to the laws of physics, with increased heat production and demands on electric power. This sets practical limits on the maximum clockspeed that is possible to achieve and resulted in a major paradigm shift in the way processors are built.

2.1.1 Introduction to Parallel Computing

As the limit of single processor performance was approached, the industry increasingly looked to parallel systems to be able to achieve better performance than what is possible on a frequency constrained single processor. Multi-core CPUs are now common in most new PCs (2-6 cores at the time of writing). By putting more processor cores on a single chip, performance is theoretically be multiplied by the number of cores. Several lower frequency, and hence cooler low-power cores are more easily housed inside a computer, than a single extremely hot, power consuming one. The trend in HPC computing are so called computing clusters made up of several commodity servers or PCs, connected by a network. The most powerful computing systems in the world today with thousands of processors, are clusters. These have taken over much of the use for large proprietary mainframe computers. Both these trends call for parallel programs that use all of the available processors effectively. However, doing so with two or more processors, is difficult in many cases, because of how access to the data must be managed. The data needs to be shared between the processors either through a system bus, or, even worse, a low bandwidth, high latency network. Slowly, we are seeing more effective use of the available computing power in such systems, and more recently even for non-scientific applications[4].

2.1.2 General Purpose GPU (GPGPU) Computing

A component that is becoming more commonplace in modern PCs, are powerful graphics processors. Originally, the graphics card in a PC was only tasked with outputting a 2D image onto a screen, but in the 90s, specialized graphics accelerators came to market that accelerated 3D graphics for games. This made it possible to include realistic graphics in games that was never before possible, and has also become an important part of modern gaming consoles (such as the Playstation and Xbox). A professional market also existed for these devices, where 3D graphics were used for design and construction (CAD/CAM), but it was the consumer cards that sparked the rapid developments in this area.

Shader programs

A graphics card traditionally includes a lot of fixed-function specialized hardware for rendering a 3D-image with real-time framerates (25-30+ fps). Such an image is generated from vertices and polygons that the programmer outputs from an application. The rendering of simple dots, lines and single-colored polygons are not enough to produce realistic looking graphics on their own. Over the years, more effects have been added into the hardware. Eventually, graphics programmers needed to be able to create their own effects that were relevant to their application, and so a level of programmability was introduced into the graphics cards with the addition of so called shader programs. The word GPU became commonplace, describing the new co-processor supplementary to the CPU. Later, three types of programmable shaders were standardized:

- Vertex shader - Performs operations on single vertices, like changing their color for special effects.
- Pixel/fragment shader - Carries out operations on single screen pixels, instead of points in 3D space.
- Geometry shader - Can change the geometry of the scene itself.

Although these shaders allow much programmability of the GPU, they are still very much tied to a graphics pipeline, in that all operations are performed on graphics primitives. Nevertheless, efficient programs have been written only using shaders (GLSL/HLSL) and extensions (like NVIDIA Cg). For instance, a Lattice Boltzmann method fluid simulation[5] was created using such a method on cluster of PCs with Geforce 4 GPUs.

Expanding the programmability beyond shaders

Newer generations of GPUs (for example NVIDIA's Geforce 8-series and AMD/ATI: Radeon 2000-series and newer) incorporate a feature called ”unified shaders”, which was part of the DirectX 10[1] standard, and is supported by GPUs which implement this widely used API. Unified shaders leads to that all the three types of shader programs mentioned above, share the same type of operations. Previously, some functionality was only available to certain shader types. This lead to the development of more general computing elements by the graphics cards manufacturers to support these unified operations, and so there was no longer any point in having specialized hardware for each type of shader.

With support for unified shaders added, a framework for more general computation on the GPU was achievable. This was made possible by allowing a fourth type of program to run on the processing elements in addition to the three types of shaders: compute kernels. These are programs, often compiled from high-level languages, that run on the same processing elements as the shader programs, with up to several thousands of processing elements on a single GPU, and offer the developer massive parallelism. At the time of writing, the highest-end single consumer GPU from NVIDIA was the Geforce GTX480 with 480 shader cores, and the Radeon 5870

[1]Information about unified shaders in DirectX 10: `http://msdn.microsoft.com/en-us/library/ee418282%28VS.85%29.aspx`

from AMD with 1600 shader cores (the performance, functionality and grouping of these are slightly different, and cannot in terms of numbers of processors be compared directly).

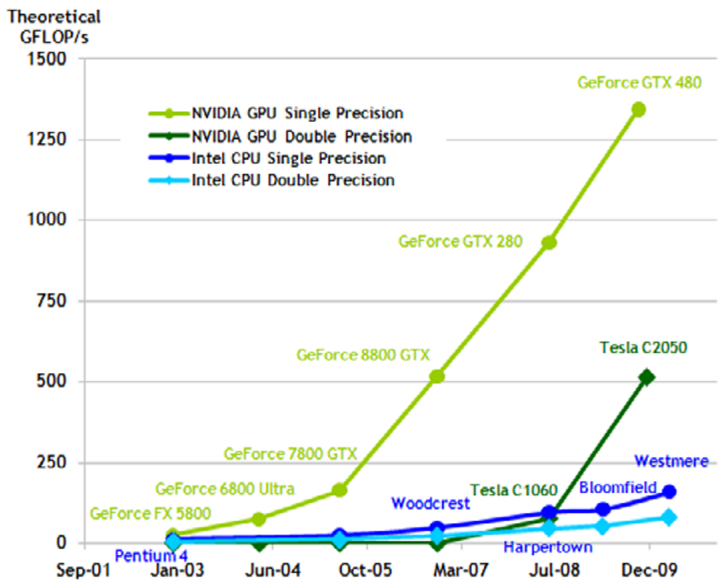


Figure 2.1: Development of theoretical computational power on CPUs vs (NVIDIA) GPUs in the last decade, from [6] with permission from NVIDIA.

This large number of processing elements combines into a huge amount of processing power on a single chip when one considers the compute capability offered by all the cores (see Figure 2.1). and compare this to a modern CPU, that has 2-4 cores as mentioned earlier. For programs that map well to the architecture, they can experience a massive speedup. But although this looks brilliant in theory, some limitations arise when writing programs for the GPU, that will be discussed in detail later in this chapter. Because the GPU functionality was originally made for drawing graphics, each processing element does not offer the general functionality of a CPU core, and attention has to be paid to the way in which many cores access memory. Some applications are inherently better suited than others for execution on GPUs, [7] lists the following as characteristics of applications that are well suited:

- *Computational requirements are large.* GPUs perform a massive amount of computations in real-time, and are most efficient if given a substantial amount of work to do at once. Preferably, the application should have a high ratio of numerical operations compared to memory accesses, but this can be helped by favorable access patterns. Heavy branching from complicated logic that makes threads diverge, can also significantly reduce performance.

- *Parallelism is substantial.* Due to the massive parallel design of GPUs, the application must be very parallelizable and so the computational domain of the application should map well onto the architecture of the GPU, *i.e.* a large number of threads is often needed to mask memory access time.

- *Throughput is more important than latency.* Since real-time graphics operate on a millisecond timescale, GPUs are optimized for throughput on a scale that is perceivable by the human eye. On the time scale of microprocessors, this is fairly long, which have lead to long pipelines that are optimized for throughput over latency.

2.2 OpenCL

The obvious step after developing multi-purpose hardware and making it possible to run more general programs on GPUs, is to ease the burden of creating the software. Using graphics-based languages or programming in an assembly language is tedious and makes development of more complex programs prohibitively difficult. To alleviate this problem, GPU manufacturers created APIs and translators for more general programming languages, to ease development for programmers not familiar with the inner workings of the GPU and allowing for more rapid development and reuse of code. The first widely used framework for writing high-level programs for GPGPU was Compute Unified Device Architecture (CUDA) from NVIDIA. Originally, it was only possible to write programs in C or FORTRAN, but the platform has also been extended to other languages. All the high-level languages are compiled down to a common bytecode, PTX-files, that is then compiled into binary code that is executed on the GPU (see Figure 2.2). This removes most of the dependencies on the graphics pipeline, and allows more scientific applications to be run. The main framework in focus in this thesis for GPGPU is

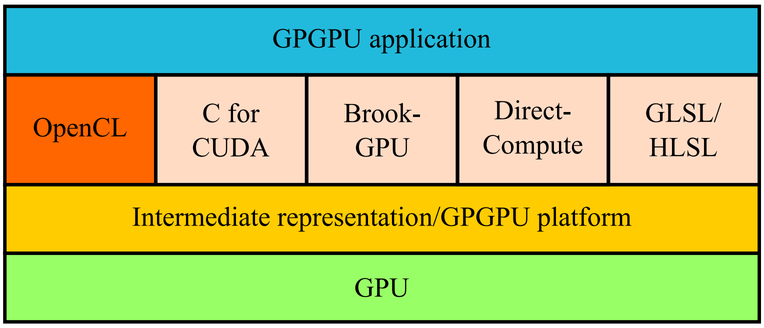


Figure 2.2: GPGPU computing platform and languages, based on figure in [6].

OpenCL. Not only is it supported as one of the languages and APIs that can be used to construct applications on the NVIDIA CUDA platform, but it is also a new vendor/platform-independent standard that can increase the propagation of GPGPU and stream processing into traditional areas of computing.

2.2.1 The OpenCL Standard

OpenCL is an "open standard for parallel programming of heterogeneous systems"[2], the OpenCL framework is said to be comprised of the following components (from [2]):

- *OpenCL Platform Layer*: The platform layer allows the host program to discover OpenCL devices and their capabilities and create contexts (execution environments).
- *OpenCL Runtime*: The runtime allows the host program to manipulate contexts once they have been created.
- *OpenCL Compiler*: The OpenCL compiler creates program executables that contain OpenCL kernels. The OpenCL C programming language implemented by the compiler supports a subset of the ISO C99 language with extensions for parallelism.

At the time of writing, the first stable version of the OpenCL standard, 1.0[8], had been out for over a year, and the second stable version (1.1[2]) had just been released. The standard is managed by the Khronos group, which is a consortium funded by industry members. Besides the OpenCL standard, it also manages the widely used OpenGL standard for writing graphics applications. Contributors to the standard included leading hardware vendors such as: AMD/ATI, Apple, ARM, IBM, Intel and NVIDIA. The broad participation of the industry when developing the standard, ensured broad cooperation on implementing the standard on a wide variety of platforms. The known publicly available implementations at the time of writing were the following (with stable releases):

- CPU/GPU implementation from Apple included in their operating system MAC OSX. GPU acceleration available both on NVIDIA and AMD GPUs.
- CPU/GPU implementation from AMD supports both their own GPUs and the newer multicore X86 CPUs (including those from Intel) with SSE3 vector operation support. Provides a multi-platform SDK and drivers for Windows and Linux.

[2]OpenCL home page: `http://www.khronos.org/opencl/`

- GPU implementation from NVIDIA accelerated on their own GPUs. Provides a multi-platform SDK and drivers for Windows, Linux and MAC OSX.
- CPU/accelerator implementation from IBM providing support for acceleration on their Cell processor. Provides an SDK for Linux.

2.2.2 The OpenCL Language

The OpenCL specification contains an API and a set of extensions to the C programming language, that allow special C functions (declared as kernels) to be executed on the computing device.

When writing OpenCL programs, it typically consists of two parts: One part executes on the host (CPU) and accesses the OpenCL runtime and platform layers, and the other part (kernel) executes on the computing device (GPU/multi-core CPU/accelerator). The host code specify how to configure kernel execution (how many threads should be executed, and how they relate to each other) and memory management (the GPU and CPU do not typically share memory, so data often needs to be explicitly copied between their address spaces). The kernel itself, is regular C99-code with some restrictions[2]:

- Many of of the standard library functions are not available, *i.e.* standard C IO functions.
- There is no stack available, meaning that there is no real function calls (functions will be automatically inlined), and no recursion. But it is possible for extensions to the standard to provide this.
- All kernel invocations must return void.
- Pointer use is somewhat restricted, *e.g.* pointers to pointers are not legal, no function pointers are permitted and no dynamic allocation of memory is possible.
- Static variables are not available, data must be persisted explicitly between kernel invocations.
- Built-in functions must be used to retrieve information about the platform, synchronization primitives and functions for performing some mathematical operation effectively on the GPU.
- Support for debugging the kernel code it is running on a GPU is often very limited.

The exact details are dependent upon the implementation in question. The platform is modular, with each implementation free to implement extensions to utilize available features (ie. double-precistion floating point support). In addition, for the kernel to run effectivly some considerations need to be made, in short:

- Great attention needs to be paid to the way memory is accessed (see Subsection 2.2.6).
- The program will be inherently multithreaded, and must behave accordingly (see Subsection 2.2.5).
- Branching where some work-items/threads diverge can waste a lot of resources, as work-items execute the same code on multiple processing elements at the time. Branches have to be serialized on the processing elements.
- The GPU has support for a lot of specific types of operations (primarily simple single-precision floating-point arithmetic), other operations should be avoided as much as possible.

Extra considerations to make kernels execute with high performance, can be alleviated by hardware designed to address specific issues. This is an ongoing process, and for NVIDIA each new generation of capabilities are called the "compute capability" of a certain GPU. From 1.0 which was implemented with the Geforce 8-series of graphics cards, to present version 2.0 in the Geforce 400-series and the newest Tesla cards. Until now, many of the improvements have come in the form of increased flexibility when accessing memory while preserving high performance.

HPC applications are becoming an increasingly important factor when new GPUs are designed as evident by the Fermi architecture from NVIDIA [9], and to a certain degree the Cypress architecture from AMD [10].

2.2.3 Alternatives to OpenCL

Due to the special requirements in this thesis, OpenCL was chosen as the target platform. Before going into detail on how OpenCL programs are executed, some alternatives will be mentioned. The C for CUDA language seems to be the most widespread at the time of writing, but it is mostly specific to NVIDIA GPUs, and has less vendor support for executing on other types of devices (CPUs[3] and accelerators). The equivalent to CUDA from ATI, is called Stream and did not gain the same foothold as CUDA before OpenCL support emerged, but offered many of the same features and support a C-like language called Brook[4] that was developed at Stanford University with an open source implementation available. Although BrookGPU in itself is not GPU vendor specific, Stream is specific to AMD graphics

[3]MCUDA is a community/research project to effectively compile CUDA code to CPUs: `http://impact.crhc.illinois.edu/mcuda.php`

[4]BrookGPU project: `http://graphics.stanford.edu/projects/brookgpu/`

processors, in the same way as CUDA is for NVIDIA.

Additional languages and APIs exist, but are mostly have minority developer communities behind them (FORTRAN, Java and C# on CUDA), have limited platform support (DirectCompute) or are made to solve a limited set of problems. Other, perhaps more high-level, standards, such as HMPP[5], may prevail in the future, programming massively parallel processors like GPUs is an ever-changing field, where one of the most important goals is to make them accessible to new applications and new developers.

2.2.4 OpenCL on Embedded Devices

GPGPU has to date been a concept tied to traditional servers and workstations. As embedded systems get more advanced over time, they become a new domain where utilizing GPUs for diverse tasks has potential. Many mobile phones at the time of writing contain a simple GPU integrated with the CPU (sharing the same memory), which is mainly used for rendering the graphical user interface and games. As they get more powerful and versatile, it is anticipated they will be used in ways similar to how it has been done on PCs. Popular mobile GPUs include the ARM Mali, POWERVR SGX from Imagination Technologies and NVIDIA Tegra, but hardware support from these manufacturers alone do not make it possible to use features on these devices. At the time of writing, these devices were mostly programmable through the use of shader languages such as the OpenGL ES graphics standard. There had been much discussion of the possibility of using OpenCL on such devices, and some beta-level software is available to a restricted set of developers, but wide support for writing OpenCL programs on embedded devices is not present at the time of writing.

The OpenCL 1.1 specification [2] defines a specialized “embedded profile” for developing OpenCL applications on embedded devices. The embedded profile is a subset of the complete specification adapted specifically to embedded devices. In summary, these are a set of additional restrictions that apply to the “full profile”[2]:

- 64-bit integers are not supported.
- Only optional support for 3D images (textures), and restrictions to interpolation of 2D and 3D images.
- Differences in default rounding modes for single-precision floating point operations.
- The division and square root operations on floating point numbers can have more relaxed rounding demands.

[5]CAPS enterprise, makers of HMPP for hybrid computing: `http://www.caps-entreprise.com/`

- Differences in conversions between single- and half-precision floating point numbers.
- Less rigid rounding from integers to floating point numbers.
- Custom device info parameters. These are metrics about the computing device that can be retrieved at run-time.

These are fairly acceptable restrictions to the specification, and makes it possible to easily port OpenCL "full-profile" (desktop) programs to the embedded profile.

In the context of this thesis, it would be very interesting to the case study involving audio processing port to an embedded device. The Speex library (see 3.4) is already made to run on embedded devices. It supports the DSP (Digital Signal Processors) on Analog Devices' Blackfin microprocessors, has optimized procedures for the ARM4 microprocessor, support for the Symbian operating system, and support for CPUs without floating point capability. Running Speex on an embedded device has some obvious applications, as it could be used for many kinds of voice communication, primarily for mobile phones. It might be beneficial to offload much of the computational load on such a device to the GPU or other supported OpenCL devices. If such tasks were offloaded from the CPU, they might be performed faster and more energy efficient, enabling better audio quality as well as freeing the CPU for other tasks. In this thesis, embedded OpenCL implementations as treated as a purely theoretical subject, because, as mentioned, no comprehensive development platforms was available to the public at the time of writing.

2.2.5 Concepts of Kernel Execution

As mentioned, GPUs contain a lot of computation cores. The physical difference in surface area distribution of the different components for the CPU compared to a GPU chip, is depicted in Figure 2.3. The GPU dedicates significantly more space to the smaller and simpler computational units (the ALUs), instead of the fewer but more functional units of the CPU. CPUs also have to support the considerable legacy of the X86 architecture and its flexible general computing nature, this increases the amount of control logic needed. In addition, the GPU eliminates/reduces the global cache, and instead distributes some fast memory among groups of computational cores. The larger amount of computational cores, increases the demands on parallelization of the application that will execute on them, which is a burden the developer, and preferably to a certain degree the compiler, is responsible for.

The most basic unit that executes on a GPU, is termed a *work-item* in OpenCL (CUDA: thread), which is the same concept often used for parallel applications on CPUs. Each work-item/thread can have its own variables and in theory its own

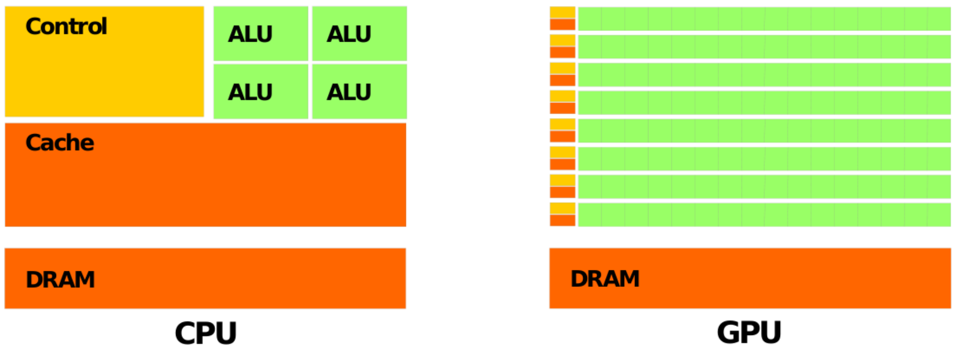


Figure 2.3: Overview of the area distribution on CPU and GPU chips, from [6] with permission from NVIDIA.

separate control flow, and is an instance of a *kernel function*. But, as will become apparent later in the section, it is not as simple as a 1:1 mapping between running threads and independent cores (as there is on most CPUs). In OpenCL, a group of work-items is called exactly that, a *work-group* (CUDA: block), which again can be divided in 1, 2 or 3 dimensions, depending on what is appropriate for the problem at hand.

Typically, these work-groups can contain several hundreds of work-items in each dimension defined. These work-groups are assigned to a unit on the GPU, that is called a *compute unit* in OpenCL (NVIDIA/CUDA: *streaming multiprocessor*). Each of these compute units are independent of each other, and schedule work-items (in groups of 32, called *warps* in CUDA) to run on the ALU units that it contains. They are built after the so-called SIMT principle, which stands for Single Instruction, Multiple Threads. This is similar to the concept of SIMD (Single Instruction, Multiple Data), in that multiple computational units execute the same operation (instruction) at the same time, each working on a different piece of data. In contrast to SIMD, SIMT work-items/threads can branch and so can include some control logic, but even though this is possible, branching is serialized and so each work-item is not completely independent. If even one of the work-items running on a compute unit diverges from the others, only the work-items following the branch can continue executing in parallel, the rest have to wait and then execute their own branch later. All this means that the programmer must pay close attention to the way that work is divided between the work-groups/compute units, and to avoid branching behavior in work-items as much as possible.

A group of work-groups, is again collected in what is called a *NDRange* (CUDA: grid), this consists of a 1, 2 or 3-dimensional index space of work-groups. The dimensions of the NDRange, in the same way as for work-groups, can be determined by the problem at hand. In practice, the maximum size of the NDRange is much larger than the work-groups, since it is not bound to the physical resources of the multiprocessors in the way a work-group is. GPUs from the same generation, often have the same basic compute units, but with different numbers of them. On the GPUs in the NVIDIA Geforce 200-series with 240 shader cores (30 SMs of 8 SPs each), more than 30 000 threads can be scheduled at once[11].

An overview of the grid/block/thread hierarchy can be found in Figure 2.4.

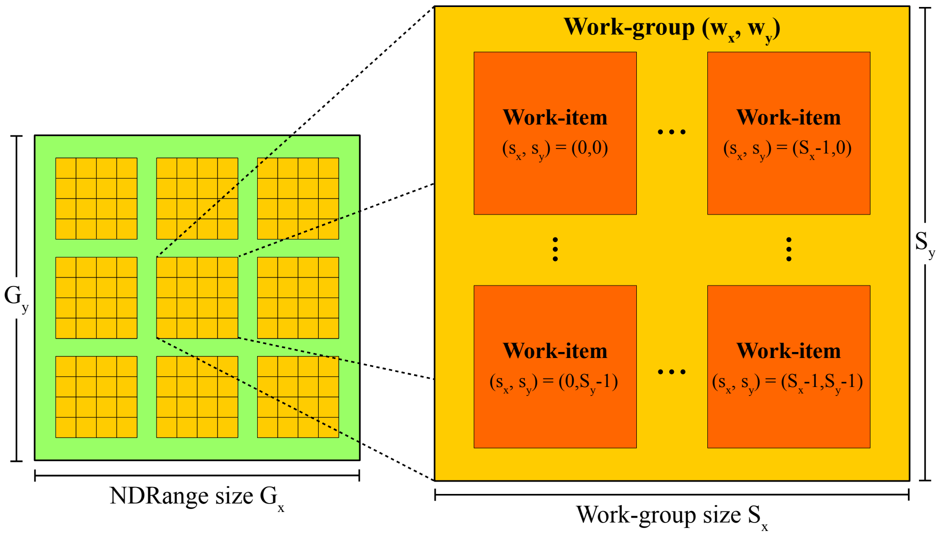

Work-group (w_x, w_y)
Work-item
(s_x, s_y) = (0,0)
Work-item
(s_x, s_y) = (S_x-1,0)
Work-item
(s_x, s_y) = (0,S_y-1)
Work-item
(s_x, s_y) = (S_x-1,S_y-1)
G_y
S_y
NDRange size G_x
Work-group size S_x

Figure 2.4: OpenCL NDRange of work-groups. (CUDA: Grid of blocks), based on figure in [2].

2.2.6 Memory Hierarchy

Just like a CPU, GPUs have access to various types of memory that vary in speed and capacity. To utilize the memory effectively across hundreds of cores, and deal with the long memory latencies involved with accessing the large DRAM memory (termed “global memory” from now on), other types of faster, but smaller capacity memory are also available to the GPU (see Figure 2.5). One also needs to remember that before data can be operated on by the GPU from global memory, it must be first copied from the main CPU memory. To date this copy operation, must be carried out over the PCI-Express bus, which has a maximum bandwidth of only 8 GB/s[6] in each direction. As mentioned earlier, this limits the use of GPUs for

[6]Press release of PCIe 2.0 (retrieved 20.July 2010): `http://www.pcisig.com/news_room/PCIe2_0_Spec_Release_FINAL2.pdf`

some problems, they are best applied to problems where a significant amount of calculations are involved.

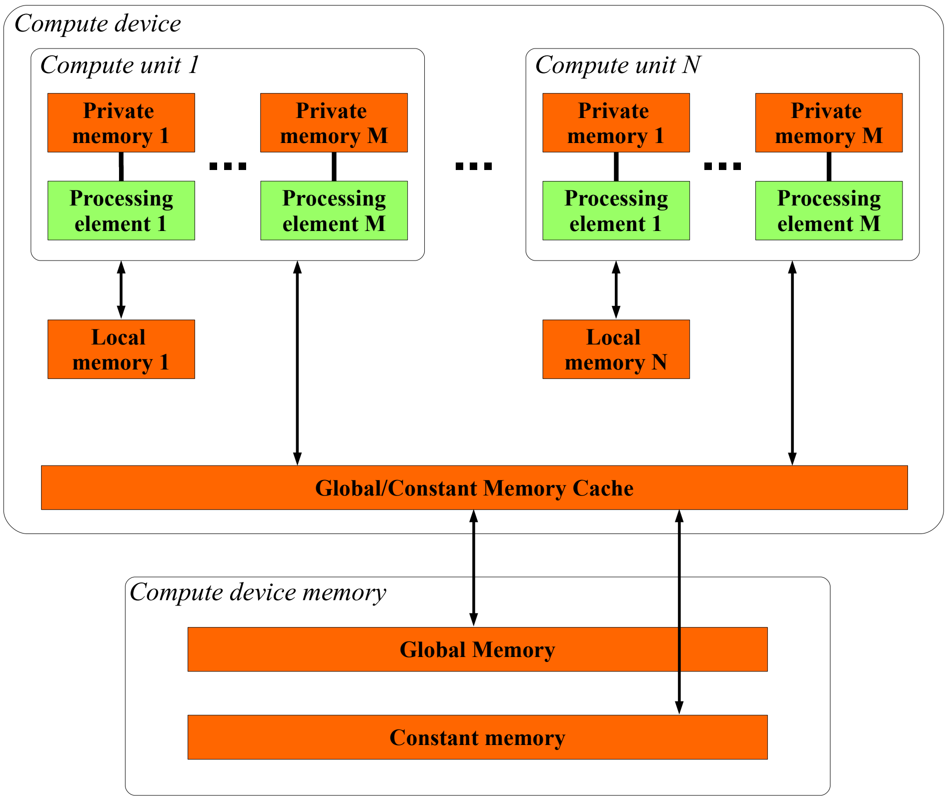


Figure 2.5: OpenCL memory hierarchy, based on figure in [2].

Primarily, there are three types of memory available to the GPU. The following listing is based on information in [11], with specifications mainly for NVIDIA GPUs, but the princicples apply to ATI GPUs as well:

- Global memory that is accessible by all work items, a large majority of the total memory resources is of this type. It is large (6 GB on the current high-end NVIDIA graphics cards), has reasonably high bandwidth, but can have several hundred clock cycles of access latency. The bandwidth for sequential accesses, is in fact much higher[7] than the DRAM memory in most PCs and

[7]The specifications for the Tesla C2050 lists the memory bandwidth as 144 GB/s. Source (retrieved 20.July 2010): `http://www.nvidia.com/object/product_tesla_C2050_C2070_us.html`

servers[8], but despite the higher bandwidth, a lot more cores need to access the same memory. The access latency can become a large problem if accesses are not organized effectively. Reads and writes from/to global memory from a number of work-items should be *coalesced* (combined into a larger operations) by reading/writing as much as possible, in specific patterns, depending on the GPU generation (see the NVIDIA OpenCL Programming Guide [6] for more details). Additionally, some other categories of memory is ultimately stored in global memory, but used for special purposes:

   – Private memory (called local memory in CUDA), is a small piece of memory available to each thread. Currently this is 512 KB per work-item on the Fermi architecture GPUs[6]. In practice, private memory usually acts as slow spillover storage for faster registers, so it should be avoided as much as possible. Newer NVIDIA GPUs[9] have implemented caching of private memory, which can help performance if this is a problem in the application.
   – Constant memory, which is 64 KB (all current NVIDIA GPUs[6]) shared among all work-items and is typically used for storing constants (hence the name). This is cached on the compute units, and can be much faster than accessing regular global memory when a value is read multiple times (same as registers on cache hits). The values can not be altered by kernels, they must be predefined or copied to from the host before the kernel is launched.
   – Image objects (texture memory in CUDA), is based on a method to bind a section of global memory as a cached texture. This is cached by each compute unit, similar to constant memory, and can lead to faster lookups, especially for 2-dimensional data. A heritage from visualization, is that values in texture memory are automatically interpolated without additional computational cost, which can be very useful for some applications. Textures can only be written to by compute kernels under certain conditions, and have generally been used as read-only data by the GPU.

- Local memory (called shared memory in CUDA terminology) available for each work-group and is local to each compute unit (32 KB per compute unit on NVIDIA Fermi[6] and ATI Cypress[10] GPUs). If a work-group has high re-use of data, temporarily storing the values here can significantly reduce access latency as it is roughly 100 times faster[11] than global memory. Another automatic use of local memory, is for "function" arguments passed to the kernel. Some considerations must be taken with read-patterns from local memory, to avoid *bank conflicts*. Bank conflicts arise because local memory is organized into memory banks that can be access simultaneously if all the threads are requesting separate addresses in a linear fashion, but problems

[8]The specifications for the Intel Core i7-980 Extreme Edition list it as having a 25.6 GB/s maximum memory bandwidth. Source (retrieved 20.July 2010): `http://ark.intel.com/Product.aspx?id=47932&processor=i7-980X&spec-codes=SLBUZ`

arise when the access patterns prohibits all the banks to be accessed at once (further details can be found in the [6] and depend on the specific GPU architecture).

- Registers local to each compute unit represents the fastest data storage, but the amount available to each work-group is limited, so if many registers are used, it can limit the amount of concurrent work-items that is possible to run on each compute unit. If the kernel uses too many registers, they spill over into private memory (physically located in global memory), which considerably slows the program down. The maximum amount of registers available to each compute unit, is 32 768 32-bit registers on NVIDIA Fermi-based GPUs[6].

2.3 Load Balancing

In this section load balancing and how it might be applied to GPUs and sound processing applications (such as our implementation) is discussed.

2.3.1 History of Load Balancing

Load balancing in general, is an ago-old problem in compute science. Traditionally it has been used to distribute the work-load for large tasks among a number of systems, *e.g.* it is commonly employed to serve large web sites or various web services. This was a most common use-case in a world where all computers only had a single processor with a single core that needs to do all processing. With many similar CPUs installed side by side in newer systems, as well as including other computing devices, like GPUs, load balancing also becomes an issue internally on a single machine. Load balancing is also closely related to scheduling, which is a much explored problem of assigning batches of work to a finite number of resources, *i.e.* scheduling processes for execution on a single CPU. For applications running on the CPU, the operating system can schedule processes for execution on different cores at the same time, but it can not automatically assign a process to several CPUs/cores at the same time to speed up execution, the program has to do so explicitly itself. If one wishes to make use of all the available resources, the application has to implement some form of load-balancing itself to distribute work to where the resources are available at the time.

2.3.2 Traditional Load Balancing Techniques

Static load balancing is the concept of assigning work to processors before any tasks start executing. This is a challenging theoretical problem that is categorized as np-complete. The following are listed as some of the major practical static load-balancing techniques in [12]:

- Round robin algorithms - distribute work in sequential order of processes, coming back to the first when all the processes have been given a task.
- Randomized algorithms - selects processes at random to take tasks.
- Recursive bisection - recurisvely divides the problem into subproblems of equal computational effort while minimizing message-passing.
- Simulated annealing - an optimization technique that avoids the locally best solution (end time) in favor of the best one globally.
- Genetic algorithm - an optimization technique based on evolution in nature.

These algorithms are useful if all the parameters of the system are predetermined before execution starts, but problems can occur if for instance other tasks share the processors or the interconnect is not completely isolated (which is often not the case for ethernet networks). The idea of "off-loading" tasks from processors is also not congruent with static load balancing, because we do not usually know the work-load at the processors forward in time.

Dynamic load balancing, as the name suggests, takes the current overall state of the processors into account. This adds some additional overhead during execution, but allows for much greater flexibility for real-world environments. In [12], dynamic load balancing is divided into two categories:

- Centralized - Tasks are distributed from a central location with a master process that controls several slave processes. The tasks can be arranged in a prioritized queue that can grow dynamically during execution. Termination on the end of execution can be simple with one central work-pool, the master process simply signals all the slave processes.
- Desentralized - Tasks are passed between arbitrary processes and finally reporting to a single process when they are completed. This alleviates the dependency on the single process within the work-pool, it may be better to have completely independent worker/slave processes or smaller groups of processes that share a work pool. A problem with desentralized load-balancing is that determining when to terminate can become a problem, but several algorithms exists to solve this.

An example of a centralized load-balancing approach for modern processors is Grand Central Dispatch[13] (GCD), which is a technology created to effectively utilize multi-core CPUs between several applications. The library implementation has been released as the open source project libdispatch[9] and is primarily available in Apple Mac OS X at the time of writing. It acts as an abstraction layer to the application developer and groups tasks into *blocks*, that are organized in *queues*. The queues from the application are managed centrally by GCD, which is a system level component that can manage work depending on the existing queues and the overall status of the system. The blocks are removed from the queues by GCD and assigned to threads for execution.

2.3.3 Load Balancing on GPUs

There does not exist very much research in load balancing between the CPU and GPU on a single heterogenous system. More common is the task of load-balancing between several fully utilized GPUs[14], and between a cluster of heterogenous computers with both CPUs and GPUs. In the case study in this thesis, work should be either executed as fast as possible or offload to either the CPU/GPU, if computing resources are used for other purposes. In this regard, a dynamic load balancing scheme is required. It is also implied that it must be centralized with due to the technology in GPUs currently available, the CPU still needs to be strongly involved when offloading work to the GPU.

A large challenge when considering dynamically load balancing involving a GPU, is that there is currently no practical way of gathering information about what the work-load of the GPU is at a point in time. In fact, to date it has not been possible for several programs with separate processes to share the same GPU in controlled way. NVIDIA aimed to improve on this area with their Fermi architecture [9] which makes it possible for multiple kernels to run on the same GPU to utilize it fully. But, this hardware had not been available long enough at the time of writing for this to be incorporated into the case study. Problems with sharing a GPU still exist even with these improvements, for instance when another application needs to use the GPU for rendering graphics at the same time and how this combines with GPGPU applications.

These limitations lead to more simplified algorithms for load balancing on the GPU, where some strong assumptions need to be made about the overall state of the system. GPGPU platforms must still evolve further before resources on the GPU can be shared and utilized by multiple applications throughout the system, in the same way as the CPU.

A system similar to Grand Central Dispatch (see Section 2.3.2) could be a possible

[9]libdispatch web page: `http://libdispatch.macosforge.org/`

solution on GPUs, and some of same principals of execution queues are found in OpenCL. Although the principles are appealing, some problems arise when transferring the same principles to GPUs, first among which are memory accesses/locks and fine-grained thread control that are more easily achieved with the general CPU architectures. Tight control of thread execution, atomic operations and synchronization across all threads are not well supported operations on GPUs today, or incur a large performance penalty. GPU hardware designers have shown to cater more to the needs of GPGPU in newer generations of their architectures, and this could lead to such general load balancing being realistically achievable in the future.

2.3.4 Auto Tunable GPU Algorithms

Because of differences in computing devices that can be utilized by OpenCL applications, auto tunable algorithms become even more important than they were with more traditional CPU-based hardware. Auto tuning involves a program to be self-optimizing for the available hardware resources with a given set of capabilities. For a OpenCL application in production this becomes very important, because there are significant differences between the capabilities of GPUs, other accelerators and multi-core CPUs. Modern GPUs come in a number of different configurations when it comes to the number of available compute units and their grouping. Some differences exists inside the same generation of chips from the same manufacturer because of price/power differentiations of their products. Between generations major changes can occur in how the computational units are organized, for instance between the NVIDIA GT200 (“Tesla”) and GF100 (“Fermi”) architectures[9].

Even larger differences are found between GPU manufacturers. At the time of writing, the competing products from AMD and NVIDIA had fairly different architectures and specifications, even though the performance is comparable (depending on the application). Where the ATI Radeon 5870[10] has 1600 so-called processing elements in groups of 5 as stream cores, again grouped in 16 as SIMD engines, the NVIDIA Geforce GTX 480[11] has 480 “CUDA cores”, in groups of 32 as streaming multiprocessors. A program written for one architecture, does not nescessarily exceed the level of performance on the other without custom tuning. Currently, the compiler optimizations available in the GPU platforms is limited, so algorithms and libraries can get sizeable performance increases[15][1] by having the ability to auto tune and adapt to the environment in which it is run.

A few attempts have been made at creating auto tunable algorithms on GPUs. Nukada and Matsuoka[1] have created an auto tunable FFT library for GPUs

[10]ATI Radeon 5870 specs (retrieved 25.June 2010): `http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-specifications.aspx`

[11]NVIDIA GTX 480 specs (retrieved 25.June 2010): `http://www.nvidia.com/docs/IO/90025/GTX_480-470_Web_Datasheet.Final3.pdf`

with NVIDIA CUDA. It auto-generates an optimized kernel function and tries different combinations of threads and block sizes for a given FFT size and batch size at compile time (for the kernel function). With a small time penalty during compilation, they achieve up to 5.2x-8.0x faster performance than the NVIDIA CUFFT library (CUDA ver.2.1). This functionality would be a valuable addition to a production version of the program in our case study. For more information about the FFT library used in the case study, see Section 3.7. This does have some elements of the library from [1] as the code is dynamically generated, but does not implement the dynamic thread/block optimization.

Ideally, one would want the compiler to optimize the code for any given architecture to almost the same degree as an expert programmer would be able to do. Still, such a compiler has not been created, but to create it one would need a model for the performance with the different parameters possible. This was explored by Baghsorkhi *et al.*[16] by translating CUDA programs into so-called work flow graphs. They measured a fairly accurate execution time compared to their model on a matrix multiplication kernel, FFT and prefix sum kernels. An accurate model could potentially remove the need for "compile and try" auto tuning as in [1], and could lead to better automatic compiler optimizations.

But in the current state of auto tuning GPU algorithms, they are fairly complex to implement for non-trivial applications. Although attractive, it is not very feasible to apply auto tuning properties to prototype implementations without adding significant additional development time. However, as auto tuning library functions and algorithms become available, they will be used to speed up segments of compute intensive programs.

Chapter 3

Sound Preprocessing with Speex

The area of sound processing will be investigated in particular in this thesis, with regards to parallelization on GPU is preprocessing of human speech which is done in the Speex speech codec. This chapter contains background information on the case study and on sound processing in general. It begins with Section 3.1 that introduces a few fundamental digital audio concepts. Section 3.2 gives an overview of earlier work done with audio processing of GPUs. Section 3.3 introduces the problem area of the case study: Acoustic Echo Cancellation. Section 3.4 and Section 3.5 introduces the Speex library and its preprocessing components. Section 3.6 gives an introduction to adaptive filters and a specific implementation (MDF). Section 3.7 introduces the FFT and its implementation on GPUs.

3.1 Fundamentals of Digital Audio

To introduce a few digital audio concepts that will be used later in this chapter, first a few fundamental principals will be described.

The sound that is used in the context of this thesis, is always recorded by a microphone connected to the computing system in some way. The sound waves hitting the microphone membrane represent a continuous signal, but this has to be discretized into a finite series of numbers before they can be represented and processed by a computer. This conversion is termed *sampling*, and the end product is a series of *samples*, which is simply a series of numbers. An example of how a series of samples look when plotted over time[1], see Figure 3.1. The top plot shows the samples over a very short period of time with the small dots representing individual samples, the bottom plot is zoomed out to a larger time interval where the overall characteristics of the signal become visible.

The format of the numbers used to store samples can vary, but usually they are signed or unsigned integers stored with 8 to 24 bits precision. Since the samples represent a discrete version of the continuous sound waves, the more often the samples are stored per time-unit, the more closely they resemble the sampled signal. The frequency with which these samples are stored, is termed the *sampling rate*, and is usually in the range of 8kHz (telephones) to 192kHz (professional recordings and high-end music/movie formats). A common reference, the Compact Disc format. has a 44.1kHz sampling frequency and store the samples at 16-bit signed integers and audio stored with these parameters is often called "CD-quality". Since

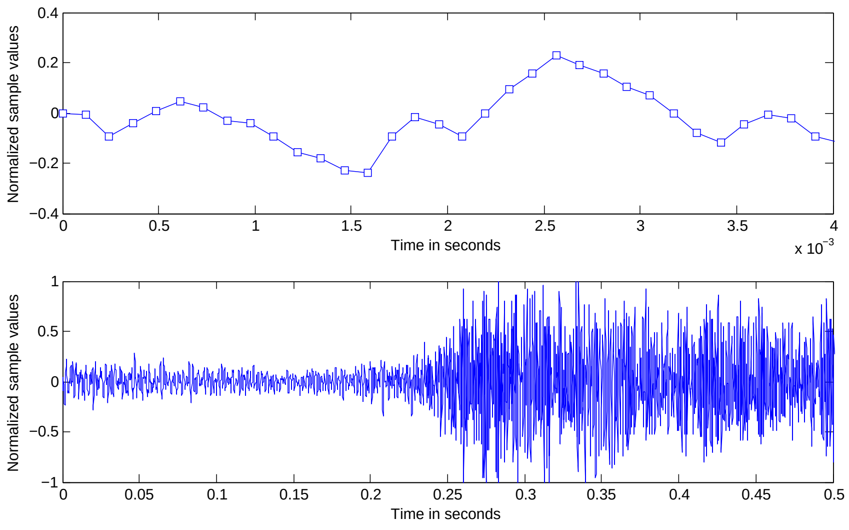


Figure 3.1: Waveform visualization, showing normalized floating point sample values over time.

computer hardware and software is generally most effective when given more than single values at the time to work, samples are often grouped into *frames*. As will be discussed later in this thesis, the most appropriate frame size is usually defined by the hardware architecture at hand, the operations to be performed with the frame and required filter latency. After processing of the samples is done, there are often even tighter constraints on the frame size due to network standards and the audio hardware used to playback the stream of samples.

[1]Plot is based on the "Visulizing Sound" demo in the MathWorks MATLAB software.

3.2 Audio Processing on GPU

Performing general computations on GPUs in itself is a fairly new venue of research, even more so for audio processing on GPUs. Very little has been done on using the GPU for such processing, but some examples do exist. Investigations into the feasibility of using GPUs for processing sound, have often lead to mixed degrees of success[17]. Often, GPU sound processing involves utilizing the GPU for 3D sound processing in games/visualization (which already uses the GPU)[17][18][19][20], or for heavier processing like speech recognition [21].

Cowan and Kapralos[4] investigated using the GPU for creating spatial sound for games and virtual environments by means of convolution. This is aimed at giving the user an impression of the space that he/she is moving in through an immersive sound experience. A large hall would for instance sound very different from a small office, first and foremost in terms of reverberation time, but also from other factors. Their implementation was developed using a shader language and achieved large speedups for larger data sets (larger number of audio samples). Jedrzejewski and Marasek[20] used ray tracing to calculate the detailed paths sound travels in a virtual environment, the methods resembles those used in ray tracing for visualization, which can be parallelized on GPUs[22]. The increased performance by using shader programs on GPUs allowed for more intricate scenarios in real time. Tsingos *et al.*[23] used an algorithm similar to level of detail to group sound sources in a virtual environment to avoid excessive computational load. They achieved comparatively higher performance when many sound sources are involved by offloading pre-mixing to the GPU using shader programs.

Trebien and Oliveira[18] investigated implementation of recursive 1D filters with their application to re-synthesizing sound effects when objects hit different materials such as wood, glass, plastic etc. They achieved an effective offloading of the CPU and a 2-4 times speedup over their CPU version.

In general, previous research has utilized shader programs, partly because they were the most viable option at the time, and partly because they easily integrate into real-time rendering of virtual environments. Few have investigated audio processing on modern GPGPU frameworks, using CUDA or OpenCL for development. These frameworks brings many advantages in ease of implementation and control, but some tradeoffs are necessary (see Section 5.2).

3.3 Acoustic Echo Cancellation

The focus area of this thesis within audio processing, is acoustic echo cancellation. But before describing the details of how it is canceled is described in detail in the next chapters, the problem of acoustic echo needs to be defined and understood. Acoutic echo describes a problem that occurs when using an at least two-way audio communication system where the recording device (microphone) can pick up the incoming sound sent from the other party through the playback device (loudspeaker). The sound from the speaker travels through a room that has some unknown acoustic properties into the microphone and gets sent back to the original sender. Delays with network transmission further the increases the delay of the sound, in addition to the time it takes the sound to travel from speaker to microphone, and reverberation in the room where the sound is played. A person in the other end will then hear an echo of their own transmission affected by the room in the other end, maybe even together with a new transmission.

This problem is not prevalent in a telephone that has a handset with a speaker in one end and a speaker in the other, or a headset with isolated earpieces that does not leak sound to the room or the microphone. But if a speaker-phone is used where the sound is played into the room or some types of hands-free equipment is used, the acoustic echo is more audible. A common use case in recent years is also to use a full computer for VoIP communication, often in a more casual manner than a traditional telephone, where a microphone is mounted by the computer and sound is played through speakers, maybe even an external sound system.

Acoustic echo cancellation is the act of “subtracting” the echo of the sound played through the speakers from the sound picked up by the microphone. The challenge lies in correctly predicting the effect the speaker, microphone and room acoustics have had on the sound before it was picked up by the microphone.

3.4 Speex

The acoustic echo canceller we have chosen to build our work on in this thesis, is part of a library called Speex[2]. It is an open source patent-free audio compression format designed for speech. The project aims to lower the barrier of entry for voice applications by providing a free alternative to existing, often expensive, proprietary speech codecs. Having a free software speech codec opens possibilities for many new applications without having to pay licensing fees for the underlying technology used to transfer speech as files or as a stream over a network. The Speex project is part of the non-profit Xiph.Org Foundation, that also maintains the relatively popular audio codecs FLAC (Free Lossless Audio Codec) and Vorbis, as well as the video codec Theora and the Ogg media container format.

Some of the features of the Speex codec itself are [3]:

[2]Speex website, and source of definition: `http://www.speex.org/`

- *Narrowband* (8 kHz), *wideband* (16 kHz), and *ultra-wideband* (32 kHz) compression in the same bitstream.
- *Intensity stereo encoding* for encoding a stereo signal without the need to transfer a full dual-mono signal.
- *Packet loss concealment* for transferring audio over an unreliable network (such as the Internet) with as few interruptions as possible.
- *Variable bitrate operation (VBR)* for better utilizing the available bandwidth by varying the bitrate depending on the content of the audio stream.
- *Discontinuous Transmission (DTX)* for stopping transmission when there is nothing to trasmit (in conjunction with VAD (see list below) and VBR).
- *Fixed-point* port, for running on embedded devices without floating point support.

In addition, the codec includes a preprocessing module, with, among others, the following features[3]:

- *Acoustic Echo Canceller (AEC)* used to avoid echo occuring on the remote end when audio playback is captured by the microphone. More details can be found about this module in later sections.
- *Noise suppression* for reducing the amount of background noise in the signal.
- *Automatic Gain Control (AGC)* for adjusting recording volume to a reference level, regardless of specific setups with differing input volume.
- *Voice Activity Detection (VAD)* for distinguishing between background noise and speech.
- *Adaptive Jitter Buffer* for make sure that the audio stream arrives in time and the correct order when transmitting it over a network, mainly by buffering.
- *Resampler* for converting audio between different sample rates.

Some of the software that currently make use of Speex[3]:

- *Adobe Flash*: A general framework for animations, multimedia content and video streaming often used on the Web that has Speex as one of its supported audio codecs[4].

[3]Speex application list (retrieved 15.July 2010): `http://www.speex.org/software/`

[4]Flash Player 10 Datasheet (retrieved 15.July 2010): `http://www.adobe.com/products/flashplayer/pdfs/Flash_Player_10_Datasheet.pdf`

- *Asterisk*: An open-source PBX (Private Branch Extension), or telephony system. It is typically used to route calls and provide services within a larger organization, for which commercial systems have typically been used. Interoperability between different telephone techonolgies (ie. PSTN/ISDN vs. VoIP) is one of its strengths, and Speex adds to the possible audio protocols that can be used.
- *Ekiga*: A videoconferencing application.
- *Google Talk*: A combined instant messaging and VoIP application where Speex is one of the available codecs[5].
- *LinPhone*: A graphical VoIP client.
- *Mumble*: A voice chat application for gamers, which uses much of the available Speex functionality to improve audio quality. This software project was started by the co-advisor for this thesis, Thorvald Natvig.
- *Wengo*: A VoIP service with an open source client that uses Speex.
- *A large number of games*: Speex is an attractive alternative for game creators to use for in-game voice-over and voice communication, that often are integral components of modern games.

Since Speex is open source software, most of the applications/libraries that uses it, are also free/open-source software.

3.5 Preprocessing in Speex

Preprocessing of sound when using Speex, is primarily done for two reasons:

- *Compressability*: With a voice codec, it is very important to remove unecessary noise (and unwanted sounds) from the signal before it is compressed, to dedicate as much as possible of the available bitrate to transmit the voice itself, and not noise. This ultimately leads to better voice clarity.
- *Quality*: The other obvious reason for doing preprocessing is improved audio quality for the end-users. Most of the time, one is only interested in hearing as clearly as possible the voice of the person one is communicating with, even through low-bandwidth network transmissions and in difficult acostic environments.

[5]Google Talk Speex support (retrieved 15.July 2010): `http://googletalk.blogspot.com/2006/10/speex-support.html`

The amount of preprocessing that should be done to the signal will always depend on the most common use case for the application, and how much processing power is available to it. The preprocessing filters are designed to be flexible, so that they execute in real-time on everything from small embedded processors to modern X86 processors with SSE instructions.

3.6 Adaptive Filters

A method used for acoustic echo cancellation, is to use adaptive filters. An adaptive filter is a self-adjusting algorithm that tries to approximate a real system, for instance room acoustics, based on a input signal over time. These filters are attractive to use for echo cancellation because exactly modeling all possible room acoutics, speakers and microphones physically correct in a single algorithm would be an insurmountable task. Instead some metrics of the quality of the result from the filter can be used to adapt it as best possible to physical system.

The challenge with adaptive filters is to make it accurate, and at the same time flexible enough to allow for changes in the system. When doing acoustic echo cancellation the system might change when for instance the microphone is moved around the room, or people or objects are moved around inside the room, altering the acoustics slightly.

The problem to be solved, is depicted in Figure 3.2. An adaptive echo cancellation filter takes the remote signal ($x(n)$), and earlier error values from the output. The signal produced by the filter is then subtracted from the signal that is coming from the microphone. The echo is produced by the sound output at the local side being picked up by the microphone ($y(n)$), together with the new speech that is to be transmitted ($v(n)$). The echo is affected by the room acoustics, which is an unknown system that the filter tries to adapt to. Notice that the echo is only removed from the output signal, and only improved for the remote end.

The algorithm used by Speex for echo cancellation, is based on a Multidelay Block Frequency domain adaptive Filter (MDF) design[25], and more specifically the Alternatively Updated MDF (AUMDF) variant (see Section 3.6.2). The implementation in Speex is pragmatically contructed with MDF as a base, with some modifications that improve the performance in practice. It is one of the better acoustic echo cancellers available as open-source without patent conflicts. Many echo cancellation algorithms are hidden within embedded products and proprietary systems/software. Earlier work by Herikstad[26] investigated performing solely the Fast Fourier Transforms (FFTs) in this filter on GPUs using CUDA and CUFFT. Since then, the available implementation has improved somewhat, with support for multiple microphones and speakers, but most of the general algorithm remains the same.

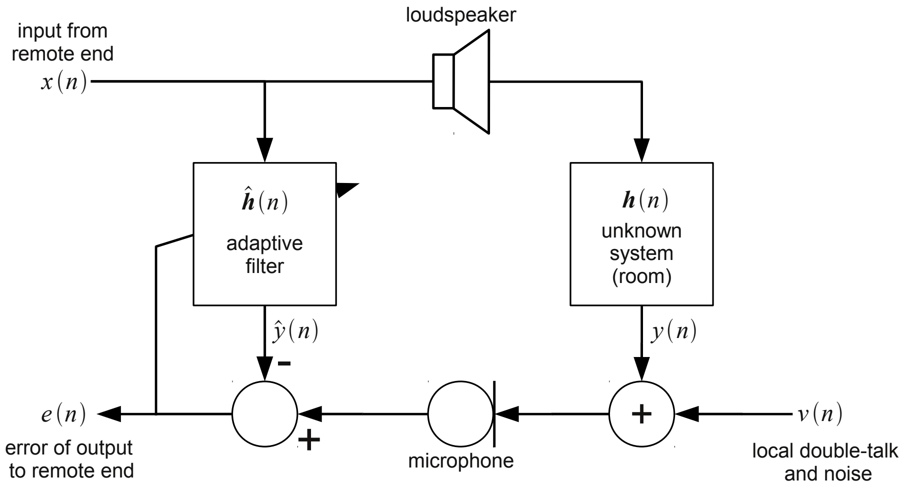


Figure 3.2: Block diagram of echo cancellation system. Referenced from [24].

It is important to note that the acoustic echo canceller component of Speex is independent of the voice codec, and can be used for any sort of audio signal. The filter itself takes raw sample data as input and output, which means it can be easily combined with other software packages.

3.6.1 MDF

MDF[25] is a variation of the Least Mean Squares (LMS) filter, where an adaptive filter is used to approximate an unknown desired filter. It approximates the filter by finding filter coefficients that minimize the least mean squares of the error signal (the difference between the desired and actual signal, see Figure 3.2). Least mean squared error is a common statistical error quantity used in signal processing.

MDF is a frequency domain implementation of an LMS filter where blocks of fixed size that can be smaller than the filter length, is processed in the frequency domain. There exists other methods, such as FLMS for computing a LMS filter in the frequency domain, but it is inflexible with respect to the size of the blocks of data that needs to be transformed into the frequency domain using FFTs (see Section 3.7). If the block size used is of the same size as the FLMS size, then it becomes identical to MDF. With smaller blocks, it is possible to make better use of embedded hardware for FFTs, and thus avoid performance problems and inaccuracies that can occur with larger FFTs, and with smaller blocks the latency of the filter can be reduced so it can adapt faster to changes.

For the rest of this thesis, the total length of all the blocks in the MDF, will be termed the *echo tail*. This tail consists of a number of blocks (frames) back in time, often of the size that fits well with high-performance FFT implementations. The length of the echo tail determines how long backward in time the filter can cancel echo, this depends on the acoustics of the room in question. In addition to room reverberation, audio equipment and computers also add delay. The sample rate also play into the length of the echo tail, as a higher sample rate will need a longer tail to cover the same period of time, and will naturally be more computationally demanding.

The following, is an overview of the flow of input frames through the filter (see Figure 3.3, enumerated according to the following list):

1. The current and last frames are taken as input and combined using a overlap-save or overlap-add technique[27]. This allows the block filter to better approximate a continuous filter.
2. The overlapped frame is transformed into the frequency domain using a 1-dimensional real-to-complex FFT. The size of the FFT is twice the size of the input frames.
3. The transformed frame is added to the stored blocks (the "echo tail"). The blocks are stored in a FIFO buffer, the oldest one is removed.
4. All the blocks are multiplied with corresponding stored weights.
5. All the multiplied values of the blocks and weights, are added together into one block.
6. The block is transformed back to the time domain using a 1-dimensional complex-to-real FFT (inverse order of above).
7. Half the points in the transformed block is output as the filtered frame.
8. After the final frame is calculated, the value is subtracted from the initial frame to find the error value.
9. The error frame is padded with N/2 zeroes and transformed into the frequency domain using a real-to-complex FFT.
10. Each weight is updated by adding them to the calculated least-mean-square of the transformed error frame.

The major addition to the plain LMS algorithm in the Speex implementation, is additional double-talk detection[24]. A special way to update the learning rate of the filter is used based on the current amount of noise and double-talk in the signal.

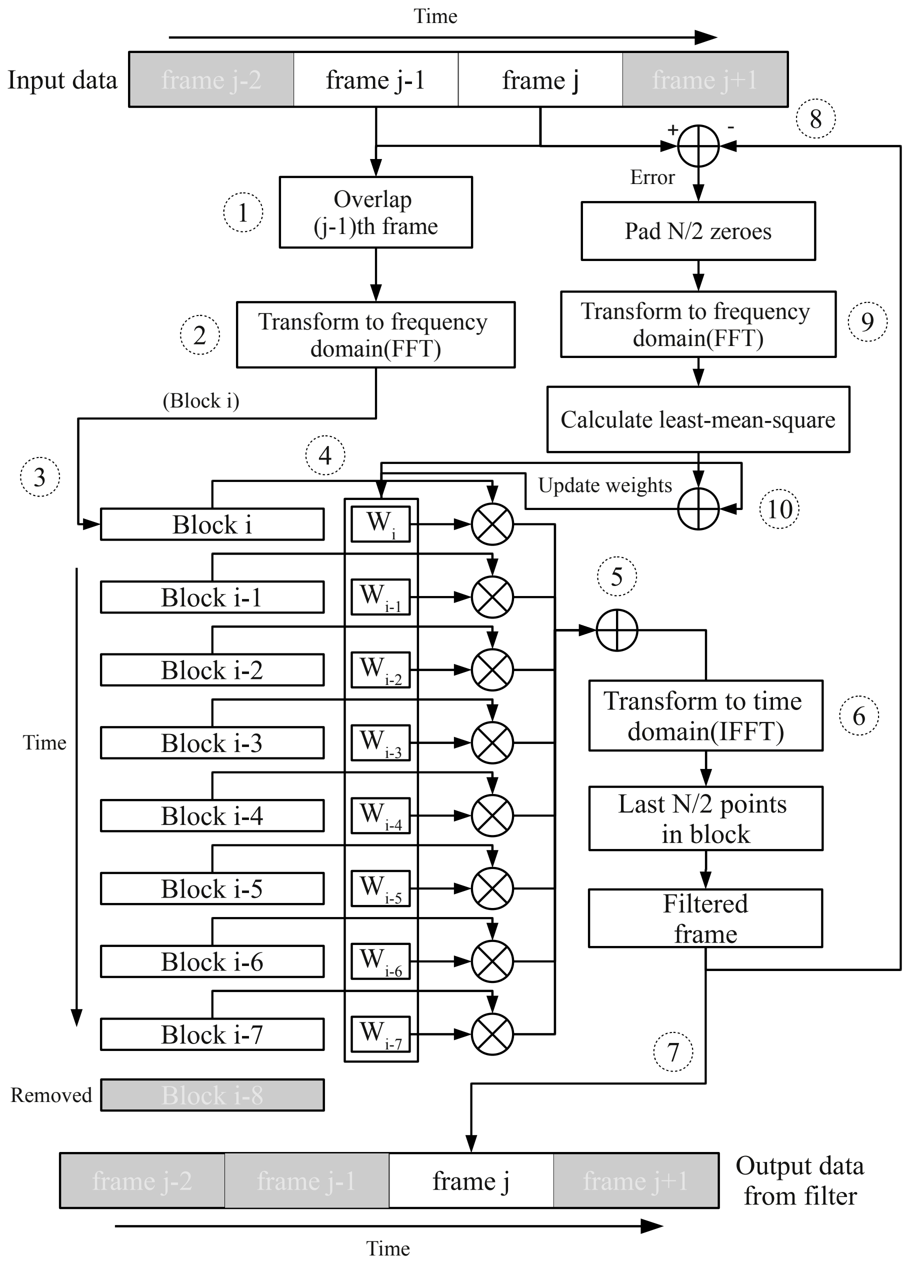


Figure 3.3: Block diagram of an MDF with an 8-block echo tail. Designed after illustrations in [25] and [26].

Double-talk occurs when the person in front of the microphone is talking at the same time as speech is played through the speakers from the far-end. Double-talk can confuse an adaptive filter if not handled specifically, ideally the cancellation of the speech picked up from the speaker should not be affected by the person talking. In the Speex implementation, a contiuous learning rate variable is used that can be adjusted to compensate for disturbances that should not effect the filter. The learning rate is given as:

$$\hat{\mu}_{opt}(k,\ell) = min\left(\hat{\eta}(\ell)\frac{\left|\hat{Y}(k,\ell)\right|^2}{\left|\hat{E}(k,\ell)\right|^2}, \mu_{max}\right) \tag{3.1}$$

Here, k is the frequency index, ℓ is is the frame index. $\hat{Y}(k,\ell)$ is the echo signal transformed into the frequency domain, $\hat{E}(k,\ell)$ is the frequency domain version of the error signal. $\eta(\ell)$ is a frequency independent leakage coefficient and μ_{max} is a design parameter for the maximum adaption rate, and prevents the filter from becoming unstable. When double-talk is present in the microphone signal, the filter should adapt at a much slower rate than normal. Double talk is differentiated from a change in echo path (which also leads to large errors) by a leakage estimate. If the echo path changes there is a large correlation between the power spectra of the error and the estimated echo, but not during double-talk.

3.6.2 AUMDF

The particular MDF implementation in Speex, is a variant called Alternatively Updated MDF (AUMDF). The result of this algorithm is that only a single weight is updated for each frame that is processed by the filter, in contrast to a separate weight for each frame of the echo tail as with regular MDF. Without AUMDF, both an FFT and an IFFT would need to be run for each weight that make up the echo tail and for each input and output channel. This can be very computationally demanding, especially for longer echo tails. If for instance a frame size of 512 samples is used with an echo tail of 20 (combined) frames and two output channels, this corresponds to 40 FFTs and 40 IFFTs of length 1024. However this case, AUMDF reduces this to 2 FFTs and 2 IFFTs, but with a trade off in accuracy.

3.7 Fast Fourier Transform

In many audio processing applications it is more efficient to operate on sound in the frequency domain. This can often leave to favorable performance over alternative methods, but depends on a fast transformation of the input data, which naturally is recorded in the time domain. The transformation used in general is called the

Disrete Fourier Transform (DFT), but what we are interested in is a fast algorithm for signal processing, and these are termed Fast Fourier Transforms (FFTs). The two terms are often interchanged, but FFT will be mostly used in the rest of this text (although DFT is the correct term when discussing the mathematical transformation itself). Cooley and Tukey[28] popularized the first FFT algorithm in the era of electronic computers, which could perform a DFT in $Nlog(N)$ operations.

The FFT is a very important part of Speex in general, including the echo filtering procedure. It has an abstraction layer that makes it possible to use several different implementation, and at the time of writing the following were supported: The FFTW (Fastest Fourier Transform in the West) library, the KISS (Keep It Simple, Stupid) library, the FFT implementation from the MKL (Intel Math Kernel Library), as well as a custom implementation from the OGG project (smallft) and a implementation with support for for fixed-point architectures.

FFTW is a highly optimized FFT library for CPU, and its integration with Speex was used as a starting point when developing the OpenCL implementation, because it is well documented and easily available, due to the fact that it is free software. It utilizes so-called "plans" that dynamically optimize the operation based on the parameters of the transform. The two operations that are used by Speex are one-dimensional DFTs, for purely real to complex, and purely complex to real transforms. The fact that the input is purely real or complex, opens up for further performance benefits (the manual states a factor of two improvement in speed and memory usage).

The DFT transform is defined by the FFTW library as [29]:

$$Y_k = \sum_{j=0}^{n-1} X_j e^{-2\pi jk\sqrt{-1}/n} \tag{3.2}$$

Here n is the number of real numbers in X. The real to complex transform only outputs $n/2+1$ complex number, the rest is redundant because of symmetry.

The inverse (complex to real) transform is defined as:

$$Y_k = \sum_{j=0}^{n-1} X_j e^{2\pi jk\sqrt{-1}/n} \tag{3.3}$$

This results in n real numbers. FFTW calculates so-called unnormalized transforms, so to get the original real numbers back after both a forward and inverse transform, one needs to multiply the input to the forward transform by $1/n$. This is performed by the Speex wrapper functions for FFTW.

While FFTW is a well-written library, in this project it must be possible to calculate the FFT purely on the GPU. CUFFT is a closed-source (but included in the CUDA SDK) FFT library from NVIDIA, that computes transforms on the GPU, and has an API that has a high resemblance to that of FFTW. It would be an excellent candidate for replacing FFTW in Speex, and exactly this pomt was explored by Herikstad[26]. It has also been used in as a building block in numerical simulations, for instance by Stantchev *et al.*[30] which used it computational kernels for simulating plasma turbulence and compared it to a highly optimized CPU implementation. CUFFT is also commonly used as a benchmark for GPU FFT implementations [1]. It is highly beneficial to store both input and output of the FFT in GPU memory, if most of the sound preprocessing algorithm is to be implemented on GPU. This is necessary to avoid as many data transfers over the PCI-E bus as much as possible as it is a slow bus (compared to global memory on the GPU), to avoid the extra latency such an operation would introduce.

CUFFT has this functionality, but in this project it is not a feasible option because it is only available on the CUDA platform from NVIDIA and requires an NVIDIA GPU. This project investigates using OpenCL to create a more general implementation, and using CUFFT would significantly restrict the number of platforms that could utilize the library. This is important both to support traditional GPUs from other vendors, but also for possible use on embedded devices with accelerator chips or smaller GPUs, or even just multi-core CPUs with the same codebase.

Several FFT alternatives were considered for the OpenCL implementation:

- *Create a custom FFT from scratch:* Creating a custom FFT implementation in OpenCL from scratch, could result in a small, simple implementation that only includes the transformations necessary for Speex. But implementing efficient FFTs on GPUs, as explored earlier, is not a trivial problem. It would also limit benefits from library developments in the future, that can occur in for instance CUFFT.
- *Use a FFT implementation from available OpenCL SDKs:* Both the released OpenCL SDKs from AMD and NVIDIA include examples for FFTs, but none of them are particularly well suited for the specific transform required by this work. It might also pose trouble (among others, with license/copyright) to decouple the examples from the SDK which they are a part of, and include them in Speex. They may also include platform-specific optimizations by the GPU vendors.
- *Use another third-party implementation:* Apple has released an open-source simple FFT library for OpenCL[6], that should work across platforms, and is actively maintained. This makes it possible to utilize a versatile library with the possibility of adapting it to our spesific needs. A few shortcomings of the library include the lack of (official) OpenCL CPU-support, lack of a specialized real-to-complex and complex-to-real transforms with improved

[6]Apple OpenCL FFT library (retrieved 20.July 2010): `http://developer.apple.com/mac/library/samplecode/OpenCL_FFT/`

performance (which CUFFT also lacks) and the use of some Mac-specific libraries (but only in the example-application).

The library that was chosen was the Apple FFT library, but with a few modifications as described in Section 4.4. It is based on work in [31] and [32], so it can be assumed to be fairly well optimized.

Chapter 4

Optimizing Speex Echo Cancellation for OpenCL

This case study reimplements the MDF echo filtering component of the Speex library using OpenCL. The goal was to create a completely compatible function to the existing one for our testing parameters. The existing implementation is written in pure C for use both on PCs and embedded devices without floating point support. Its datastructures are stored in a large C struct which is replicated to be compatible with our implementation. This makes it possible to exchange the existing filter with the OpenCL implementation without much difficulty. Several development platforms were used to make sure that the OpenCL code was widely compatible (small differences exist between implementations). These were the AMD Stream platform (CPU/GPU) and the NVIDIA CUDA 3.0 SDK, both on Linux-based systems.

This chapter starts with Section 4.1 that describes how the new implementation is integrated with Speex. Further, Section 4.2 introduces the application created to test the OpenCL implementation and Section 4.3 describes in more detail how timing and verification was done. Section 4.4 details how the FFT was implemented on GPU and how it was adapted to the dataformat expected by the Speex library. Section 4.5 lists some of the steps taken to parallelize the echo cancellation filter. Documentation of the kernel functions (in the OpenCL implementation) themselves can be found in Section 4.6. After the initial implementation was created, a few experiences affected further development. Section 4.7 describes some of the challenges faced developing the OpenCL for different hardware platforms, while Section 4.8 examines some of the performance limitations found during development and a few of the efforts to best utilize the computing power in the GPU are listed in Section 4.9. Lastly, Section 4.10 investigates how co-processing can be applied to the filter and a simplistic solution is presented for current systems.

4.1 Integration with Speex

The OpenCL implementation was made to fit into the Speex framework as far as possible. The echo filter code mainly consists of three components:

- *A filter state*: This is a very large C structure with all the buffers (26 in total) used by the filter. It contains both temporary values and values that persist between input frames.
- *Initialization/cleanup*: Mainly allocates and frees the buffers in the state struct, as well as setting the default values and constants.
- *The main filtering function*: This is called for each input frame and contains all the logic of the filter.
- *Subroutines*: Common subroutines that are called by the main filtering function.

The filter state is reused exactly as it is, to be able to run the filters with the exact same input data and to compare intermediate results. Initialization and cleanup functions are completely replaced by calls to the OpenCL runtime and buffers are created similar to those that already exist in the filter state. The main filtering function is replaced with the OpenCL implementation that uses the same prototype definition and is compatible with the existing implementation. Some of the filter logic is copied from the library version, but most is rewritten from scratch. Some of the subroutines are replicated, but many were combined with other operations (see Section 4.8).

4.2 The Testing Application

Included in the Speex library is a testing application for the echo cancellation functionality, simply called “testecho”. This application was used as the basis for testing both the original library function and the OpenCL version. It reads two files from disk: A file with the reference (without echo) audio, and a file with echoed audio. Both input files are raw samples (no container format) with a configurable sample rate. Frames from both the files are passed to the echo cancellation filter, which in turn returns a single output frame that is written to an output file. Parameters that can be adjusted, include frame size (how large frames are read and processed at the time, by default 128 samples), and the echo tail length (how far back data is stored in the filter, by default 1024 samples). The original application does not support multi-channel input/output (*i.e.* the use of several microphones and/or speakers).

The modifications made to accommodate the OpenCL version are fairly simple:

- An additional initialization function was added to initialize the OpenCL environment, and allocate the needed buffers.
- Calls to the initialization and cleanup functions of the timing and verification data (see Section 4.3).
- A new function to copy the complete echo cancellation state in order to be able to call both the library and the OpenCL version with the same input data. The filter changes several of the buffers and values in the state by design, not only the output buffer.
- A call to the OpenCL version of the speex_echo_cancellation() function, that processes each frame.
- The application was adapted for different input parameters to create a more realistic test scenario with a higher sample rate, larger frame sizes and multiple channels (see Chapter 5 for more details).

The testing application consists of these main steps:

1. Initialization phase
   (a) Open files for the speaker signal, microphone signal and the output. Optionally these files can store interleaved data for several channels.
   (b) Allocate input/output buffers, used as temporary storage when reading/writing files. The size of these buffers must be of a multiple of the frame size chosen and the number of channels. Separate buffers are used for the OpenCL version, in order to always provide it with the same data.
   (c) Initialize the echo state, which is a large struct that is used for storing all data used by the echo canceller. This contains among other things the echo tail, temporary buffer and configuration parameters. A separate version is created for the OpenCL version using the same format as the original program.
   (d) Initialize the Speex preprocessor state. Similar to the echo state, this holds information for the preprocessor, which among other things removes noise before the output is written to file. This functionality was not altered in any way in this case study.

   (e) Initialize OpenCL and allocate device buffers required (corresponding to those in the echo state). This step sets up the OpenCL environment, retrieves a reference to the compatible hardware device(s) on the system and compiles the OpenCL kernel code at runtime.
   (f) Initialize the timing system by allocating any memory required.

2. Main loop

   (a) Read data from the input files (microphone and speaker signals) into the I/O buffers.
   (b) Initialize verification and timing checkpoints by allocating the memory needed and setting initial values.
   (c) If both the original CPU filter and the OpenCL versions are run side-by-side for debugging, some additionals steps are taken to assure that they have the same input data. First, a complete copy is made in main/CPU memory of the echo state of both the original and OpenCL filters (as the echo state is also part of the input data). In addition, the I/O buffers are copied, so that both filters are passed the same data. This can be skipped if only one is run (production conditions).
   (d) The main echo cancellation function is called.
   (e) Cleanup functions for the verification and timing checkpoints to reset the data before the next iteration.
   (f) Run the output through the rest of the Speex preprocessor.
   (g) Write the output frame(s) to file.

3. Cleanup

   (a) Print timing statistics (average iteration time etc.) and clean up system memory used for timing.
   (b) Clean up the echo state for the original filter state and the copy. This is done using original functionality in Speex.
   (c) Clean up the Speex preprocessor state.
   (d) Close all opened files.
   (e) Free I/O buffers, and their copies.
   (f) Free allocated OpenCL device memory and associated variables.

This process is illustrated in Figure 4.1. Grey boxes are timing/verification functions that are disabled for benchmarking. Orange boxes are functionality implemented using OpenCL. The white boxes represent the original functionality of the test program. Functions that are horizontally grouped do similar tasks. At several occasions, the OpenCL functionality is performed in tandem with the original functionality. The program as shown, is only run for debugging purposes, since most computations are replicated. A production version would either perform load

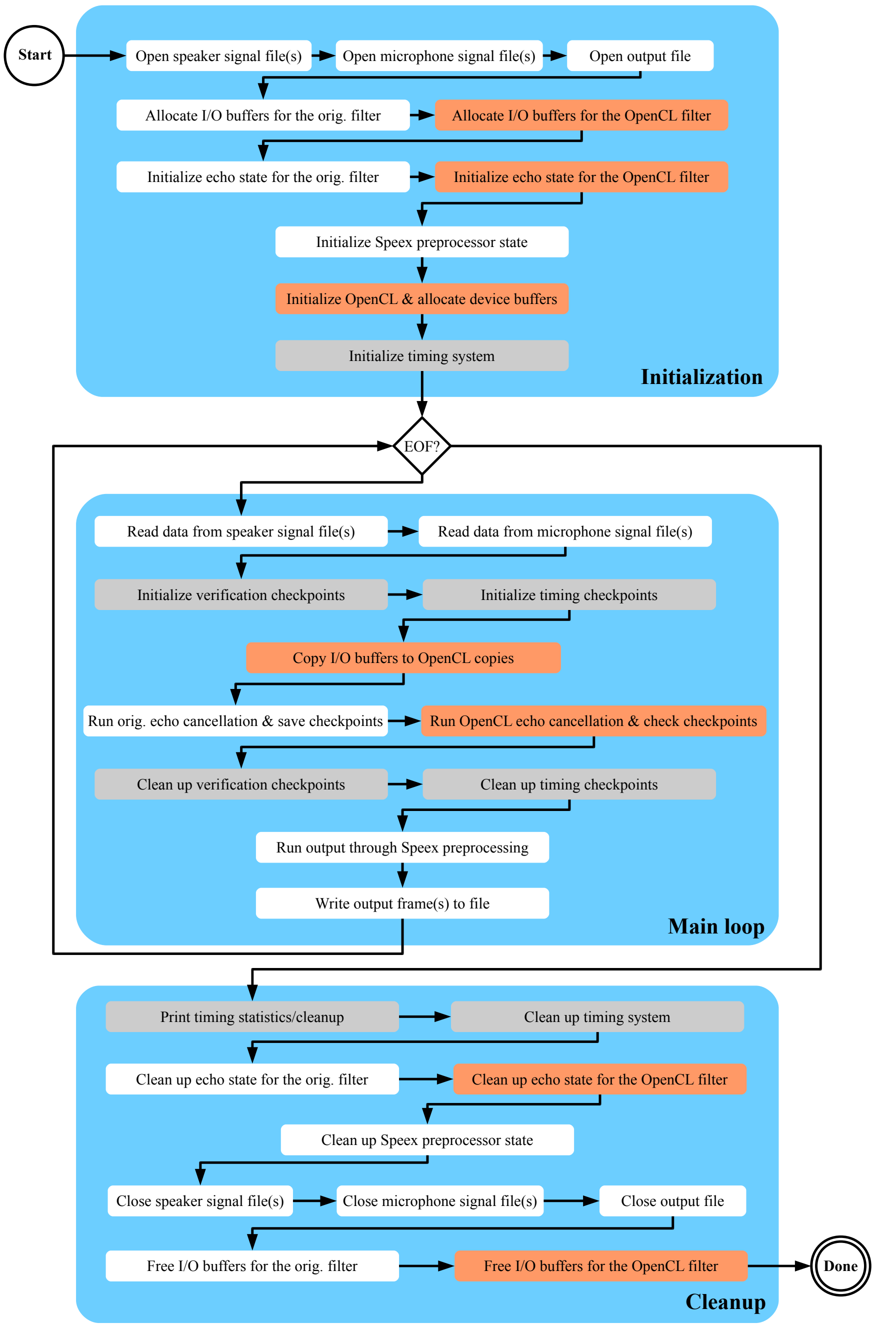

Start
Open speaker signal file(s)
Open microphone signal file(s)
Open output file
Allocate I/O buffers for the orig. filter
Allocate I/O buffers for the OpenCL filter
Initialize echo state for the orig. filter
Initialize echo state for the OpenCL filter
Initialize Speex preprocessor state
Initialize OpenCL & allocate device buffers
Initialize timing system
Initialization
EOF?
Read data from speaker signal file(s)
Read data from microphone signal file(s)
Initialize verification checkpoints
Initialize timing checkpoints
Copy I/O buffers to OpenCL copies
Run orig. echo cancellation & save checkpoints
Run OpenCL echo cancellation & check checkpoints
Clean up verification checkpoints
Clean up timing checkpoints
Run output through Speex preprocessing
Write output frame(s) to file
Main loop
Print timing statistics/cleanup
Clean up timing system
Clean up echo state for the orig. filter
Clean up echo state for the OpenCL filter
Clean up Speex preprocessor state
Close speaker signal file(s)
Close microphone signal file(s)
Close output file
Free I/O buffers for the orig. filter
Free I/O buffers for the OpenCL filter
Done
Cleanup

Figure 4.1: Flow diagram of test application.

balancing between the original and OpenCL versions (see Section 4.10 for a simple load-balancer), or run one of the versions exclusively. The logic for load balancing is not shown.

4.3 Timing and Verification

Simple frameworks were created for the purpose of easing debugging and to time the different implementations.

Verification

To avoid writing custom testing code for all stages of the echo filter, a simple framework was created for vertification of the results. This involve storing snapshots of select buffers from the echo state at certain points in time called “checkpoints”. These are numbered single values or buffers, stored with one of the datatypes float, integer or short. Reference values are generated during the run of the original library implementation. When the OpenCL version is run with the same input data, it calls functions that checks the data against the reference values. If the difference between the values is larger than the specified margin of error, both the reference and the erranous data is printed to console for investigation. The data can be stored either in device global memory or main memory. In the case where it is stored on the GPU, it is transferred to a temporary buffer on the CPU. The verification can be easily disabled for benchmarking as it adds a significant amount of overhead to the total computations.

Timing

Timing the program is performed in much the same manner as verification. Each implementation calls a function that stores the elapsed time in an array as it progresses together with a string describing that particular point. At the end of the filter, the values are printed in a list to console, relative to the first value. Since the only operating system used for benchmarking was Linux, the sys/time.h header was used with its `gettimeofday()` function that has theoretically down to a microsecond precision, but depends on the available hardware clocks on the system. This timing framework was used only to measure the overall progress of the filter, including overhead. GPU timers were not used to measure kernel execution time as the NVIDIA Compute Profiler was used when optimizing the kernels individually.

4.4 Integrating the FFT on GPU

As described in Section 3.7, the Apple OpenCL FFT library was chosen as the best available candidate meeting the requirements of the case study. Some adaptions were required because of the data format used by Speex and the FFT library. The differences between FFT libraries is usually handled in Speex, through the use of FFT wrapper functions with generic prototypes which are then used throughout the framework. Adding a GPU FFT-library to this wrapper was explored by Herikstad[26], but transfer of data to and from device (GPU) memory then needs to be included unless the whole framework is implemented on GPU. For this case study, separate FFT functions are created instead, so as to execute FFTs using device memory only in the OpenCL implementation of the echo cancelling. The process of implementing the FFTs went through several iterations before the final solution was explored:

1. Firstly, the library test program was adapted to run on Linux, it was dependent upon Mac OS X specific libraries for timing and validating the results. FFTW was used to validate the results, and standard Unix timing API was used for timing. Several unnecessary test-cases was removed, and the test-case with 1-dimensional DFT with real data (similar to the one used by Speex) was implemented. Functions to convert to/from Speex FFT format was implemented and tested against FFTW by looking at the existing FFT wrappers.

2. The kernel functions for converting the input/output data was added to the echo cancellation source files. This made each transformation a three-stage process (different sets for inverse and forward transforms):

   (a) *Prepare* - The input data is copied (with offset if needed) into a buffer with the correct format for the OpenCL FFT.

   (b) *Execute* - The FFT is executed using the external library functions.

   (c) *Finalize* - The output data is copied (again, with offset) into the output buffer with the correct format for Speex.

   As discussed in Section 4.8, this leads to very high overhead and overall poor performance because of the complex program structure, so a new solution was needed. The OpenCL FFT library also has some initialization overhead for each program run, as it dynamically generates kernel functions depending on the device it is executing on.

3. The final solution involved taking the source code for the kernel generated by the OpenCL FFT library, and including it among the rest of the kernel functions for the echo filter and integrating it with the prepare and finalize kernels to reduce the whole process into a single kernel call. It still copies the data to separate input/output buffers used by the FFT to be able to change

> the data format without ruining the original content. But the combined kernel proved to have much higher performance and properly support batch runs with several transforms in parallel.
>
> A problem with this approach, is that the FFT size is hard coded in the implementation for specific hardware. But it would not necessarily create performance problems on such small sizes (256/1024 samples) as one move to other devices. A larger problem would be to if different frame sizes were to be used for filtering, but it should not be a large challenge to adapt to the sizes needed, based on the source code output of the OpenCL FFT library for those sizes.

As mentioned, the format for the data in Speex and FFTW and the OpenCL FFT library have some differences. The buffers in Speex are mostly made to be as small as possible to allow for embedded devices with little memory available, where the FFT libraries often are made to be as flexible as possible with regards to possible input. Another issue is that the OpenCL FFT library only has complex to complex transforms available, which are slightly different than transforms involving purely real numbers. The complex to complex transforms can be used with real input data with some minor changes to the data[33], but one does not get the performance benefits of the specialized implementations, as is available with FFTW.

4.4.1 FFT Data Preparation

The data preparation step for the forward FFT, taking into account both the different data formats and utilizing a complex-complex DFT, is described in the Algorithm 1. The *input* parameter is an input array of n real numbers in the time domain (this is also the FFT size) and x is the array of complex numbers passed to the FFT. Note that all arrays are zero indexed.

It is a very simple algorithm where each real number is mapped to the corresponding real part of a complex number in the time domain. So for an FFT of length 256, one would have 256 real numbers as input and 256 complex numbers as output of the preparation stage.

Algorithm 1 Preparation step for OpenCL forward FFT.

```
$m \leftarrow 1/n$
for $i = 0$ to $n$ do
    $x[i].real \leftarrow input[i] \times m$
    $x[i].imag \leftarrow 0$
    $i \leftarrow i + 1$
end for
```

Preparing data for an inverse FFT is slightly trickier, since we have $n/2+1$ complex numbers in the frequency domain as input, packed as an interleaved float array. This interleaving means that the real component of the first number is stored in $X[0]$, the imaginary component is stored as $X[1]$, the real component of the second number is stored as $X[2]$ and so on.

The inverse FFT is still executed with the same length as the forward FFT, even though the input is $n/2+1$ complex numbers. As described in [33], to execute (an inverse) complex to real FFT, one needs to mirror the numbers around a middle element. So for instance, for an FFT size of 256, the 129th element should be the same as the 127th (with the imaginary part negated), the 130th should be the same as the 126th and so on. This mirroring will cause the output data after the inverse FFT to be 256 complex numbers in the time domain, with only real components. This is all summarized in Algorithm 2. *input* is the input array of $n/2+1$ complex numbers stored interleaved (offset by -1 after the first element, so the last element is the imaginary part of the last complex number) and X is the array of complex numbers passed to the inverse FFT.

Algorithm 2 Preparation step for OpenCL inverse FFT.

```
$X[0].real \leftarrow input[0]$
$X[0].imag \leftarrow 0$
$X[128].real \leftarrow input[255]$
$X[128].imag \leftarrow 0$
**for** $i = 1$ to $n/2 - 1$ **do**
    $X[i].real \leftarrow input[2 \times i - 1]$
    $X[i].imag \leftarrow input[2 \times i]$
    $i \leftarrow i + 1$
**end for**
**for** $i = n - 1$ to $n/2 + 1$ **do**
    $X[i].real \leftarrow X[n - i].real$
    $X[i].imag \leftarrow -X[n - i].imag$
    $i \leftarrow i + 1$
**end for**
```

4.4.2 FFT Data Finalization

The finalization functions convert the output data from the OpenCL FFT library back to the format that Speex expects. The forward FFT function is listed in Algorithm 3. X is the result of the FFT ($n/2+1$ complex numbers) and *output* is the array of interleaved complex numbers passed back to Speex.

The algorithm itself is simple, but note that the imaginary part of the first and last numbers are truncated.

Algorithm 3 Finalization step for OpenCL forward FFT.

```
$output[0] \leftarrow X[0]$
for $i = 1$ to $n/2$ do
    $output[2 \times i - 1] \leftarrow X[i].x$
    $output[2 \times i] \leftarrow X[i].y$
    $i \leftarrow i + 1$
end for
```

As mentioned earlier, the result of the inverse FFT when the input is mirrored correctly, are purely real numbers. So the finalization algorithm listed in Algorithm 4 simply extracts all the n real parts of the complex numbers and stores them in the output array. x is the result of the inverse FFT.

Algorithm 4 Finalization step for OpenCL inverse FFT.

```
for $i = 0$ to $n$ do
    $output[i] \leftarrow x[i].x$
    $i \leftarrow i + 1$
end for
```

4.5 Parallelization of MDF

OpenCL programs must be written to be inherently parallel in order to fully utilize the available hardware. But the original MDF implementation in Speex is written to be completely sequential, except for any possible optimizations done by the compiler, although this is not available used in current software.

4.5.1 Calculate Frames in Parallel

The filter loops through all the samples in the input frame or combined window (current and previous frame). In most cases, these are trivially parallelizable and can be calculated independently. Typical frame sizes allow for a elegant mapping to the computational units on the GPU with parallelism in the order of hundreds, and can be executed in a few operations. Although this exploits the computational units in an efficient way, it does not add enough computational load by itself to mask memory accesses and allow for good throughput. Optimally, these operations are good candidates to combine with other operations to keep the number of kernel launches to a minimum(see Section 4.8), but certain of these operations have dependencies that limit them to being executed on a single thread, so execution on a CPU is preferred.

4.5.2 Calculate I/O Channels and Echo Tail in Parallel

In addition to the frames themselves, the outer loops often consist of a number of input or output channels. These can parallelized in larger blocks, adding an additional level of computation. The FFTs are also mostly executed one time for each channel, which translates into larger batches. But as is the case with frames, some operations have dependencies that are not possible to run in parallel. The echo tail is in the same class as the channels, and this is most dominant when updating the weights themselves (see Section 4.9 for more information about how this can even further affect performance).

4.5.3 Identify Independent Sections

The overall architecture of the echo filter is, for the most part, a pipeline. This requires that the operations are executed in order and with some form of global synchronization. The operations that can be executed out of order can benefit overall performance if they are performed when it is most fitting and preferably in parallel with other operations.

4.6 OpenCL Kernel Functions

Using the techniques listed in Section 4.5, the the final implementation was created as an alternative version of the `speex_echo_cancellation()` function with subroutines, which is called `speex_echo_cancellation_opencl()`. It can coexist in the library together with the original function, to facilitate testing and validation.

The following is a list of all the kernel functions that make up the OpenCL implementation (see also Figure 4.2 for an overview), sorted by first appearance in the code, but grouped into two groups: MDF and FFT kernels. The FFT kernels are not purely FFT related, but contain some integrated functionality to reduce the number of kernel calls required. If not listed otherwise, combined kernels are always made task parallel on the combined tasks unless some data dependency is present. Some descriptions are based on documentation from the Speex source code.

4.6.1 MDF Kernel Functions

clMDF_prepare_shift_preemph_freq_data The first kernel in the filter that is run on the device, this kernel applies pre-emphasis to the input data (after it

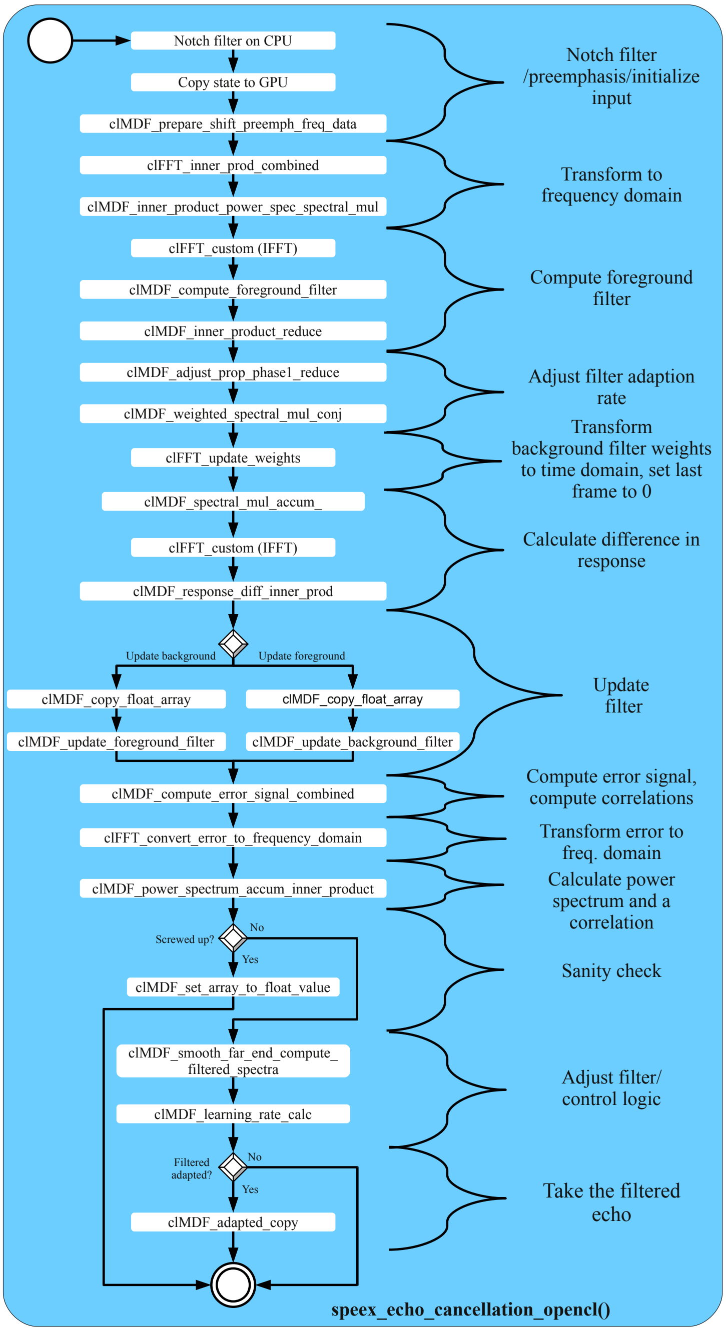

Notch filter on CPU
Copy state to GPU
clMDF_prepare_shift_preemph_freq_data
clFFT_inner_prod_combined
clMDF_inner_product_power_spec_spectral_mul
clFFT_custom (IFFT)
clMDF_compute_foreground_filter
clMDF_inner_product_reduce
clMDF_adjust_prop_phase1_reduce
clMDF_weighted_spectral_mul_conj
clFFT_update_weights
clMDF_spectral_mul_accum_
clFFT_custom (IFFT)
clMDF_response_diff_inner_prod
Update background
Update foreground
clMDF_copy_float_array
clMDF_copy_float_array
clMDF_update_foreground_filter
clMDF_update_background_filter
clMDF_compute_error_signal_combined
clFFT_convert_error_to_frequency_domain
clMDF_power_spectrum_accum_inner_product
Screwed up?
No
Yes
clMDF_set_array_to_float_value
clMDF_smooth_far_end_compute_filtered_spectra
clMDF_learning_rate_calc
Filtered adapted?
No
Yes
clMDF_adapted_copy
speex_echo_cancellation_opencl()
Notch filter /preemphasis/initialize input
Transform to frequency domain
Compute foreground filter
Adjust filter adaption rate
Transform background filter weights to time domain, set last frame to 0
Calculate difference in response
Update filter
Compute error signal, compute correlations
Transform error to freq. domain
Calculate power spectrum and a correlation
Sanity check
Adjust filter/ control logic
Take the filtered echo

Figure 4.2: Flow diagram of the OpenCL MDF implementation

has been converted to floating point numbers from integers on the host), shifts the main time domain buffer (the echo tail) to make room for new data (far end signal) and inserts the far end signal after applying pre-emphasis. Applying pre-emphasis to the input is only parallelized using a single work-group of the device (because synchronization is needed), and has to loop through a number samples depending on the frame size and the number of input channels. Shifting the main time domain buffer is parallelized for both samples[1] and output channels using several work-groups. The shifting of the frequency domain buffer is parallelized on samples and echo tail frames, but have to loop through speakers for synchronization.

clMDF_inner_prod_power_spec_spectral_mul Combines an inner product calculation and the first part of a "spectral multiply accumulate" operation. More details can be found in the description for each dedicated kernels. The spectral multiply accumulate operation requires an additional reduction before the finished result can be used. The inner product is paralellized by samples and needs to loop through output channels. The spectral multiply accumulate operation is paralellized by samples, echo tail frames, input channels and output channels.

clMDF_compute_foreground_filter A simple function calculates the difference between the error buffer and the input buffer for a frame, and stores it in a separate slot in the error buffer. It is parallelized by samples and input channels.

clMDF_inner_product_reduce This is the general-purpose inner product kernel, although often combined with other operations. It is used when a combination is not practical or possible, and is employed as a template for the combined operations and multiplies the real and imaginary parts of the interleaved array composed of complex numbers. All the products are then added together into a single scalar value, this operation is done in local memory using a common reduction pattern from the NVIDIA and ATI SDKs (basically a binary tree pattern). The kernel is parallelized on samples and can also run in several work-groups if batch processing is needed. The results are the added together on the host. usually a very small number of values, so the overhead of the memory read operation will dominate.

clMDF_adjust_prop_phase1_reduce Performs one of the calculations when resolving the new adaption rate of the filter. Calculates the square value of an array of elements and adds them all together into a single value. Again, the reduction pattern occurs in local memory. This kernel parallelized on samples, input channels, output channels and echo tail frames. The input to this kernel is the output of `clMDF_weighted_spectral_mul_conj` from the previous frame.

[1]To clarify, when it says that a kernel function is parallelized by samples, this means that each thread/work-item is given a single sample each to process, from a frame or window (2 frames).

clMDF_weighted_spectral_mul_conj “Computes weighted cross-power spectrum of a half-complex (packed) vector with conjugate“. At its core, this function performs an operation on the echo tail in the frequency domain and the error transformed into the frequency domain. There are two complex numbers: $X_i = a + bi$ (from the echo tail and $E_i = c + di$ (from the error) and the function calculates $S = (a \times c + b \times d) + (-b \times c + a \times d)i$. The function is only run if the input signal is not saturated (consists of extremal values), at which point the whole filter is uneffective. It is parallelized on samples, input channels, output channels and echo tail frames.

clMDF_spectral_mul_accum This is special case kernel that calculates cross-power spectras for a batch of frames to be added together. In addition to setting three unrelated buffers to zero to avoid overhead. The spectral mul. accum. operation is essentially a multiplication and addition of complex numbers on signals transformed to the frequency domain. For two complex numbers $X_i = a + bi$ and $Y_i = c + di$, the function calculates $S_i = (a \times c - b \times d) + (b \times c + a \times d)i$ and the results are added together for all frames in the echo tail into a single frame. The main calculations are parallelized on samples and input channels, but a loop through echo tail frames is required because they are accumulated in global memory. Clearing the three buffers is only parallelized for samples (since they are the size of a single frame).

clMDF_combined_response_diff_inner_prod Is a combined kernel consisting of two inner products (see `clMDF_inner_product_reduce`) and a calculation of a difference in response. The response difference is stored in the buffer for the error in the time domain and is calculated as the difference between the input and the results of `clMDF_spectral_mul_accum_long` between the echo tail and the weights, and is transformed into the time domain. All operations are parallelized on samples and the response difference must be executed in the same work-group as one of the inner products because of a data dependency.

clMDF_update_foreground_filter This kernel applies a smooth transition to the error in the time domain when updating the foreground filter so not to introduce blocking artifacts by multiplying with a pre-calculated smoothing function based on cosine. It is parallelized on samples and input channels. The update itself is performed by an earlier call to `clMDF_copy_float_array`.

clMDF_update_background_filter The corresponding function for the background filter consists of a a number of copy operations between temporary buffers with time domain data and a subtraction from the input. It is parallelized on samples and input channels.

clMDF_error_signal_combined This is a combined kernel of three inner products in addition to calculating the error signal, and it also checks for saturation in the microphone signal. Since the calculation of the error signal has a high degree of data dependency, it is performed on CPU, with only some initial preparation of the data being performed on the device to minimize memory transfer. All the operations are parallelized on samples and input channels.

clMDF_power_spectrum_inner_product This kernel combines an inner product with two computations of the power spectrum for the echo and filter response. If we have a complex number $X_i = a + bi$ from a signal transformed into the frequency domain, the power spectrum is calculated as $(P_i = a^2 + b^2)$. These results are then accumulated across input channels into a single frame. The function is parallelized on samples only, for the power spectrum each work item will then loop through the input channels to avoid race conditions with the additions.

clMDF_set_array_to_float_value A simple utility function that sets the specified number of of values (with an optional offset) in a float array to a specified value. It is only used when the filter detects a large problem and haves to reset itself, which does not happen regularly. It was a useful tool for development, but was eventually combined with other operations in most cases to avoid additional overhead for small data sizes. The function is parallelized on array elements.

clMDF_smooth_far_end_compute_filtered_spectra This kernel combines three tasks: Smoothing the far-end energy estimate over time, computing the filtered spectra and some cross-correlations. Smoothing the far-end energy estimate involves multiplication with constants and the addition of the power spectrum of the echo tail. Computing the filtered spectra and the two cross-correlation is done in the same operation as there are data-dependencies between the two. The cross-correlation must be reduced down to single values and this is done using shared memory in a similar manner as with the inner products. All the operations are parallelized by samples and they are only run on a single frame.

clMDF_learning_rate_calc This kernel includes some calculations done when computing the new learning rate for the filter. The operations that depend on device buffers are done on the device itself to avoid overhead. The result depends on whether the filter has been adapted or not. The function is parallelized on samples, and is only run for a single frame.

clMDF_adapted_copy A simple kernel that stores the difference between the raw input and output in a buffer for use in the next iteration, but only if the filter has adapted. It is separated into its own kernel because it depends on the results from on the host, since some control logic is not transferred to the device. It is parallelized by samples.

4.6.2 FFT Kernel Functions

clFFT_inner_prod_combined Combines an inner product calculation with a forward FFT transform. See more detailed description in the descriptions of `clMDF_inner_product_reduce` and `clFFT_custom`. This is a special case combined kernel that calculates the inner product of the main frequency domain buffer with itself after performing the transformation from the main time domain buffer. This FFT and inner product is only run in a batch size of 1, since it is just run once for each filtered frame.

clFFT_custom This is the general purpose FFT/IFFT kernel based on the code output by the Apple OpenCL FFT library. It combines the computational kernels themselves with preparation and finalization steps (see Section 4.4) to convert into, or out of, the Speex/FFTW data format. A separate set of temporary input/output buffers are maintained while the application is run using temporary storage for the FFT. This also allows some flexibility in that it is possible to set an offset into the data to be transformed. In our case study, we used hard-coded FFT kernels for the sizes of 256 and 1024 elements (allows for frame sizes of 128 and 512) and these use 64 and 128 work items per work-group respectively. Some local memory is used for each work-group, as well as registers to store temporary results. At these small sizes (below 2048), the FFT itself can be computed without using any global memory for temporary storage using the latest GPU hardware, by doing the whole transform using a single work-group. The transformation can also be done in batches where data is stored in consecutive blocks that are the same size as the FFT. This will greatly benefit performance over running a single FFT several times as it better the utilizes massively parallel hardware by adding more work-groups, for instance, where data from several input channels can be processed in a single batch.

clFFT_update_weights This function combines and parallelizes the process of updating weights for the complete echo tail, in contrast to a single frame with AUMDF (see Section 4.9.1). As the FFTs are run in a single work-group for each frame they handle, each isolated frame is handled by a work-group. First, an inverse FFT is performed to transform the weights into the time domain, next the first half of the weights are then set to 0 and finally the weights are transformed back into the frequency domain. The FFT core functionality is the same as with the `clFFT_custom` kernel repeated twice, only with some additional offsets for batch processing. The kernel is parllelized by samples (thread to sample ratio varies by frame/FFT size), echo tail frames, output channels and input channels. This makes the kernel a highly parallel, and potentially the function with the most scope for speedup compared with a single threaded CPU implementation. Because of problems experienced when running this kernel on ATI GPU hardware (see Section 5.1 for hardware specifications), this kernel had to be split in two due to a lack of registers/memory available for each thread. The IFFT is then performed in the first kernel, along with setting of values, and the FFT is then performed in a second kernel. This had a negative impact on performance, but the original unified kernel is easily enabled on platforms with sufficient hardware support to run it.

clFFT_convert_error_to_frequency_domain This kernel combines converting both the error and the filter response to the frequency domain, as well as setting the first half of the values in the filter response to zero. This is done by using the same FFT core as in `clFFT_custom`, but letting some of the work-groups handle each transform. In the majority of cases, adding additional work-groups resulted in little impact on performance for GPUs since several computational units were typically idling. The kernel is parallelized on samples in the same manner as with the other FFT kernels, but is also parallelized on input channels, running them in batch.

4.7 Executing on Different Platforms

As briefly mentioned earlier, the platforms we focused on for OpenCL were the combined CPU and GPU implementation from AMD (ATI Stream) and the GPU implementation from NVIDIA. In principle, the same kernel functions should be able to execute on all the platforms with correct results. However, as the devices can still impose limitations on work-group sizes, memory size and so on, the CPU implementation is least restricted in such regard, but at the same time it has the least scope for parallelism. Large parts of the program need to be serialized for execution, even though it is run on multiple CPU cores, because it has a lot less inherent parallelism than GPUs, even with the availability of vector instructions. This can both expose errors that occur when the code is executed in serial instead of parallel, but also hide errors that only occur in parallel. Testing the code frequently on both CPU and GPU was important.

What was found to be the largest problem between the implementations, were error reporting and the recovery from errors. The kernel functions were developed mainly on NVIDIA GPUs, new faults in the code was found when it was moved to the ATI platform. Some of these were obvious errors such as array indices out of bounds, which did not show up as fatal faults on the NVIDIA implementation. This indicate a less rigid memory protection on the NVIDIA implementation on the particular platform used. Errors related to barriers and synchronization were also found, and in some instances could completely lock down a host system with ATI GPU for branches that included barriers, which can lead to complete deadlocks of the GPU, including screen output. In some cases additional synchronization was also required to get the correct results. Ultimately, most of these flaws were rooted in human error, but additional work was nonetheless created in order to make it work on all platforms. Support for error reporting was also fairly poor in some cases, this can partly be attributed to the fact that OpenCL is a recent technology and most of the developer kits available were the first or second generation of production release, and not a deficiency of the standard.

Initially, when we started the case study, the ATI Stream SDK was only out in version 2.0, which supported OpenCL 1.0 but lacked a number of features. Fortunately, a new version (2.1) was released before benchmarking was started. The new release includes many new features to make the platform relatively comparable with the NVIDIA CUDA SDK. These included byte addressable memory (to be able to use datatypes smaller than 4 bytes without bit-shifting), support for 2D and 3D textures, double precision support, support for AMD-specific vector operations and OpenCL interoperability. Recently vendors has also started cooperating on a common standard, installable client driver (ICD[2]), for switching between OpenCL implementations on a single system. For instance, this allows the AMD CPU implementation to be installed side-by-side with the NVIDIA GPU implementation. However having GPUs from different vendors installed on the same system can still be problematic depending on the operating system, additionally the display drivers are often not made to co-exist with those of other vendors.

4.8 Preliminary Performance Findings and Tuning

In this section, some preliminary performance findings will be discussed, that were used to optimize the code during development. Detailed benchmarking and discussion can be found in Chapter 5, while optimization of the FFT has already been discussed in Section 4.4. Most of the optimizations have already been mentioned. Especially when using small echo tails, the overhead incurred simply by executing on a GPU was significant. Initially most loops and sub-routines had their own kernels, but quickly this strategy incurred too much overhead for repeated kernel calls. So, in addition to basic optimizations of memory access patterns for the GPU, the most important optimization was to subsume kernel calls into larger kernels and to avoid calling kernels inside loops. As not all kernels were run with the same size ND range/grid, certain work-groups (blocks) tended to branch off from execution

[2]OpenCL extension registry entry for cl_khr_icd (version 7, dated March 2, 2010): `http://www.khronos.org/registry/cl/extensions/khr/cl_khr_icd.txt`

and ending earlier than others. While not optimal for very large data sets, it can save significant time if overhead is the dominant consumer of time. In some kernels, it was also possible to simply have parallel work-groups with their own task (of approximately the same length) which resulted in effective task parallelization of the tasks through the masking of execution time.

Some kernels were also found to be very poor candidates for parallelization, but these were often operating on a fairly small amount of data, which gave an acceptable running time because of the overhead that is always incurred on kernel launches. These were mostly loops with data dependencies, for instance when the array value in the current iteration was dependent upon the previous iteration. One example from the original Speex echo cancellation source can be found in Listing 4.1 (fixed point code in this code snippet is removed, and some types are resolved to their primitive types for clarity). Writing values to the *out* array itself can be parallelized, but $out[i]$ is dependent upon *vout*, which is dependent upon $mem[0]$, which is dependent on $mem[0]$ from previous iteration. In this particular instance, the function is always executed on the CPU since it is the very first function in the filter, which means the calculation can be done on CPU before the data is copied to GPU memory. Similar cases exist, where doing so would incur a lot of extra overhead, so it must still be performed inefficiently on the GPU since all the data is located there anyway.

```c
static inline void filter_dc_notch16(const int *in, float radius,
    float *out, int len, float *mem, int stride)
{
  int i;
  float den2;
  den2 = radius*radius + .7*(1-radius)*(1-radius);
  for (i=0;i<len;i++)
  {
    float vin = in[i*stride];
    float vout = mem[0] + vin;
    mem[0] = mem[1] + 2*(-vin + radius*vout);
    mem[1] = vin - (den2*vout);
    out[i] = radius * vout;
  }
}
```

Listing 4.1: Notch filter function from the echo cancellation filter in Speex 1.2 RC1. This loop is poorly parallelizable on GPU because of depedencies.

4.9 Increasing the Computational Load

Since overhead is such a large consideration with GPU implementations, some options were evaluated for increasing computation load and thereby decreasing the effect of implementation overhead.

4.9.1 Using MDF Instead of AUMDF

As described in Section 3.6.2, the AUMDF[24] algorithm is used in Speex originally, only updates a single weight in the echo tail for each input frame. This updating operation includes some copy and subtraction operations, but more importantly, both an inverse and forward FFT. On a CPU, the optimization of only performing this for a single weight saves significant iterative operations for a small decrease in accuracy of filter operation, but on the GPU these operations can be easily parallelized for the complete tail, so they do not incur as large a performance penalty. The filter was made to easily operate as a traditional MDF filter, and this was done for both CPU and GPU versions.

4.9.2 Longer Echo Tails

As described in Section 3.6.1, the echo cancelling filter stores an echo tail of earlier frames for comparison against the new frame in order to detect echoes. The length of this tail determines the maximum length of the echo that is possible to cancel. A long echo tail makes it possible to cancel echo in larger rooms with longer reverberation times, a long distance between microphones and speakers and compensate for delays in the audio and computer equipment. However, a long echo tail where it is not needed, does have an impact of the adaption rate to new conditions.

The echo tail length can have a significant impact on performance where the filter needs to loop through the whole tail. This happens when new data is added to the echo tail, computing the filter itself, adjusting adaption rate, computing the weigth gradient, computing response difference and when copying between background and foreground filters. All these operations take longer time as the length of the echo tail increases. In addition, updating the weights in the echo tail themselves take longer when using a plain MDF algorithm instead of AUMDF.

The default echo tail length in the Speex echo canceller test program is 8 frames of 128 samples, or 1024 samples. At an 8000hz sampling rate, this corresponds to 128ms. We experimented with setting this tail with up to 256 frames (approx. 4 secs.) to increase the computation time. Another case for testing longer echo tails is that when using higher sampling rates, longer tails, in terms of number of samples, need to be used to compensate for each sample representing a smaller time period. Speex has an ultra-wideband codec with a default 32kHz sampling rate, which would require four times longer echo tails to cover the same time period (be usable in the same room). Higher sample rates are often used for video conferencing and higher end phone conferences. The Mumble VoIP application uses an echo tail of 4800 samples (10 frames) with a sample rate of 48kHz by default.

4.9.3 Longer Input Frames

The frame size determines how many samples from the input data are loaded for each run of the filter. This is closely tied to the length of the FFT used. For file encoding on PCs, longer frames (within reasonable limits) lead to shorter processing time for the whole file. This could potentially increase performance on GPU, since smaller 1D FFTs can not fully utilize the resources on modern GPUs. There are three main problems with increasing this size:

- Using a longer FFT is plausible on desktop CPUs and GPUs, but might not scale well on embedded hardware. An embedded GPU typically has fewer computational cores than desktop GPUs. Embedded platforms might also have fixed size specialized hardware for doing FFTs, or a CPU that can only calculate a limited size FFT in reasonable time in software.
- The most common use case for the Speex codec is for VoIP, operating over a slow network such as limited Internet connections. In such conditions, smaller frame sizes are desired to reduce the effect of network latency and to increase fault-tolerance if frames are lost.
- Having longer frames lead to a more coarse-grained echo filter that adapts slower, since frames are fewer and farther between.

The test program uses a frame size of 128 samples by default, which leads to a FFT size of 256 samples (two frames are always combined). Because of the design of the FFT implementation used, power of 2 (divided by two) frame sizes are preferred. The default frame size for the Mumble VoIP application is 480 samples in combination with a 48kHz sampling rate.

A simple benchmark was conducted with different frame sizes (above 64 samples) using the reference implementation from Speex, the results can be found in Figure 4.3. The test was conducted on an Intel quad-core workstation (see Section 5.1 for detailed specifications of Machine A). AUMDF was enabled to make the effects of the echo tail length as small as possible. An initial tail length of 20 480 samples was used, and rounded up when the tail was not a multiple of the frame size. Only single input and output channels were used, and the measurements are the average for 1024 frames.

As is evident from the graph, the default frame size of 128 samples is not optimal for this CPU, but few performance gains are found for frame sizes above 768. GPUs may scale up to even larger frame sizes before the performance per sample stagnates. Another consideration must also be taken, if the filter is to be used in a real-time system with large frame sizes. The frame size widely used on Ethernet networks in general, is 1518 bytes[34]. Larger frames (9 kB) are supported[35], but are not seeing widespread use other than for very high-bandwidth local networks

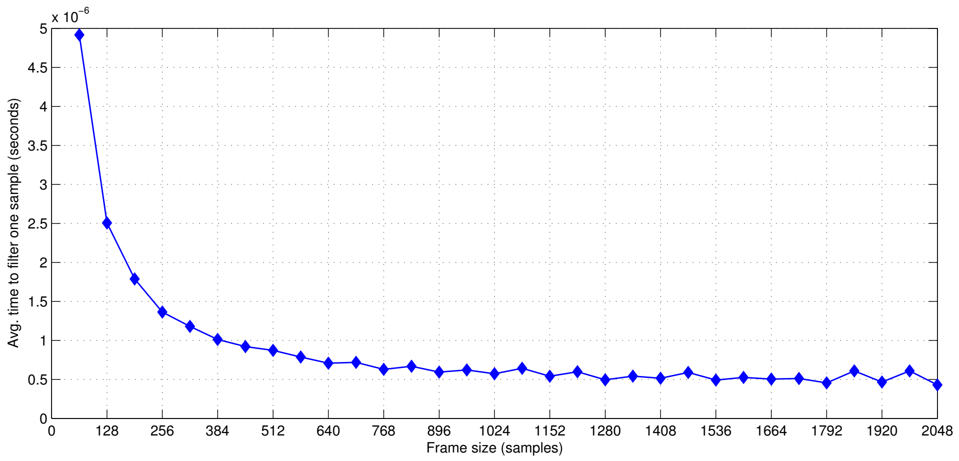


Figure 4.3: Scalability of MDF filter with increasing frame size.

(gigabit ethernet and above) where their use can increase the maximum throughput. Due to the fact that a speech codec is not intended to be a high bandwidth application, and continuity problems that can occur when using such large frames for audio transfer, we choose not to focus on frame sizes larger than the 1518 byte standard.

Each frame in Speex consists of a group of samples stored as the short integer datatype in C. This means that each sample is stored using 16 bits or 2 bytes. The largest multiple of 64 that could ideally fit into a 1518 byte packet is 704 samples. At this point, the CPU performance has plateaued. To keep the implementation simple, the frame size of 512 samples was further experimented with on GPU in addition the default 128, since it leads to an FFT size of 1024.

4.9.4 Multiple Input/Output Channels

Professional deployments of voice communication systems often include several microphones to better remove negative artifacts in the sound from the room, and to cover all the participants. Even some newer mobile devices are constructed with multiple microphones to better reduce the effect of noise. Several speakers are also common, first and foremost for stereo sound, but one can also see surround sound becoming commonplace in the future. For the input data, the number of microphones multiplies the amount of data (size of the buffer). For output data, the number of speakers multiplies the amount of data. Depending on the mix of functions used, this can also multiply the amount work that needs to be done.

To investigate the extent to which the number of channels impact performance, a simple benchmark was conducted. The execution time for one frame to be processed by the echo filter (average over 1024 frames) was timed. The default frame size (128 samples) and tail length (1024 samples) were used, and AUMDF was disabled. The same PC as in 4.9.3 was used.

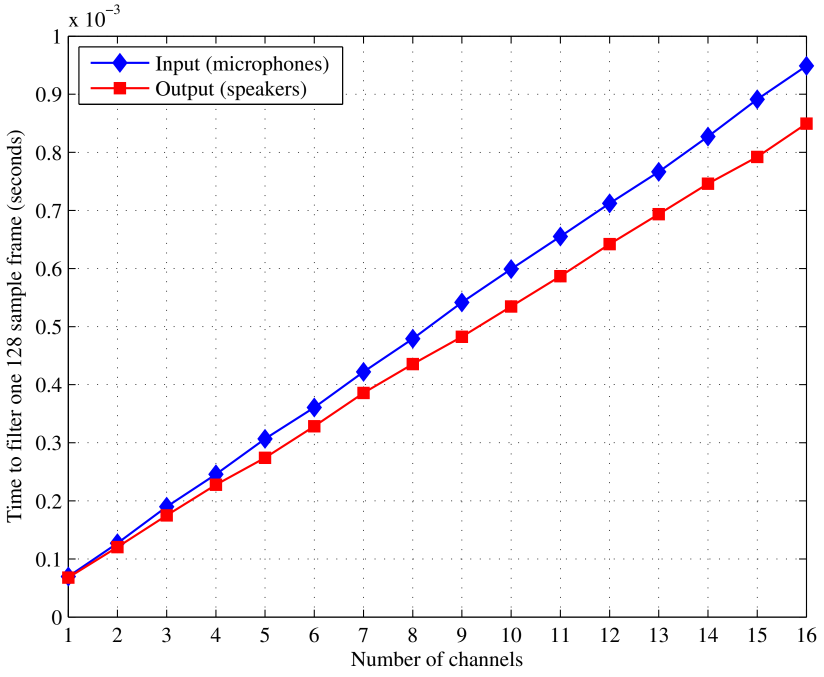


Figure 4.4: Scalability of MDF filter with increasing input *or* output channels. Shows the effect an increasing either the number of input or output channels.

First, the number of input and output channels were increased separately, as seen in Figure 4.4. This resulted in fairly linear scaling properties both for the number of input and output channels, but input channels always have the highest impact. In Figure 4.5, both the number of input and output channels were set to the same value, and this shows an exponentially shaped growth in computation time.

At first, the echo cancelling was implemented with only mono (single-channel) capabilities on GPU for simplicity, but to increase the computational load support for multiple input and output was implemented.

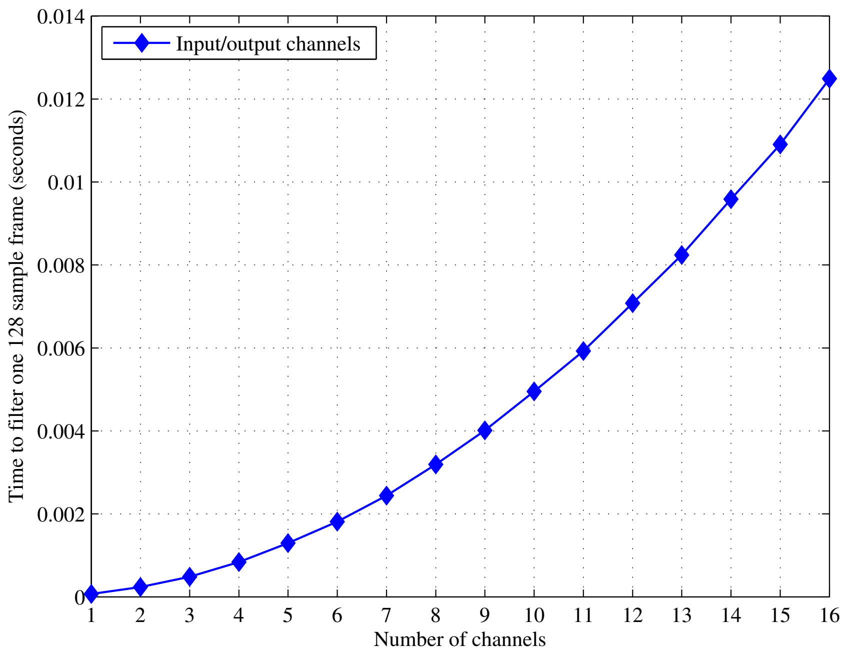


Figure 4.5: Scalability of MDF filter with increasing input *and* output channels. The numer of input and output channels are set to the same value.

4.9.5 Processing Several Filesections in Parallel

In the application used for testing the case study, a file with raw sample data is taken as input frame by frame, and processed into raw sample data as output into another file. In theory, it would be possible to look a different sections of the file in parallel to better utilize the GPU. This would mean that the filter would need to re-adapt for each starting point, or some presumptions would need to be made after the first section(s). The total time for processing the file would be much lower, but some loss in audio quality would result.

In this thesis, this possiblity is procluded because it should be possible to use the filter in a real-time system, which is the most common use for Speex. The file encoding is only used to simulate an incoming stream of data from a network, and not to encode files as such. If a fast file encoder was desired, this could be an option to examine.

4.10 Implementing Co-processing

The reason for investigating the use of OpenCL in this case study, except for portability between GPUs, was that it facilitates co-processing in a much more seamless manner than earlier GPGPU frameworks. A simple co-processing scheme was implemented, but other possibilities are left as further work.

4.10.1 OpenCL on CPU

Earlier GPGPU languages and platforms have often focused purely on execution on the GPU, or in best case had a simplified emulator layer that could execute the GPU code on a CPU. With OpenCL, CPU implementations have been created to fully utilize modern multi-core processors and use vector instructions (such as SSE) where appropriate. The CPU can be used as another computing device by the OpenCL implementation, often with relaxed limitations (with regards to maximum number of threads and local memory) compared to GPUs. However, because of fewer hardware restrictions in their architecture they also tend to be more useful as a debugging tool. AMD released such an implementation as part of their Stream SDK. NVIDIA has no CPU implementation yet, but the two implementations can be used side-by-side. Writing programs for use in stream processing, such as on GPUs, is rather different than the traditional CPU implementations, so having the possibility of running them on a CPU can therefore be of major benefit when debugging. The OpenCL kernel programs are inherently multi-threaded, so they have the potential to be significantly faster than a single-threaded CPU implementation for certain applications.

For co-processing, this sort of portability is ideal, as very little specific code has to be written for the CPU version after the GPU version is created, and computing can be interleaved between devices on a kernel-function basis. But as we find in Chapter 5, in our case the performance of the OpenCL version on CPU is significantly worse than the existing library. CPU-usage when running the GPU-implementation is also currently fairly high. These performance problems hampered further work on the subject in the case study. A simple load-balancer that can be used for co-processing was implemented and is described in the next section, but it does in most cases not gain much available CPU-time in practice.

4.10.2 A Simple Load-balancer

As mentioned in Section 2.3, dynamic load balancing between CPUs and GPUs during run-time is not a well-explored subject. One of the reasons for this, is that the tools for doing so have until recently been lacking. With OpenCL, it is possible to actually execute the program using both types of processors for the first time. However, the toolchains for utilizing all the available computing power in a system has not to matured enough for it to be easily implemented.

One of the most significant problems with using the GPU to offload common tasks from different applications in the system, is the lack of support for running multiple applications (besides graphical rendering) using the same GPU at the same time. Traditionally, the GPU has been completely occupied by a game or other real time visualization for 3D rendering, and perhaps a few general computations running shader programs or using GPGPU programs. These applications typically are modal, in that they occupy all the users attention at a point in time. But this is not a common use case for sound processing and many other general computational tasks.

At the time of writing, it was possible for several processes to send work to the GPU, but there is no real mechanism for these to effectively co-exist with regards to the performance each would require. If there are other applications using the GPU for GPGPU tasks or perhaps rendering, there is no way to asess how much capacity (both processing power and memory) is available on the GPU at a point in time, as is possible with CPUs in most modern systems. The ability to run several kernel functions at once was made possible by NVIDIA with the release of their Fermi architecture[9] (also see Section 4.10.3). This should be the first step towards more advanced multi-tasking capabilities on GPUs, but this technology was not available when we started developing our implementation, and there was still no practical way to determine overall load levels on a GPU.

A true load-balancing system would consider both the total load level on the CPU and GPU to be able to realistically select the appropriate device to use for execution. Seeing as it is only feasible to have control over the load level on the CPU, a simple implementation was created within this limitation. In some applications, one can assume that the GPU is available all time, but that would not be the case in future systems where more and more applications offload calculations to the GPU, or for instance with graphics intensive 3D games.

The load balancing system implemented is outlined in Figure 4.6. In our implementation, it switches between the reference implementation from Speex on CPU, and the OpenCL version on GPU. This was done because of the limited performance of the OpenCL implementation on CPU (see benchmarks in Section 5.5), but could just as well have been used between the CPU and GPU OpenCL versions. In such case, it could benefit greatly from using mapped device memory between the host and device when executing on CPU.

The natural granularity at which to perform load-balancing in the test application using the echo cancellation filter was the processing of a single frame, which is the unit of the main loop. The loop was outfitted with some extra logic that checks the system CPU load every second and evaluates whether the load is high enough (a threshold would be determined by the requirements of the system/application) to switch to GPU execution. If the CPU implementation manages to utilize all the CPU cores (the current reference does not fully occupy a multi-core CPU), another method of determining whether to switch to GPU execution might be needed, for instance the number of other applications currently running or an explicit interface that can be called by other CPU-demanding applications. This might also be influenced by power-demands to execute in the least power consuming fashion as possible depending on the performance demand to the application, especially suited to embedded systems. When the value falls under a certain threshold, CPU-execution could be resumed. A small function was made to read the system CPU

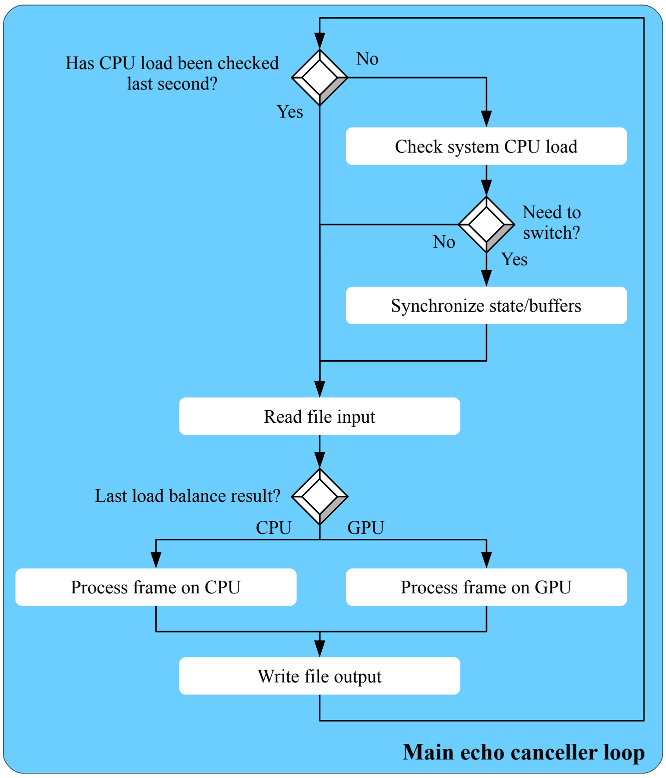


Figure 4.6: Simple CPU/GPU load balancer program flow for echo cancellation, as implemented.

load from the operating system[3]. This was done on a GNU/Linux system and the functionality is operating system specific, so a new version would have to be created for use on other operating systems (should be easily adaptable on UNIX-

[3]CPU load function based on the following script (retrieved 10.July 2010): `http://colby.id.au/node/39`

based OSes). It opens the system file `/proc/stat`, which contains several statistics about resource utilization. The lines of concern here are the lines beginning with "cpu". On a multi-core system there can be several lines, starting with "cpu0", "cpu1", "cpu2" and so on. But the line of most interest is the one only starts with "cpu", which is a summary of all the following. It is the only line of interest, unless a more detailed analysis based on individual cores is required. Following "cpu", are integers that represent the number of milliseconds that the system has been in (since boot)[4]: User mode, user mode with low priority (nice), system mode, I/O wait, IRQ (hardirq) and softirq (lower priority than hardirq). By summing together these numbers to get total utilization and by comparing to the previous value obtained, a single CPU load percentage can be derived. There is no point in querying this file several times a second, so it is only polled by the main loop when it has been at least a second since the last check.

When it has been decided to switch device, synchronization of memory needs to take place in order to be able to continue where the other device left off. This is a time-consuming operation, so there should be a obvious threshold to switching between devices, requiring a high/low CPU load to be observed over time, and switching at all during execution could be problematic in a real-time application. Some slowness is already built into the system when polling the `/proc/stat` only once a second, but this should be used with an additional heuristic to improve this in a production environment. It must be noted that the synchronization time is largely a hardware/GPGPU platform implementation limitation (see latency measurements in Section 5.2) and will improve over time.

4.10.3 Task Parallelism

A natural extension to running sound processing on GPU, would be massively parallel sound processing that can better utilize the GPU. Since the amounts of data processed in this case study are not enough to saturate all the cores in the GPU all the time, much of its capacity is left unused. A major use case for such functionality, would be in conjunction with the Asterisk PBX software (which can already use the Speex codec). Asterisk can be used as the hub of all telephony services in an organization, everything from regular calls and conference calls, to automated voice services and voicemail. If it processes a number of calls from "dumb" handsets, for instance simpler mobile phones with limited processing power, it could run the filtering for all the calls at once.

Another (while not so common) use case would be for encoding several audio files in parallel. Two main approaches could be used to achieve this with OpenCL.

[4]Source for information about /proc/stats is the RHEL 4 reference guide (retrieved 10.July 2010): `http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/en-US/Reference_Guide/s2-proc-stat.html`

1. Rewriting all the kernel functions to accept several audio streams as input/output. This would mean that all the streams would have to be synchronized when being run through the echo filter, as well as large modifications to the existing code base. It could also be problematic for real-time systems such as those used for VoIP, since all the audio streams would have to be synchronized, and switching the number of simultaneous streams while the program is running would be a challenge.

2. Run the audio streams independent of each other, in their own thread/process on the CPU. This would require a solution for "multi-tasking" on the GPU, which is not supported, or is still in its infancy at the time of writing. Although each kernel function is instantiated as many threads, running several kernels at once has not traditionally been possible. As already mentioned, NVIDIA is bringing concurrent kernel executions in their Fermi architecture devices [9] (see Figure 4.7). It requires new types of barriers that do not affect all threads on the GPU, and so have explicit control of their execution. OpenCL organizes all kernel calls in a queue, and it would be possible for the implementation to execute several kernels in parallel between barriers, but no such behavior is supported in current implementations. The main method for task-parallelism using OpenCL is to use multiple command queues, but the hardware and software support for this is limited at the time of writing. Compute kernels can then be queued on command queues completely independent of each other, which can reside on different threads on the host. This approach to task parallelism is not a part of our implementation, but can be added at a later point in time on platforms with proper hardware support without major restructuring.

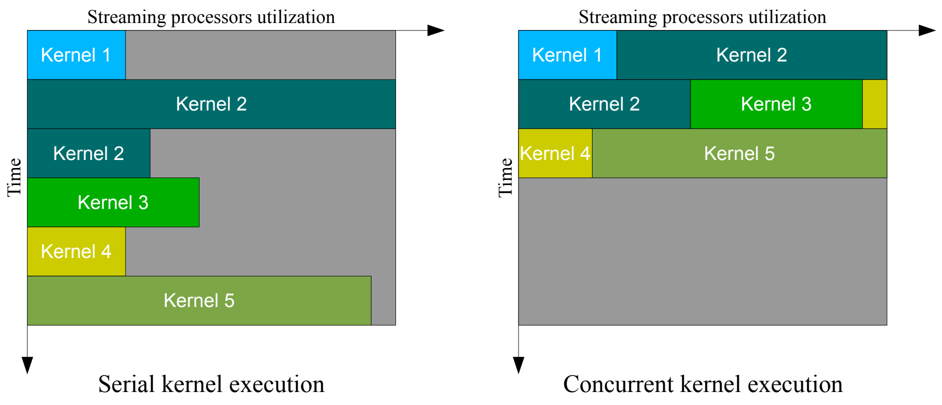


Figure 4.7: Concurrent vs. serial kernel execution. Designed after illustration in [9].

As the OpenCL implementations were not mature enough to implement the second approach as we started our implementation, and the first alternative requires a large amount of additional work, task parallelism is considered future work.

Chapter 5

Benchmarking and Results

In this chapter, the performance of the resulting application from the case study is investigated, along with some related metrics. Section 5.1 describes the hardware and software of the systems used for benchmarking in later sections. Section 5.2 looks at the latency of certain operations on several GPGPU implementations and its impact on the performance of the case study. Section 5.3 contains some performance testing of the FFT used, compared against a well-established GPU FFT library (CUFFT). Section 5.4 describes the testing input used and contains some notes on accuracy.

Benchmarking of the case study itself starts in 5.5 with testing the impact of echo tail length on performance, and continues in Section 5.6 where scalability with multiple channels is tested. Section 5.7 contains analysis of which kernel functions dominate running times. Section 5.8 looks at the CPU load and considers how this affects load balancing the application. Lastly, Section 5.9 summarizes the findings and discusses the performance issues faced.

5.1 System Specifications

Two systems were used for benchmarking in order to guage the cross-platform performance and portability of OpenCL, since the ability to execute on GPUs from different vendors and on multi-core CPUs is its big advantage over earlier frameworks. The two machines were each configured with GPUs from the two major vendors at the time of writing. If not specified specifically, machine A was used for all CPU tests involving the reference implementation from Speex, while machine B was used for all OpenCL CPU testing. FFTW 3.2.1 was used as the

Hardware	
CPU	Intel Core 2 Quad Q9550
CPU clockspeed	2.83 GHz
Memory size	4 GB
Graphics card #1	NVIDIA Geforce 280 GTX
Graphics card #1 memory	1GB RAM
Graphics card #2	NVIDIA Tesla C2050
Graphics card #2 memory	3GB RAM
Software	
OS	Ubuntu 9.10
Kernel version	2.6.31-14
Kernel CPU architecture	x86_64
NVIDIA graphics driver ver.	256.35
NVIDIA CUDA toolkit/SDK ver.	3.1
GCC version	4.4

Table 5.1: Specification for benchmarking machine A.

FFT library when testing the reference implementation, as it was found to generally have better overall performance than "smallft", which is the built-in default FFT used by Speex.

5.1.1 Machine A

This machine had an Intel quad-core CPU and was used for benchmarking the GPU-only OpenCL implementation from NVIDIA. It contained two different NVIDIA GPUs: One of the previous generation with CUDA compute capability 1.3 (GTX280), and one of the current generation with CUDA compute capability 2.0[9] (C2050). Detailed specification can be found in Table 5.1.

5.1.2 Machine B

This machine had an AMD quad-core CPU and was used for benchmarking the OpenCL implementation from AMD/ATI. The hardware is from the same generation as machine A, and has fairly similar end-user performance even though it is based on hardware from different vendors. It contained a single ATI Radeon 5870 GPU, which was the fastest single GPU based graphics card from ATI at the time of writing. Detailed specification can be found in Table 5.2.

Hardware	
CPU	AMD Phenom II X4 965
CPU clockspeed	3.40 GHz
Memory size	4 GB
Graphics card #1	ATI Radeon 5870
Graphics card #1 memory	1GB RAM
Software	
OS	Ubuntu 10.04
Kernel version	2.6.32-22
Kernel CPU architecture	x86_64
ATI Catalyst graphics driver ver.	10.6
ATI Stream SDK ver.	2.1
GCC version	4.4

Table 5.2: Specification for benchmarking machine B.

5.2 Latency of GPGPU Implementations

As mentioned in Section 4.8, latency/overhead when calling OpenCL kernels became an issue while implementing the case study. To quantify this problem, a benchmark was created to test both OpenCL and the until now de-facto GPGPU platform, NVIDIA C for CUDA. The test was run on three different GPUs (two NVIDIA, one ATI), and one CPU, where both CUDA and OpenCL implementations were tested on the same NVIDIA GPUs. The tests were:

1. Launch of a trivial kernel. This is an empty kernel with no arguments, launched in only a single thread.
2. Launch using a more realistic kernel. This was based on an actual kernel function used in the case study, but the body of code removed. It has 14 arguments in the OpenCL version (12 in the CUDA version because of the way shared memory is allocated). These were six pointers to floating point buffers, one pointer to a buffer of short integers, one floating point number, four unsigned integers and two pointers to a local/shared memory buffer. The kernel was launched in 1024 threads.
3. Memory write of a single floating point value to a buffer in global memory.
4. Memory write of 1024 floating point values to a buffer in global memory.
5. Memory write of 8096 floating point values to a buffer in global memory.
6. Memory read of a single floating point value from a buffer in global memory.
7. Memory read of 1024 floating point values from a buffer in global memory.

8. Memory read of 8096 floating point values from a buffer in global memory.

The three different sizes of memory operations were included to show that the smaller operations are dominated by overhead and to get an idea of at what size the bandwidth has any effect. All tests were run for 10 000 iterations with a `cudaThreadSynchronize()` or `clFinish()` synchronization call at the end of each iteration. The average latency was calculated at the end of the program run, and an average over three programs runs was used. The tests were run on the machines with the respective GPUs, the CPU test was run on machine B (the AMD-based CPU). The results are listed in Table 5.3 and shown in Figure 5.1.

	C2050 (CUDA)	GTX280 (CUDA)	C2050 (OpenCL)	GTX280 (OpenCL)	ATI 5870 (OpenCL)	CPU (OpenCL)
Simple kernel	$7\mu s$	$10\mu s$	$16\mu s$	$65\mu s$	$31\mu s$	$17\mu s$
Realistic kernel	$19\mu s$	$23\mu s$	$24\mu s$	$75\mu s$	$127\mu s$	$63\mu s$
Mem write 4B	$5\mu s$	$8\mu s$	$38\mu s$	$44\mu s$	$298\mu s$	$15\mu s$
Mem write 4KB	$6\mu s$	$10\mu s$	$38\mu s$	$51\mu s$	$287\mu s$	$18\mu s$
Mem write 32KB	$19\mu s$	$27\mu s$	$72\mu s$	$71\mu s$	$310\mu s$	$21\mu s$
Mem read 4B	$7\mu s$	$9\mu s$	$43\mu s$	$46\mu s$	$53\mu s$	$19\mu s$
Mem read 4KB	$8\mu s$	$11\mu s$	$39\mu s$	$55\mu s$	$73\mu s$	$19\mu s$
Mem read 32KB	$23\mu s$	$25\mu s$	$75\mu s$	$70\mu s$	$208\mu s$	$21\mu s$

Table 5.3: Latency of GPGPU platforms. Averages are rounded to closest microsecond.

The data shows a large difference in latency even on current platforms, and this will certainly impact any application where latency is an issue such as sound processing.

The older more mature C for CUDA language has in general, lower latency on NVIDIA hardware. This could be caused both by a difference in the internal implementation, but also because C for CUDA is created specifically for the CUDA platform with more direct access, where OpenCL is another abstraction layer on top of this platform. Most operations display up to several times longer latency on OpenCL versus CUDA, but kernel launches on the GF100 (Fermi) architecture card show promise for better performance. Even more than on the CUDA platform, there is a marked difference between the GT200 architecture card (GTX280) and the GF100 (C2050) card on kernel launch latency. The GF100 architecture includes the possibility to launch several kernels in parallel (using C for CUDA), and it seems that the changes made to the architecture benefit latency in general.

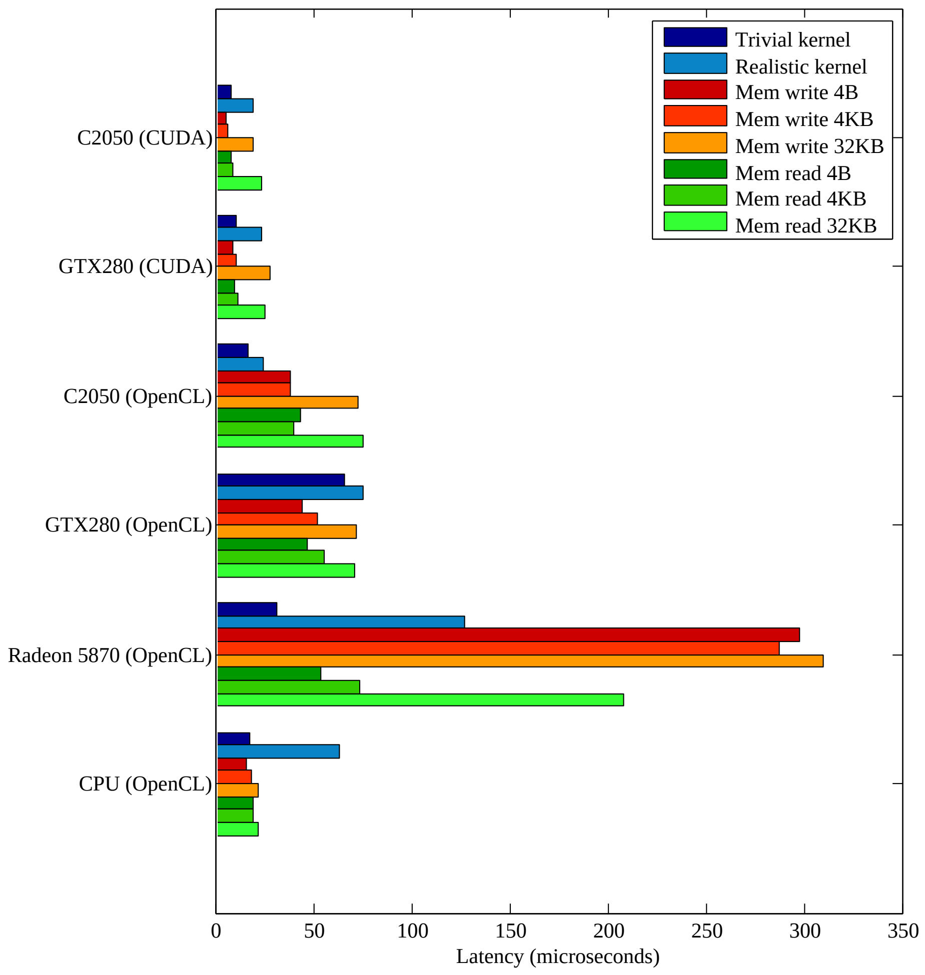


Figure 5.1: Latency of GPGPU platforms. Two workstations was used for the benchmarking (see 5.1).

When looking at the ATI 5870 GPU with OpenCL, we see some interesting numbers. The simple kernel launch is positioned between the GT200 and GF100 cards from NVIDIA with an acceptable latency, but still substantially higher than with C for CUDA on the NVIDIA cards. The realistic kernel is almost twice as slow as the slowest NVIDIA card, which shows that scaling seems to be worse for a large number of threads and kernel arguments. Memory reads on the ATI GPU is in general slower than the NVIDIA cards, with the largest transfer being more than twice as slow. But latency for memory reads seem to be reasonable as shown by the smallest (4B) read, which is only slightly slower. The most pertinent problem here is memory writes that seem to have around 300μs latency, compared to around $40\mu s$ on the NVIDIA cards for OpenCL. The performance might even out on larger sizes, but this can still be a large challenge depending on the application. These results show that a even single small memory write operation to transfer the input data into GPU memory, might take longer than it would take the original CPU implementation in the case study to completely process the frame with computationally light parameters.

The ATI CPU implementation displays much lower latency in general than their GPU implementation. Kernel launches with few arguments are on par with the C2050 using OpenCL, but larger kernel launches with a large number of threads are slower and more similar to the GTX280, this can be possibly attributed to the overhead of starting many work-items/threads on the CPU. Memory operations are faster than all the other cards/implementations, except for the NVIDIA cards using CUDA for certain operations. Even though this makes it fairly reasonable when it comes to latency compared to the GPUs, it is still a substantial amount of overhead for simply executing programs on the CPU. This means that executing every single operation on the CPU using kernel functions is not feasible to gain good performance, and kernels should only be launched for larger operations that can benefit from more threads and vectorization, and where the overhead can be amortized.

As discussed earlier in Section 4.8, latency as measured here, became a large problem in trying to get any performance gains on GPU with the OpenCL implementation. Smaller operations should in general be grouped together into larger kernel functions, but the kernel functions must be split when global synchronization is needed, and developing large monolithic kernels can also become unmanageable. A platform with lower latency would benefit both performance and ease of development significantly. All the implementations used here lack maturity, and there are a number of variables that could impact the results: The version of the respective SDKs used, the display driver version, the operating system etc. But it does show that there is a large variation in latency on GPGPU platforms, and not all are yet mature enough to effectively do real-time sound processing. The benefit obtained from the cross-platform nature of OpenCL does seem to come at the cost in latency, and as these toolkits are proprietary, it is up to the vendors to improve the performance.

5.3 FFT Performance

The FFT is perhaps the single most computationally demanding component of the program. To verify that the performance of the FFT we were using in our implementation on GPUs was reaching an acceptable performance level, we performed a simplistic benchmark against the CUFFT library from NVIDIA. This library is implemented in CUDA as part of the CUDA SDK, so the best available NVIDIA

GPU was used for benchmarking, namely the Tesla C2050 in machine A as listed in Section 5.1. We compared this against the kernel function that was used in the echo canceller implementation from the Apple OpenCL FFT library. Preparation and finalization code was commented out for the benchmark, as no corresponding functionality exists in the CUFFT library kernels. Although higher performing FFT implementations exist for GPUs, CUFFT works well as a standard benchmark with good performance. It is also a production library that is flexible, well tested and with a standardized API based on the popular FFTW library.

The FFT sizes tested were the two used in benchmarking the echo filter, N=256 and N=1024, which corresponds to input frame sizes of 128 and 512 for the filter. It is important to note that the OpenCL FFT kernel used in the N=1024 case was based on a modified series of radices compared the original library version so as to retain compatibility with ATI GPUs, so some minor performance loss may have occurred. The results were obtained by executing the FFT and synchronizing the GPU threads with `cudaThreadSynchronize()` or `clFinish()` after 1024 iterations with varying batch sizes and calculating the average time for each. The maximum batch size in the echo cancellation filter is calculated using the following expression:

$$N_{max} = M \times C \times K \tag{5.1}$$

Here, M is the number of frames in the echo tail (usually 10-20), C is the number of input channels and K is the number of output channels. So the larger batch sizes here are mostly important for long echo tails combined with a large system with many input and output channels. As we can see in Figure 5.2 and Table 5.4,

Batch size	1	32	64	96	128	160	192	224	256
CUFFT N=256	15	15	16	17	17	19	19	21	22
CUFFT N=1024	16	20	25	31	37	44	51	58	62
OpenCL N=256	22	22	23	25	27	29	30	33	35
OpenCL N=1024	25	29	36	41	49	54	59	64	70

Table 5.4: Average execution time of FFT libraries on GPUs (all values are in microseconds per complete batch).

the OpenCL FFTs generally performs slower than the CUFFT counterparts. We also see a constant performance gap both for N=256 and N=1024 as the batch size is increased, but their performance is comparable and scales in a similar manner, which is what we wanted to investigate. As we can see, for N=256, the batch size has little effect on the execution time, which shows that excess processing capacity is still available at this size. But for N=1024 the batch size shows a significant increase in execution time in a linear fashion (implying that batches are very easily parallelizable).

Some of the performance difference between the CUFFT and OpenCL implemen-

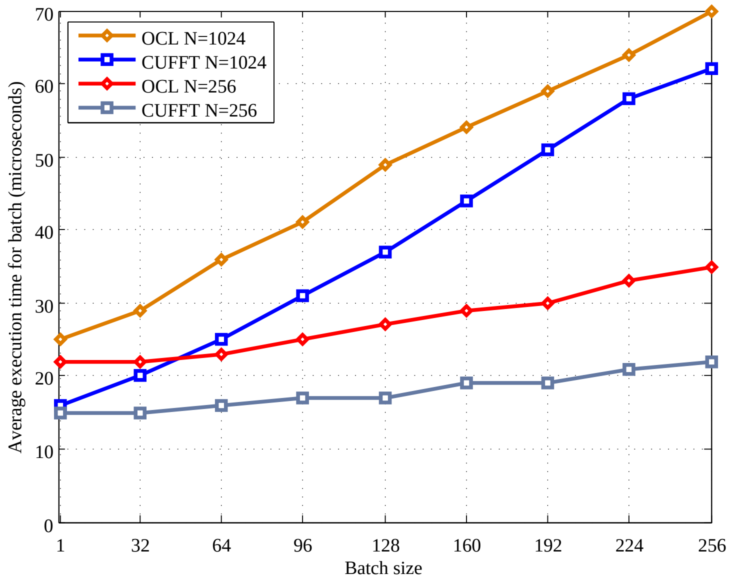


Figure 5.2: Performance of FFT libraries on GPUs.

tations might be due to library optimizations, but might also to some degree be attributed to the difference between the CUDA and OpenCL frameworks, especially on such small data sets. The latency of launching the kernel and the proceeding synchronization adds significant overhead on this time-scale. We can indeed see that the execution times for N=256 using both CUFFT and OpenCL is in line with the latency numbers found for launching an empty kernel in Section 5.2. The fact that the execution time does not increase greatly for this size, means that it is basically bound by latency for realistic batch sizes (it is of course usually combined with other calculations as well).

5.4 Test Data and Accuracy

To test the effectiveness of the filter in a realistic manner, a recording done with real equipment and carried out in a real environment is required. In order to effectively cancel out the echo using the filter, the input data must be recorded with high-quality audio hardware, mainly ck synchronization between input and output. Signals with synthetic echoes (for instance created with audio editor applications), will confuse the adaptive filter and severly reduce performance because it is constructed to mimic a real environment with natural variations. For testing, a signal was recorded with a music signal in the background and a person speaking at the same time. The music clip was then used as the signal sent to the speakers (far-end). The filter will then ideally remove all the music and only leave the persons voice in the output audio. In a real scenario, the music would be exchanged with another person talking.

To test that the filter gave fairly accurate results, a function was created that generates a list of an scalar values that represent how close the OpenCL implementation output is to the reference. The scalars are the output of a utility function (`get_residual_error()`) calculates the root square difference between the array output of the the existing `speex_echo_get_residual()` function in the reference imlementation called with the two echo states (OpenCL and reference). The `speex_echo_get_residual()` function outputs the spectrum for estimated echo of the given echo state.

5.5 Performance Impact of Echo Tail Length

The first benchmark run, was used to test the impact of different echo tail lengths on execution time. The echo tail length impacts several of the more computationally demanding sections of the filter, first among which is the filter update itself. We are testing with AUMDF turned off here (see Section 4.9 for more background), which means that one FFT and a IFFT is added to the batch in the weight update kernel function for each frame the echo tail increases with (for a single channel filter). Longer echo tails should benefit parallel architectures, a length of 10-20 frames is often a realistic maximum length in practice (depending on the combination of sample rate and frame size), but we double that limit to emphasize the scalability. As mentioned earlier, the OpenCL version of the filter was made to support two frame sizes: 128 and 512 samples, both are tested here for all the available OpenCL platforms, as well as the reference implementation on CPU. All results in this section are presented in plots, tables with the results can be found in Appendix A.

5.5.1 Frames of 128 Samples

The results of the benchmark with frames of 128 samples and echo tail lengths between 2 (256 samples) and 40 (5120) can be found in Figure 5.3. It is immediately obvious that the reference implementation has significantly faster run times than the OpenCL filter on any of the available implementations and hardware. In some ways, the GPU results found here reflect the results in Section 5.2, as the overhead seems to completely dominate the execution time. On the shortest tails, the ATI Radeon 5870 GPU is the slowest with execution times varying between 5000-6000 microseconds regardless of tail length. and it does not appear to increase significantly for larger sizes, so it would seem to have no problem with the throughput. It should be noted that we found the running times of the kernel functions of this

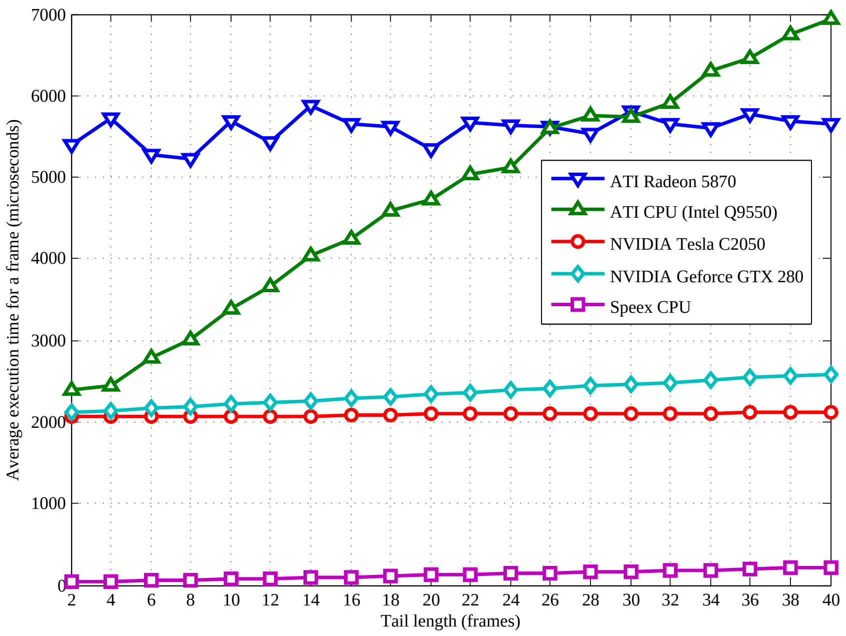


Figure 5.3: Average execution time of a single frame with increasing echo tail lengths (128 sample frames).

implementation to vary a lot compared to the other implementations, with a standard deviation of up to around 2000 microseconds, but this is also a consequence of larger average values.

The CPU-implementation of OpenCL in the ATI SDK starts out with about the same latency as the NVIDIA GPU implementations, but its run times increase linearly at a steep rate as the tail lengths are increased, and uses longer than the ATI Radeon from around tail lengths of 26 frames and larger. This indicates that our OpenCL implementation does not scale well with such few computational cores available and it has not been specifically optimized for this purpose. More information on OpenCL CPU performance hurdles in the current implementation can be found in Section 5.9.

The two NVIDIA GPUs have the shortest execution times of the OpenCL implementations, with the newer Tesla C2050 card showing the best GPU performance. The latency differences between these two cards found in Section 5.2 did not make much differences here, but this may among other things be caused by the mix of kernel calls and memory transfers which combined reduces the difference in practice. As the tail lengths are scaled up, the architectural difference become visible, the Geforce GTX280 experiences a fairly linear increase in performance almost parallel to the reference implementation, but the run times when using the C2050 barely increase at all at longer tail lengths, which show that it has a lot of spare capasity.

In general we see that the OpenCL implementation is not likely to be practical for such small frame sizes with realistic echo tail lengths. This might not pose a problem with modern systems, as this frame size is most likely to be used for lower quality signal with fairly low sample rates, for instance telephones sampling at 8kHz. The best case performance here is the C2050 with a 40 frame tail, which is still over 9 times slower than the reference. The data sizes involved are too small to warrant execution on a device where we have found the latencies of a single or a few memory transfers or empty kernel launches are longer than the reference implementation takes to complete processing of a frame.

5.5.2 Frames of 512 Samples

An avenue for increasing the size of the data sets with only single input and output channels, is to increase the frame size, *i.e.* the number of samples to be processed at once. With larger frame sizes, the size of the useful echo tails also increase in terms of the number of samples, now the minimum 2 frame tail is 1024 samples and the maximum 40 frame tail measured 20480 samples. As the latency dominated with 128 sample long frames, this levels the playing field to a certain degree. Longer frames are often used in practice in modern systems with enough bandwidth available, where higher sample-rates such as 48kHz are often used. Each sample now represent a shorter time interval, a hence more samples are needed in the echo tail to cover the same time interval as with the lower sample rate.

Execution times with increasing tail lengths with frames of 512 samples are plotted in Figure 5.4. Still, we see that the OpenCL implementations are in general slower than the reference implementation, but the gap is now smaller. The complete plot of the ATI CPU-implementation is not included as it scaled in a way similar as with the shorter frames, and is significantly slower than any of the other implementations on larger sizes (see the table in Appendix A for a complete listing).

Because of the poor scalability of our OpenCL implementation on CPU, this metric was excluded from most of the other benchmarks in this chapter. The ATI GPU-implementation is still troubled by the high latency already from the shortest echo tails, but scales fairly well on to longer tails, and it scales better than the reference implementation, although it never closes the gap at reasonable echo tail lengths. Note that the weight update kernel function had to be split in two on the Radeon 5870 with this frame size because of lack of memory available to each thread, so performance was slightly affected by this change (but was still parallelized into the same number of threads).

The GTX280 again shows fairly good performance, and the running times increases

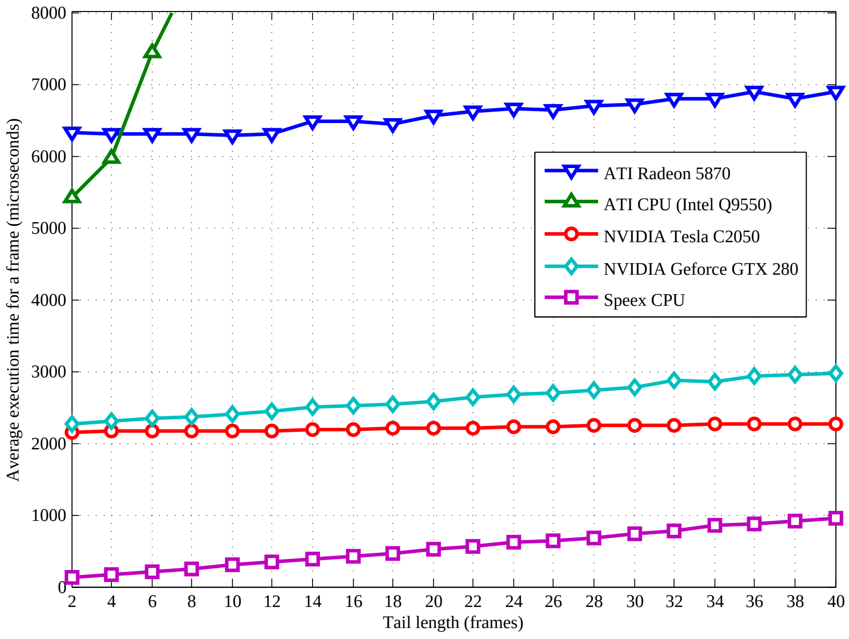


Figure 5.4: Average execution time of a single frame with increasing echo tail lengths (512 sample frames).

at a slower rate than the reference implementation as the echo tail grows. Even for the longest tails tested here, it is still 2.4x as slow as the reference. The C2050 again has the lowest latency and performance is barely affected by the tail length. At 10-20 frame long tails, it still performs significantly slower than the reference, but reduces the difference as the tail length is increased further. A 40 frame echo tail is already longer than what is commonly in used practice (represents 0.43 seconds with a 48kHz sample rate), so we focused on other factors that can be scaled to utilize the available parallelism for other purposes.

At these frame sizes, the best GPU implementation performance is becoming more comparable with the reference, which might make offloading computations reasonable prospect. However, the GPUs should display better performance to make up for the extra power consumption compared to the CPU implementation (for the hardware tested here).

This frame size was used for the rest of the benchmarking tests (unless otherwise specified) to increase the numerical intensity to a reasonable level on the GPUs.

5.6 Performance Impact of Multiple Channels

In addition to increasing the echo tail length, the other parameters that can significantly increase execution time of the echo cancellation program, are the number of input and output channels used. As it was ascertained in Section 5.5 that the GPUs in some cases had little problem with throughput, increasing the number of audio channels is a way to benefit from this idle capacity. As in the last section, tables containing the results can be found in Appendix A, in addition to the plots presented here.

5.6.1 Scalability of Input Channels

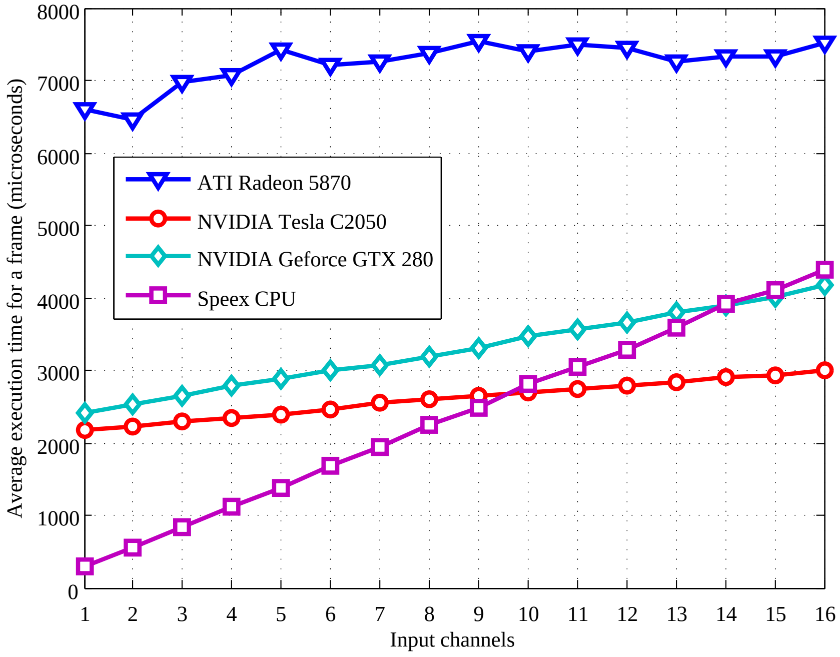


Figure 5.5: Average execution time of a single frame with increasing number of input channels (512 sample frames).

The result of the benchmark for processing 512 sample frames with a 10 frame echo tail and 1-16 input channels can be found in Figure 5.5. As noted earlier, the ATI Radeon 5870 displays long latency already using a single channel, but the execution time remains at a fairly constant level (slowly increasing). Even though it scales well, it does not overtake the CPU implementation in terms of performance within the 16 channel limit set here. Both the NVIDIA GPUs perform better than the reference from 9 (for the Tesla C2050) and 14 (for the Geforce GTX280) channels respectively. All the implementations, except for the ATI GPU, display fairly linear scaling properties within these parameters, with the reference having the greatest degradation of performance.

Even though the GPUs show good scaling properties as the number of input channels is increased, a large number of input channels for the filter function are rarely used in practice. Either the input channels are mixed together before being sent to the filter, or separate filter instances are used. Another reason for benchmarking with multiple input channels, is that it is somewhat analogous to having task parallelism with multiple filters at the same time, as mentioned in Section 4.10.3 (but difficult in practice with OpenCL at the time of writing on some platforms), this shows potential for use in larger systems such as a PBX.

5.6.2 Scalability of Output Channels

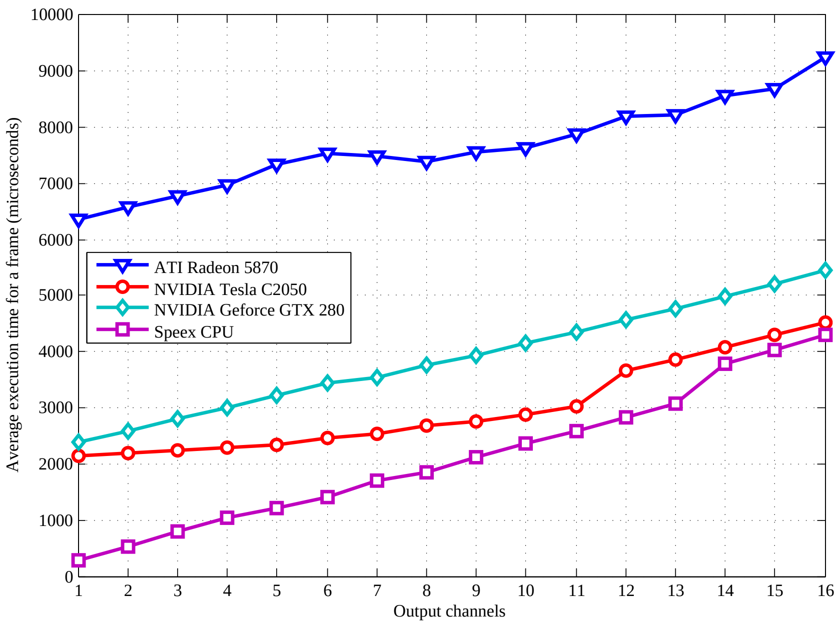


Figure 5.6: Average execution time of a single frame with increasing number of output channels (512 sample frames).

As an alternative to increasing the number of input channels, the number of output channels can also be adjusted. Results using a similar benchmark as for input channels can be found in Figure 5.6, the same parameters are used, but only a single input channel is present.

Some of the same characteristics can be found as with the test for variable number of input channels, but the reference implementation running times does not increase as sharply compared to the GPUs. This is the first benchmark which we see a significant change in the ATI GPU running time as the parameter is scaled, although it still is hindered by overhead.

The Tesla C2050 generally has similar performance to the reference above 14 output channels, with the reference only being about 5% faster at 16 channels. But the Geforce GTX280 never converges to the level of the reference implementation, although the difference is smaller at higher number of channels, and it is 1.24x slower using 16 channels.

5.6.3 Scalability of Both Input and Output Channels

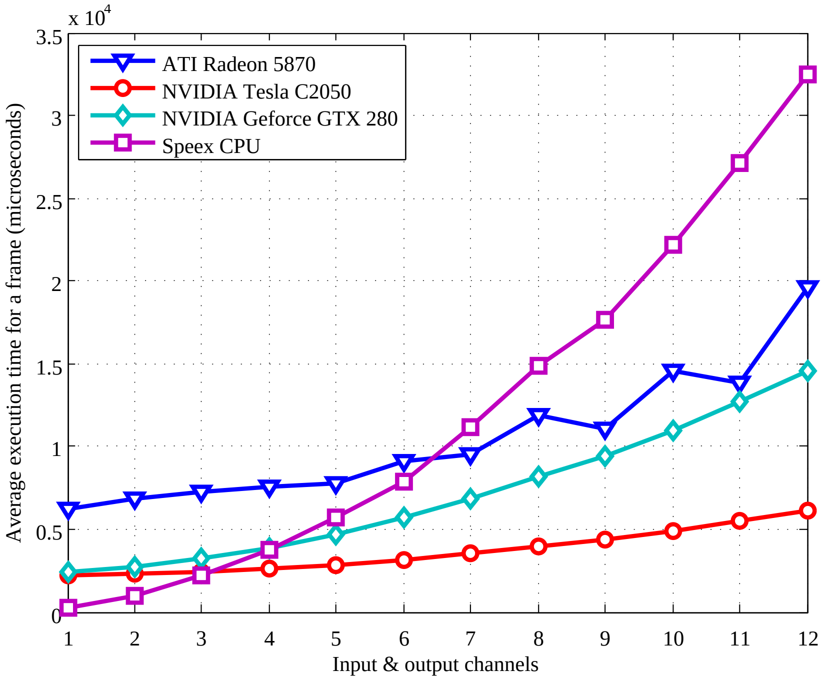


Figure 5.7: Average execution time of a single frame with increasing number of both input *and* output channels (512 sample frames).

To really test the scalability of the GPU implementations, a benchmark was also performed where both the number of input and output channels were increased at the same time. As is shown in Section 4.9 this increases execution time on CPU in a exponential fashion. The results can be found in Figure 5.7.

Although not the most practical benchmark, it does show that if the number of channels is increased enough, all the GPU implementation performs better than the reference. The NVIDIA GPUs overtake the performance of the reference already at three to four input and output channels, but the ATI GPU overtakes it at six channels. At 12 channels, the performance of the Tesla is 5.3x faster than the reference. Perhaps of particular interest here, is that the performance of the Geforce GTX280 increases to close to the level of the ATI Radeon 5870 at the higher number of channels. This shows that the ATI GPU has good performance when the overhead is not the dominant factor in the total execution time.

It is worth noting here that the time represented by a 512 sample frame with a sample rate of 48kHz is about 10668 microseconds, the reference implementation passes this threshold after seven channels. Above this limit, the filter could not possibly be used in a real-time system, the C2050 never exceeds the limit in this benchmark with the longest running time at 6122 microseconds at 12 channels. It should also be noted that this is an absolute upper bound, and the real time window will in practice be shorter.

5.6.4 A Reasonable Multi-channel Use Case

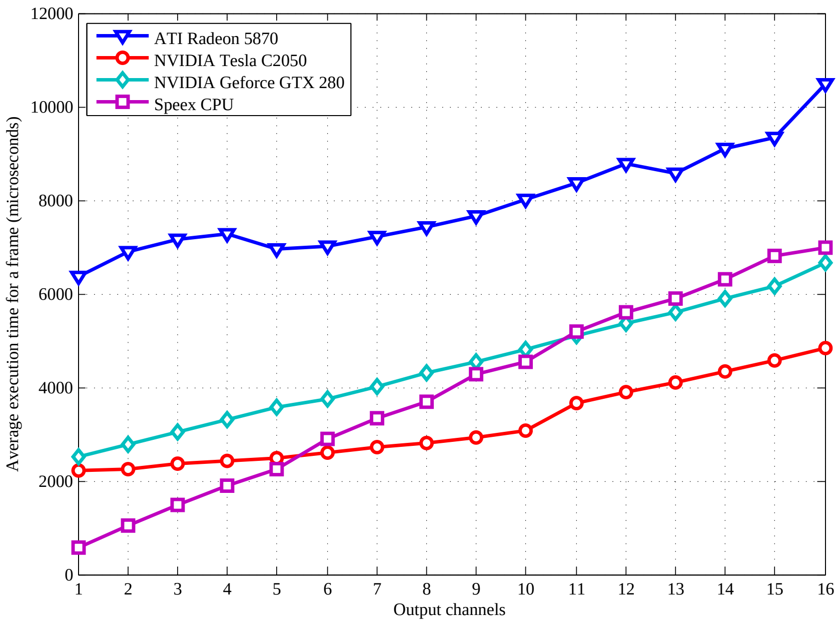


Figure 5.8: Average execution time of a single frame with two input channels and an increasing number of output channels (512 sample frames).

A more practical, but still fairly computationally intensive scenario was setup to test a more reasonable mix of input and output channels. The number of input channels was simply set to two, and the number output channels was increased similar to what was done in Section 5.6.2. The results can be found in Figure 5.8.

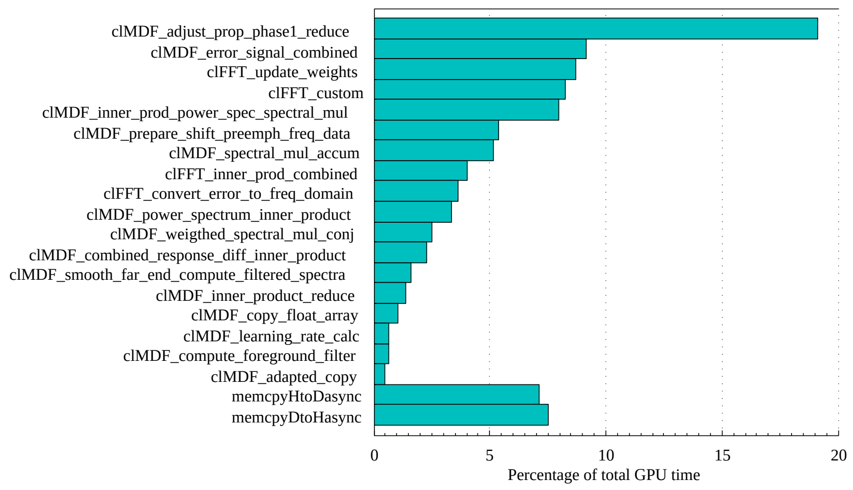
The Tesla C2050 GPU overtakes the performance of the reference for five output channels and more, which makes for a very practical application. Five speakers is not unreasonable in modern sound systems, as surround sound systems with four to five speakers or more are becoming widespread. At 16 output channels, it performs about 50% quicker than the reference ($4831\mu s$ vs. $6997\mu s$), but this is of limited practical use. The GTX280 needs 11 channels before it produces better performance than the reference. The Radeon 5870 never overtakes the reference, but displays an improved performance ratio compared to the reference on a large number of output channels, it is 33% slower at 16 output channels. The performance of the two latter GPUs in not very useful in practice, but could be used to offload from the CPU under certain circumstances.

5.7 Profiling Kernel Functions

To examine the execution time of functions on the GPU, one can either time them explicitly inside the application, or use a profiler. The NVIDIA Compute Profiler can be used to retrieve data about the running time of CUDA/OpenCL kernel functions on NVIDIA GPUs, together with a host of other performance metrics to help developers optimize the device code. A similar solution was also available form ATI, but not for the operating system and development environment used here.

Two use cases were run with the profiler, first with a set of very computationally light parameters (128 sample frames, 1280 sample echo tail, single channel), and then a more computationally demanding set (512 sample frames, 10240 sample echo tail, 4 input and 4 output channels) to gain insight in what kernels dominates in both cases. These are just a small sampling of the possible parameter configurations, but are interesting to get an overview and to show the difference between the two devices used here, the Geforce GTX280 and the Tesla C2050.

For the result of the profiler run with the less computationally example, see Figure 5.9. In general, we see that the C2050 has more equally distributed running time, with the longest running kernel (`clFFT_custom`) taking up 11% of the total time, where the GTX280 uses 19% of its GPU time on the `clMDF_adjust_prop_phase1_reduce` kernel, which is significantly faster on the C2050. This shows that the new Fermi architecture helps hide memory access time such as for `clMDF_adjust_prop_phase1_reduce`, which is not particularly computationally intensive, and instead kernels including FFTs, which involve more computations, dominates. With this configuration the



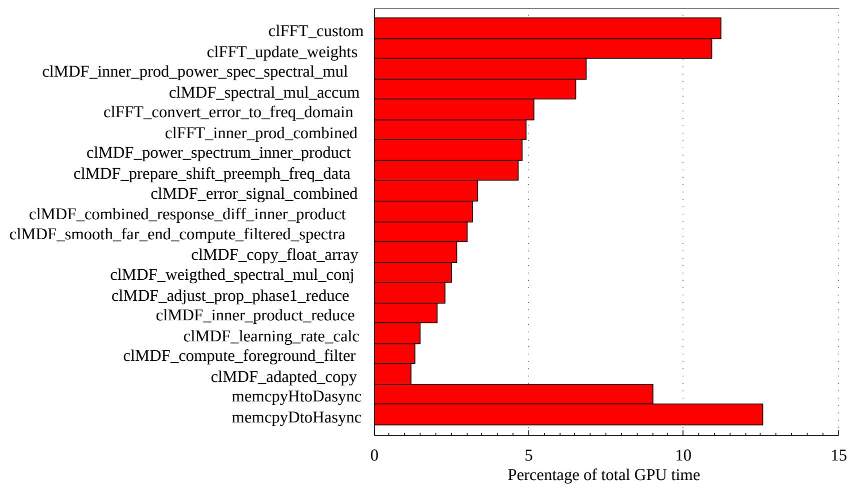
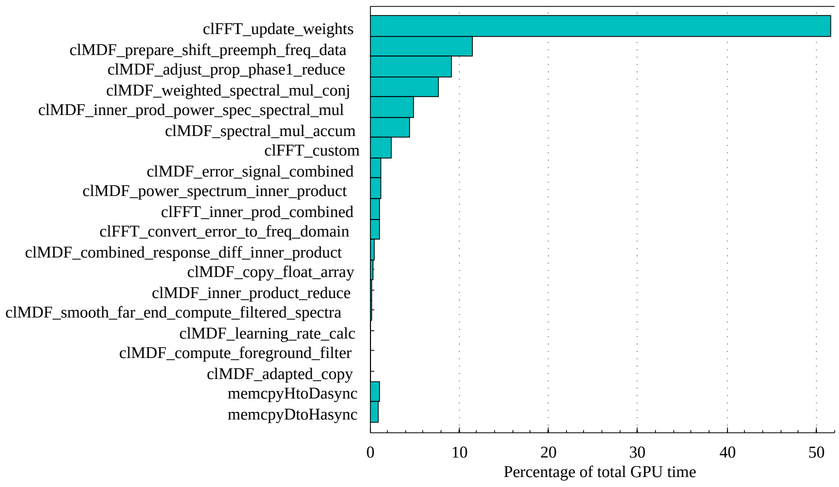


Figure 5.9: Kernel function exeuction time with NVIDIA Geforce GTX280 (top) and Tesla C2050 (bottom), 128 sample frames, single input/output channel. Notice the different scales on the X-axis.

running time inside the kernel will to a certain degree be amortized by overhead, so it might not be worth optimizing kernels to improve a few percent on smaller sizes.

The more computationally demanding profiler run, can be seen in Figure 5.10, and shows more similarity between the two devices as the frame size and number of channels are scaled up. The weight update kernel, which predictably is the most computationally demanding function in the filter, dominates the running time of both. The speed of the FFT implementation now becomes very important (see



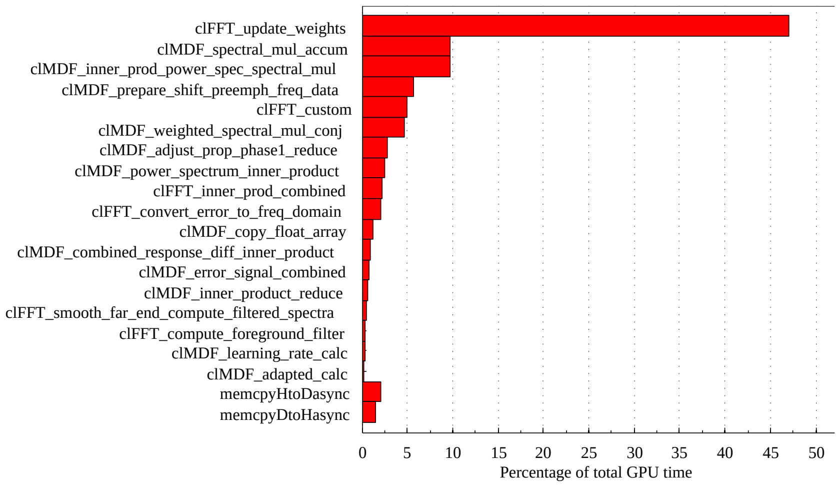


Figure 5.10: Kernel function exeuction time with NVIDIA Geforce GTX280 (top) and Tesla C2050 (bottom), 512 sample frames, four input/output channel.

Section 4.6 for more information on this kernel). It is also interesting to see the memory copy operations listed as the two bottom bars, which now takes up a much smaller fraction of the overall running time.

5.8 Performance With Load Balancing and CPU Load

A requirement for actually offloading computations to a GPU, or another co-processor, is that it actually decreases the work that needs to be carried out by the CPU. To get an idea of the actual CPU usage by the different OpenCL implementations tested, a simple scenario was set up to test the filter while running a script that measures system CPU load (same as the one that the load balancer is based on, see Section 4.10.2). The original Speex reference implementation was tested, together with the ATI CPU implementation, ATI GPU implementation and the NVIDIA GPU implementation. These were tested using the corresponding systems listed in Section 5.1, both systems were quad-core systems, and 100% CPU usage means that all the cores are occupied. The test scenario used 128 sample frames, with a 40 frame long echo tail, 2 input channels and 2 output channels, which should be enough to generate a representative load on the system.

As expected, the reference implementation uses 25% of the total CPU time, being that it is not written to utilize multiple threads, but fully utilize a single core. Both the NVIDIA and ATI GPU implementations averaged slightly below this, but still mostly above 20%. For NVIDIA, the Tesla C2050 generated a higher CPU load than the Geforce GTX280 (10-15%), but performed the task faster. This can be attributed to the test application simply having some CPU-demanding components, among other things most noticably file I/O. With longer execution times on the GPU, the CPU can be idle for longer periods of time while waiting for the GPU to finish. The Tesla C2050 had a CPU load similar to the ATI Radeon 5870, despite offering nearly three times the performance in this scenario.

The relatively high CPU load on the ATI GPU could be caused by the higher overhead of the implementation found in Section 5.2, but also by the fact that it depends on the X Window System on Linux systems to run, which NVIDIA does not. Extra CPU load is added by the rendering of the graphical console itself and certain background tasks that are common with such a system. The ATI CPU implementation is multithreaded and results in a much larger load on the CPU, with utilization between 80-85% for this application. While this shows that most of the CPU is used, it still leaves 10-15% idle capacity on the CPU, which is not ideal. Other OpenCL application can push the CPU usage with this implementation to close to 100%, but this shows that more complex application without long running kernel functions can struggle to reach such a high usage. As there is often some overhead in spawning a large number of threads on the CPU, more long-running kernels will probably benefit more from being run with OpenCL, over a traditional single-threaded implementation. As noted earlier, our OpenCL kernel functions have not been optimized with CPU execution in mind, so there might be scope for improvement here.

5.9 Summary of Results

After benchmarking the OpenCL implementation, some conclusions can be drawn.

5.9.1 Performance Is Often Bound by Latency

As we have seen, the OpenCL version of the filter is only usable in certain computationally intensive configurations in its current form. All the different kernel functions and several transfer operations that are called during a normal program run, collectively have too large overhead in the operations themselves to be usable on small frame sizes like 128 samples and with single input/output. Several methods could be used to reduce this, first among which is to even further combine kernel functions and reduce the number of kernel arguments to only the absolutely essential data. But this can be tedious work, unintended consequences to the performance can be hard to keep track of with different possible combinations of kernels. It does not get easier by the goal that it should be optimized against different hardware platforms and still retain compatibility. Auto tunable methods as discussed in Section 2.3.4 would be ideal, but as mentioned it can be challenging to apply them to a larger application compounded of a variety of different operations.

More operations could also be executed faster on the host (CPU), but there is a delicate balance between execution time and the time it takes to transfer the necessary memory back and forth to the device. Part of the problem here, is that the filter uses a lot of different buffers in memory, that can be hard to keep track of. The current implementation is fairly naive in its handling of the data structure, since it has all the data available on the device. By analyzing the life-cycle of the buffers and only transfer what is absolutely necessary and maybe reuse or combine them where possible, some overhead could be removed. The reference implementation is written to function even on embedded devices with limited amount of memory available, in contrast to most devices used by OpenCL today that have a fairly large amount of DRAM available to them. This could help increase performance if allocating more memory can be used to reduce the number of transfers and kernel arguments, and thus overhead.

5.9.2 Performance Scales Well With Demanding Parameters

When scaling the filter to large frames, long tails and several channels, the GPUs show better scaling properties than the reference implementation when they overcome the initial threshold of latency of about 2ms of the NVIDIA devices. When scaling above this threshold, the GPU performs the tasks faster than the CPU, and a few compute intensive kernel functions dominates the running time. The problem is that this threshold is already at the limit of running times with practical parameters of the filter. A good traditional multi-core CPU implementation might also close this gap up to the GPU performance at realistic parameters, since we only found single-digit speedups. But this does still not eliminate the fact that the performance is good enough for offloading processing to the device, which might free the CPU for other tasks.

At demanding parameters to the filter, a few of the compute kernels that dominate running times, are completely dependent upon the FFT implementation. In these cases, significant performance gains might be possible by using an even more optimized FFT implementation. We discovered some performance difference between the library used and the CUFFT library in Section 5.3, so there should be some possible performance gains available by tweaking the existing library version used or switching to another library altogether. Memory accesses patterns to global memory also becomes more important at these sizes, so a more detailed analysis might reveal further possibilities for optimizations. But again, there is often a fine balance between kernel code that executes well on "simpler" parameters with as little latency as possible and code that scales well when they need to process more data.

There is a point to be made that switching GPGPU framework to CUDA at the time of writing could have gained some performance due to lower latency found in Section 5.3, but this would limit hardware compatibility and co-processing potential. It remains to be seen in the future if better implementations can close this gap, or other vendors can catch up with other solutions. A performance difference was also found by Zhang *et.al.*[36] between CUDA and OpenCL for CT reconstruction and image recognition. They found a 9 times speedup with their CUDA implementation over OpenCL (but still much faster than their CPU implementation), but they were using early versions of the vendor implementations.

5.9.3 OpenCL CPU Performance Is Lacking

The load-balancing possibilities of OpenCL is one of the more appealing features of the technology, but we found CPU-performance with OpenCL to be very much lacking in our benchmarks. This for the most part due to the FFT library not being optimized for CPU execution and that no specific optimizations were added to echo cancellation code, but we also found significant overhead by execution through OpenCL, compared to (comparatively) almost no overhead with a traditional CPU program. This made the CPU implementation of OpenCL mainly into a debugging tool during development, but some steps can be taken to improve on this:

- Memory should be shared in a more rational way between host and device when the device is a CPU, mapping the memory can lead to large performance benefits in such cases.

- Another FFT implementation for OpenCL should be used that is optimized for CPUs (if available), or it could simply be performed by existing multi-threaded FFT libraries for CPUs, although this increases code complexity. The kernels that includes FFTs dominated running times when more demanding parameters were used, even more so than on GPU. If an OpenCL FFT with similar or better performance than traditional single-threaded CPU libraries is not available, it contradicts the reason for running the filter with OpenCL on CPU, as it is the most attractive operation to parallelize.

- CPU optimization should be written into the computational kernels and the host code to utilize the CPU better. A very large number of work-items/threads might not always be the answer on CPUs, and restrictions to memory performance is different. Optimally, all this should be combined with auto tunable methods.

Chapter 6

Conclusions and Future Work

In this thesis, we investigated co-processing between CPUs and GPUs for audio processing, more specifically acoustic echo cancellation which is used to avoid sending the received audio back to the initial sender as an echo. Since GPUs are massively parallel devices that often have spare capacity, off-loading computations to them can have great benefits for applications that need available CPU time for other purposes, for instance for decoding video in video conferencing or calculating the next move by an artificial opponent in a computer game. On embedded devices, off-loading computations to the GPU may be crucial to achieve the performance needed or conserve power. Execution on GPUs also opens up the possibility of massive parallel sound processing, such as in large telephony systems.

The echo filter implemented in this thesis is based on the original C-code from the Speex library. Our optimized programs developed to the new OpenCL standard can be executed both on ATI/NVIDIA GPUs and on multi-core CPUs. Tests were created to validate each step of the program against the original reference version, so they can be run side-by-side using the same input-data for verification. The Apple OpenCL FFT library was used as a basis for the optimized transform written for both CPUs and GPUs on OpenCL.

6.1 Conclusions

Benchmarks of our two implementations (one CPU-based and one with GPU co-processing, both implemented in OpenCL) showed that the more computationally demanding configurations of the echo cancellation filter favored execution on GPUs. When more input and output channels are added, the execution time of the reference (Speex) implementation increased steeply and this allowed the massive parallelism of GPUs to be utilized to increase performance. We found speedups of up to 5.3x when using 12 input and 12 output channels, and comparable performance for a more realistic configuration of 2 input channels and 5 or more output channels. What made this increase in performance possible was that FFTs of large data sizes dominated the execution time when scaling the parameters, and this was a well parallelizable algorithm on GPUs.

However, on less demanding input configurations, our GPU implementation suffered from the overhead of the OpenCL vendor implementations used. For processing single channel audio data with smaller frame sizes, the overhead completely dominated the running time on the GPU. This meant that the performance was not comparable to the reference CPU version, which was for instance 77x faster than the fastest OpenCL implementation on frames of 128 samples run in single channel with an echo tail of 10 frames.

Large differences were found in the overhead both for kernel execution and memory transfers between different GPGPU frameworks and vendor implementations of OpenCL, as well as between different generations of GPU hardware. For instance, we found the fastest GPU tested with CUDA to be almost 60x times faster at a single byte memory write to the device than the slowest OpenCL implementation from another vendor. The large variations found indicate that OpenCL vendor implementations still have a way to go before they reach maturity. Differences were also found between the available FFT libraries for CUDA and OpenCL, with the GPU-vendor supported CUFFT library coming out on top.

Overhead from the OpenCL implementation severely impacts performance of the echo cancellation filter, as it depends on a long series of kernel because of the large number of varying operations performed. Memory transfers can also be limiting, as some tasks are best executed on the CPU. This was shown in single-channel performance where the fastest OpenCL implementation had a minimum running time of about 2ms, and it did not increase significantly from this level until multiple channels were added. The overhead incurred by the implementation can be reduced by further combining kernel functions, but this could affect performance scalability. In general, GPUs are not very well suited to sub-milliseconds tasks such as this, as stated by Owens *et.al.*[7]. This problem can be alleviated by new architectures where the CPU and GPU have access to the same main memory to eliminate memory transfers and reducing overall overhead. Such architectures are already common in embedded devices, an certain PC platforms.

However, with multiple channels, the GPU performance is already sufficient for offloading tasks from the CPU. A prototype load-balancer was created to offload the CPU by executing the filter on the GPU, if the overall system CPU load becomes too high. Because of lacking performance from the OpenCL CPU implementation (mainly due to a GPU only optimized FFT library), the load-balancer switches between the reference implementation on the CPU and the OpenCL version on GPU. In computationally demanding configurations, this can free available CPU time for other applications, but executing on the GPU itself occupies the majority of a single CPU core, so this is most useful in multi-core systems. Offloading the CPU can be very important in embedded devices, which increasingly contain advanced GPUs and will have GPGPU frameworks available to them in the future.

Overall, we found it plausible to implement audio processing on GPUs, but detailed analysis of the application at hand should be performed, so as to determine whether it can generate enough computational load to utilize the massive parallelism inherent in GPUs and overcome the entry barrier represented by implementation overhead. This can improve as the frameworks and hardware for GPGPU matures, so it becomes even more attractive for co-processing.

6.2 Future Work

Many possibilities exist for further work on the subject of this thesis, a few are listed in this section.

6.2.1 Further Optimize the OpenCL Implementation

Although some tuning was done on the OpenCL implementation, most of the time was still used simply creating the first implementation and making sure it executes correctly on different platforms. Several avenues exist for further optimizing the existing codebase. One possibility is to experiment with the ordering and division of tasks between kernels, simply reducing the number of kernel functions could improve performance by reducing overhead. A pragmatic approach could also be taken to further mix execution on CPU and GPU, but the challenge is then to minimize memory transfer while taking advantage of some operations that is quicker on the CPU.

6.2.2 Investigate Massively Parallel Audio Processing

A way to further increase the workload on the GPU beyond using multiple channels, is to run several independent audio streams in parallel as would be the case for instance with the Asterisk PBX. A scheme for how this could be implemented was proposed in 4.10.3. Such an approach would be possible with both OpenCL and CUDA at the time of writing, but this only become available with the newest generation of hardware.

6.2.3 Explore Other Algorithms for Echo Cancellation

Other methods exists for echo cancellation than the particular one used in Speex (AUMDF). One alternative, described in [37], is using the Discrete Cosine Transform, Extended Lapped Transform and Lapped Orthogonal Transform in an LMS filter. It would be interesting to perform a comparison of the accuracy and performance of such a filter on parallel architectures. The basic transforms have already been shown to perform well on GPUs[38]. In general, signal processing algorithms often follow a pipeline structure, but this does not necessarily translate well to GPU execution. Custom algorithms that utilize massively parallel architectures would be ideal for this task, but requires more research into signal processing.

6.2.4 Integrate the Code Better Into the Library

While the current code is built to integrate into the existing Speex library, it is by no means production ready in a sense that it could be directly incorporated without any changes. It needs to be better integrated with the build system, stabilized and tested on more platforms to be a portable part of the library. It might be interesting to adapt it to a larger pattern used for implementing pieces of the library in OpenCL.

6.2.5 Investigate Converting Other Parts of Speex to OpenCL

The echo cancellation filter is only a small part of Speex, other functionality could possibly be converted to OpenCL for running on GPU/accelerators. Maybe the most obvious component would be the decoding of the audio stream itself, but it will be a fairly challenging task. Other preprocessing filters such as noise cancellation and voice activity detection might also be possible candidates. It might also be possible to add new filters that are considered too arithmetically intensive for the CPU.

6.2.6 Utilize the GPU Version in an Application

Currently, the OpenCL version of the echo filter is only tested using the "testecho" program that comes with Speex. It would be beneficial to have a complete application that could demonstrate processing on different hardware. On could imagine a pure demo application to demonstrate the concepts, or for instance a custom version of the Mumble application. Such an application should exploit the echo filter with parameters that are beneficial for offloading the CPU.

6.2.7 Running on an Embedded Device

As embedded devices are becoming more powerful, they are also equipped with more powerful GPUs and accelerator chips. The platform-agnostic approach of OpenCL would benefit these platform greatly, especially since they often have less powerful CPUs and need to take advantage of all their hardware to achieve high performance. By executing a program where it is most appropriate with regards to power consumption is also very important on these devices. The Speex codec itself should be a prime candidate for software packages that could be of great use on embedded devices, especially mobile phones, which are increasingly merging with VoIP and other online services such as video conferencing.

6.3 Concluding Remarks

As can shown from this work, off-loading computations to accelerators such as GPU can offer attractive, energy-efficient ways of increasing system performance, especially for real-time applications where CPU-load is a critical issue. The optimization techniques developed in this thesis are therefore expected to be of increasing importance as GPUs and CPUs become more and more integrated, especially on embedded devices. This makes latencies become less of an issue and hence the value of our results stronger. This is especially true for applications such as audio processing on just a few channels which has less computational load per memory access compared to traditional GPGPU applications, such as for instance dense linear algebra

Bibliography

[1] A. Nukada and S. Matsuoka. Auto-tuning 3-d fft library for cuda gpus. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–10, New York, NY, USA, 2009. ACM. 1, 24, 25, 39

[2] A. Munchi. *OpenCL specification, Version 1.1, Revision 33.* Khronos Group, June 2010. `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`. 1, 12, 13, 15, 18, 19

[3] J. Valin. *The Speex Codec Manual Version 1.2 Beta 3.* Xiph.org Foundation, December 2007. `http://www.speex.org/docs/manual/speex-manual.pdf`. 2, 30, 31

[4] B. Cowan and B. Kapralos. Spatial sound for video games and virtual environments utilizing real-time gpu-based convolution. In *Future Play '08: Proceedings of the 2008 Conference on Future Play*, pages 166–172, New York, NY, USA, 2008. ACM. 8, 29

[5] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. Gpu cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society. 9

[6] NVIDIA OpenCL programming guide for the CUDA architecture version 3.1. Guide, May 2010. 10, 11, 17, 20, 21

[7] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879 –899, may 2008. 10, 96

[8] A. Munchi. *OpenCL specification, Version 1.0, Revision 48.* Khronos Group, September 2009. `http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf`. 12

[9] Nvidia's next generation cuda compute architecture: Fermi. White paper, September 2009. `http://www.nvidia.com/content/PDF/fermi_white_`

papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. 14, 20, 23, 24, 66, 69, 72

[10] Amd: Unleashing the power of parallel compute - with commodity ati radeon 5800 series gpu. Course notes, OpenCL Parallel Programming for Computing and Graphics, SIGGRAPH Asia 2009, December 2009. http://sa09.idav.ucdavis.edu/docs/SA09_AMD_IHV.pdf. 14, 20

[11] Nvidia opencl best practices guide - cuda toolkit 3.1. Guide, April 2010. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_OpenCL_BestPracticesGuide.pdf. 18, 19, 20

[12] B. Wilkinson and M. Allen. *Parallel Programming - Techniques and Applications Using Networked Workstations and Parallel Computers.* Pearson Prentice-Hall, second edition, 2005. International edition. 22

[13] Apple grand central dispatch. Technology Brief, 2009. http://images.apple.com/macosx/technology/docs/GrandCentral_TB_brief_20090903.pdf. 23

[14] D. Cederman and P. Tsigas. On dynamic load balancing on graphics processors. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 57–64, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Assciation. 23

[15] S. Kamil, C. Cy, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, 19-23 2010. 24

[16] S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp, and W.W. Hwu. An adaptive performance modeling tool for gpu architectures. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 105–114, New York, NY, USA, 2010. ACM. 25

[17] E. Gallo and N. Tsingos. Efficient 3d audio processing on the gpu. In *Proceedings of the ACM Workshop on General Purpose Computing on Graphics Processors.* ACM, August 2004. 29

[18] F. Trebien and M.M. Oliveira. Realistic real-time sound re-synthesis and processing for interactive virtual worlds. *Vis. Comput.*, 25(5-7):469–477, 2009. 29

[19] Q. Zhang, L. Ye, and Z. Pan. Physically-based sound synthesis on gpus. In *Entertainment Computing - ICEC 2005*, volume 3711, pages 328–333. Springer Berlin / Heidelberg, 2005. 29

[20] M. Jedrzejewski and K. Marasek. Computation of room acoustics using programmable video hardware. In K. Wojciechowski, B. Smolka, H. Palus, R.S. Kozera, W. Skarbek, and L. Noakes, editors, *Computer Vision and Graphics.* Springer Netherlands, 2006. 29

[21] P.R. Dixon, T. Oonishi, and S. Furui. Harnessing graphics processors for the fast computation of acoustic likelihoods in speech recognition. *Comput. Speech Lang.*, 23(4):510–526, 2009. 29

[22] H. Ludvigsen and A.C. Elster. Real-time ray tracing using Nvidia OptiX. In *Eurographics Short Papers*, pages 65 –68. The Eurographics Association, 2010. 29

[23] N. Tsingos, E. Gallo, and G. Drettakis. Breaking the 64 spatialized sources barrier. *Gamasutra Audio Resource Guide 2003*, May 2003. 29

[24] J Valin. On adjusting the learning rate in frequency domain echo cancellation with double-talk. *IEEE Transactions on Audio, Speech and Language Processing*, 15(3):1030–1034, March 2007. 34, 35, 60

[25] J. Soo and K.K. Pang. Multidelay block frequency domain adaptive filterr. *IEEE Transactions on Acoustics, Speech and Signal processing*, 38(2):373–376, February 1990. 33, 34, 36

[26] Å. Herikstad. Gpu sound processing, December 2008. Specialization Project TDT4590, Complex Computer Systems, NTNU. 33, 36, 39, 47

[27] A. Daher, E.-H. Baghious, G. Burel, and E. Radoi. Overlap-save and overlap-add filters: Optimal design and comparison. *Signal Processing, IEEE Transactions on*, 58(6):3066 –3075, june 2010. 35

[28] J. M. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comp.*, 19:297, 1965. 38

[29] M Frigo and S.G. Johnson. *FFTW for version 3.2.2.* Massachusetts Institute of Technology, July 2009. `http://www.fftw.org/fftw3.pdf`. 38

[30] G. Stantchev, D. Juba, W. Dorland, and A. Varshney. Using graphics processors for high-performance computation and visualization of plasma turbulence. *Computing in Science Engineering*, 11(2):52 –59, march-april 2009. 39

[31] V. Volkov and B. Kazian. Fitting fft onto the g80 architecture. University of California, Berkeley, May 2008. 40

[32] N.K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press. 40

[33] S.W. Smith. *The Scientist & Engineer's Guide to Digital Signal Processing.* California Technical Pub., first edition, 1997. Chapter 12 retrieved at 25.May at `http://www.dspguide.com/CH12.PDF`. 48, 49

[34] IEEE, editor. *802.3-2008 IEEE Standard for Information Technology–Telecommunications and Information Exchange Between Systems–Specific Requirements Part 3: CSMA/CD Access Method and Physical Layer Specifications.* New York, 2008. 61

[35] J. Silvestre, V. Sempere, and T. Albero. Impact of the use of large frame sizes in fieldbuses for multimedia applications. In *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on*, volume 1, pages 8 pp. –440, 19-22 2005. 61

[36] W. Zhang, L. Zhang, S. Sun, Y Xing, Y. Wang, and J. Zheng. A preliminary study of opencl for accelerating ct reconstruction and image recognition. In *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, pages 4059 –4063, oct. 2009. 92

[37] H. S. Malvar. *Signal Processing with Lapped Transforms.* Artech house, first edition, 1992. 98

[38] A.A. Aqrawi. Effects of compression on data-intensive algorithms. Master's thesis, Norwegian University of Science and Technology, 2010. 98

Appendices

Appendix A

Benchmark Time Measurements

Since a few of the tables with time measurements from Chapter 5 were rather large, with all the measurements done, the data has been collected in this appendix as a reference.

Performance Impact of Echo Tail Length

Shorthand table-headings:

- *5870* - ATI Radeon 5870 GPU with OpenCL.
- *OCLCPU* - AMD Phenom II X4 965 CPU with the ATI OpenCL implementation for CPU.
- *C2050* - NVIDIA Tesla C2050 GPU with OpenCL.
- *GTX280* - NVIDIA Geforce GTX280 GPU with OpenCL.
- *Speex* - Intel Core 2 Quad Q9550 with the reference CPU implementation in the Speex library.

Frames of 128 Samples Average execution time of a single frame with increasing echo tail lengths (128 sample frames). All values are in microseconds.

Echo tail length	5870	OCLCPU	C2050	GTX280	Speex
2 frames	5400	2391	2062	2114	35
4 frames	5714	2440	2068	2141	39
6 frames	5269	2794	2062	2166	54
8 frames	5227	3015	2064	2190	63
10 frames	5694	3381	2070	2216	74
12 frames	5428	3658	2074	2239	81
14 frames	5868	4045	2075	2263	92
16 frames	5648	4246	2084	2291	102
18 frames	5621	4583	2093	2316	110
20 frames	5348	4729	2096	2339	122
22 frames	5665	5038	2101	2361	130
24 frames	5631	5122	2097	2388	144
26 frames	5622	5604	2104	2414	148
28 frames	5528	5759	2103	2441	159
30 frames	5803	5733	2103	2461	167
32 frames	5645	5909	2105	2487	177
34 frames	5607	6302	2105	2512	188
36 frames	5768	6466	2114	2541	201
38 frames	5695	6753	2116	2562	209
40 frames	5654	6944	2116	2588	223

Frames of 512 Samples Average execution time of a single frame with increasing echo tail lengths (512 sample frames). All values are in microseconds.

Echo tail length	5870	OCLCPU	C2050	GTX280	Speex
2 frames	6317	5422	2151	2274	118
4 frames	6302	5966	2167	2310	168
6 frames	6300	7432	2157	2335	208
8 frames	6299	8588	2159	2369	251
10 frames	6291	9842	2163	2401	304
12 frames	6311	10467	2175	2444	339
14 frames	6485	11768	2185	2497	388
16 frames	6482	12435	2191	2511	429
18 frames	6432	14060	2197	2545	456
20 frames	6552	14763	2201	2583	511
22 frames	6617	15994	2210	2630	557
24 frames	6661	16661	2216	2670	613
26 frames	6645	17912	2218	2700	646
28 frames	6701	18650	2238	2731	668
30 frames	6712	23575	2248	2778	731
32 frames	6786	24603	2253	2866	778
34 frames	6788	25838	2257	2856	851
36 frames	6891	26889	2266	2929	873
38 frames	6786	28320	2269	2960	912
40 frames	6899	29119	2270	2965	947

Performance Impact of Multiple Channels

Scalability of Input Channels Average execution time of a single frame with increasing number of input channels (512 sample frames). All values are in microseconds.

Input channels	5870	C2050	GTX280	Speex
1 channel	6600	2168	2410	295
2 channels	6449	2221	2519	564
3 channels	6987	2287	2642	832
4 channels	7059	2331	2782	1117
5 channels	7434	2383	2876	1384
6 channels	7223	2465	2993	1673
7 channels	7226	2548	3078	1943
8 channels	7373	2601	3189	2248
9 channels	7541	2643	3296	2474
10 channels	7408	2685	3460	2810
11 channels	7488	2737	3567	3037
12 channels	7449	2779	3652	3280
13 channels	7268	2839	3790	3578
14 channels	7341	2911	3894	3918
15 channels	7335	2938	4020	4109
16 channels	7514	3002	4186	4386

Scalability of Output Channels Average execution time of a single frame with increasing number of output channels (512 sample frames). All values are in microseconds.

Output channels	5870	C2050	GTX280	Speex
1 channel	6368	2156	2395	293
2 channels	6587	2205	2588	529
3 channels	6786	2250	2801	799
4 channels	6980	2295	3011	1040
5 channels	7337	2346	3219	1219
6 channels	7545	2461	3445	1399
7 channels	7482	2536	3544	1693
8 channels	7400	2672	3752	1843
9 channels	7557	2749	3943	2122
10 channels	7637	2868	4154	2373
11 channels	7884	3034	4361	2594
12 channels	8188	3675	4561	2820
13 channels	8219	3872	4770	3077
14 channels	8576	4084	4976	3789
15 channels	8688	4299	5197	4044
16 channels	9248	4529	5451	4305

Scalability of Both Input and Output Channels Average execution time of a single frame with increasing number of both input *and* output channels (512 sample frames). All values are in microseconds.

Input/output channels	5870	C2050	GTX280	Speex
1 channel	6248	2176	2397	294
2 channels	6812	2283	2741	1011
3 channels	7304	2457	3243	2222
4 channels	7542	2629	3886	3708
5 channels	7790	2868	4660	5742
6 channels	9082	3145	5663	7901
7 channels	9565	3571	6881	11216
8 channels	11875	3952	8196	14873
9 channels	11100	4389	9433	17692
10 channels	14590	4921	10973	22230
11 channels	13821	5545	12701	27161
12 channels	19657	6122	14530	32472

A Reasonable Multi-channel Use Case Average execution time of a single frame with two input channels and an increasing number of output channels (512 sample frames). All values are in microseconds.

Output channels	5870	C2050	GTX280	Speex
1 channel	6360	2211	2505	560
2 channels	6898	2264	2766	1038
3 channels	7153	2364	3035	1494
4 channels	7270	2425	3301	1893
5 channels	6946	2487	3559	2253
6 channels	7007	2606	3750	2884
7 channels	7230	2725	4017	3335
8 channels	7434	2804	4309	3682
9 channels	7676	2923	4542	4276
10 channels	8022	3087	4818	4556
11 channels	8376	3650	5094	5199
12 channels	8765	3909	5357	5589
13 channels	8588	4094	5609	5909
14 channels	9108	4325	5895	6314
15 channels	9335	4560	6170	6812
16 channels	10499	4831	6670	6997

Appendix B

Source Code

In this appendix, you will find relevant source code. Host side OpenCL code has not been included, only device code.

B.1 Test Program

This is the source code for the test application, containing the main function used for testing and benchmarking. Some benchmarking code has been removed from this listing, but it only runs the main program several time with increasing echo tail lengths, input channels or output channels to avoid recompilation of kernel functions.

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <fcntl.h>
#include "speex/speex_echo.h"
#include "speex/speex_preprocess.h"

#define NN 512
#define TAIL 512*20

#define MAX_ITERATIONS 10000
// If benchmarking on number of channels, set these values to the maximum value
    that
// is going to be tested
#define NUM_CHANNELS 1  // OpenCL version is limited to max 64 channels
#define NUM_SPEAKERS 1
```

```
#define ENABLE_LOAD_BAL 1 // Enable load balancer

// Used to determine between the Speex CPU reference implementation and the
     OpenCL version
enum MDF_DEVICE
{
    OPENCL = 0,
    CPU_REFERENCE = 1,
};

// Load monitor global variables
FILE *fp;
char *cpustring;
struct timeval last_cpu_check, curr_time;
int exec_dev = CPU_REFERENCE;
int req_dev = CPU_REFERENCE;
int prev_total = 0;
int prev_idle = 0;

int get_system_CPU_load();
void interleave_input(short *echo_in, short *ref_in, short *echo_out, short *
     ref_out);

int main(int argc, char **argv)
{
   FILE *echo_fd[NUM_SPEAKERS];
   FILE *ref_fd[NUM_CHANNELS];
   FILE *e_fd[2];       // Write one output from CPU and one from OpenCL
   FILE *errors_file_fd;
   // Frame read from file
   short *echo_buf = (short *)malloc(sizeof(short)*NN*NUM_SPEAKERS);
   // Frame passed to filter, interleaved channels
   short *echo_buf_interleaved = (short *)malloc(sizeof(short)*NN*NUM_SPEAKERS);
   short *ref_buf = (short *)malloc(sizeof(short)*NN*NUM_CHANNELS);
   short *ref_buf_interleaved = (short *)malloc(sizeof(short)*NN*NUM_CHANNELS);
   short *e_buf = (short *)malloc(sizeof(short)*NN*NUM_CHANNELS);
   short *ref_buf_copy = (short *)malloc(sizeof(short)*NN*NUM_CHANNELS);
   short *echo_buf_copy = (short *)malloc(sizeof(short)*NN*NUM_SPEAKERS);
   short *e_buf_copy = (short *)malloc(sizeof(short)*NN*NUM_CHANNELS);
   float *residual_error = (float *)malloc(sizeof(double)*MAX_ITERATIONS);
   SpeexEchoState *st;
   SpeexEchoState *st_copy;
   SpeexPreprocessState *den;
   SpeexPreprocessState *den_copy;
   int sampleRate = 48000;
   int last_cpu_usage = 0;
   int i = 0;
   int j = 0;
   int run = 1;
   char clear_key;

   cpustring = (char *)malloc(sizeof(char) * 80);
   #ifdef ENABLE_LOAD_BAL
   gettimeofday(&last_cpu_check, NULL);
   #endif

   speex_echo_opencl_init_dev(NN);

   for(i=0;i<MAX_ITERATIONS;i++)
      residual_error[i] = 0.0;

   if (argc != 4)
   {
      fprintf(stderr, "testecho mic_signal.sw speaker_signal.sw output.sw\n");
      exit(1);
   }
   // Output input files, just open several version of the single-channel files
   for(i=0;i<NUM_CHANNELS;i++)
```

```
    {
       printf("Open loop input channels.\n");
       ref_fd[i] = fopen(argv[1], "rb");
    }
    for(i=0;i<NUM_SPEAKERS;i++)
    {
       printf("Open loop output channels.\n");
       echo_fd[i] = fopen(argv[2], "rb");
    }
    // Write to two output files
    printf("Opening regular output file.\n");
    e_fd[0] = fopen(argv[3], "wb");
    printf("Opening OpenCL output file.\n");
    char *ocl_out_string = (char *)malloc(sizeof(char)*500);
    sprintf(ocl_out_string, "%s_ocl", argv[3]);
    e_fd[1] = fopen(ocl_out_string, "wb");

    printf("Tried to open files.\n");

    // Check that the output file could be opened
    if(e_fd[0] == NULL)
    {
       fprintf(stderr, "The output file could not be opened for writing!\n");
       exit(1);
    }
    else if(e_fd[1] == NULL)
    {
       fprintf(stderr, "The output file for OpenCL could not be opened for
            writing!\n");
       exit(1);
    }
    // Check that the input files for all channels could be opened
    else
    {
       for(i=0;i<NUM_SPEAKERS;i++)
       {
          if(echo_fd[i] == NULL)
          {
             fprintf(stderr, "One of the given filenames does not exist!\n");
             exit(1);
          }
       }
       for(i=0;i<NUM_CHANNELS;i++)
       {
          if(ref_fd[i] == NULL)
          {
             fprintf(stderr, "One of the given filenames does not exist!\n");
             exit(1);
          }
       }
    }
    printf("Files are open.\n");

    printf("Running regular echo cancellation.\n");

    st = speex_echo_state_init_mc(NN, TAIL, NUM_CHANNELS, NUM_SPEAKERS);
    st_copy = speex_echo_state_init_mc(NN, TAIL, NUM_CHANNELS, NUM_SPEAKERS);
    den = speex_preprocess_state_init(NN, sampleRate);
    den_copy = speex_preprocess_state_init(NN, sampleRate);
    speex_echo_ctl(st, SPEEX_ECHO_SET_SAMPLING_RATE, &sampleRate);
    speex_echo_ctl(st_copy, SPEEX_ECHO_SET_SAMPLING_RATE, &sampleRate);
    speex_preprocess_ctl(den, SPEEX_PREPROCESS_SET_ECHO_STATE, st);
    speex_preprocess_ctl(den_copy, SPEEX_PREPROCESS_SET_ECHO_STATE, st_copy);

    speex_echo_opencl_init_mem(st);
    speex_echo_opencl_execution_time_init();

    while (run && i < MAX_ITERATIONS)
    {
       printf("Starting loop iteration echo.\n");
       // Check if end of file has been reached, for all channels
       for(j=0;j<NUM_SPEAKERS;j++)
       {
           if(feof(echo_fd[j]))
                   run = 0;
       }
       for(j=0;j<NUM_CHANNELS;j++)
       {
           if(feof(ref_fd[j]))
                   run = 0;
       }
       if(run)
       {
          for(j=0;j<NUM_SPEAKERS;j++)
          {
             fread(echo_buf + j*NN, sizeof(short), NN, echo_fd[j]);
          }
          for(j=0;j<NUM_CHANNELS;j++)
          {
             fread(ref_buf + j*NN, sizeof(short), NN, ref_fd[j]);
          }
          interleave_input(echo_buf, ref_buf, echo_buf_interleaved,
               ref_buf_interleaved);
          init_checkpoints();
          init_timingpoints();
          // Uncomment to sync the OpenCL and CPU state for each frame
          // copy_echo_state(st, st_copy);
          memcpy(ref_buf_copy, ref_buf_interleaved, sizeof(short)*NN*NUM_CHANNELS
               );
          memcpy(echo_buf_copy, echo_buf_interleaved, sizeof(short)*NN*
               NUM_SPEAKERS);

          #ifdef ENABLE_LOAD_BAL
          if(exec_dev == CPU_REFERENCE)
          #endif
                 // Run reference implementation
                 speex_echo_cancellation(st, ref_buf_interleaved,
                      echo_buf_interleaved, e_buf);
          #ifdef ENABLE_LOAD_BAL
          else if(exec_dev == OPENCL)
                 // Run OpenCL implementation when load balancing (to the same
                      output)
                 speex_echo_cancellation_opencl(st_copy, ref_buf_copy,
                      echo_buf_copy, e_buf);
          #else
                 // Run OpenCL implementation when not load balancing (to
                      separate output)
                 speex_echo_cancellation_opencl(st_copy, ref_buf_copy,
                      echo_buf_copy, e_buf_copy);
          #endif

          cleanup_checkpoints();
          cleanup_timingpoints();
          // Store residual error divided by number of frames
          #ifndef ENABLE_LOAD_BAL
          if(i<MAX_ITERATIONS)
                 residual_error[i] = get_residual_error(st, st_copy) / NN;
          #endif

          // Output reference output/loadbalanced output
          speex_preprocess_run(den, e_buf);
          fwrite(e_buf, sizeof(short), NN*NUM_CHANNELS, e_fd[0]);
          // Output OpenCL output to separate file
          #ifndef ENABLE_LOAD_BAL
          speex_preprocess_run(den_copy, e_buf_copy);
          fwrite(e_buf_copy, sizeof(short), NN*NUM_CHANNELS, e_fd[1]);
```

```
         #endif

         i++;
         #ifdef ENABLE_LOAD_BAL
         gettimeofday(&curr_time, NULL);
         // Every second, check CPU usage
         if(((long long) (curr_time.tv_sec - last_cpu_check.tv_sec) * 1000000 + (curr_time.tv_usec - last_cpu_check.tv_usec)) > 1000000.0)
         {
            last_cpu_usage = get_system_CPU_load();
            last_cpu_check = curr_time;

            if(last_cpu_usage > 50)
            {
                    if(exec_dev == CPU_REFERENCE)
                    {
                        printf("Switching to OpenCL exeuction (i=%d, cpu=%d).\n", i, last_cpu_usage);
                        exec_dev = OPENCL;
                        copy_echo_state(st, st_copy);
                        speex_echo_opencl_sync_to_GPU(st);
                    }
            }
            else
            {
                    if(exec_dev == OPENCL)
                    {
                        printf("Switching to reference CPU exeuction (i=%d, cpu=%d).\n", i, last_cpu_usage);
                        exec_dev = CPU_REFERENCE;
                        printf("Copying echo state\n");
                        copy_echo_state(st_copy, st);
                        printf("Syncing buffers\n");
                        speex_echo_opencl_sync_to_CPU(st);
                    }
            }

            printf("CPU use: %d\n", last_cpu_usage);
         }
         #endif
      }
      printf("Ended loop iteration echo.\n");
   }

   // Write out residual errors to file
   errors_file_fd = fopen("errors.txt", "wb");
   if(errors_file_fd != NULL)
   {
      for(i=0;i<MAX_ITERATIONS;i++)
          fprintf(errors_file_fd, "%f ", residual_error[i]);
   }
   fclose(errors_file_fd);

   speex_echo_opencl_print_execution_time_stats();
   speex_echo_opencl_execution_time_cleanup();

   printf("Cleaning up original state.\n");
   speex_echo_state_destroy(st);
   printf("Cleaning up copy state.\n");
   speex_echo_state_destroy(st_copy);
   printf("Cleaning up OpenCL.\n");
   speex_echo_opencl_cleanup_mem();

   speex_preprocess_state_destroy(den);
   speex_preprocess_state_destroy(den_copy);
   for(i=0;i<NUM_SPEAKERS;i++)
   {
      fclose(echo_fd[i]);
```

```
    }
    for(i=0;i<NUM_CHANNELS;i++)
    {
       fclose(ref_fd[i]);
    }
    for(i=0;i<2;i++)
    {
       fclose(e_fd[i]);
    }

    speex_echo_opencl_cleanup_dev();

    printf("Cleaning up copy arrays.\n");
    free(residual_error);
    free(ref_buf_copy);
    free(echo_buf_copy);
    free(e_buf_copy);
    free(echo_buf);
    free(echo_buf_interleaved);
    free(ref_buf);
    free(ref_buf_interleaved);
    free(e_buf);
    printf("Return, exiting.\n");
    return 0;
}

/**
 * Linux-specific function for retrieving system CPU load
 */
int get_system_CPU_load()
{
        int diff_idle;
        int diff_total;
        int diff_usage;
        int total;
        int idle;

        int stat_user = 0;
        int stat_nice = 0;
        int stat_system = 0;
        int stat_idle = 0;
        int stat_iowait = 0;
        int stat_irq = 0;
        int stat_softirq = 0;

        // Very simple/naive cpu load monitor for Linux systems
        // Source: http://colby.id.au/node/39
        if((fp = fopen("/proc/stat", "r")) != NULL) {

                fgets(cpustring, 80, fp);
                //printf("/proc/stat:\n%s", cpustring);
                sscanf(cpustring, "cpu  %d %d %d %d %d %d %d", &stat_user, &
                     stat_nice, &stat_system, &stat_idle, &stat_iowait, &
                     stat_irq, &stat_softirq);

                total = stat_user + stat_nice + stat_system + stat_idle +
                     stat_iowait + stat_irq + stat_softirq;

                idle = stat_idle;

                diff_idle = idle - prev_idle;
                diff_total = total - prev_total;
                if(diff_total > 0)
                {
                        //printf("Calc: (1000*(%d-%d)/%d + 5) / 10\n",
                             diff_total, diff_idle, diff_total);
                        diff_usage = (1000*(diff_total-diff_idle)/diff_total +
                             5) / 10;
```

```c
                }

                prev_total = total;
                prev_idle = idle;

                fclose(fp);

                return diff_usage;
        }
        else
                return -1;

}

/**
 * Utility-function to interleave multichannel input from files
 * _in buffers contain concecutive frames from each of the channels:
 *      1. frame chan 1, 1. frame chan 2, 2. frame chan 1, 2. frame chan 2 etc
    ...
 * _out buffers contains the _in buffers interleaved:
 *      1. sample chan 1, 1. sample chan 2, 2.sample chan 1, 2.sample chan 2 etc
    ...
 */
void interleave_input(short *echo_in, short *ref_in, short *echo_out, short *
    ref_out)
{
        int i, speak, chan;
        for(speak=0;speak<NUM_SPEAKERS;speak++)
        {
                for(i=0;i<NN;i++)
                {
                        echo_out[i*NUM_SPEAKERS + speak] = echo_in[speak*NN + i
                            ];
                }
        }
        for(chan=0;chan<NUM_CHANNELS;chan++)
        {
                for(i=0;i<NN;i++)
                {
                        ref_out[i*NUM_CHANNELS + chan] = ref_in[chan*NN + i];
                }
        }
}
```

Listing B.1: Main function.

B.2 OpenCL Implementation (Device Code/Kernel Functions)

The kernel functions that are actually run on the device is listed below. All functions are annotated with the local and global work size that are used to execute them.

```c
#define WORD2INT(x) ((x) < -32767.5f ? (int)-32768 : ((x) > 32766.5f ? (int)
    32767 : floor(0.5f+(x))))
#pragma OPENCL EXTENSION cl_khr_byte_addressable_store : enable

/**
```

```
 * Applies pre-emphasis to the input data (after it has been converted to float
     on the host), shifts the main time domain buffer (the echo tail) to make
     room for new data (far end signal) and inserts the far end signal after
     applying pre-emphasis.
 * Globalsize: N,M,K
 * Localsize: 64,1,1
 * If device runs out of local memory, global memory temporary buffers are used
     in place of tmpMemX and tmpMemD here (remember to adapt to global indices)
 */
__kernel void clMDF_prepare_shift_preemph_freq_data(__global float *input,
     __global float *x, __global float *X, __global float *tempX, __global short
      *far_end, __global float *memD, __global float *memX, float preemph,
     unsigned int frame_size, unsigned int window_size, unsigned int
     num_channels, unsigned int num_speakers, __local int* tmpMemX, __local
     float *tmpMemD)
{

        unsigned int i = get_global_id(0);      // Goes to window_size (N)
        unsigned int j = get_global_id(1);      // Goes to M
        unsigned int k = get_global_id(2);      // Goes to K
        unsigned int speak = get_global_id(2);  // Goes to K
        unsigned int lsize = get_local_size(0);
        unsigned int frame_ratio = frame_size / lsize;
        unsigned int tid = get_local_id(0);

        unsigned int chan;      // Usually just one channel, but loop if it not
             the case

        // If I is witin frame size, do prepare and shift_preemph
        if(i < lsize && j == 0 && k == 0)
        {

                // clMDF_prepare
                if(i == 0)
                {
                        for(chan = 0; chan < num_channels; chan++)
                        {
                                tmpMemD[chan*frame_size + frame_ratio*tid + 0] =
                                      memD[chan];
                                for(int s = 1;s<frame_ratio;s++)
                                        tmpMemD[lsize*chan*frame_ratio +
                                             frame_ratio*tid + s] = input[chan*
                                             frame_size + frame_ratio*i + s -
                                             1];
                        }
                }
                else
                {
                        for(chan = 0; chan < num_channels; chan++)
                        {
                                for(int s = 0;s<frame_ratio;s++)
                                        tmpMemD[lsize*chan*frame_ratio +
                                             frame_ratio*tid + s] = input[chan*
                                             frame_size + frame_ratio*i + s -
                                             1];
                        }
                }
        }

        barrier(CLK_LOCAL_MEM_FENCE);

        if(i < lsize && j == 0 && k == 0)
        {
                for(chan = 0; chan < num_channels; chan++)
                {
                        for(int s = 0;s<frame_ratio;s++)
                        {
                                input[chan * frame_size + frame_ratio*i + s] =
                                        input[chan*frame_size + frame_ratio*i + s]
                                        - (preemph * tmpMemD[lsize*chan*frame_ratio
                                         + frame_ratio*tid + s]);
                        }

                }

        }

        barrier(CLK_GLOBAL_MEM_FENCE);

        if(i < frame_size/2 && j == 0)
        {
                // clMDF_shift_preemph
                // Temporary value in place of memX (updated on each iteration
                    in sequential code)
                // Notice that we only store the values needed for update, not
                    the shift
                // This is done to save local mem. (only frame_size*K, instead
                    of window_size)
                if(i == 0)
                {
                        tmpMemX[speak*frame_size] = memX[speak];
                        tmpMemX[speak*frame_size + 1] = far_end[speak];
                        memX[speak] = far_end[(frame_size - 1)*num_speakers +
                            speak];
                }
                else
                {
                        tmpMemX[speak*frame_size + 2*i] = far_end[((i*2-1) *
                            num_speakers) + speak];
                        tmpMemX[speak*frame_size + 2*i + 1] = far_end[((i*2) *
                            num_speakers) + speak];
                }
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        if(i < frame_size/2 && j == 0)
        {
                // Shift data from the last frame in the time domain data
                x[speak*window_size + i*2] = x[speak*window_size + 2*i +
                    frame_size];
                x[speak*window_size + i*2+1] = x[speak*window_size + 2*i + 1 +
                    frame_size];

                // Insert new data in the new frame and apply preemphasis
                x[speak*window_size + i*2 + frame_size] = far_end[(2*i)*
                    num_speakers + speak] - (preemph * tmpMemX[speak*frame_size
                     + i*2]);
                x[speak*window_size + i*2 + 1 + frame_size] = far_end[(2*i + 1)*
                    num_speakers + speak] - (preemph * tmpMemX[speak*frame_size
                     + i*2 + 1]);
        }

        // clMDF_shift_freq_data
        for(speak = 0; speak < num_speakers; speak++) {
                X[(j+1)*window_size*num_speakers+speak*window_size+i] = tempX[j*
                    window_size*num_speakers+speak*window_size+i];
        }
}

/**
 * Generic inner product function, len is the size of the buffers
 * Globalsize: len/2
 * Localsize: len/2
 * Based on reduction kernel from the NVIDIA CUDA SDK (reduce2 kernel)
 * Optional (naive) support for batch runs, batch_size is the number of elements
      in the batch, batch_pitch is the distance between elements in the batch (
     must be the same for x and y).
```

```
*/
__kernel void clMDF_inner_product_reduce(__global float *x, unsigned int
    x_offset, __global float *y, unsigned int y_offset, __global float *g_odata
    , unsigned int n, unsigned int batch_size, unsigned int batch_pitch,
    __local float* sdata)
{
        // load shared mem
        unsigned int tid = get_local_id(0);
        unsigned int i = get_global_id(0);
        unsigned int xi = i*2 + x_offset;
        unsigned int yi = i*2 + y_offset;

        if(i < n) {
                sdata[tid] = 0.0f;

                for(int k=0;k<batch_size;k++)
                {
                        sdata[tid] += x[k*batch_pitch + xi] * y[k*batch_pitch +
                            yi];
                        sdata[tid] += x[k*batch_pitch + xi + 1] * y[k*
                            batch_pitch + yi + 1];
                }
        }
        else
                sdata[tid] = 0.0f;

        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

        // do reduction in shared mem
        for(unsigned int s=get_local_size(0)/2; s>0; s>>=1)
        {
                if (tid < s)
                {
                        sdata[tid] += sdata[tid + s];
                }
                barrier(CLK_LOCAL_MEM_FENCE);
        }

        // write result for this block to global mem
        if (tid == 0)
                g_odata[get_group_id(0)] = sdata[0];
}

/**
 * This kernel combines an inner product with two computations of the power
     spectrum for the echo and filter response.
 * Globalsize: len/2
 * Localsize: 64
 * Combined kernel with 3*power_spectrum_accum and 1*inner_product_reduce
 */
__kernel void clMDF_power_spectrum_inner_product(
        __global float *x, __global float *E, __global float *Y, __global float
            *X,
        __global float *Rf, __global float *Yf, __global float *Xf,
        // Offsets to power pectrum accum
        unsigned int E_offset, unsigned int Y_offset, unsigned int X_offset,
        // Offsets to inner product
        unsigned int x1_offset, unsigned int x2_offset,
        unsigned int C, unsigned int K, unsigned int window_size,
        unsigned int N_power_spectrum,
        unsigned int n_inner_product,
        __global float *temp_odata,
        __local float* ldata)
{

        unsigned int j = get_global_id(0);
        unsigned int i = j * 2 - 1;

        unsigned int tid = get_local_id(0);
        unsigned int xi = j*2 + x1_offset;
        unsigned int yi = j*2 + x2_offset;

        // Optimize for regular case (mono mic and speaker)
        if(C == 0 && K == 0)
        {
                if(j == 0)
                {
                        Rf[j] += E[0] * E[0];
                        Yf[j] += Y[0] * Y[0];
                        Xf[j] += X[0] * X[0];
                }
                else if(i < (N_power_spectrum-1))
                {
                        Rf[j] += E[i] * E[i] + E[i+1] * E[i+1];
                        Yf[j] += Y[i] * Y[i] + Y[i+1] * Y[i+1];
                        Xf[j] += X[i] * X[i] + X[i+1] * X[i+1];
                }
                if(j == (get_global_size(0) - 1))
                {
                        i += 2;
                        Rf[j+1] += E[i] * E[i];
                        Yf[j+1] += Y[i] * Y[i];
                        Xf[j+1] += X[i] * X[i];
                }
        }
        // In case of multi-channel
        else
        {
                if(j == 0)
                {
                        unsigned int chan_offset, speak_offset;
                        for(unsigned int chan = 0;chan<C;chan++)
                        {
                                chan_offset = chan*window_size;
                                Rf[j] += E[0 + chan_offset] * E[0 + chan_offset
                                    ];
                                Yf[j] += Y[0 + chan_offset] * Y[0 + chan_offset
                                    ];
                        }
                        for(unsigned int speak = 0;speak<K;speak++)
                        {
                                speak_offset = speak*window_size;
                                Xf[j] += X[0 + speak_offset] * X[0 +
                                    speak_offset];
                        }
                }
                else if(i < (N_power_spectrum-1))
                {
                        unsigned int chan_offset, speak_offset;
                        for(unsigned int chan = 0;chan<C;chan++)
                        {
                                chan_offset = chan*window_size;
                                Rf[j] += E[i + chan_offset] * E[i + chan_offset]
                                     + E[i+1 + chan_offset] * E[i+1 +
                                    chan_offset];
                                Yf[j] += Y[i + chan_offset] * Y[i + chan_offset]
                                     + Y[i+1 + chan_offset] * Y[i+1 +
                                    chan_offset];
                        }
                        for(unsigned int speak = 0;speak<K;speak++)
                        {
                                speak_offset = speak*window_size;
                                Xf[j] += X[i + speak_offset] * X[i +
                                    speak_offset] + X[i+1 + speak_offset] * X[i
                                    +1 + speak_offset];
                        }
                }
                if(j == (get_global_size(0) - 1))
                {
                        i += 2;
                        unsigned int chan_offset, speak_offset;
                        for(unsigned int chan = 0;chan<C;chan++)
                        {
                                chan_offset = chan*window_size;
                                Rf[j+1] += E[i + chan_offset] * E[i + chan_offset];
                                Yf[j+1] += Y[i + chan_offset] * Y[i + chan_offset];
                        }
                        for(unsigned int speak = 0;speak<K;speak++)
                        {
                                speak_offset = speak*window_size;
                                Xf[j+1] += X[i + speak_offset] * X[i + speak_offset];
                        }
                }
        }

        // Again, optimize for 1 speaker
        if(K == 0)
        {
                if(j < (n_inner_product/2)) {
                        ldata[tid] = 0.0f;
                        ldata[tid] = fma(x[xi], x[yi], ldata[tid]);
                        ldata[tid] = fma(x[xi+1], x[yi+1], ldata[tid]);
                }
                else
                        ldata[tid] = 0.0f;

                barrier(CLK_LOCAL_MEM_FENCE);

                // do reduction in shared mem
                for(unsigned int s=get_local_size(0)/2; s>0; s>>=1)
                {
                        if (tid < s)
                        {
                                ldata[tid] += ldata[tid + s];
                                //ldata[tid] = 1.0;
                        }
                        barrier(CLK_LOCAL_MEM_FENCE);
                }

                // write result for this block to global mem
                if (tid == 0)
                        temp_odata[get_group_id(0)] = ldata[0];
        }
        // Several speakers
        else
        {
                unsigned int x_speak_offset;
                for(unsigned int speak = 0;speak<K;speak++)
                {
                        x_speak_offset = speak*window_size;
                        if(j < (n_inner_product/2)) {
                                ldata[tid] = 0.0f;
                                ldata[tid] = fma(x[xi + x_speak_offset], x[yi + x_speak_offset], ldata[tid]);
                                ldata[tid] = fma(x[xi+1 + x_speak_offset], x[yi+1 + x_speak_offset], ldata[tid]);
                        }
                        else
                                ldata[tid] = 0.0f;

                        barrier(CLK_LOCAL_MEM_FENCE);

                        // do reduction in shared mem
                        for(unsigned int s=get_local_size(0)/2; s>0; s>>=1)
                        {
                                if (tid < s)
                                {
                                        ldata[tid] += ldata[tid + s];
                                        //ldata[tid] = 1.0;
                                }
                                barrier(CLK_LOCAL_MEM_FENCE);
                        }

                        barrier(CLK_LOCAL_MEM_FENCE);

                        // write result for this block to global mem
                        if (tid == 0)
                                temp_odata[get_group_id(0) + get_num_groups(0)*speak] = ldata[0];

                        barrier(CLK_LOCAL_MEM_FENCE);
                }
        }

}

/**
 * This is special case kernel that calculates cross-power spectras for a batch of frames added together,
 * in addition to setting three unrelated buffers to zero to avoid overhead.
 * Globalsize: N/2,C
 * Localsize: 32,1
 * lMem is assumed to be of size N
 */
__kernel void clMDF_spectral_mul_accum(__global float *X, __global float *Y, __global float *acc, __global float *Rf, __global float *Yf, __global float *Xf, unsigned int N, unsigned int num_speakers, unsigned int M)
{
        int i = get_global_id(0);
        unsigned int tid = get_local_id(0);
        unsigned int chan = get_global_id(1);
        unsigned int c;
        unsigned int M_org = M / num_speakers;

        // Temporary accum registers
        float tmp1 = 0.0f;
        float tmp2 = 0.0f;
        float tmp3 = 0.0f;

        if(i == 0)
        {
                for(c=0;c<M;c++)
                {
                        tmp1 += X[c*N + 0]*Y[chan*N*num_speakers*M_org + c*N + 0];
                        tmp2 += (X[c*N + 1]*Y[chan*N*num_speakers*M_org + c*N + 1] - X[c*N + 2]*Y[chan*N*num_speakers*M_org + c*N + 2]);
                }
        }
        else if(i < (get_global_size(0) - 1))
        {
                for(c=0;c<M;c++)
                {
                        tmp1 += (X[c*N + 2*i - 1]*Y[chan*N*num_speakers*M_org + c*N + 2*i - 1] - X[c*N + 2*i]*Y[chan*N*num_speakers*M_org + c*N + 2*i]);
                        tmp2 += (X[c*N + 2*i]*Y[chan*N*num_speakers*M_org + c*N + 2*i - 1] + X[c*N + 2*i - 1]*Y[chan*N*num_speakers
                                *M_org + c*N + 2*i]);
                }
        }
        else if(i == (get_global_size(0) - 1))
        {
                for(c=0;c<M;c++)
                {
                        tmp1 += (X[c*N + 2*i - 1]*Y[chan*N*num_speakers*M_org + c*N + 2*i - 1] - X[c*N + 2*i]*Y[chan*N*num_speakers*M_org + c*N + 2*i]);
                        tmp2 += (X[c*N + 2*i]*Y[chan*N*num_speakers*M_org + c*N + 2*i - 1] + X[c*N + 2*i - 1]*Y[chan*N*num_speakers*M_org + c*N + 2*i]);
                        tmp3 += X[c*N + (N-1)]*Y[chan*N*num_speakers*M_org + c*N + (N-1)];
                }
        }

        // Write out to global memory
        if(i > 0 && i < (get_global_size(0) - 1))
        {
                acc[chan*N + 2*i - 1] = tmp1;
                acc[chan*N + 2*i] = tmp2;
        }
        else if(i == 0)
        {
                acc[chan*N] = tmp1;
        }
        else if(i == (get_global_size(0) - 1))
        {
                acc[chan*N + 2*i - 1] = tmp1;
                acc[chan*N + 2*i] = tmp2;
                acc[chan*N + 2*i + 1] = tmp3;
        }

        if(chan == 0)
        {
                Rf[i] = 0.0f;
                Yf[i] = 0.0f;
                Xf[i] = 0.0f;
                if(i == get_global_size(0) - 1)
                {
                        Rf[N/2] = 0.0f;
                        Yf[N/2] = 0.0f;
                        Xf[N/2] = 0.0f;
                }
        }
}

/**
 * Compute foreground filter
 * Globalsize: frame_size,C
 * Localsize: 32,1
 */
__kernel void clMDF_compute_foreground_filter(__global float *e, __global float *input, unsigned int window_size, unsigned int frame_size, unsigned int num_channels)
{
        if(get_global_id(1) < num_channels)
        {
                unsigned int i = get_global_id(0);
                unsigned int chan = get_global_id(1);

                e[chan*window_size + i] = input[chan*frame_size + i] - e[chan*window_size + i + frame_size];
        }
}

```

```
/**
 * Performs one of the calculations when resolving the new adaption rate of the
     filter.
 * Calculates the square value of an array of elements and adds them all
     together the a single value.
 * Globalsize: N,1,M
 * Localsize: 128,1,1
 */
__kernel void clMDF_adjust_prop_phase1_reduce(__global float *W, __global float
    *g_odata, unsigned int M, unsigned int window_size, unsigned int n,
    unsigned int P, __local float* sdata)
{
        unsigned int j = get_global_id(0);
        unsigned int i = get_global_id(2);
        unsigned int tid = get_local_id(0);

        sdata[tid] = 0.0f;
        // Loop through channels and add to local memory
        for(int p=0;p<P;p++)
        {
                sdata[tid] += W[p*window_size*M + i*window_size+j] * W[p*
                    window_size*M + i*window_size+j];
                barrier(CLK_LOCAL_MEM_FENCE);
        }

        // Do local memory reduction
        for(unsigned int s=get_local_size(0)/2; s>0; s>>=1)
        {
                if (tid < s)
                {
                        sdata[tid] += sdata[tid + s];
                }
                barrier(CLK_LOCAL_MEM_FENCE);
        }

        // Write result for this work-group to global mem
        if (tid == 0)
                g_odata[get_global_id(2)*get_num_groups(0) + get_group_id(0)] =
                    sdata[0];

}

/**
 * Compute weighted cross-power spectrum of a half-complex (packed) vector with
     conjugate
 * Globalsize: N/2,M,C*K
 * Localsize: 128,1,1
 */
__kernel void clMDF_weighted_spectral_mul_conj(__global float *w, __global float
     *p, __global float *X, __global float *Y, unsigned int Y_offset, __global
    float *prod, __global float *W, unsigned int window_size, unsigned int
    num_channels, unsigned int num_speakers)
{

        unsigned int i = get_global_id(0) * 2 - 1;
        unsigned int j = get_global_id(0);
        unsigned int u = get_global_id(1);
        unsigned int tid = get_global_id(0);

        int speak = get_global_id(2) / num_channels;
        int chan = get_global_id(2) - speak*num_channels;

        unsigned int X_offset = (u+1)*window_size*num_speakers+speak*window_size
            ;
        Y_offset = chan*window_size;
        unsigned int W_offset = chan * window_size * num_speakers *
            get_global_size(1) + u * window_size * num_speakers + speak *
            window_size;
	unsigned int p_offset = u;

	float temp, temp2;

	if(tid == 0)
	{
		temp = p[p_offset] * w[0];
		temp *= X[X_offset + 0] * Y[Y_offset + 0];
		W[W_offset] += temp;
		if(get_global_id(1) == 0)
			prod[0] = temp;
	}
	else if(tid <= get_global_size(0) - 1)
	{
		temp = temp2 = p[p_offset] * w[j];
		temp *= ((X[X_offset + i] * Y[Y_offset + i]) + X[X_offset + i +
		    1] * Y[Y_offset + i+1]);
		temp2 *= ((-X[X_offset + i + 1] * Y[Y_offset + i]) + X[X_offset
		    + i] * Y[Y_offset + i + 1]);
		W[W_offset + i] += temp;
		W[W_offset + i + 1] += temp2;

		if(get_global_id(1) == 0)
		{
			prod[i] = temp;
			prod[i+1] = temp2;
		}
	}
	if(tid == get_global_size(0) - 1)
	{
		i += 2;
		j += 1;
		temp = p[p_offset] * w[j];
		temp *= (X[X_offset + i] * Y[Y_offset + i]);
		W[W_offset + i] += temp;
		if(get_global_id(1) == 0)
			prod[i] = temp;
	}

}

/**
 * Simply sets a value in an array for each work item, offset by the given value
     .
 * Globalsize: Array size
 * Localsize: Determined by impl.
 */
__kernel void clMDF_set_array_to_float_value(__global float *input_array,
     unsigned int input_offset, float value)
{

	unsigned int i = get_global_id(0);

	input_array[input_offset + i] = value;

}

/**
 * Difference in response calculations, combined with inner product
 * Globalsize: frame_size,2
 * Localsize: 128,1
 * Offsets are special cases
 * temp_e is a temporary buffer assumed to be at least of the same size as e
 */
__kernel void clMDF_combined_response_diff_inner_product(__global float *e,
     __global float *temp_e, __global float *y, __global float *input,
							unsigned int C,
							unsigned int window_size,
                                                unsigned int frame_size,
                                                __global float *temp_odata,
                                                __global float *temp_e2,
                                                __local float *ldata)
{

        unsigned int i = get_global_id(0);
        unsigned int task = get_global_id(1);

        unsigned int tid = get_local_id(0);
        unsigned int xi;
        unsigned int yi;
        unsigned int n_inner_product = frame_size;

        barrier(CLK_GLOBAL_MEM_FENCE);
        if(task == 0)
        {
                for(unsigned int chan=0;chan<C;chan++)
                {
                        temp_e[chan*window_size + i] = e[chan*window_size+
                             frame_size + i] - y[chan*window_size+frame_size + i
                             ];
                        e[chan*window_size + i] = input[chan*frame_size + i] - y
                             [chan*window_size+frame_size + i];
                }

                // Dbf
                // mdf_opencl_inner_prod(ocl_e, chan*st->window_size, ocl_e,
                     chan*st->window_size, st->frame_size);
                // Runs on all threads, to make sure that they read/write to the
                      same position
                // in global memory as for the response diff.
                xi = i;
                yi = i;

                if(i < n_inner_product) {
                        ldata[tid] = 0.0f;
                        for(unsigned int chan = 0;chan<C;chan++)
                                ldata[tid] += temp_e[xi + chan*window_size] *
                                     temp_e[yi + chan*window_size];
                }
                else
                        ldata[tid] = 0.0f;

                barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

                // do reduction in shared mem (accuracy problems on GPU?)
                for(unsigned int s=get_local_size(0)/2; s>0; s>>=1)
                {
                        if (tid < s)
                        {
                                ldata[tid] += ldata[tid + s];
                        }
                        barrier(CLK_LOCAL_MEM_FENCE);
                }

                barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

                // write result for this block to global mem
                if (tid == 0)
                {
                        temp_odata[get_group_id(0)] = ldata[0];
                }
        }
        else
        {
                for(unsigned int chan=0;chan<C;chan++)
                {
                        temp_e2[chan*window_size + i] = input[chan*frame_size + i] - y[chan*window_size+frame_size + i];
                }
                // See
                // mdf_opencl_inner_prod(ocl_e, chan*st->window_size, ocl_e, chan*st->window_size, st->frame_size);
                // Runs on all threads, to make sure that they read/write to the same position
                // in global memory as for the response diff.
                xi = i;
                yi = i;

                if(i < n_inner_product) {
                        ldata[tid] = 0.0f;
                        for(unsigned int chan = 0;chan<C;chan++)
                                ldata[tid] += temp_e2[xi + chan*window_size] * temp_e2[yi + chan*window_size];
                }
                else
                        ldata[tid] = 0.0f;

                barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

                // do reduction in shared mem (accuracy problems on GPU?)
                for(unsigned int s=get_local_size(0)/2; s>0; s>>=1)
                {
                        if (tid < s)
                        {
                                ldata[tid] += ldata[tid + s];
                        }
                        barrier(CLK_LOCAL_MEM_FENCE);
                }

                barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

                // write result for this block to global mem
                if (tid == 0)
                {
                        temp_odata[get_group_id(0) + get_num_groups(0)] = ldata[0];
                }
        }

}

/**
 * Update foreground filter
 * Globalsize: frame_size, C
 * Localsize: Determined by impl.
 */
__kernel void clMDF_update_foreground_filter(__global float *e, __global float *window, __global float *y, unsigned int window_size, unsigned int frame_size)
{

        unsigned int i = get_global_id(0);
        unsigned int j = get_global_id(1);

        e[j*window_size+i+frame_size] = (window[i+frame_size] * e[j*window_size+i+frame_size]) + (window[i] * y[j*window_size+i+frame_size]);

}

/**
 * Update background filter
 * Globalsize: frame_size, C
 * Localsize: Determined by impl.
 */
```

```
__kernel void clMDF_update_background_filter(__global float *y, __global float *
    e, __global float *input, unsigned int window_size, unsigned int frame_size
    )
{

        unsigned int i = get_global_id(0);
        unsigned int j = get_global_id(1);

        y[j*window_size+i+frame_size] = e[j*window_size+i+frame_size];
        e[j*window_size+i] = input[j*frame_size+i] - y[j*window_size+i+
             frame_size];

}

/**
 * This is a combined kernel of three inner products in addition to calculating
     the error signal, it also checks for saturation in the microphone signal.
 * Globalsize: frame_size,C,4
 * Localsize: 128,1,1
 */
__kernel void clMDF_error_signal_combined(__global float *input, __global float
    *e, __global short *in, __global float *temp_results, __global float *
    temp_reductions, unsigned int window_size, unsigned int frame_size,
    unsigned int num_channels, __global int *saturated, __global float *y,
    __local float *sdata)
{
        unsigned int i = get_global_id(0);
        unsigned int j = get_global_id(1);
        unsigned int chan = get_global_id(1);
        unsigned int tid = get_local_id(0);
        unsigned int i_counter, chan_counter;   // Loop variables
        unsigned int input_set = get_global_id(2);

        barrier(CLK_LOCAL_MEM_FENCE);

        // This is an arbitrary test for saturation in the microphone signal
        if (in[i*num_channels+chan] <= -32000 || in[i*num_channels+chan] >=
             32000)
        {
                if (saturated[0] == 0)
                        saturated[0] = 1;
        }

        // Sey
        // clMDF_inner_product_reduce(ocl_e, chan*st->window_size+st->frame_size
             , ocl_y, chan*st->window_size+st->frame_size, g_odata, st->
             frame_size, sdata)
        if(input_set == 1)
        {
                temp_results[i*num_channels + chan] = input[chan*frame_size + i]
                     - e[chan*window_size + frame_size + i];

                e[chan*window_size + frame_size + i] = e[chan*window_size + i];
                e[chan*window_size + i] = 0;

                barrier(CLK_GLOBAL_MEM_FENCE);

                if(i < frame_size) {
                        sdata[tid] = 0.0f;
                        sdata[tid] += e[chan*window_size + frame_size + i] * y[
                             chan*window_size + frame_size + i];
                }
                else
                        sdata[tid] = 0.0f;

                barrier(CLK_LOCAL_MEM_FENCE);

                // do reduction in shared mem
                for(unsigned int s=get_local_size(0)/2; s>0; s>>=1)
                {
                        if (tid < s)
                        {
                                sdata[tid] += sdata[tid + s];
                        }
                        barrier(CLK_LOCAL_MEM_FENCE);
                }

                // write result for this block to global mem
                if (tid == 0)
                {
                        temp_reductions[get_group_id(0)*num_channels + chan] =
                            sdata[0];
                }

                barrier(CLK_GLOBAL_MEM_FENCE);
        }

        // Syy
        // clMDF_inner_product_reduce(ocl_y, chan*st->window_size+st->frame_size
            , ocl_y, chan*st->window_size+st->frame_size, g_odata, st->
            frame_size, sdata)
        else if(input_set == 2)
        {
                if(i < frame_size) {
                        sdata[tid] = 0.0f;
                        sdata[tid] += y[i + chan*window_size + frame_size] * y[i
                             + chan*window_size + frame_size];
                }
                else
                        sdata[tid] = 0;

                barrier(CLK_LOCAL_MEM_FENCE);

                // do reduction in shared mem
                for(unsigned int s=get_local_size(0)/2; s>0; s>>=1)
                {
                        if (tid < s)
                        {
                                sdata[tid] += sdata[tid + s];
                        }
                        barrier(CLK_LOCAL_MEM_FENCE);
                }

                // write result for this block to global mem
                if (tid == 0)
                {
                        temp_reductions[get_num_groups(0)*num_channels +
                            get_group_id(0)*num_channels + chan] = sdata[0];
                }
        }

        // Sdd
        // clMDF_inner_product_reduce(ocl_input, chan*st->frame_size, ocl_input,
             chan*st->frame_size, g_odata, st->frame_size, sdata)
        else if(input_set == 3)
        {
                if(i < frame_size) {
                        sdata[tid] = 0.0f;
                        sdata[tid] += input[i + chan*frame_size] * input[i +
                            chan*frame_size];
                }
                else
                        sdata[tid] = 0;

                barrier(CLK_LOCAL_MEM_FENCE);

                // do reduction in shared mem
                for(unsigned int s=get_local_size(0)/2; s>0; s>>=1)
                {
                        if (tid < s)
                        {
                                sdata[tid] += sdata[tid + s];
                        }
                        barrier(CLK_LOCAL_MEM_FENCE);
                }

                // write result for this block to global mem
                if (tid == 0)
                        temp_reductions[get_num_groups(0)*num_channels*2 +
                            get_group_id(0)*num_channels + chan] = sdata[0];
        }
}

/**
 * This kernel combines three tasks: Smoothing the far-end energy estimate over
     time, computing the filtered spectra and some cross-correlations.
 * Computing the filtered spectra and the two cross-correlation is done in the
     same operation as there are data-dependencies between the two.
 * Globalsize: frame_size
 * Localsize: 128
 */
__kernel void clMDF_smooth_far_end_compute_filtered_spectra(__global float *
    power, __global float *Xf, __global float *Rf, __global float *Eh, __global
     float *Yf, __global float *Yh,  float ss_1, float ss, float spec_average,
    unsigned int frame_size, __global float *temp_space, __global float *
    temp_eh_yh, __local float* sdata)
{
        unsigned int tid = get_global_id(0);
        unsigned int ltid = get_local_id(0);
        int j;
        float Eh_local, Yh_local;

        if(tid <= frame_size)
        {
                power[tid] = (ss_1 * power[tid]) + 1 + (ss * Xf[tid]);

                j = tid;
        }

        // Calculate values to shared memory and update global memory values
        if(tid <= frame_size)
        {
                sdata[ltid] = (Rf[tid] - Eh[tid]) * (Yf[tid] - Yh[tid]);
                sdata[get_local_size(0) + ltid] = (Yf[tid] - Yh[tid]) * (Yf[tid]
                     - Yh[tid]);
                Eh[tid] = (1-spec_average)*Eh[tid] + spec_average*Rf[tid];
                Yh[tid] = (1-spec_average)*Yh[tid] + spec_average*Yf[tid];
        }
        else
        {
                sdata[ltid] = 0.0f;
                sdata[get_local_size(0) + ltid] = 0.0f;
        }

        barrier(CLK_LOCAL_MEM_FENCE);

        // Do reduction in shared memory
        for(unsigned int s=get_local_size(0)/2; s>0; s>>=1)
        {
                if (ltid < s)
                {
                        sdata[ltid] += sdata[ltid + s];
                        sdata[get_local_size(0) + ltid] += sdata[get_local_size
                            (0) + ltid + s];
                }
                barrier(CLK_LOCAL_MEM_FENCE);
        }

        // write result for this block to global mem
        if (ltid == 0)
        {
                temp_space[2*get_group_id(0)] = sdata[0];       // Pey
                temp_space[2*get_group_id(0) + 1] = sdata[get_local_size(0)];
                     // Pyy
        }
}

/**
 * Some calculations done when computing the new learning rate for the filter.
 * Primarily calculations that depend on device data, the rest is done on the
     host
 * Globalsize: frame_size + 32
 * Localsize: Determined by impl.
 */
__kernel void clMDF_learning_rate_calc(__global float *Yf, __global float *Rf,
     __global float *power_1, __global float *power, float RER, unsigned int
     frame_size, float leak_estimate, float adapt_rate, int adapted)
{
        unsigned int i = get_global_id(0);

        if(i < frame_size + 1)
        {
                if(adapted == 1)
                {
                        float r, e;
                        // Compute frequency-domain adaptation mask
                        r = leak_estimate * Yf[i];
                        e = Rf[i] + 1;
                        if (r > 0.5f * e)
                                r = 0.5f * e;
                        r = (0.7f * r) + (0.3f * (float)(RER * e));
                        power_1[i] = r / (e * power[i] + 10.0f);
                }
                else
                {
                        power_1[i] = adapt_rate / (power[i] + 10.0f);
                }
        }
}

/**
 * A simple kernel storing the difference between the raw input and output in a
     buffer for use in the next iteration, but only if the filter has adapted.
 * Globalsize: frame_size
 * Localsize: Determined by impl.
 */
__kernel void clMDF_adapted_copy(__global float *last_y, __global short *in,
     __global int *out, unsigned int frame_size)
{
        unsigned int i = get_global_id(0);
        last_y[frame_size + i] = in[i] - out[i];
}

/**
 * Combines an inner product calculation and the first part of a "spectral
     multiply accumulate" operation. More details can be found for dedicated
     kernels on both.
 * Globalsize: N/2,C,2
 * Localsize: 128,1,1
 * Based on reduction kernel from the NVIDIA CUDA SDK (reduce2 kernel)
 */
__kernel void clMDF_inner_prod_power_spec_spectral_mul(__global float *Xf,
    __global float *data_time_domain, __global float *data_freq_domain,
    __global float *foreground, __global float *temp_reduction, __global float
    *acc, unsigned int N, unsigned int K, unsigned M, int frame_size, __local
    float* ldata)
{
        // mdf_opencl_power_spectrum_accum(data_freq_domain, 0, ocl_Xf, 0, st->
            window_size)
        if(get_global_id(2) == 0 && get_global_id(1) == 0)
        {
                unsigned int j = get_global_id(0);
                unsigned int i = j * 2 - 1;

                if(j == 0)
                {
                        for(int speak=0;speak<K;speak++)
                        {
                                Xf[j] += data_freq_domain[speak*N + 0] *
                                    data_freq_domain[speak*N + 0];
                        }
                }
                else if(i < (N-1))
                {
                        for(int speak=0;speak<K;speak++)
                        {
                                Xf[j] += data_freq_domain[speak*N + i] *
                                    data_freq_domain[speak*N + i] +
                                    data_freq_domain[speak*N + i+1] *
                                    data_freq_domain[speak*N + i+1];
                        }
                }
                if(j == (get_global_size(0) - 1))
                {
                        i += 2;
                        for(int speak=0;speak<K;speak++)
                        {
                                Xf[j+1] += data_freq_domain[speak*N + i] *
                                    data_freq_domain[speak*N + i];
                        }
                }
        }
        // mdf_opencl_spectral_mul_accum(data_freq_domain, ocl_foreground, 0,
            ocl_Y, 0, st->window_size, st->M*st->K)
        // Only first part, needs reduce kernel as well
        // X=data_freq_domain
        // Y=foreground
        else if(get_global_id(2) == 1)
        {
                int i = get_global_id(0);
                unsigned int tid = get_local_id(0);
                unsigned int chan = get_global_id(1);
                unsigned int c;
                unsigned int M_org = M / K;

                // Temporary accum registers
                float tmp1 = 0.0f;
                float tmp2 = 0.0f;
                float tmp3 = 0.0f;

                if(i == 0)
                {
                        for(c=0;c<M;c++)
                        {
                                tmp1 += data_freq_domain[c*N + 0]*foreground[
                                    chan*N*K*M_org + c*N + 0];
                                tmp2 += (data_freq_domain[c*N + 1]*foreground[
                                    chan*N*K*M_org + c*N + 1] -
                                    data_freq_domain[c*N + 2]*foreground[chan*N
                                    *K*M_org + c*N + 2]);
                        }
                }
                else if(i < (get_global_size(0) - 1))
                {
                        for(c=0;c<M;c++)
                        {
                                tmp1 += (data_freq_domain[c*N + 2*i - 1]*
                                    foreground[chan*N*K*M_org + c*N + 2*i - 1]
                                    - data_freq_domain[c*N + 2*i]*foreground[
                                    chan*N*K*M_org + c*N + 2*i]);
                                tmp2 += (data_freq_domain[c*N + 2*i]*foreground[
                                    chan*N*K*M_org + c*N + 2*i - 1] +
                                    data_freq_domain[c*N + 2*i - 1]*foreground[
                                    chan*N*K*M_org + c*N + 2*i]);
                        }
                }
                else if(i == (get_global_size(0) - 1))
                {
                        for(c=0;c<M;c++)
                        {
                                tmp1 += (data_freq_domain[c*N + 2*i - 1]*
                                    foreground[chan*N*K*M_org + c*N + 2*i - 1]
                                    - data_freq_domain[c*N + 2*i]*foreground[
                                    chan*N*K*M_org + c*N + 2*i]);
                                tmp2 += (data_freq_domain[c*N + 2*i]*foreground[
                                    chan*N*K*M_org + c*N + 2*i - 1] +
                                    data_freq_domain[c*N + 2*i - 1]*foreground[
                                    chan*N*K*M_org + c*N + 2*i]);
                                tmp3 += data_freq_domain[c*N + (N-1)]*foreground
                                    [chan*N*K*M_org + c*N + (N-1)];
                        }
                }

                // Write out to global memory
                if(i > 0 && i < (get_global_size(0) - 1))
                {
                        acc[chan*N + 2*i - 1] = tmp1;
                        acc[chan*N + 2*i] = tmp2;
                }
                else if(i == 0)
                {
                        acc[chan*N] = tmp1;
                }
                else if(i == (get_global_size(0) - 1))
                {
                        acc[chan*N + 2*i - 1] = tmp1;
                        acc[chan*N + 2*i] = tmp2;
                        acc[chan*N + 2*i + 1] = tmp3;
                }
        }
}
```

Listing B.2: MDF OpenCL kernel functions

B.3 OpenCL Implementation (FFT Device Code/Kernel Functions)

The kernel functions containing FFT functions that are actually run on the device is listed here below. The core FFT kernel is only listed in complete in the first kernel (clFFT_custom), but is marked by a comment in the preceeding kernels ("// FFT KERNEL FUNCTION NOT LISTED, see clFFT_custom"). Only the hardcoded kernels for frames with 512 samples is included, but the kernels for 128 samples are very similar. A major difference between the two, is that with frames of 512 samples, the local work size is 128, with frames of 128 samples it is 64 (because of the radices used). The global size is the local size multiplied by the batch size.

```
// This file contains FFT-related kernels:
// * Conversion between Speex format and Apple OpenCL FFT format
// * Hardcoded FFT kernels based on Apple OpenCL FFT (to avoid library overhead)
// * Various kernels where the FFT is combined with other operations
#pragma OPENCL EXTENSION cl_khr_byte_addressable_store : enable
#ifndef M_PI
#define M_PI 0x1.921fb54442d18p+1
#endif
float2 complexMul(float2 a,float2 B) { return (float2)(mad(-(a).y, (B).y, (a).x
     * (B).x), mad((a).y, (B).x, (a).x * (B).y));}
#define conj(a) ((float2)((a).x, -(a).y))
#define conjTransp(a) ((float2)(-(a).y, (a).x))

#define fftKernel2(a,dir) \
{ \
    float2 c = (a)[0];    \
    (a)[0] = c + (a)[1];  \
    (a)[1] = c - (a)[1];  \
}

#define fftKernel2S(d1,d2,dir) \
{ \
    float2 c = (d1);   \
    (d1) = c + (d2);   \
    (d2) = c - (d2);   \
}

#define fftKernel4(a,dir) \
{ \
    fftKernel2S((a)[0], (a)[2], dir); \
    fftKernel2S((a)[1], (a)[3], dir); \
    fftKernel2S((a)[0], (a)[1], dir); \
    (a)[3] = (float2)(dir)*(conjTransp((a)[3])); \
    fftKernel2S((a)[2], (a)[3], dir); \
    float2 c = (a)[1]; \
    (a)[1] = (a)[2]; \
    (a)[2] = c; \
}

#define fftKernel4s(a0,a1,a2,a3,dir) \
{ \
    fftKernel2S((a0), (a2), dir); \
    fftKernel2S((a1), (a3), dir); \
    fftKernel2S((a0), (a1), dir); \
    (a3) = (float2)(dir)*(conjTransp((a3))); \
    fftKernel2S((a2), (a3), dir); \
    float2 c = (a1); \
    (a1) = (a2); \
    (a2) = c; \
}

#define bitreverse8(a) \
{ \
    float2 c; \
    c = (a)[1]; \
    (a)[1] = (a)[4]; \
    (a)[4] = c; \
    c = (a)[3]; \
    (a)[3] = (a)[6]; \
    (a)[6] = c; \
}

#define fftKernel8(a,dir) \
{ \
        const float2 w1  = (float2)(0x1.6a09e6p-1f,  dir*0x1.6a09e6p-1f);  \
        const float2 w3  = (float2)(-0x1.6a09e6p-1f, dir*0x1.6a09e6p-1f);  \
        float2 c; \
        fftKernel2S((a)[0], (a)[4], dir); \
        fftKernel2S((a)[1], (a)[5], dir); \
        fftKernel2S((a)[2], (a)[6], dir); \
        fftKernel2S((a)[3], (a)[7], dir); \
        (a)[5] = complexMul(w1, (a)[5]); \
        (a)[6] = (float2)(dir)*(conjTransp((a)[6])); \
        (a)[7] = complexMul(w3, (a)[7]); \
        fftKernel2S((a)[0], (a)[2], dir); \
        fftKernel2S((a)[1], (a)[3], dir); \
        fftKernel2S((a)[4], (a)[6], dir); \
        fftKernel2S((a)[5], (a)[7], dir); \
        (a)[3] = (float2)(dir)*(conjTransp((a)[3])); \
        (a)[7] = (float2)(dir)*(conjTransp((a)[7])); \
        fftKernel2S((a)[0], (a)[1], dir); \
        fftKernel2S((a)[2], (a)[3], dir); \
        fftKernel2S((a)[4], (a)[5], dir); \
        fftKernel2S((a)[6], (a)[7], dir); \
        bitreverse8((a)); \
}

#define bitreverse4x4(a) \
{ \
        float2 c; \
        c = (a)[1];  (a)[1]  = (a)[4];  (a)[4]  = c; \
        c = (a)[2];  (a)[2]  = (a)[8];  (a)[8]  = c; \
        c = (a)[3];  (a)[3]  = (a)[12]; (a)[12] = c; \
        c = (a)[6];  (a)[6]  = (a)[9];  (a)[9]  = c; \
        c = (a)[7];  (a)[7]  = (a)[13]; (a)[13] = c; \
        c = (a)[11]; (a)[11] = (a)[14]; (a)[14] = c; \
}

#define fftKernel16(a,dir) \
{ \
    const float w0 = 0x1.d906bcp-1f; \
    const float w1 = 0x1.87de2ap-2f; \
    const float w2 = 0x1.6a09e6p-1f; \
    fftKernel4s((a)[0], (a)[4], (a)[8],  (a)[12], dir); \
    fftKernel4s((a)[1], (a)[5], (a)[9],  (a)[13], dir); \
    fftKernel4s((a)[2], (a)[6], (a)[10], (a)[14], dir); \
    fftKernel4s((a)[3], (a)[7], (a)[11], (a)[15], dir); \
    (a)[5]  = complexMul((a)[5], (float2)(w0, dir*w1)); \
    (a)[6]  = complexMul((a)[6], (float2)(w2, dir*w2)); \
    (a)[7]  = complexMul((a)[7], (float2)(w1, dir*w0)); \
    (a)[9]  = complexMul((a)[9], (float2)(w2, dir*w2)); \
    (a)[10] = (float2)(dir)*(conjTransp((a)[10])); \
    (a)[11] = complexMul((a)[11], (float2)(-w2, dir*w2)); \
    (a)[13] = complexMul((a)[13], (float2)(w1, dir*w0)); \
    (a)[14] = complexMul((a)[14], (float2)(-w2, dir*w2)); \
    (a)[15] = complexMul((a)[15], (float2)(-w0, dir*-w1)); \
    fftKernel4((a), dir); \
    fftKernel4((a) + 4, dir); \
    fftKernel4((a) + 8, dir); \
    fftKernel4((a) + 12, dir); \
    bitreverse4x4((a)); \
}

#define bitreverse32(a) \
{ \
    float2 c1, c2; \
    c1 = (a)[2];   (a)[2] = (a)[1];   c2 = (a)[4];   (a)[4] = c1;   c1 = (a)[8];
```

```
            (a)[8] = c2;    c2 = (a)[16];  (a)[16] = c1;   (a)[1] = c2; \
    c1 = (a)[6];   (a)[6] = (a)[3];   c2 = (a)[12];  (a)[12] = c1;  c1 = (a)[24];  (a)[24] = c2;   c2 = (a)[17];  (a)[17] = c1;   (a)[3] = c2; \
    c1 = (a)[10];  (a)[10] = (a)[5];  c2 = (a)[20];  (a)[20] = c1;  c1 = (a)[9];   (a)[9] = c2;    c2 = (a)[18];  (a)[18] = c1;   (a)[5] = c2; \
    c1 = (a)[14];  (a)[14] = (a)[7];  c2 = (a)[28];  (a)[28] = c1;  c1 = (a)[25];  (a)[25] = c2;   c2 = (a)[19];  (a)[19] = c1;   (a)[7] = c2; \
    c1 = (a)[22];  (a)[22] = (a)[11]; c2 = (a)[13];  (a)[13] = c1;  c1 = (a)[26];  (a)[26] = c2;   c2 = (a)[21];  (a)[21] = c1;   (a)[11] = c2; \
    c1 = (a)[30];  (a)[30] = (a)[15]; c2 = (a)[29];  (a)[29] = c1;  c1 = (a)[27];  (a)[27] = c2;   c2 = (a)[23];  (a)[23] = c1;   (a)[15] = c2; \
}

#define fftKernel32(a,dir) \
{ \
    fftKernel2S((a)[0],  (a)[16], dir); \
    fftKernel2S((a)[1],  (a)[17], dir); \
    fftKernel2S((a)[2],  (a)[18], dir); \
    fftKernel2S((a)[3],  (a)[19], dir); \
    fftKernel2S((a)[4],  (a)[20], dir); \
    fftKernel2S((a)[5],  (a)[21], dir); \
    fftKernel2S((a)[6],  (a)[22], dir); \
    fftKernel2S((a)[7],  (a)[23], dir); \
    fftKernel2S((a)[8],  (a)[24], dir); \
    fftKernel2S((a)[9],  (a)[25], dir); \
    fftKernel2S((a)[10], (a)[26], dir); \
    fftKernel2S((a)[11], (a)[27], dir); \
    fftKernel2S((a)[12], (a)[28], dir); \
    fftKernel2S((a)[13], (a)[29], dir); \
    fftKernel2S((a)[14], (a)[30], dir); \
    fftKernel2S((a)[15], (a)[31], dir); \
    (a)[17] = complexMul((a)[17], (float2)(0x1.f6297cp-1f, dir*0x1.8f8b84p-3f)); \
    (a)[18] = complexMul((a)[18], (float2)(0x1.d906bcp-1f, dir*0x1.87de2ap-2f)); \
    (a)[19] = complexMul((a)[19], (float2)(0x1.a9b662p-1f, dir*0x1.1c73b4p-1f)); \
    (a)[20] = complexMul((a)[20], (float2)(0x1.6a09e6p-1f, dir*0x1.6a09e6p-1f)); \
    (a)[21] = complexMul((a)[21], (float2)(0x1.1c73b4p-1f, dir*0x1.a9b662p-1f)); \
    (a)[22] = complexMul((a)[22], (float2)(0x1.87de2ap-2f, dir*0x1.d906bcp-1f)); \
    (a)[23] = complexMul((a)[23], (float2)(0x1.8f8b84p-3f, dir*0x1.f6297cp-1f)); \
    (a)[24] = complexMul((a)[24], (float2)(0x0p+0f, dir*0x1p+0f)); \
    (a)[25] = complexMul((a)[25], (float2)(-0x1.8f8b84p-3f, dir*0x1.f6297cp-1f)); \
    (a)[26] = complexMul((a)[26], (float2)(-0x1.87de2ap-2f, dir*0x1.d906bcp-1f)); \
    (a)[27] = complexMul((a)[27], (float2)(-0x1.1c73b4p-1f, dir*0x1.a9b662p-1f)); \
    (a)[28] = complexMul((a)[28], (float2)(-0x1.6a09e6p-1f, dir*0x1.6a09e6p-1f)); \
    (a)[29] = complexMul((a)[29], (float2)(-0x1.a9b662p-1f, dir*0x1.1c73b4p-1f)); \
    (a)[30] = complexMul((a)[30], (float2)(-0x1.d906bcp-1f, dir*0x1.87de2ap-2f)); \
    (a)[31] = complexMul((a)[31], (float2)(-0x1.f6297cp-1f, dir*0x1.8f8b84p-3f)); \
    fftKernel16((a), dir); \
    fftKernel16((a) + 16, dir); \
    bitreverse32((a)); \
}

__kernel void \
clFFT_1DTwistInterleaved(__global float2 *in, unsigned int startRow, unsigned int numCols, unsigned int N, unsigned int numRowsToProcess, int dir) \
{ \
   float2 a, w; \
   float ang; \
   unsigned int j; \
        unsigned int i = get_global_id(0); \
        unsigned int startIndex = i; \
         \
        if(i < numCols) \
        { \
            for(j = 0; j < numRowsToProcess; j++) \
            { \
                a = in[startIndex]; \
                ang = 2.0f * M_PI * dir * i * (startRow + j) / N; \
                w = (float2)(native_cos(ang), native_sin(ang)); \
                a = complexMul(a, w); \
                in[startIndex] = a; \
                startIndex += numCols; \
            } \
        }         \
} \

/**
 * Generic FFT/IFFT kernel
 * Kernel output from the Apple OpenCL FFT lib for 1D input of length 1024, with
      modified radices (8 - 8 - 4 - 4) for ATI compatability
 * Globalsize: 128*S (S is a general batch size parameter)
 * Localsize: 128
 */
__kernel void clFFT_custom(__global float *speex_in, unsigned int
     speex_in_offset, __global float2 *in, __global float2 *out, __global float
     *speex_out, unsigned int speex_out_offset, int dir, int S)
{
        __local float sMem[1088];
        int i, j, r, indexIn, indexOut, index, tid, bNum, xNum, k, l;
        int s, ii, jj, offset;
        float2 w;
        float ang, angf, ang1;
        __local float *lMemStore, *lMemLoad;
        float2 a[8];
        int lId = get_local_id(0);
        int groupId = get_group_id(0);
        ii = lId;
        jj = 0;

        offset =  mad24(groupId, 1024, ii);
        __global float2 *out_orig = out;        // Keep this pointer for
             finalize code
        int workItemOffset = lId * 8 + groupId * 1024;

        // Do preparation step to convert from Speex FFT format
        // There is one thread for each four positions in the FFT length

        // clFFT_Forward
        if(dir == -1)
        {
                for(int t=0;t<8;t++)
                {
                        in[workItemOffset + t].x = speex_in[workItemOffset + t +
                              speex_in_offset] * (1.0f / 1024.0f);
                        in[workItemOffset + t].y = 0.0f;
                }

                barrier(CLK_GLOBAL_MEM_FENCE);
                barrier(CLK_LOCAL_MEM_FENCE);
        }
        // clFFT_Inverse
        else if(dir == 1)
        {
```

```
        int groupOffset = groupId * 1024;
        if(lId == 0)
        {
            in[groupOffset + 0].x = speex_in[groupOffset + speex_in_offset + 0];
            in[groupOffset + 0].y = 0.0f;
            for(int t=1;t<4;t++)
            {
                in[groupOffset + t].x = speex_in[groupOffset + speex_in_offset + (2*t - 1)];
                in[groupOffset + t].y = speex_in[groupOffset + speex_in_offset + (2*t - 1) + 1];
            }
            in[groupOffset + 512].x = speex_in[groupOffset + speex_in_offset + 1024 - 1];
            in[groupOffset + 512].y = 0.0f;
        }
        else
        {
            for(int t=0;t<4;t++)
            {
                in[groupOffset + 4*lId + t].x = speex_in[groupOffset + speex_in_offset + lId*8 - 1 + 2*t];
                in[groupOffset + 4*lId + t].y = speex_in[groupOffset + speex_in_offset + lId*8 + 2*t];
                //in[groupOffset + 8*lId + t].x = 9000.0f;
                //in[groupOffset + 8*lId + t].y = 9000.0f;
            }
        }

        barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);

        // Generate negative frequencies for symmetric input
        // Source: http://www.dspguide.com/ch12/1.htm
        // For the remaining n/2 threads that needs to mirror
        for(int t=0;t<4;t++)
        {
            in[groupOffset + 512 + lId*4 + t].x = in[groupOffset + 1024 - (512 + lId*4 + t)].x;
            in[groupOffset + 512 + lId*4 + t].y = -in[groupOffset + 1024 - (512 + lId*4 + t)].y;
        }
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    in += offset;
    out += offset;
    a[0] = in[0];
    a[1] = in[128];
    a[2] = in[256];
    a[3] = in[384];
    a[4] = in[512];
    a[5] = in[640];
    a[6] = in[768];
    a[7] = in[896];
    fftKernel8(a+0, dir);
    angf = (float) ii;
    ang = dir * ( 2.0f * M_PI * 1.0f / 1024.0f ) * angf;
    w = (float2)(native_cos(ang), native_sin(ang));
    a[1] = complexMul(a[1], w);
    ang = dir * ( 2.0f * M_PI * 2.0f / 1024.0f ) * angf;
    w = (float2)(native_cos(ang), native_sin(ang));
    a[2] = complexMul(a[2], w);
    ang = dir * ( 2.0f * M_PI * 3.0f / 1024.0f ) * angf;
    w = (float2)(native_cos(ang), native_sin(ang));
```

```
        a[3] = complexMul(a[3], w);
        ang = dir * ( 2.0f * M_PI * 4.0f / 1024.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[4] = complexMul(a[4], w);
        ang = dir * ( 2.0f * M_PI * 5.0f / 1024.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[5] = complexMul(a[5], w);
        ang = dir * ( 2.0f * M_PI * 6.0f / 1024.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[6] = complexMul(a[6], w);
        ang = dir * ( 2.0f * M_PI * 7.0f / 1024.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[7] = complexMul(a[7], w);
        lMemStore = sMem + ii;
        j = ii & 7;
        i = ii >> 3;
        lMemLoad = sMem + mad24(j, 130, i);
        lMemStore[0] = a[0].x;
        lMemStore[130] = a[1].x;
        lMemStore[260] = a[2].x;
        lMemStore[390] = a[3].x;
        lMemStore[520] = a[4].x;
        lMemStore[650] = a[5].x;
        lMemStore[780] = a[6].x;
        lMemStore[910] = a[7].x;
        barrier(CLK_LOCAL_MEM_FENCE);
        a[0].x = lMemLoad[0];
        a[1].x = lMemLoad[16];
        a[2].x = lMemLoad[32];
        a[3].x = lMemLoad[48];
        a[4].x = lMemLoad[64];
        a[5].x = lMemLoad[80];
        a[6].x = lMemLoad[96];
        a[7].x = lMemLoad[112];
        barrier(CLK_LOCAL_MEM_FENCE);
        lMemStore[0] = a[0].y;
        lMemStore[130] = a[1].y;
        lMemStore[260] = a[2].y;
        lMemStore[390] = a[3].y;
        lMemStore[520] = a[4].y;
        lMemStore[650] = a[5].y;
        lMemStore[780] = a[6].y;
        lMemStore[910] = a[7].y;
        barrier(CLK_LOCAL_MEM_FENCE);
        a[0].y = lMemLoad[0];
        a[1].y = lMemLoad[16];
        a[2].y = lMemLoad[32];
        a[3].y = lMemLoad[48];
        a[4].y = lMemLoad[64];
        a[5].y = lMemLoad[80];
        a[6].y = lMemLoad[96];
        a[7].y = lMemLoad[112];
        barrier(CLK_LOCAL_MEM_FENCE);
        fftKernel8(a+0, dir);
        angf = (float) (ii >> 3);
        ang = dir * ( 2.0f * M_PI * 1.0f / 128.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[1] = complexMul(a[1], w);
        ang = dir * ( 2.0f * M_PI * 2.0f / 128.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[2] = complexMul(a[2], w);
        ang = dir * ( 2.0f * M_PI * 3.0f / 128.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[3] = complexMul(a[3], w);
        ang = dir * ( 2.0f * M_PI * 4.0f / 128.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[4] = complexMul(a[4], w);
        ang = dir * ( 2.0f * M_PI * 5.0f / 128.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[5] = complexMul(a[5], w);
        ang = dir * ( 2.0f * M_PI * 6.0f / 128.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[6] = complexMul(a[6], w);
        ang = dir * ( 2.0f * M_PI * 7.0f / 128.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[7] = complexMul(a[7], w);
        lMemStore = sMem + ii;
        j = (ii & 63) >> 3;
        i = mad24(ii >> 6, 8, ii & 7);
        lMemLoad = sMem + mad24(j, 136, i);
        lMemStore[0] = a[0].x;
        lMemStore[136] = a[1].x;
        lMemStore[272] = a[2].x;
        lMemStore[408] = a[3].x;
        lMemStore[544] = a[4].x;
        lMemStore[680] = a[5].x;
        lMemStore[816] = a[6].x;
        lMemStore[952] = a[7].x;
        barrier(CLK_LOCAL_MEM_FENCE);
        a[0].x = lMemLoad[0];
        a[1].x = lMemLoad[32];
        a[2].x = lMemLoad[64];
        a[3].x = lMemLoad[96];
        a[4].x = lMemLoad[16];
        a[5].x = lMemLoad[48];
        a[6].x = lMemLoad[80];
        a[7].x = lMemLoad[112];
        barrier(CLK_LOCAL_MEM_FENCE);
        lMemStore[0] = a[0].y;
        lMemStore[136] = a[1].y;
        lMemStore[272] = a[2].y;
        lMemStore[408] = a[3].y;
        lMemStore[544] = a[4].y;
        lMemStore[680] = a[5].y;
        lMemStore[816] = a[6].y;
        lMemStore[952] = a[7].y;
        barrier(CLK_LOCAL_MEM_FENCE);
        a[0].y = lMemLoad[0];
        a[1].y = lMemLoad[32];
        a[2].y = lMemLoad[64];
        a[3].y = lMemLoad[96];
        a[4].y = lMemLoad[16];
        a[5].y = lMemLoad[48];
        a[6].y = lMemLoad[80];
        a[7].y = lMemLoad[112];
        barrier(CLK_LOCAL_MEM_FENCE);
        fftKernel4(a+0, dir);
        fftKernel4(a+4, dir);
        angf = (float) (ii >> 6);
        ang = dir * ( 2.0f * M_PI * 1.0f / 16.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[1] = complexMul(a[1], w);
        ang = dir * ( 2.0f * M_PI * 2.0f / 16.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[2] = complexMul(a[2], w);
        ang = dir * ( 2.0f * M_PI * 3.0f / 16.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[3] = complexMul(a[3], w);
        angf = (float) ((128 + ii) >>6);
        ang = dir * ( 2.0f * M_PI * 1.0f / 16.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[5] = complexMul(a[5], w);
        ang = dir * ( 2.0f * M_PI * 2.0f / 16.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[6] = complexMul(a[6], w);
        ang = dir * ( 2.0f * M_PI * 3.0f / 16.0f ) * angf;
        w = (float2)(native_cos(ang), native_sin(ang));
        a[7] = complexMul(a[7], w);
        lMemStore = sMem + ii;
        j = ii >> 6;
        i = ii & 63;
        lMemLoad = sMem + mad24(j, 256, i);
        lMemStore[0] = a[0].x;
        lMemStore[256] = a[1].x;
        lMemStore[512] = a[2].x;
        lMemStore[768] = a[3].x;
        lMemStore[128] = a[4].x;
        lMemStore[384] = a[5].x;
        lMemStore[640] = a[6].x;
        lMemStore[896] = a[7].x;
        barrier(CLK_LOCAL_MEM_FENCE);
        a[0].x = lMemLoad[0];
        a[1].x = lMemLoad[64];
        a[2].x = lMemLoad[128];
        a[3].x = lMemLoad[192];
        a[4].x = lMemLoad[512];
        a[5].x = lMemLoad[576];
        a[6].x = lMemLoad[640];
        a[7].x = lMemLoad[704];
        barrier(CLK_LOCAL_MEM_FENCE);
        lMemStore[0] = a[0].y;
        lMemStore[256] = a[1].y;
        lMemStore[512] = a[2].y;
        lMemStore[768] = a[3].y;
        lMemStore[128] = a[4].y;
        lMemStore[384] = a[5].y;
        lMemStore[640] = a[6].y;
        lMemStore[896] = a[7].y;
        barrier(CLK_LOCAL_MEM_FENCE);
        a[0].y = lMemLoad[0];
        a[1].y = lMemLoad[64];
        a[2].y = lMemLoad[128];
        a[3].y = lMemLoad[192];
        a[4].y = lMemLoad[512];
        a[5].y = lMemLoad[576];
        a[6].y = lMemLoad[640];
        a[7].y = lMemLoad[704];
        barrier(CLK_LOCAL_MEM_FENCE);
        fftKernel4(a+0, dir);
        fftKernel4(a+4, dir);
        out[0] = a[0];
        out[128] = a[4];
        out[256] = a[1];
        out[384] = a[5];
        out[512] = a[2];
        out[640] = a[6];
        out[768] = a[3];
        out[896] = a[7];

        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

        // Do finalization step to convert to Speex FFT format
        // There are one thread for each four positions in the FFT length

        // clFFT_Forward
        if(dir == -1)
        {
                int workItemOffset2 = lId * 4 + groupId*1024;

                // The first thread for each FFT length
                if(lId == 0)
                {
                        speex_out[speex_out_offset + groupId*1024 + 0] =
                            out_orig[workItemOffset2 + 0].x;
                        speex_out[speex_out_offset + groupId*1024 + 1] = out_orig[workItemOffset2 + 1].x;
                        for(int t=1;t<4;t++)
                        {
                                speex_out[speex_out_offset + groupId*1024 + 2*t] = out_orig[workItemOffset2 + t].y;
                                speex_out[speex_out_offset + groupId*1024 + 2*t + 1] = out_orig[workItemOffset2 + t + 1].x;
                        }
                }
                else
                {
                        for(int t=0;t<4;t++)
                        {
                                speex_out[speex_out_offset + workItemOffset + 2*t] = out_orig[workItemOffset2 + t].y;
                                speex_out[speex_out_offset + workItemOffset + 2*t + 1] = out_orig[workItemOffset2 + t + 1].x;
                        }
                }
        }
        // clFFT_Inverse
        else if(dir == 1)
        {
                for(int t=0;t<8;t++)
                {
                        speex_out[speex_out_offset + workItemOffset + t] = out_orig[workItemOffset + t].x;
                }
        }
        // End of finalization step
}

/**
 * Specialized kernel for updating the filter weights in batches
 * Globalsize: 128*M,C,K
 * Localsize: 128,1,1
 */
__kernel void clFFT_update_weights(__global float *W, __global float2 *in, __global float2 *out, __global float *wtmp, int S)
{
        __local float sMem[1088];
        int i, j, r, indexIn, indexOut, index, tid, bNum, xNum, k, l;
        int s, ii, jj, offset;
        int chan = get_global_id(1);
        int speak = get_global_id(2);
        int num_speakers = get_global_size(2);
        int num_channels = get_global_size(1);
        float2 w;
        float ang, angf, ang1;
        __local float *lMemStore, *lMemLoad;
        float2 a[8];
        int lId = get_local_id( 0 );
        int groupId = get_group_id( 0 );
        const float m = 1.0f / 1024.0f;
        int groupOffset = groupId * 1024 * num_speakers;        // Group offset for float
        int groupOffset2 = groupId * 512 * num_speakers;        // Group offset for float2
        unsigned int cs_offset = chan*1024*num_speakers*S + speak*1024; // Channel/speaker offset
        unsigned int W_offset = cs_offset;
        ii = lId;
        jj = 0;

        // Do IFFT first
        int dir = 1;

        offset = cs_offset + groupId * 1024 * num_speakers + ii;
        __global float2 *out_orig = out;        // Keep this pointer for finalize code
        __global float2 *in_orig = in;

        int workItemOffset = lId * 8 + groupOffset + cs_offset;

        barrier(CLK_LOCAL_MEM_FENCE);

        // Do preparation step to convert from Speex FFT format
        // There is one thread for each four positions in the FFT length

        if(lId == 0)
        {
                in[groupOffset + cs_offset + 0].x = W[groupOffset + cs_offset + 0];
                in[groupOffset + cs_offset + 0].y = 0.0f;
                for(int t=1;t<4;t++)
                {
                        in[groupOffset + cs_offset + t].x = W[groupOffset + cs_offset + (2*t - 1)];
                        in[groupOffset + cs_offset + t].y = W[groupOffset + cs_offset + (2*t - 1) + 1];
                }
                in[groupOffset + cs_offset + 512].x = W[groupOffset + cs_offset + 1024 - 1];
                in[groupOffset + cs_offset + 512].y = 0.0f;
        }
        else
        {
                for(int t=0;t<4;t++)
                {
                        in[groupOffset + cs_offset + 4*lId + t].x = W[groupOffset + cs_offset + lId*8 - 1 + 2*t];
                        in[groupOffset + cs_offset + 4*lId + t].y = W[groupOffset + cs_offset + lId*8 + 2*t];
                }
        }

        barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);

        // Generate negative frequencies for symmetric input
        // Source: http://www.dspguide.com/ch12/1.htm
        // For the remaining n/2 threads that needs to mirror
        for(int t=0;t<4;t++)
        {
                in[groupOffset + cs_offset + 512 + lId*4 + t].x = in[groupOffset + cs_offset + 1024 - (512 + lId*4 + t)].x;
                in[groupOffset + cs_offset + 512 + lId*4 + t].y = -in[groupOffset + cs_offset + 1024 - (512 + lId*4 + t)].y;
        }

        // FFT KERNEL FUNCTION NOT LISTED, see clFFT_custom

        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

        // Do finalization step to convert to Speex FFT format
        // There are one thread for each four positions in the FFT length
        for(int t=0;t<8;t++)
        {
                wtmp[workItemOffset + t] = out_orig[workItemOffset + t].x;
                //wtmp[workItemOffset + t] = 1.0f;
        }

        // End of finalization step
        barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);

        // Set to zero
        wtmp[groupOffset + cs_offset + 512 + 4*lId] = 0.0f;
        wtmp[groupOffset + cs_offset + 512 + 4*lId + 1] = 0.0f;
        wtmp[groupOffset + cs_offset + 512 + 4*lId + 2] = 0.0f;
        wtmp[groupOffset + cs_offset + 512 + 4*lId + 3] = 0.0f;

        in = in_orig;
        out = out_orig;

        // Do forward FFT
        dir = -1;

        // Do preparation step to convert from Speex FFT format
        // There is one thread for each four positions in the FFT length

        // clFFT_Forward
        for(int t=0;t<8;t++)
        {
                in_orig[workItemOffset + t].x = wtmp[workItemOffset + t] * m;
                //in_orig[workItemOffset + t].x = 1.0f;
                in_orig[workItemOffset + t].y = 0.0f;
        }

        barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);
        // End of preparation step

        offset = cs_offset + groupId * 1024 * num_speakers + ii;
        // FFT KERNEL FUNCTION NOT LISTED, see clFFT_custom

        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

        // Do finalization step to convert to Speex FFT format
        // There are one thread for each four positions in the FFT length
        // Not adapted to multi-channel filter, only mono
        int workItemOffset2 = lId * 4 + groupOffset + cs_offset;

        // The first thread for each FFT length
        if(lId == 0)
        {
                W[cs_offset + groupOffset + 0] = out_orig[workItemOffset2 + 0].x
                    ;
                W[cs_offset + groupOffset + 1] = out_orig[workItemOffset2 + 1].x
                    ;
                for(int t=1;t<4;t++)
                {
                        W[cs_offset + groupOffset + 2*t] = out_orig[
                            workItemOffset2 + t].y;
                        W[cs_offset + groupOffset + 2*t + 1] = out_orig[
                            workItemOffset2 + t + 1].x;
                }
        }
        // note: cs_offset is included in workItemOffset
        else if(groupId < S)
        {
                for(int t=0;t<4;t++)
                {
                        W[workItemOffset + 2*t] = out_orig[workItemOffset2 + t].
                            y;
                        W[workItemOffset + 2*t + 1] = out_orig[workItemOffset2 +
                             t + 1].x;
                        //W[workItemOffset + 2*t] = 9000.0f;
                        //W[workItemOffset + 2*t + 1] = 8000.0f;
                        //W[workItemOffset + 2*t] = wtmp[groupOffset + cs_offset
                             + 8*lId + t];
                        //W[workItemOffset + 2*t + 1] = wtmp[groupOffset +
                            cs_offset + 8*lId + t];
                }
        }
        // End of finalization step
}

/**
 * Specialized kernel for updating the filter weights in batches
 * Split in two for ATI, as it crashes on a single long kernel
 * Globalsize: 128*M,C,K
 * Localsize: 128,1,1
 */
__kernel void clFFT_update_weights_ati_pt1(__global float *W, __global float2 *in, __global float2 *out, __global float *wtmp, int S)
{
        __local float sMem[1088];
        int i, j, r, indexIn, indexOut, index, tid, bNum, xNum, k, l;
        int s, ii, jj, offset;
        int chan = get_global_id(1);
        int speak = get_global_id(2);
        int num_speakers = get_global_size(2);
        int num_channels = get_global_size(1);
        float2 w;
        float ang, angf, ang1;
        __local float *lMemStore, *lMemLoad;
        float2 a[8];
        int lId = get_local_id( 0 );
        int groupId = get_group_id( 0 );
        const float m = 1.0f / 1024.0f;
        int groupOffset = groupId * 1024 * num_speakers;        // Group offset for float
        int groupOffset2 = groupId * 512 * num_speakers;        // Group offset for float2
        unsigned int cs_offset = chan*1024*num_speakers*S + speak*1024; // Channel/speaker offset
        unsigned int W_offset = cs_offset;
        ii = lId;
        jj = 0;

        // Do IFFT first
        int dir = 1;

        offset = cs_offset + groupId * 1024 * num_speakers + ii;
        __global float2 *out_orig = out;        // Keep this pointer for finalize code
        __global float2 *in_orig = in;

        int workItemOffset = lId * 8 + groupOffset + cs_offset;

        barrier(CLK_LOCAL_MEM_FENCE);

        // Do preparation step to convert from Speex FFT format
        // There is one thread for each four positions in the FFT length

        if(lId == 0)
        {
                in[groupOffset + cs_offset + 0].x = W[groupOffset + cs_offset + 0];
                in[groupOffset + cs_offset + 0].y = 0.0f;
                for(int t=1;t<4;t++)
                {
                        in[groupOffset + cs_offset + t].x = W[groupOffset + cs_offset + (2*t - 1)];
                        in[groupOffset + cs_offset + t].y = W[groupOffset + cs_offset + (2*t - 1) + 1];
                }
                in[groupOffset + cs_offset + 512].x = W[groupOffset + cs_offset + 1024 - 1];
                in[groupOffset + cs_offset + 512].y = 0.0f;
        }
        else
        {
                for(int t=0;t<4;t++)
                {
                        in[groupOffset + cs_offset + 4*lId + t].x = W[
                            groupOffset + cs_offset + lId*8 - 1 + 2*t];
                        in[groupOffset + cs_offset + 4*lId + t].y = W[
                            groupOffset + cs_offset + lId*8 + 2*t];
                }
        }

        barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);

        // Generate negative frequencies for symmetric input
        // Source: http://www.dspguide.com/ch12/1.htm
        // For the remaining n/2 threads that needs to mirror
        for(int t=0;t<4;t++)
        {
                in[groupOffset + cs_offset + 512 + lId*4 + t].x = in[groupOffset
                     + cs_offset + 1024 - (512 + lId*4 + t)].x;
                in[groupOffset + cs_offset + 512 + lId*4 + t].y = -in[
                    groupOffset + cs_offset + 1024 - (512 + lId*4 + t)].y;
        }

        barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);

        // FFT KERNEL FUNCTION NOT LISTED, see clFFT_custom

        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

        // Do finalization step to convert to Speex FFT format
        // There are one thread for each four positions in the FFT length
        for(int t=0;t<8;t++)
        {
                wtmp[workItemOffset + t] = out_orig[workItemOffset + t].x;
                //wtmp[workItemOffset + t] = 1.0f;
        }

        // End of finalization step
        barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);

        // Set to zero
        wtmp[groupOffset + cs_offset + 512 + 4*lId] = 0.0f;
        wtmp[groupOffset + cs_offset + 512 + 4*lId + 1] = 0.0f;
        wtmp[groupOffset + cs_offset + 512 + 4*lId + 2] = 0.0f;
        wtmp[groupOffset + cs_offset + 512 + 4*lId + 3] = 0.0f;
}

/**
 * Specialized kernel for updating the filter weights in batches
 * Split in two for ATI, as it crashes on a single long kernel
 * Globalsize: 128*M,C,K
 * Localsize: 128,1,1
 */
__kernel void clFFT_update_weights_ati_pt2(__global float *W, __global float2 *
     in, __global float2 *out, __global float *wtmp, int S)
{
        __local float sMem[1088];
        int i, j, r, indexIn, indexOut, index, tid, bNum, xNum, k, l;
        int s, ii, jj, offset;
        int chan = get_global_id(1);
        int speak = get_global_id(2);
        int num_speakers = get_global_size(2);
        int num_channels = get_global_size(1);
        float2 w;
        float ang, angf, ang1;
        __local float *lMemStore, *lMemLoad;
        float2 a[8];
        int lId = get_local_id( 0 );
        int groupId = get_group_id( 0 );
        const float m = 1.0f / 1024.0f;
        int groupOffset = groupId * 1024 * num_speakers;        // Group offset
             for float
        int groupOffset2 = groupId * 512 * num_speakers;        // Group offset
             for float2
        unsigned int cs_offset = chan*1024*num_speakers*S + speak*1024; //
             Channel/speaker offset
        unsigned int W_offset = cs_offset;
        ii = lId;
        jj = 0;

        // Do IFFT first
        int dir = 1;

        //offset =  mad24(groupId, 1024, ii);
        offset = cs_offset + groupId * 1024 * num_speakers + ii;
        __global float2 *out_orig = out;        // Keep this pointer for
             finalize code
        __global float2 *in_orig = in;

        int workItemOffset = lId * 8 + groupOffset + cs_offset;

        barrier(CLK_LOCAL_MEM_FENCE);

        // Do forward FFT
        dir = -1;

        // Do preparation step to convert from Speex FFT format
        // There is one thread for each four positions in the FFT length

        // clFFT_Forward
        for(int t=0;t<8;t++)
        {
                in_orig[workItemOffset + t].x = wtmp[workItemOffset + t] * m;
                in_orig[workItemOffset + t].y = 0.0f;
        }

        barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);

        // End of preparation step

        offset = cs_offset + groupId * 1024 * num_speakers + ii;
        // FFT KERNEL FUNCTION NOT LISTED, see clFFT_custom

        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

        // Do finalization step to convert to Speex FFT format
        // There are one thread for each four positions in the FFT length
        // Not adapted to multi-channel filter, only mono
        // groupOffset = groupId * 256 * num_speakers;
        int workItemOffset2 = lId * 4 + groupOffset + cs_offset;
        //workItemOffset = lId * 2 + groupOffset;
        //unsigned int W_offset = speak*1024 + chan*1024*num_speakers*S;

        // The first thread for each FFT length
        if(lId == 0)
        {
                W[cs_offset + groupOffset + 0] = out_orig[workItemOffset2 + 0].x
                     ;
                W[cs_offset + groupOffset + 1] = out_orig[workItemOffset2 + 1].x
                     ;
                for(int t=1;t<4;t++)
                {
                        W[cs_offset + groupOffset + 2*t] = out_orig[
                             workItemOffset2 + t].y;
                        W[cs_offset + groupOffset + 2*t + 1] = out_orig[
                             workItemOffset2 + t + 1].x;
                }
        }
        // note: cs_offset is included in workItemOffset
        else if(groupId < S)
        {
                for(int t=0;t<4;t++)
                {
                        W[workItemOffset + 2*t] = out_orig[workItemOffset2 + t].y;
                        W[workItemOffset + 2*t + 1] = out_orig[workItemOffset2 + t + 1].x;
                }
        }
        // End of finalization step
}

/**
 * Specialized kernel for converting the error to the frequency domain in one operation
 * Globalsize: 128*C,2
 * Localsize: 128,1
 */
__kernel void clFFT_convert_error_to_freq_domain(__global float *e, __global float *E, __global float *y, __global float *Y, __global float2 *in, __global float2 *in2, __global float2 *out, __global float2 *out2, int S)
{
        __local float sMem[1088];
        int i, j, r, indexIn, indexOut, index, tid, bNum, xNum, k, l;
        int s, ii, jj, offset;
        float2 w;
        float ang, angf, ang1;
        __local float *lMemStore, *lMemLoad;
        float2 a[8];
        int lId = get_local_id(0);
        int taskId = get_global_id(1);
        int groupId = get_group_id(0);
        const float m = 1.0f / 1024.0f;
        int groupOffset = groupId * 1024;       // Group offset for float
        int groupOffset2 = groupId * 512;       // Group offset for float2
        ii = lId;
        jj = 0;

        offset =  mad24(groupId, 1024, ii);
        __global float2 *out_orig = out;        // Keep this pointer for finalize code
        __global float2 *in_orig = in;
        __global float2 *out2_orig = out2;      // Keep this pointer for finalize code
        __global float2 *in2_orig = in2;

        int workItemOffset = lId * 8 + groupOffset;

        barrier(CLK_LOCAL_MEM_FENCE);

        // Do FFT
        // Equiv: speex_echo_opencl_fft_execute2(ocl_e, chan*st->window_size, ocl_E, chan*st->window_size, 1, commandQueue);
        if(taskId == 0)
        {
                // Do FFT first
                int dir = -1;

                // Do preparation step to convert from Speex FFT format
                // There is one thread for each four positions in the FFT length

                // clFFT_Forward
                for(int t=0;t<8;t++)
                {
                        in_orig[workItemOffset + t].x = e[workItemOffset + t] *
                             m;
                        in_orig[workItemOffset + t].y = 0.0f;
                }

                // End of preparation step

                barrier(CLK_LOCAL_MEM_FENCE);

                offset =  mad24(groupId, 1024, ii);
                // FFT KERNEL FUNCTION NOT LISTED, see clFFT_custom

                barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

                // Do finalization step to convert to Speex FFT format
                // There are one thread for each four positions in the FFT
                     length
                // Not adapted to multi-channel filter, only mono

                int workItemOffset2 = lId * 4 + groupOffset;

                // The first thread for each FFT length
                if(lId == 0)
                {
                        E[groupOffset + 0] = out_orig[workItemOffset2 + 0].x;
                        E[groupOffset + 1] = out_orig[workItemOffset2 + 1].x;
                        for(int t=1;t<4;t++)
                        {
                                E[groupOffset + 2*t] = out_orig[workItemOffset2
                                     + t].y;
                                E[groupOffset + 2*t + 1] = out_orig[
                                     workItemOffset2 + t + 1].x;
                        }
                }
                else
                {
                        for(int t=0;t<4;t++)
                        {
                                E[workItemOffset + 2*t] = out_orig[
                                     workItemOffset2 + t].y;
                                E[workItemOffset + 2*t + 1] = out_orig[
                                     workItemOffset2 + t + 1].x;
                        }
                }

                // End of finalization step

        }
        // Set to value and do FFT
        // Equiv: mdf_opencl_set_array_to_float_value(ocl_y, chan*st->
             window_size, 0, st->frame_size);
        //          speex_echo_opencl_fft_execute2(ocl_y, chan*st->window_size,
             ocl_Y, chan*st->window_size, 1, commandQueue);
        else
        {
                // Set to zero
                y[groupOffset + 4*lId] = 0.0f;
                y[groupOffset + 4*lId + 1] = 0.0f;
                y[groupOffset + 4*lId + 2] = 0.0f;
                y[groupOffset + 4*lId + 3] = 0.0f;

                // Do FFT
                int dir = -1;

                barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);

                // Do preparation step to convert from Speex FFT format
                // There is one thread for each four positions in the FFT length
                for(int t=0;t<8;t++)
                {
                        in2[workItemOffset + t].x = y[workItemOffset + t] * m;
                        in2[workItemOffset + t].y = 0.0f;
                }

                barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);

                offset =  mad24(groupId, 1024, ii);
                // FFT KERNEL FUNCTION NOT LISTED, see clFFT_custom
                // a small modification is made to use the in2 and out2 buffers

                barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

                // Do finalization step to convert to Speex FFT format
                // There are one thread for each four positions in the FFT
                     length

                int workItemOffset2 = lId * 4 + groupOffset;

                // The first thread for each FFT length
                if(lId == 0)
                {
                        Y[groupOffset + 0] = out2_orig[workItemOffset2 + 0].x;
                        Y[groupOffset + 1] = out2_orig[workItemOffset2 + 1].x;
                        for(int t=1;t<4;t++)
                        {
                                Y[groupOffset + 2*t] = out2_orig[workItemOffset2
                                     + t].y;
                                Y[groupOffset + 2*t + 1] = out2_orig[
                                    workItemOffset2 + t + 1].x;
                        }
                }
                else
                {
                        for(int t=0;t<4;t++)
                        {
                                Y[workItemOffset + 2*t] = out2_orig[
                                    workItemOffset2 + t].y;
                                Y[workItemOffset + 2*t + 1] = out2_orig[
                                    workItemOffset2 + t + 1].x;
                        }
                }

                // End of finalization step
        }
}

/**
 * Specialized kernel for doing a forward FFT and then an inner product
 * Globalsize: 128*K
 * Localsize: 128
 */
__kernel void clFFT_inner_prod_combined(__global float *data_time_domain,
     __global float2 *in, __global float2 *out, __global float *data_freq_domain
     , __global float *Xf, __global float *temp_reduction, int frame_size, int S
     )
{
        __local float sMem[1088];
        int i, j, r, indexIn, indexOut, index, tid, bNum, xNum, k, l;
        int s, ii, jj, offset;
        int lId = get_local_id(0);
        int groupId = get_group_id(0);
        int groupOffset = groupId * 1024;       // Group offset for float
        int groupOffset2 = groupId * 512;       // Group offset for float2
        unsigned int xi = get_local_id(0)*4 + frame_size + groupOffset; // Used
             for inner product
        unsigned int yi = get_local_id(0)*4 + frame_size + groupOffset; // Used
```

```
            for inner product
        float2 w;
        float ang, angf, ang1;
        __local float *lMemStore, *lMemLoad;
        float2 a[8];
        const float m = 1.0f / 1024.0f;
        ii = lId;
        jj = 0;

        // Do FFT
        int dir = -1;

        offset =  mad24(groupId, 1024, ii);
        __global float2 *out_orig = out;        // Keep this pointer for
            finalize code
        __global float2 *in_orig = in;

        int workItemOffset = lId * 8 + groupOffset;

        barrier(CLK_LOCAL_MEM_FENCE);

        // Do preparation step to convert from Speex FFT format
        // There is one thread for each four positions in the FFT length
        for(int t=0;t<8;t++)
        {
                in[workItemOffset + t].x = data_time_domain[workItemOffset + t]
                    * m;
                in[workItemOffset + t].y = 0.0f;
        }

        barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);

        // End of preparation step
        offset =  mad24(groupId, 1024, ii);
        // FFT KERNEL FUNCTION NOT LISTED, see clFFT_custom

        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

        // Do finalization step to convert to Speex FFT format
        // There are one thread for each four positions in the FFT length

        int workItemOffset2 = lId * 4 + groupOffset;

        // The first thread for each FFT length
        if(lId == 0)
        {
                data_freq_domain[groupOffset + 0] = out_orig[workItemOffset2 +
                    0].x;
                data_freq_domain[groupOffset + 1] = out_orig[workItemOffset2 +
                    1].x;
                for(int t=1;t<4;t++)
                {
                        data_freq_domain[groupOffset + 2*t] = out_orig[
                            workItemOffset2 + t].y;
                        data_freq_domain[groupOffset + 2*t + 1] = out_orig[
                            workItemOffset2 + t + 1].x;
                }
        }
        else
        {
                for(int t=0;t<4;t++)
                {
                        data_freq_domain[workItemOffset + 2*t] = out_orig[
                            workItemOffset2 + t].y;
                        data_freq_domain[workItemOffset + 2*t + 1] = out_orig[
                            workItemOffset2 + t + 1].x;
                }
        }

        // End of finalization step

        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

        // Calculate inner product (last reduction is done on host)
        if(get_local_id(0) < 128) {
                sMem[lId] = 0.0f;
                sMem[lId] += data_time_domain[xi] * data_time_domain[yi];
                sMem[lId] += data_time_domain[xi+1] * data_time_domain[yi+1];
                sMem[lId] += data_time_domain[xi+2] * data_time_domain[yi+2];
                sMem[lId] += data_time_domain[xi+3] * data_time_domain[yi+3];
        }
        else
                sMem[lId] = 0.0f;

        barrier(CLK_LOCAL_MEM_FENCE);

        // do reduction in shared mem
        for(unsigned int s = get_local_size(0) / 2; s > 0; s >>= 1)
        {
                if(lId < s)
                {
                        sMem[lId] += sMem[lId + s];
                }
                barrier(CLK_LOCAL_MEM_FENCE);
        }

        // write result for this block to global mem
        if (lId == 0)
        {
                temp_reduction[get_group_id(0)] = sMem[0];
        }
}

/**
 * Regular inverse FFT, combined with the spectral mul accum (long) reduce
     kernel
 * Globalsize: 128*K
 * Localsize: 128
 */
__kernel void clFFT_inverse_spectral_mul_accum_reduce(__global float *Y,
     __global float2 *in, __global float2 *out, __global float *e, __global
     float *temp_acc, int M, int N, int S)
{
        __local float sMem[1088];
        int i, j, r, indexIn, indexOut, index, tid, bNum, xNum, k, l;
        int s, ii, jj, offset;
        float2 w;
        float ang, angf, ang1;
        int dir = 1;    // Inverse
        __local float *lMemStore, *lMemLoad;
        float2 a[8];
        int lId = get_local_id(0);
        int groupId = get_group_id(0);
        const float m = 1.0f / 1024.0f;
        int groupOffset = groupId * 1024;
        ii = lId;
        jj = 0;

        offset =  mad24(groupId, 1024, ii);
        __global float2 *out_orig = out;        // Keep this pointer for
             finalize code
        int workItemOffset = lId * 8 + groupOffset;


        // Do the spectral mul accum reduce, 8 elementents for each thread
        for(int t=0;t<8;t++)
        {
                Y[groupOffset + lId*8 + t] = 0.0f;
        }

        for(int c=0;c<M;c++)
        {
                for(int t=0;t<8;t++)
                {
                        Y[groupOffset + lId*2 + t] += temp_acc[groupId*M*N + c*N
                             + lId*8 + t];
                }
        }

        barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);

        // Do preparation step to convert from Speex FFT format
        // There is one thread for each four positions in the FFT length

        if(lId == 0)
        {
                in[groupOffset + 0].x = Y[groupOffset + 0];
                in[groupOffset + 0].y = 0.0f;
                for(int t=1;t<4;t++)
                {
                        in[groupOffset + t].x = Y[groupOffset + (2*t - 1)];
                        in[groupOffset + t].y = Y[groupOffset + (2*t - 1) + 1];
                }
                in[groupOffset + 512].x = Y[groupOffset + 1024 - 1];
                in[groupOffset + 512].y = 0.0f;
        }
        else
        {
                in[groupOffset + 4*lId].x = Y[groupOffset + lId*8 - 1];
                in[groupOffset + 4*lId].y = Y[groupOffset + lId*8];
                in[groupOffset + 4*lId + 1].x = Y[groupOffset + lId*8 + 1];
                in[groupOffset + 4*lId + 1].y = Y[groupOffset + lId*8 + 2];

                for(int t=0;t<4;t++)
                {
                        in[groupOffset + 4*lId + t].x = Y[groupOffset + lId*8 -
                            1 + 2*t];
                        in[groupOffset + 4*lId + t].y = Y[groupOffset + lId*8 +
                            2*t];
                }
        }

        barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);

        for(int t=1;t<=4;t++)
        {
                in[groupOffset + 1024 - lId*4 - t].x = in[groupOffset + lId*4 +
                     t].x;
                in[groupOffset + 1024 - lId*4 - t].y = -in[groupOffset + lId*4 +
                      t].y;
        }

        barrier(CLK_LOCAL_MEM_FENCE);

        // FFT KERNEL FUNCTION NOT LISTED, see clFFT_custom

        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

        // Do finalization step to convert to Speex FFT format
        // There are one thread for each four positions in the FFT length

        // clFFT_Inverse
        for(int t=0;t<8;t++)
        {
                e[workItemOffset + t] = out_orig[workItemOffset + t].x;
        }

        // End of finalization step
}
```

Listing B.3: MDF OpenCL kernel functions for 512 sample frames