HAW
HAMBURG

Masterarbeit

Stefan Heidtmann

Implementation and Evaluation of Optimization Strategies for Audio Signal Convolution

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Stefan Heidtmann

Implementation and Evaluation of Optimization Strategies for Audio Signal Convolution

Mastertarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Wolfgang Fohl
Zweitgutachter: Prof. Dr. Andreas Meisel

Eingereicht am: 8. Juli 2019

Stefan Heidtmann

Thema der Arbeit

Implementierung und Evaluierung von Optimierungsstrategien für die Faltung von Audio Signalen

Stichworte

OpenCL Faltung, Nachhall, Faltungshall, Diskrete faltung, Schnelle Faltung, Audioverarbeitung, AVX, SIMD

Kurzzusammenfassung

Der Fokus dieser Thesis ist die Evaluierung von C++ und OpenCL Implementierung der diskreten und der partitionierten schnellen Faltung. Für eine aussagekräftigen Evaluierung der Rechenleistung der Algorithmen auf der CPU und der GPU muss die Leistung der Prozessoren ausgelasten werden. Um das zu erreichen wurden verschiedene Optimierungsstrategien implementiert um deren Einfluss auf Leistung getestet.

Stefan Heidtmann

Title of Thesis

Implementation and Evaluation of Optimization Strategies for Audio Signal Convolution

Keywords

OpenCL, Convolution, Reverb, Convolution Reverb, Discrete Convolution, Fast Convolution, Uniformly-Partitioned Convolution, Audio Procesing, AVX, SIMD

Abstract

The focus of the thesis is the implmentation of the discrete convolution and the uniformly-partitioned convolution on the CPU with C++ and on the GPU with OpenCL. When comparing the performance of algorithms on different processor types, it is uttermost importance to exhaust their processing power. To obtain the maximum performance

optimization of the code is required. There are several types of strategies available for optimizing the code. Their effect on the performance is evaluate to show which are absolute necessary and which have only a negligible effect.

Contents

List of Figures vii

List of Tables x

Accronyms xii

1 Introduction 1

2 Related Work 3

3 Digital Audio Signals and Real-time Digital Audio Processing 5

4 Convolution and its Implementation for Audio Processing 7

4.1 Discrete Convolution 8
4.2 Fast Convolution 9
4.2.1 Unpartitioned Convolution 9
4.2.2 Partitioned Convolution 10
4.3 Segmented Convolution 12
4.3.1 Overlap Add 12
4.3.2 Overlap Save 13

5 Basics of Performance Optimizations 15

5.1 Code Optimization 15
5.1.1 Operator Replacing 15
5.1.2 Loop unrolling 16
5.1.3 Register Optimization 17
5.2 Advanced Vector Extensions 17
5.3 FFTW 18
5.4 OpenCL 18
5.4.1 GPU Architecture for OpenCL 19

6 Project Architecture **21**
6.1 Third Party Libraries 22
6.2 EngineCore 22
6.2.1 Audio Normalization 23
6.3 AudioIO 24
6.4 Convolution Engine Libraries 24
6.5 Test Environment 24
6.6 VST Plugin 24

7 Convolution Engine Interface **25**
7.1 Interface Functions 25
7.1.1 Resize 26
7.1.2 SetFilterResponse 26
7.1.3 ProcessAudio 26
7.1.4 ClearBuffer 27
7.1.5 Error Handling 27
7.2 Dynamic Loading 27

8 Convolution Engine Implementations **28**
8.1 Discrete Convolution Engine 29
8.1.1 Optimized Discrete Convolution 29
8.1.2 AVX Discrete Convolution 32
8.2 Unpartitioned Convolution Engine 32
8.2.1 AVX Unpartitioned Convolution Engine 34
8.3 Uniformly-Partitioned Convolution Engine 34
8.4 Multithreading 35
8.5 OpenCL Implementations 36
8.5.1 Discrete Convolution 36
8.5.2 Kernel Optimization 37
8.5.3 Memory Transfer 40
8.5.4 Uniformly-Partitioned Convolution 41

9 Test Environment **43**
9.1 Test Design 43
9.2 Test Procedure File 44
9.2.1 Setting of an Environment Variable 45
9.2.2 Creation of a new Test Case 45

9.2.3 Opening of another Script File 45
9.3 Test Data 46

10 Measuring Inaccuracy and Platform Setup 47

11 Algorithmic Behavior 50
11.1 Discrete and Fast Convolution 50
11.2 Multithreading 52
11.3 GPU Memory Transfer 53

12 Evaluation 55
12.1 Hardware 55
12.2 Evaluation of the Optimizations 55
12.2.1 Discrete Convolution (CPU) 56
12.2.2 Discrete Convolution (OpenCL) 57
12.2.3 Unpartitioned and Uniformly-Partitioned Convolution 58
12.2.4 Uniformly-Partitioned Convolution with OpenCL on GPU 59
12.3 Performance comparison between CPU and OpenCL/GPU Implementations 60
12.3.1 I/O Buffer Size 61
12.3.2 Channel Number 62
12.3.3 Filter Length 63
12.4 Performance of Filter Changing 65

13 VST Convolution Plug-In 67

14 Conclusion and Outlook 69

15 Appendix 71
15.1 Convolution Engine Overview 71
15.2 Test Procedure File Variables 72
15.2.1 Set Variable 72

Glossary 78

Selbstständigkeitserklärung 79

List of Figures

4.1 Convolution of two signals f and g 8
4.2 Convolution procedure of the uniformly-partitioned convolution 11
4.3 Depiction of the overlap add process 13
4.4 Depiction of processing the Nth I/O buffer with overlap save 14

5.1 Pseudo kernel code for summing up two arrays in OpenCL 18

6.1 Building dependencies of the libraries and executables. Executable (Green), Static Library (Blue), Dynamic Library (white), Third Party Library (Red) 21

8.1 Pseudocode for the discrete convolution 29
8.2 Pseudocode for the optimized version of the discrete convolution. Shorter and more readable than the actual implementation 31
8.3 Pseudocode for the discrete convolution using vector instructions 32
8.4 Pseudocode for the fast convolution 33
8.5 Pseudocode for the fast convolution for partitioned convolution 35
8.6 Pseudocode for the computation device for the discrete convolution with OpenCL 37
8.7 Pseudocode for the host for executing the discrete convolution with OpenCL 37
8.8 Depicts the data blocks that are loaded in the local memory for the iterations of the loop (Fig. 8.9, 15). The currently loaded blocks are colored green. 38
8.9 Pseudocode for an optimized version of the discrete convolution with OpenCL. For this code to work all buffers have to have a size that is a multiple of the tile_size 39
8.10 Pseudocode for the host for executing the discrete convolution with a pipeline aproach for memory transfer and device computation 40
8.11 Pipline Models for the Open CL Implementation 41

8.12 Memory layout of the arrays for the implementation of the partitioned convolution with OpenCL. Real part of the complex number in dark the imaginary part in brighter color. The marked green part are the complex numbers a single thread handles during the convolution 42

10.1 Comparison between the engine processing times for a filter length of 44100 samples and 4 convolution channels 48
10.2 Comparison between the engine processing times for a filter length of 4410 samples and 4 convolution channels 49
10.3 Comparison between the engine processing times for a filter length of 44100 samples and 8 convolution channels 49

11.1 Sample throughput for the discrete convolution (red) and the unpartitioned convolution (blue) for different processing buffer sizes with a filter length of 44100 Samples. The 44100 samples per second line is marked in gray. 51
11.2 Sample throughput for the unpartitioned convolution for different sizes of the FFT transform, with a processing buffer size of 256. The 44100 samples per second line is marked in gray. 52
11.3 Sample throughput of a multithread engine for different numbers of channel on a Quadcore 53
11.4 Influence of the processing buffer size on the latency of the OpenCL pipeline approach. 54

12.1 Sample throughput for the unpartitioned convolution and the partitioned convolution for different parameters for the convolution. In blue the unpartitioned convolution, in red the uniformly-partitioned convolution. The default filter length is 48000 samples and a I/O buffer size of 256 samples 59
12.2 Sample throughput of the convolution engines by varying size of the I/O buffers. Start value for the buffer size is 32 samples, end 4096. Measurement points were all power of twos in between. The Central Processing Unit (CPU) implementations in blue, the OpenCL implementations in red. In grey the 48 kHz threshold for processing audio in real-time for the common sample rates 62

12.3 Sample throughput of the convolution engines by a varying number of channels for the convolution. The start number of channels is 4, and the end 48. Test were executed between start and end in increments of four. The CPU implementations in blue, the OpenCL implementations in red. In grey the 48 kHz threshold for processing audio in real-time for the common sample rates . . . 63
12.4 Sample throughput of the convolution engines by varying filter length. The start filter length is 24000, the end 480000. Tests were executed between start and end in increments of 24000. The CPU implementations in blue, the OpenCL implementations in red. In grey the 48 kHz threshold for processing audio in real-time for the common sample rates . . . 64
12.5 Sample throughput of the convolution engines by varying filter length. The start filter length is 2400, the end 20600. Tests were executed between start and end in increments of 2400. The size of the I/O buffers is in this case 64. The discrete convolution in blue, the partitioned convolution in red. In grey the 48 kHz threshold for processing audio in real-time for the common sample rates . . . 65

13.1 Screenshot of the plug-in GUI . . . 68

List of Tables

5.1 Minimum number of cycles needed to access the caches on the Haswell microarchitecture [10] 17

11.1 Prime factors of the transform sizes in Fig. 11.2 52

12.1 Comparison between the different implementations for the discrete convolution on the CPU 57

12.2 Comparison between the different implementations for the discrete convolution on the Graphics Processing Unit (GPU) with OpenCL 58

12.3 Comparison between the different implementations of the fast convolution on the CPU 58

12.4 Comparison between the different implementations for the partitioned convolution on the GPU 60

12.5 Time needed to change a 48000 samples filter for different algorithms . . . 66

Accronyms

AVX Advanced Vector Extension.

AVX2 Advanced Vector Extension 2.

CPU Central Processing Unit.

DAW Digital Audio Workstation.

DLL Dynamic Link Library.

DSP Digital Signal Processor.

FDL Frequency-domain delay-line.

FFT Fast Fourier transform.

FIR-Filter Finite Impulse Response Filter.

FPGA Field-Programmable Gate Array.

GPU Graphics Processing Unit.

IFFT Inverse Fast Fourier transform.

LTI-System Linear and Time-Invariant System.

SIMD Single Instruction, Multiple Data.

SIMT Single Instruction, Multiple Threads.

so Shared Object.

SpS Samples per Second.

SSE Streaming SIMD Extensions.

1 Introduction

Convolution reverb is a digital audio effect to simulate the reverberation of sound in an environment. These reverberations are caused by changes in the direction of the sound wave, mostly through diffuse reflexion. Through the change in the propagation direction, a wavefront can reach a specific point in an environment multiple times with variations in traveling time and amplitude, creating reverberations. These reverberations alter the perception of sound and give humans an impression of their surrounding environment.

Convolution reverb reproduces the reverberations created by an environment accurately. The effect is applied by convolving an audio signal with the impulse response of the environment, hence the name. The impulse response is an audio recording of the reverberation triggered by an impulse in the environment. The simulation of the reverb is only accurate for a specific position of the audio source and the listener position. The first is the position in the environment where the impulse was triggered, the latter where the reverberations were recorded.

The application of convolution reverb in a real-time application is complicated. The main problem with convolution reverb is the algorithmic complexity of convolution algorithms combined with the high sampling rates of audio signals. This problem is only increased by the requirement to convolve multiple audio signals to create any immersive auditive environment. As an example, the pre-rendered acoustic simulation for the concert hall of the WDR, created by Ralph Burgmeyer [5], convolved an audio signal 24 times to recreate the reverberations of a single audio source. For the creation of an immersive and interactive environment a fitting reverb is indispensable. When convolution reverb is selected, proper optimization of the convolution algorithm is required.

Practical use of convolution reverb is not limited to music enthusiasts, but the effect is also used for audio in Virtual Reality (VR). While VR is a niche product two frameworks

using convolution reverb are available. The first one is VRWork from NVIDIA [1] and the second Resonance Audio from Google [2].

The main aim of this thesis is to evaluate three different convolution algorithms and compare their performance on CPU and GPU. The algorithms are discrete convolution, unpartitioned, and uniformly-partitioned fast convolution. GPUs are often used to increase the performance of algorithms. This is enabled by their different architecture. For a performance comparison between the two processors it is necessary to fully use their computational abilities. This is important, because the aim is to compare the performance of the processors not how well optimized the code is.

An emphasis of this thesis is the optimization. Each device has its own optimization strategies. For the CPU it is standard code optimization in C++ and the vector instruction unit. Strategies for the GPU are memory optimization and transfer optimization.

The algorithms are implemented in the form of convolution engines. A convolution engine being a digital audio processor element. The purpose of the convolution engine is to convolve multiple digital audio signals with their corresponding impulse responses. Each engine is customizable in at least three parameters: I/O buffer size, filter response length, and the number of channels for the convolution.

To showcase the effect of the optimization different versions of the algorithms were implemented. Each version uses different optimization strategies. The purpose is to find out which optimizations are necessary to achieve the maximum performance and are therefore necessary to apply for comparisons between the two processors.

Evaluation of the engines is conducted in a test environment. The test environment allows the testing of the engines in similar conditions with different test parameters. The aims of the tests are to find the break-even point between CPU and GPU implementations, and between the different algorithms.

As proof of concept, a VST Plug-in was created that combined with a Digital Audio Workstation (DAW) realizes the acoustic simulation of the WDR concert hall[5] in real-time.

[1] https://developer.nvidia.com/vrworks/vrworks-audio
[2] https://resonance-audio.github.io/resonance-audio/

2 Related Work

Artificial reverberations is a relatively old topic. The development of artificial reverberations started in the 1920 with analog methods. The first digital solution where developed in the 1960 through digital filters structures. The real-time usage of convolution reverb is a development over the last 10 years that was enabled by the increasing processing power of CPUs and the usage of GPUs for the convolution. [20]

That a GPU can be used to increase the performance of different audio processing task was shown by L.Savioja, V.Välimäki, J.O. Smith. [17]. They showed that with using CUDA a GPU can be used to increase the performance of additive synthesis, discrete and fast convolution. Since hardware and the compiler for said hardware-improves with the time the result of older papers may not reflect the current state.

This is not the only paper that examined the performance of convolution algorithms on GPUs. An implementation of the uniformly-partitioned convolution with CUDA showed that the GPU performed better for larger problem sizes [22]. Similar findings were reported in a paper that implemented partitioned convolution with CUDA [14]. This paper focused on running the convolution completely on the GPU. Combination of CPUs and GPUs can be beneficial as well. This was demonstrated for non-uniformly-partitioned convolution algorithms [11]. Non-uniformly-partitioned convolution algorithms use Fourier transforms of different sizes. The idea is to execute smaller transforms on the CPU while the larger transforms are executed on the GPU. Last but not least a different paper examined the effect of pipelined algorithms for the convolution on the GPU [2].

There are three problems with these papers. Firstly the results are only accurate for hardware similar to the hardware used in the tests. There is currently no solution to this problem. A more significant problem is the evaluation of the performance. The usual approach is to compare the performance of the new GPU implementation with a reference implementation on the CPU. For an accurate comparison it is necessary to optimize the

different implementations to their optimum, or at least close to it [8]. The problem with the mentioned papers is that implementation detail to the reference implementations are only brief or completely missing. The best description to the reference implementation found in the papers is: "As a reference implementation we used CPU implementation of convolution reverb effect using FFTW for FFT and OpenMP for parallelization of Overlap-Add method"[14]. It is not comprehensible from this description if the reference implementation is ell optimized. The probable cause is the page limit of papers. This forces the authors to only describe what they deem the most important aspects, even if this means that important information is missing.

The last problem is the focus on algorithms based on fast convolution. The advantage of these algorithms is their lower algorithmic complexity. The algorithmic complexity is only a good indicator for large problem sizes. For smaller problems the details implementation decide which the faster algorithm is.

The discrete convolution is not only interesting, because of the potential better performance for certain problem sizes. A problem with real-time audio processing is the latency between input and output. The latency depends on the size of the internal audio buffer, a smaller buffer means a shorter latency. When an effect is added to an instrument in real-time this latency becomes noticeable. At which point the latency becomes noticeable depends on the instrument. The acceptable range can range from 1.4 to 42 milliseconds [12]. The processing buffer size for 1.4ms latency would be 62 samples for a sample rate of 44.1 kHz and 68 samples for 48kHz. Fast convolution works better for larger buffer sizes while the discrete convolution, at least in theory, is unaffected by the buffer size.

The fast convolution has a far more severe problem in environments where the filter can change during run time. Unlike the discrete convolution the filter has to be preprocessed before being used in the convolution process. In the preprocessing the filter is partitioned, if necessary, and transformed into the frequency domain. For a system running already running close to the capacity, this can lead to errors in the audio output. Through careful load balancing and other replacement strategies, the additional load can be reduced, but not eliminated [4].

Other applications where discrete convolution can be better than fast convolution is in usage with hybrid reverb algorithms [16] and for the smaller partition sizes of non-uniformly-partitioned convolution. Hybrid reverb algorithms only convolve the direct sound and the early reflection and use other methods to create the remaining reverberations.

3 Digital Audio Signals and Real-time Digital Audio Processing

In digital environments sound is represented in the form of a digital audio signal. Digital audio signals are discrete in time and amplitude. A signal is a sequence of samples, each sample is a amplitude value. Time information is stored by the position of the sample in the sequence and the sample rate. [24]

Common formats for the amplitude value are floating point number or signed integer. While in a sample the full range of the integer is used, the range of the floating point is limited to $-1.0 \leq x \leq +1.0$. Values that are outside of the range are clipped. When converting between the formats -1.0 in floating point is equivalent to the minimal value of the integer and $+1.0$ to the maximum. The simplest way to store a digital audio signal in a program is as an array of either integers or floating points.[3]

The quality of a digital audio signal depends on the sampling rate and the number of bits used for a sample. A higher sample rate means a higher Nyquist frequency, a higher Nyquist frequency means an extended frequency bandwidth or simply put a higher sampling rate allows an audio signal to contain higher frequencies. An audio signal has to have at least a sample rate of 44.1 kHz to contain all frequencies that can be perceived by the average human. All frequencies above the Nyquist cannot properly be sampled and it leads to aliasing. A higher bit depth allows a better resolution of the amplitude. [24]

Real-time processing of digital audio signals on a PC is always problematic. The cause of the problem are the real-time requirements of the processing. At a sample rate of 48kHz, the time the CPU has between each sample is roughly around $20\mu s$. This timeframe has to be sufficient for processing the sample. The problem is a CPU on a PC has usually other tasks as well. A short spike in the execution time can mean that the audio processing process misses the deadlines. While audio processing has only a soft deadline, missing it leads to audible artifacts in the output. Audio samples are usually

not processed sample per sample, but rather in blocks of audio data. This does not solve the problem of spikes during the execution, but the additional time needed for the execution is distributed on all samples in the buffer rather than a single sample.

The usage of audio data blocks has its advantages and disadvantages. Besides the increased robustness against fluctuations in the processing times, an algorithm can need fewer operations per sample to process the audio. The disadvantage is that they introduce latency between capturing a signal, processing it, and playing it. Capturing as well as playing audio requires a full block of audio data. The minimum achievable latency for real-time processing of audio is at least two times the block size divided by the sample rate. For real-time applications it is beneficial to use small block sizes to reduce the latency (I/O Latency).

From the view of an audio processing element the audio blocks of the input signals are buffers. The input buffer refers to the buffer that holds the data that needs to be processed, while the output buffer holds the processing result. Since most processing elements use the same buffer for in- and output the buffer in this document is referred to as I/O buffer.

4 Convolution and its Implementation for Audio Processing

As mentioned in the introduction, the reverberation of a sound wave in an environment is simulated by convolving a dry audio signal with an impulse response of the environment. This is possible, because the propagation of the sound waves in an environment is linear and time-invariant due to the superposition property of waves. A general property of Linear and Time-Invariant Systems (LTI-Systems) is that the output can be simulated by convolving the input with the impulse response of said system. In audio processing this property of LTI-Systems is used to implement Finite Impulse Response Filters (FIR-Filters). In terms of audio processing convolution reverb is simply a FIR-Filter. [24]

The convolution operation itself is an operation that takes two functions as operands and creates a new function as a result. The symbol for the convolution is the asterisk (*) and it has the following properties: distributive, associative, and commutative. The definition of the convolution depends on the functions that are convolved. In the case of implementing FIR-Filters the operands, as well as the result, are digital audio signals. One of the operands is referred to as the filter, the impulse response of said FIR-Filter, the other as the input signal. The result of the convolution is simply referred to as the output signal. There are different algorithms for convolving digital audio signals, the discrete, and the fast convolution. [19]

4.1 Discrete Convolution

Digital audio signals are discrete and finite. In this case the convolution is referred to as discrete convolution and is defined as:

$$(f * g)[n] = \sum_{m=-M}^{M} f[n-m]g[m] \tag{4.1}$$

Where -M and M are the index of the first and last element of the function g. [19]

For understanding convolution it is helpful to understand the impulse response. The impulse response describes the behavior of a LTI-System after an impulse. In digital form the impulse is an audio signal with only a single sample with the maximum amplitude. When the input is shifted in time the impulse response is shifted with the same amount and when the amplitude is changed the amplitude in each sample of the impulse response is changed by the same proportions (linearity and time-invariant).

An approach to visualizing the convolution process is to treat one of the signals as a sequence of impulses (Fig. 4.1). Each impulse creates a new response of the LTI-System. The output of the LTI-System is the filter response shifted in time by the sample position of the triggering samples with each sample multiplied by the amplitude value of the triggering sample. The result of the convolution is the sum of all the filter responses. [19]

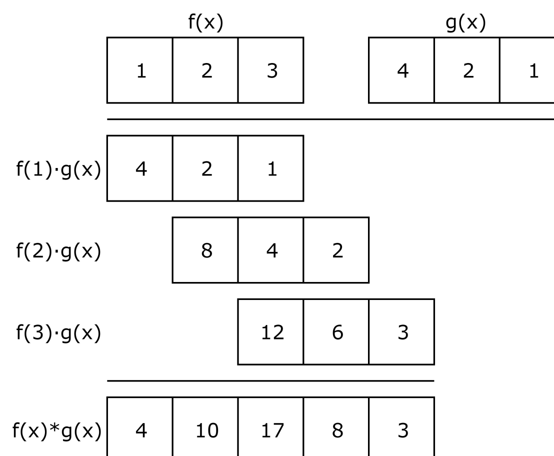

Figure 4.1: Convolution of two signals f and g

The order of algorithmic complexity of the discrete convolution of two functions $f(x)$ and $g(x)$ is $O(|f| \cdot |g|)$. Since the reverberations of the last sample in $f(x)$ have a length of $|g|$ the total length of the result is $|(f * g)| = |f| + |g| - 1$. [19]

4.2 Fast Convolution

Fast convolution is, as its name suggests, in most cases a faster way to convolve two signals. This is, because it has lower algorithmic complexity of $O(log(|f| + |g| - 1) \cdot (|f| + |g| - 1))$. The principle behind the fast convolution is that a convolution in the time domain is equal to a multiplication of the signals in the frequency domain. [19]

Theoretically, the fast convolution only works for periodic signals of equal length. The problem is that if the reverberation of a sample would exceed the length of the signal they wrap around and are added to the start of the signal. This problem is circumvented by appending zeros to the signals f and g until both signals have a length of $|f| + |g| - 1$, so that the result can be properly stored. While the wraparound technically still exists, the samples that are affected by the wraparound are all zero and thus do not have any reverberations that would be added at the beginning of the convolution result. [19]

There are different algorithm based on the fast convolution. The simplest is the unpartitioned convolution. Unpartition is referring to the unpartitioned filter, while the input may be partitioned.

4.2.1 Unpartitioned Convolution

The process for the unpartitioned convolution is first to extend the input signal and the filter to a length of input length + filter length - 1 and then transform them into the frequency domain with the Fast Fourier transform (FFT). During the transformation the format changes from real to complex numbers. In the frequency domain the complex numbers are than multiplied. The last step is to execute the Inverse Fast Fourier transform (IFFT) to transform the multiplication result back in to the time domain. [19]

4.2.2 Partitioned Convolution

The problem of the fast convolution is that the transform for the FFT need to be equally long for both the input as well as the filter. In real-time processing the size of the I/O buffer is usually far smaller the size of the filter. A larger transform is executed to transfer a relatively small signal into the frequency domain.

A solution to this problem is the segmentation or partitioning of the filter. This increases the number of operations for the multiplication in the frequency domain, but drastically reduces the number of operations for the FFT and the IFFT. There are two variants: the uniformly-partitioned convolution where the filter response is divided into equal parts and the Non-Uniformly-Partitioned Convolution.

Uniformly-Partitioned Convolution

The usual algorithm for uniformly-partitioned convolution partitions the filter response in the segments of the same same size as the input. This is not the optimal partition size for the partitions, but close to optimal for smaller segments sizes and easier to implement as an algorithms where any partition size can be chosen [23].

As preparation for the processing the filter is divided into partitions of the same size as the input buffer B. Each partition is then transformed into the frequency domain via FFT with a transform size of 2B. The first half of the transform is filled with a filter partition the other half with zeroes. [21]

The algorithm starts with transferring the input into the input save buffer (Fig. 4.2). The input save buffer holds the current input data and the data of the last input buffer. In case of the first processed input buffer the values of the last buffer are all zero. Totaling to a size of 2B. The data from the input save buffer is transformed into the frequency domain, with the oldest sample leftmost in ascending order. The transformed data of the input save buffer is then stored in a Frequency-domain delay-line (FDL) and all previously stored input shifted accordingly. [21]

After this the first entry of the FDL is multiplied with the first filter partition, the second entry with the second filter partition, etc. The FDL entries are multiplied with the filter partition by element-wise multiplication of the complex numbers. The multiplication results are then summed up in an accumulation buffer. [21]

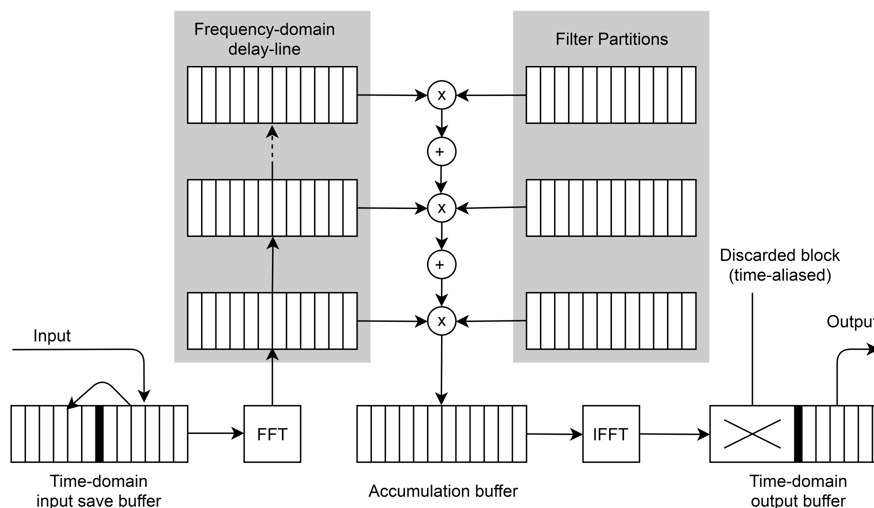

Figure 4.2: Convolution procedure of the uniformly-partitioned convolution

Lastly, the content of the accumulation buffer is transformed with the IFFT into the time domain. The first of the accumulation buffer is discarded, the second half is the convolution result. [21]

The main benefit of uniformly-partitioned convolution is that only a single FFT and IFFT are needed to convolute an I/O buffer. The main cost in the process is the multiplication of the complex numbers. The number of partitions depends on the I/O buffer size and filter length and can easily reach more than a hundred partitions for a single channel. The number of partitions at an I/O buffer size of 256 samples for a filter length of one second audio at a frequency of 48 kHz is 188.

Non-Uniformly-Partitioned Convolution

The main purpose of non-uniformly-partitioned convolution is not to increase the performance but, to reduce the I/O latency. It solves the problem that fast convolution works better for larger I/O buffer sizes, but with larger I/O buffer sizes the latency increases. In principle, non-uniformly-partitioned convolution is actually multiple convolutions with different I/O buffer sizes. In this work, this kind of optimization is not further pursued because of this aspect. A good estimate for non-uniformly-partitioned convolution would be to create multiple engines with the right parameter for the convolution and sum up the buffer processing times together.

Since the non-uniformly-partitioned convolution is not relevant to the rest of this document the term partitioned convolution is used as a term to reference the uniformly partitioned convolution.

4.3 Segmented Convolution

Digital audio is usually processed on a buffer by buffer basis. In the case of segmented convolution the input signals are divided into multiple segments that are convolved separately. This is possible because of the distributive property $((A+B)*C) = (A*C+B*C))$ of the convolution.

There are two different approaches for implementing segmented convolution: overlap add and overlap save. While both terms are more often used in the context of the fast convolution the concept behind them can be applied to the discrete convolution as well. The two approaches are basically two different approaches to handling the time behavior of the convolution algorithm. This is necessary, because the result of processing an I/O buffer is longer than the I/O buffer itself. A short way to express the difference would be: overlap add stores the result, while overlap save stores the input.

4.3.1 Overlap Add

The concept of overlap add is the implementation of the distributive property (Fig. 4.3). The first step is to convolve the audio data in the input buffer with the filter response. The result of the convolution is added to an internal buffer that stores the result. Lastly, the output buffer is filled with values of the result buffer.

Implementations of overlap add use a ring buffer to save memory as a buffer for the result. The size of the result buffer must be at least as high as the number of samples in the I/O buffer and the filter response combined. The write pointer of the ring buffer is the starting point where the result of the convolution of the input buffer and the filter response is written and is incremented in I/O buffer size steps. The read pointer points to the location from which the samples are transferred to the output buffer.

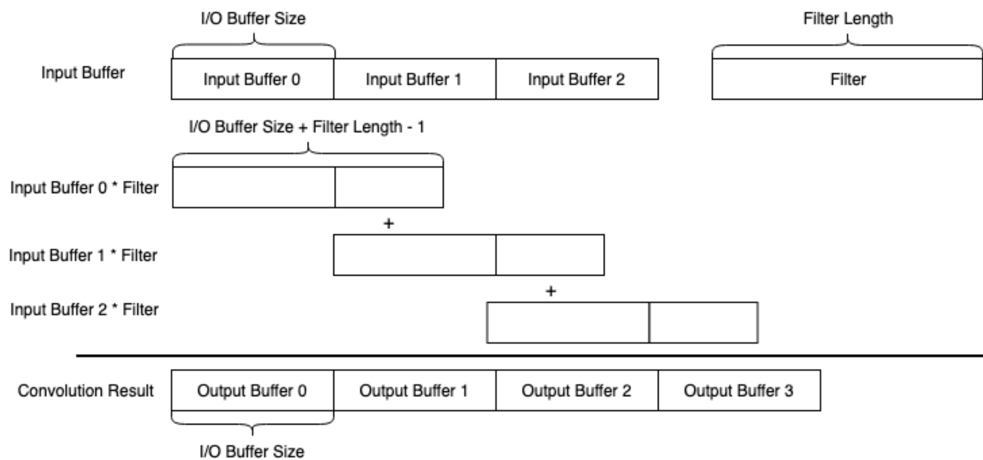

Figure 4.3: Depiction of the overlap add process

4.3.2 Overlap Save

The basic idea of overlap save is to save enough of the input buffer to calculate the values of the samples in the output buffer. The first step is to copy the data of the input buffer at the end of the save buffer. After this, the values for the samples in the output buffer are calculated. For the discrete convolution the value of a sample is calculated by the previously mentioned equation (eq. 4.1).

Overlap save for the fast convolution is slightly more complex. With the fast convolution the value of a specific sample in the result can be calculated by convolving a section of the input signal with the length of the filter response with the filter response (Fig. 4.4). The position of the last sample of the section is the position of the sample in the result. Expanding the section by X samples increases the number of calculated values by X as well. This means that the section for overlap save needs to have a total length of the filter response length + I/O buffer size - 1. The result of the convolution has the same length as the input signals. The values for the output buffer are at the end of the result. The part of the convolution results before the last segment are useless byproducts and are discarded. Because of this, the approach is also called overlap discard.

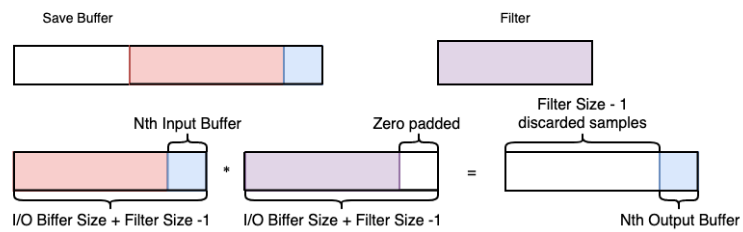

Figure 4.4: Depiction of processing the Nth I/O buffer with overlap save

5 Basics of Performance Optimizations

Convolution algorithms have high algorithmic complexity, but the choice of the algorithm is not the only influencing factor on the performance. A major contributing factor is how well optimized the code is and on what kind of device the code is executed. There are different approaches for optimizing the code. Examined strategies are standard code optimization, vector operation through intrinsic functions, and outsourcing of the code on to the GPU with OpenCL. The latter entails its own optimization.

5.1 Code Optimization

Code optimization can be achieved by various means. The options include: reducing the number of CPU operations, replacing slow operations through faster ones, reducing the memory transfer between RAM and CPU Caches as well as between cashes and the CPU registers.

One of the most helpful tools for optimizing code is the compiler. A problem for the optimization process is that information available to the programmer may not be available to the compiler. To achieve the best performance it is necessary to either apply the optimization manually or to enrich the code with as much information as possible.

The most common drawback of code optimization is that the code becomes increasingly longer and more complex. This makes the code less readable thus harder to maintain and more prone to errors.

5.1.1 Operator Replacing

Replacing of slow operations can drastically reduce the execution time of code. Two of the slowest arithmetic operations are the division and modulo operation, but both can

be replaced under certain circumstances. Other costly operations like function calls and branches cannot be replaced only avoided. [10]

The division operation can be replaced by multiplication with the multiplicative inverse when using floating point arithmetic. This is not an optimization on its own, because an additional multiplication is required, but it increases the performance when multiple values have to be divided by the same value or when the multiplicative inverse can be calculated outside of the performance critical code section. [10]

The replacement of the modulo operation has a few more conditions. Replacing the modulo operation is only possible when using integer arithmetic, both operands are positive, and the right operand is a power of two. When these conditions are met the modulo operation can be replaced by a bitwise AND:

$$x \,\%\, 2^n = x \,\&\, (2^n - 1) | x, n \,\epsilon\, \mathbb{N} \tag{5.1}$$

This is possible because of the properties of the binary system and the modulo operation, specifically the base two of the binary system. When calculating the modulo Y from X (X % Y) all digits from X with an equal or higher value than Y, are always an integer multiple of Y, whereas all digits with a value less than Y can maximally sum up to Y - 1. The modulo operation there gives as result only the last n - 1 digits of X where n is: $2^n = Y$. In the end, the modulo operation only sets all digits with a value equal or higher than Y to zero. This can far better be achieved, by a bitwise AND operation where the last n - 1 digits are one, which is the case for Y - 1.[10]

5.1.2 Loop unrolling

Loop unrolling is common practice to reduce the number of operations and jumps in the code. Loop unrolling means to reduce the number of iterations by executing the loop body multiple times in a single iteration or in the extreme the loop is completely replaced. Each time the body of the loop is executed, the condition of the loop is checked and a jump to another part of the code is executed. When the number of loop iterations is reduced the number of jumps and comparisons are reduced which is helpful since jumps are expensive operations. [10]

5.1.3 Register Optimization

When in a code block more variables are used than the CPU has registers, register spilling can occur. Register spilling is when the CPU has to store the content of the register into the cache to free up space for further operations. This slows down the execution time, because the CPU has to store and load from the cache whenever a variable needs to be switched. The transfer between the register and the cache takes multiple clock cycles (Table 5.1). Usage of the registers can only be directly controlled in assembler, but by reducing the number of currently used variables the compiler can optimize the register usage. [10]

Level	Latency
L1	4 cycles
L2	11 cycles
L3	∼34 cycles

Table 5.1: Minimum number of cycles needed to access the caches on the Haswell microarchitecture [10]

5.2 Advanced Vector Extensions

Advanced Vector Extension is an instruction set extension for the x86 architecture. The instruction set enables the usage of the Single Instruction, Multiple Data (SIMD) unit of the CPU, if the CPU has one. In Advanced Vector Extension (AVX) each register of the SIMD unit has 256-bit. This is double the register size as AVX predecessor Streaming SIMD Extensions (SSE). Each register is a vector with multiple elements. The number of elements the vector can hold depends on the size of the element, eight integers, four doubles, etc. Adding two vectors in AVX means that all elements of the first vector are added with the corresponding element of the second vector. [10]

AVX improves the performance by processing multiple data simultaneously. The problem is that AVX can only be used for uniform operations. AVX cannot be used for different operation for the elements in the vector. AVX instructions are either used directly in assembler code or by using intrinsic functions in C/C++. AVX can also be used by the compiler if enabled, but there is no guarantee that the compiler will use it. [10]

5.3 FFTW

The FFTW Library is a library for high-performance FFT. The main advantage of FFTW is that the library supports the FFT for signals of any length. Other libraries often only support a length which only has small prime factors for performance reasons. While the FFTW Library support signals of all lengths, it performs better when the length does not contain large prime factors. [7]

5.4 OpenCL

The OpenCL is a framework for developing and executing programs on different devices for parallel computing. OpenCL accomplishes this by providing a standard interface to access the hardware. The specific implementations are provided by the hardware vendors. [13]

OpenCL is used to accelerate resource-intensive parts of an application by outsourcing them on better-suited hardware. OpenCL differentiates between two kinds of devices, host and computing device. The host device acts as a master that starts the execution of OpenCL programs, the so-called kernels, and controls the memory transfer between the devices. Since OpenCL does not provide functionalities to get data from other devices all data for processing needs to be loaded to the computing device by the host. The number of computing devices a host can control depends on the hardware, but is -at least theoretically- unlimited. The kernels are written in Open CL C, a programming language that is based on the syntax of C. The compilers for OpenCL are provided by the hardware vendor. Devices from different vendors need different compilers. [13]

In OpenCL the kernel code describes what a single thread on the computing device should do. Each kernel has a three-dimensional ID that is usually used to select the elements the thread is working on (Fig.5.1). The task of the host device is to spawn the appropriate amount of threads to solve the calculation.

```
1  add(float* a, float* b, float* c){
2    int i = getID(0);
3    c[i] = a[i] + b[i];
4  }
```

Figure 5.1: Pseudo kernel code for summing up two arrays in OpenCL

The main advantage of OpenCL, and why it is used in this project, is its portability. OpenCL is not limited to the GPU but also supports execution on the CPU, FPGA and DSP. Implementation is provided by open source projects and most major hardware vendors, but not always the most recent version. Using OpenCL instead of a vendor-specific framework, like CUDA, can cause a loss in performance.

A paper reported that in the tested benchmark cases OpenCL code was compared CUDA was up to 63% slower, but also stated that properly optimized OpenCL code should perform equally well [6]. A test that compared the performance between OpenCL and CUDA for the convolution validate the statement, but the difference is not as huge [18]. While the CUDA implementation performed better with real-time support of a 7 second filter length for 24 Channels the OpenCL implementation came close with real-time support of up to 6.65 seconds with OpenCL. The conclusion of these papers is that while OpenCL can be used on different computing devices, the best performance is achieved when the code is optimized for a specific type of device.

5.4.1 GPU Architecture for OpenCL

GPUs have often smaller clock frequencies and less memory than CPUs, but have far more cores. The higher number of cores is possible, because GPUs have a different architecture. GPUs implement the Single Instruction, Multiple Threads (SIMT) model, which is an extension of SIMD.

Instead of vector arithmetic, the SIMT model uses threads to work on a large number of elements at once. The cores for the threads are relatively simplistic, at least compared to a core of the CPU, and are bundled into groups. In each group the cores share a local memory that effectively acts as a cache and an instruction unit. While all cores share the same instruction unit, the code can branch into different cases. This is possible, because a core can decide if an instruction is carried out or not. When a branch is executed the thread ignores the commands if the condition for entering the branch is not met. Since there is only a single instruction unit the branches have to be executed sequentially. While a branch is executed only the threads in the branch perform the instruction. Excessive branching can therefore have a significant impact on the performance, because only a few threads are working while the rest are idling. [9, 15, 1]

OpenCL code can be optimized to improve the performance. Optimizations which reduce the number of operations, like loop unrolling, can be applied to improve the performance.

A major difference between standard C and OpenCL C is that the management of all memory is done manually in OpenCL C, instead of relying on the CPU to utilize the caches. For the best performance on a GPU, the programmer has to properly make use of global and local memory. A common optimization strategy for the memory usage is tiling. In tiling, a problem is divided into multiple smaller parts. Each part is small enough that it can be executed in a single working group of cores. This allows the cores to load the data from the global memory into the local memory, where the processing takes place. After processing the result is loaded into the global memory. [9, 15, 1]

The advantage of GPUs is that they have a higher computing power than CPUs, because of their high number of threads. The drawbacks are that their unique architecture means that they may not necessary can make full use of their computing power and that additional data transfers between host and device are needed.

Currently, there are two different kinds of GPU, one for consumer and one for scientific application. They differentiate in the kind of floating points they use. Consumer GPU use only 32-bit floating points, with a single 64-bit floating point unit per instruction group while scientific GPU use 64-bit floating points as default. In audio processing, as well as in 3D-rendering, the accuracy provided by the 64-bit floating points is not needed so both are equally fine for usage.

6 Project Architecture

To make this project more manageable and reduce compile time it is separated into multiple parts. These are multiple static libraries, dynamic libraries (Dynamic Link Library (DLL) and Shared Object (so)), and executable files (Fig. 6.1). A special case is the VST Plugin which technically is a DLL, but has to fulfill special requirements. The test environment, as well as the VST Plugin, load the dynamic libraries during runtime. This chapter provides a brief overview of the different libraries and executables and their purpose.

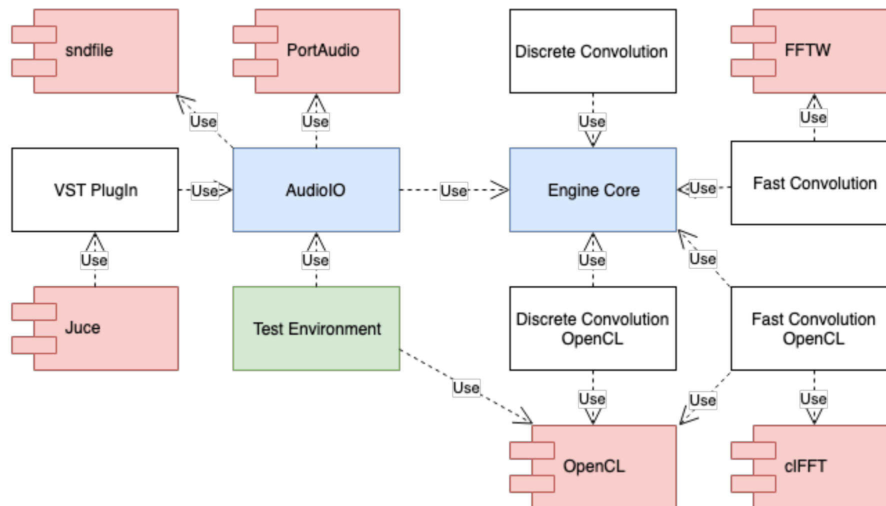

Figure 6.1: Building dependencies of the libraries and executables. Executable (Green), Static Library (Blue), Dynamic Library (white), Third Party Library (Red)

6.1 Third Party Libraries

Currently, the project uses six third-party libraries/frameworks for cross-platform support, faster code, and access to audio hardware.

- PortAudio: PortAudio is an open-source cross-platform C library for accessing the audio devices. The library supports audio drivers like Jack or ASIO.
- libsndfile: libsndfile is a cross-platform C library for reading or writing audio data into different audio formats through a uniform interface.
- FFTW: The FFTW (Fastest Fourier Transform in the West) is a cross-platform C library to compute the discrete Fourier transform.
- OpenCL: Open Computing Language is a cross-platform framework for executing code on CPU, GPU, Digital Signal Processor (DSP) and Field-Programmable Gate Array (FPGA).
- clFFT: The clFFt library is a well-optimized library for executing the fast Fourier transform with OpenCL.
- JUCE: JUCE is a cross-platform C++ application framework and was used to create the VST Plug-In. The framework makes the creation of plug-ins simpler by providing a pre-generated class to implement. The class is then wrapped by the actual plug-in through the framework. Other types of audio plug-ins are supported as well.

6.2 EngineCore

The library Engine Core provides functionalities for implementing the convolution algorithms. Core features are the Convolution Engine Interface that every convolution algorithm implements (Ch. 7), a class for dynamic loading of classes that implement the Convolution Engine Interface, and the AudioBuffer class that handles multi-channel audio data. Another function in the library is the finding of a normalization value for the convolution.

6.2.1 Audio Normalization

A problem with the convolution is that the maximum amplitude of the resulting signal is normally higher than the input signals,the amplitude value of the samples in the result could leave the value range. For integer based sample format this means under- and overflows, for floating point format clipping to the maximum or minimum value in the range (1.0 or -1.0). To prevent this the convolution result has to be normalized. The normalization of the audio is carried out by multiplying all sample values in the convolution result with the normalization parameter.

For the floating point format, the optimal value for the normalization is the multiplicative inverse of the highest absolute sample value in the convolution result. The main problem is that the optimal value can only be found after the signals are convolved. An application that uses real-time convolution could never use the optimal value.

A value that guarantees that no clipping occurs when no signal is known as the multiplicative inverse of the filter length (eq. 6.1). The only problem is that the amplitude of the normalization is far too low.

$$n_v = \frac{1}{|filterResponse|} \tag{6.1}$$

A better approach for finding a normalization value is, to sum up, the absolute values for the amplitude of the filter response (eq. 6.2). This is the highest amplitude value a sample can potentially have as the result of a convolution with the given filter response. As before the normalization value is the multiplicative inverse of said value. This normalization is the best value that still prevents clipping when only the filter is known, but after the normalization, the audio signal is still far to quiet.

$$n_v = \frac{1}{\sum_{i|n} |filterResponse(i)|} \tag{6.2}$$

The audio normalization class provides a function to analyze the filter response and suggest a normalization parameter for floating point convolution. The normalization value is the square root of the equation 6.2. The value does cannot guarantee that clipping is prevented, but the resulting signal is louder. In the end, it is only an estimation for a good value for normalization in the hope that the value found is sufficient.

There is no optimal solution for the problem of normalization. The solution is either far to quiet or does not prevent clipping. But the problem of audio normalization is not the focus of this work. When convolving multiple channels the normalization value is the same for all channels. Otherwise differences in the amplitudes in the filter response would be ignored.

6.3 AudioIO

The Audio I/O library contains classes for handling audio, a wrapper for reading and writing of audio files with libsndfile, and streaming of audio to and from devices with PortAudio. Through a shared interface device I/O and file I/O can be used interchangeably. The full list of supported audio files is found at: http://www.mega-nerd.com/libsndfile/. The full list of audio interfaces is found at: http://portaudio.com/docs/v19-doxydocs/api_overview.html.

6.4 Convolution Engine Libraries

Each of the four convolution engine libraries contains a number of convolution engines sorted by the algorithm used for the convolution and their primary usage of the CPU or the GPU. The term of convolution engines refers to the convolution algorithm that implements the convolution engine interface. Further information on the implementation can be found in chapter 7.

6.5 Test Environment

The purpose of the test environment is to test the performance of the convolution engines. The test environment is described in detail in chapter 9.

6.6 VST Plugin

The VST Plugin allows using the convolution engines in a DAW. Further information is found in chapter 13.

7 Convolution Engine Interface

The main goal of the project is the implementation of convolution engines. The purpose of a convolution engine is to convolve multichannel audio signals with the filters. For editing purposes, the channels are handled as multiple mono channels instead of an interleaved signal, like stereo. This means that the convolution of a channel is independent from the other channels.

To allow interchangeability between the engines they all need to implement the same interface. The interface was designed to accommodate two goals. The interface has to be integrable into a VST3 plug-in and only use the C++ 14 standard. While the reason for the first goal is pretty self-explanatory, the reasons for the second is to reduce the dependencies to the minimum. For dynamic loading an engine only needs to implement the interface, that is, in this case, a small single file.

7.1 Interface Functions

The interface declares the following functions for initialization, processing of audio, resetting of the engine, and error handling:

- int resize(size_t noChannels, size_t ioBufferSize, size_t minFilterLength, float normalizationValue)
- int setFilterResponse(size_t channel_index, float* filter, size_t filterLength)
- int processAudio(float** buffer)
- void clearBuffer()
- const int getLastError()
- const char* getLastErrorInfo()

The implementation does not synchronize the functions. Calling the functions from different threads is not recommended.

7.1.1 Resize

The function resizes changes the parameters for the convolution. The function has four arguments: noChannel, the number of audio channels the engine needs to process, ioBufferSize, the size of the I/O buffer, minFilterLength, the minimum filter length the engine has to support, the actual value can be higher for performance reasons, and the normalization value, the parameter for the audio normalization.

Resize is used for initializing the engine after its instantiation. The function can also be called again to reset the engine and initialize it with new parameters. While the acceptance of the parameters differs from engine to engine, the parameters must all be larger than zero and for the ioBufferSize a multiple of 32. The function returns a negative number if an error occurs.

7.1.2 SetFilterResponse

The function setFilterResponse sets the filter response used by the convolution for one channel. The function has three arguments channel_index, the index of the audio channel with which the filter response will be convolved, the second is the filter response in form of a pointer to an array and the last argument is the length of the array. The function returns a negative number if for some reason the function failed to set the filter, like selecting a non-existing channel.

7.1.3 ProcessAudio

The function processAudio has one argument: buffer. Buffer is an array over an array and is the I/O buffer. The first dimension is the channels of the buffer the second the samples of the audio signals. ProcessAudio processes the data in the buffer and fills the arrays with the result. The two dimensions are given by the resize functions. Using a buffer that is too small leads to an error. The function returns a negative number if the processing failed. For performance reasons, it is a single check if the engine was initialized

or not. The return value of called library functions, for example, when using OpenCL, are not checked.

7.1.4 ClearBuffer

For correctly implementing the time behavior of the convolution the engine needs internal buffers. When clearBuffer is called these buffers are set to zero to reset the engine. This function does not reset the filter responses.

7.1.5 Error Handling

The function getLastError returns the last occurred error as an integer. GetLastErrorInfo returns a C-String describing the cause for the error.

7.2 Dynamic Loading

The advantage of the interface is that it allows dynamic loading of the engines during run-time. The main benefit is that the application does not need to know a specific engine and can use different engines without a lot of effort. The ConvolutionEngineLoader is a class that provides functions for loading the convolution engines for Windows/VisualC++ and Linux/GCC. For the ConvolutionEngineLoader to work, the dynamic library has to provide a factory function to create the engine. The function prototype is:

```
ConvolutionEngineInterface* name(int nrParameters, int* parameters)
```

The parameters are optional to configure engines, like the number of threads the engine can use. The number and function of the parameters depend on the implementation and a full list can be found in the appendix (Ch. 15).

8 Convolution Engine Implementations

Several versions of discrete and fast convolution algorithms were implemented. Each algorithm has multiple versions that differ in the optimization and the language used (C++ or OpenCL). While the implementations of the convolution engine interface differ in the used hardware, and algorithm, the internal buffer structure is similar. All of them have a buffer for the filter, the I/O buffer, and a buffer to hold either partial results in the case of overlap add or the input signal in the case of overlap save.

The convolution algorithms are explained with the help of the pseudocode. To improve the readability the pseudocode describes only the convolution of a single channel. The iteration over the different audio channels as well as the normalization of the audio is missing. The normalization is simply a multiplication of all samples in the result. The iteration over the audio channels is an additional loop and index for all buffer accesses. The pseudocode uses the following naming scheme:

- I/O_Buffer: Buffer used to transfer the data between the Convolution Engine and the application
- Add_Buffer: Buffer that which holds the partial result when implementing Overlap Add
- Save_Buffer: Buffer that which holds the input signal for Overlap Save
- currentRingBufferPos: Pointer pointing to the current position in the ring buffer
- Filter: Buffer which holds the filter

Further missing steps are adding the I/O_Buffer to the Safe_Buffer, the partial clearing of the Add_Buffer and the setup of the filter. To further enhance readability not all optimizations are displayed. For example, most algorithms replace the modulo operation, but is usually not displayed to make the code easier to understand.

8.1 Discrete Convolution Engine

The discrete convolution engine implements, unsurprisingly, the discrete convolution using the overlap save. This implementation was created to have an unoptimized implementation as a reference of the discrete convolution for comparison with other engines. The discrete convolution engine is intended to be a worst case scenario, without actively trying to reduce the performance.

The discrete convolution engine implements the convolution by implementing the equation 8.1 to calculate a sample for the result. The only difference to the Equation 4.1 mentioned in Chapter 4 is that the filter always starts at the index 0.

$$(f * g)[n] = \sum_{m=0}^{|g|} f[n-m] \cdot g[m] \tag{8.1}$$

The convolution engine implementation needs two for loops for processing an I/O buffer (Fig. 8.1). The first for loop iterates over the samples in the I/O buffer and the second loop implements the sum in the equation.

```
1   int n = currentRingBufferPos;
2   for (int i = n; i < I/O_Buffer.size; i++, n++){
3    for (int m = 0; m < Filter.size; m++){
4      I/O_Buffer[i] += Save_Buffer[(n - m) % Save_Buffer.size] * Filter[m];
5    }
6   }
```

Figure 8.1: Pseudocode for the discrete convolution

The implementation of the discrete convolution is the implementation of the equation 8.1 with only a small modification in the form of a modulo operation (Fig. 8.1, line 4). The modulo operation is necessary because the *Save_Buffer* is a ring buffer. The access to the *Save_Buffer* in line 4 runs backwards. Through the modulo operation, the access jumps from the lowest element of the buffer to the highest.

8.1.1 Optimized Discrete Convolution

Optimizations for reducing the execution time of code are usually applied at the expense of the readability and maintainability of the code (Fig. 8.2). The most efficient way to

optimize code is to optimize the parts of the code that are executed the most, meaning mostly the bodies of loops.

When processing an I/O buffer in real-time the convolution engine only has a limited amount of time to process it. Most of the time is spent in the loops to convolve the signals. Outside the loops, the code mostly consists of memory management, transfer of audio data and management of the ring buffer. While optimizations can be applied, the time needed to execute these operations is nearly irrelevant compared to the time spent in the loops.

Reducing the number of operations in the innermost loop can have a large impact on the execution time. The innermost loop in the convolution can easily have a few millions iterations. Each clock tick saved in the loop body can have a huge impact on the processing time.

The convolution operation of a single channel in the convolution engine was optimized in several ways. The first was to unroll both loops. The first loop iterating over the samples in the output buffer was unrolled eight times (Fig. 8.2, line 3). Since the size of the I/O buffer is limited to a multiple of 32 this is not a problem since every multiple of 32 is also a multiple of eight.

The second loop that is calculating the result for the sample in the output buffer was unrolled four times (Fig. 8.2, line 2), meaning the size of the filter has to be a multiple of four to work. The length of the filter is automatically increased to the next multiple of four to avoid problems. In the worst case scenario, the size of the filter is three samples longer than needed. Compared to the thousands of samples a filter usually has the impact of these three samples negligible, especially if it means that the loop header is only executed a quarter of the time.

Simply unrolling the loop would increase the total amount of local variables leading to register spilling. Therefore the number of local variables was reduced. This leads to the cyclic changing of the *val* variables (Fig. 8.2, line 17, 20, 23 , etc.).

The last optimization was to replace the modulo operation with a bitwise AND (Fig. 8.2, line 8 - 11). This is only possible when the size of the ring buffer is a power of two. Since the ring buffer is only limited in the minimal size, the size of the ring buffer is simply rounded up to the next power of two. In the worst case the ring buffer needs nearly twice as much memory as actually needed for the convolution. Replacing the modulo operation is an optimization that sacrifices memory efficiency in favor of execution time.

```
size_t moduloMask = Save_Buffer.size - 1;
int n = currentRingBufferPos;
for (int i = 0; i < I/OBuffer.size; i += 8) {
  for (int j = 0; j < Filter.length; j += 4) {
    float filter_0 = Filter[j];
    float filter_1 = Filter[j + 1];
    float filter_2 = Filter[j + 2];
    float filter_3 = Filter[j + 3];

    float val0 = Save_Buffer[(n + i - j) & moduloMask];
    float val1 = Save_Buffer[(n + i - j - 1) & moduloMask];
    float val2 = Save_Buffer[(n + i - j - 2) & moduloMask];
    float val3 = Save_Buffer[(n + i - j - 3) & moduloMask];

    I/O_Buffer[i]     += val0 * filter_0 + val1 * filter_1
                     + val2 * filter_2 + val3 * filter_3;
    val3 = Save_Buffer[(n + i - j - 1) & moduloMask];
    I/O_Buffer[i + 1] += val3 * filter_0 + val0 * filter_1
                     + val1 * filter_2 + val2 * filter_3;
    val2 = Save_Buffer[(n + i - j - 2) & moduloMask];
    I/O_Buffer[i + 2] += val2 * filter_0 + val3 * filter_1
                     + val0 * filter_2 + val1 * filter_3;
    val1 = Save_Buffer[(n + i - j - 3) & moduloMask];
    I/O_Buffer[i + 3] += val1 * filter_0 + val2 * filter_1
                     + val3 * filter_2 + val0 * filter_3;
    val0 = Save_Buffer[(n + i - j - 4) & moduloMask];
    I/O_Buffer[i + 4] += val0 * filter_0 + val1 * filter_1
                     + val2 * filter_2 + val3 * filter_3;
    val3 = Save_Buffer[(n + i - j - 5) & moduloMask];
    I/O_Buffer[i + 5] += val3 * filter_0 + val0 * filter_1
                     + val1 * filter_2 + val2 * filter_3;
    val2 = Save_Buffer[(n + i - j - 6) & moduloMask];
    I/O_Buffer[i + 6] += val2 * filter_0 + val3 * filter_1
                     + val0 * filter_2 + val1 * filter_3;
    val1 = Save_Buffer[(n + i - j - 7) & moduloMask];
    I/O_Buffer[i + 7] += val1 * filter_0 + val2 * filter_1
                     + val3 * filter_2 + val0 * filter_3;
  }
}
```

Figure 8.2: Pseudocode for the optimized version of the discrete convolution. Shorter and more readable than the actual implementation

8.1.2 AVX Discrete Convolution

The vector arithmetic unit of a CPU allows operations on 256-bit vector, therefore enabling to add or multiply eight floats simultaneously. The vectors can be initialized by loading data from a float array (Fig. 8.3, line 5) and can also be written into a float array (Fig. 8.3, line 9). The AVX implementation always calculates eight samples for the result simultaneously. The result is calculated by loading a block of eight samples from the Save_Buffer and multiply them with a single value of the filter. The result of the multiplication is added to a result register (Fig. 8.3, line 5 - 7).

The main advantage of AVX is the potentially higher throughput by the use of vector instructions. Additionally, the header of the second loop is executed less often since eight samples are processed at the same time, the same effect as loop unrolling (Fig. 8.3, line 2).

A problem that is not addressed in the pseudocode is the wrap around of the ring buffer. The vector is a sliding window on the buffer containing eight values, that moves one sample back with every iteration of the inner loop. When the wrap around happens, the window jumps to the last sample in the buffer and loads bytes from outside the array. The simple solution is to make the buffer slightly larger and copy the first seven float values of the buffer behind the last element of the ring buffer.

```
1  for (int i = 0; i < I/OBuffer.size; i += 8){
2    size_t moduloMask = Save_Buffer.size - 1;
3    vecResult = loadValue(0);
4
5    for (int j = 0; j < Filter.size; j++){
6      vecIn = load(&Save_Buffer[(i - j) & moduloMask]);
7      vecFilter = loadValue(Filter[j]);
8      vecResult += vecIn * vecFilter;
9    }
10   store(&I/OBuffer[i], vecResult)
11 }
```

Figure 8.3: Pseudocode for the discrete convolution using vector instructions

8.2 Unpartitioned Convolution Engine

The unpartitioned convolution engine implements the fast convolution algorithm with overlap add. Fast convolution has a complexity class of $O(n \cdot log(n))$, but the algorithm

consists of three parts: the FFT to transform a signal into the frequency domain, the multiplication in the frequency domain and the IFFT to transform the multiplied signal back to the time domain. FFT and IFFT both have an algorithmic complexity of $O(n \cdot log(n))$. The implementation of the two transforms is provided by the FFTW library and are well optimized. Optimizing the fast Fourier transform to have a similar performance as the implementation provided by the library would exceed the scope of the project.

The fast convolution has the following steps (Fig. 8.4): the first step is to copy the data from the I/O buffer into a larger buffer for the transform. The usage of an extra buffer is required, because the FFT has a transform size of *I/OBuffer.size* + *filter.size* - 1. The samples in the transform buffer not filled with data from the I/O buffer are set to zero. The next step is to execute the FFT transform to transform the audio data into the frequency domain. This changes the number format from float to complex numbers. The actual convolution is carried out by multiplying the transformed I/O buffer data with the transformed filter. Afterward, the IFFT is executed to transform the convolution result back into the time domain. The result is added to the *Add_Buffer* and *I/O buffer* is filled with the result.

```
1  copy(transform_time, I/O_Buffer)
2  fft(transform_time, transform_freq)
3  for (int i = 0; i < transform_freq.size; i++){
4    transform_freq[i] *= Filter_freq[i];
5  }
6  ifft(transform_freq, transform_time)
7  n = currentRingBufferPosition;
8  for(int i = 0; i < transform_time.size; i++){
9    Add_Buffer[(i + n) % resultBuffer.size] += transform_time[i];
10 }
11 copy(I/OBuffer, Add_Buffer[n], I/OBuffer.size);
```

Figure 8.4: Pseudocode for the fast convolution

The code outside of the transforms is optimized as well, but the same optimizations are found in the optimized variant of the discrete convolution. Describing them again would be pointless and adding them to the pseudocode would only have a negative impact on the readability. For example, the implementation does not use modulo, but a bit-wise AND, even if the pseudocode would suggest otherwise (Fig. 8.4, line 8). For obvious performance reasons the filter are stored in the frequency domain. Transforming the filer whenever the processing function is called would be waste of time.

8.2.1 AVX Unpartitioned Convolution Engine

As for the discrete convolution engine, a convolution engine using AVX was implemented. The AVX version uses the vector instructions for the multiplication of the complex numbers in the frequency domain and to add the convolved I/O buffer to the result. To speed up the complex multiplication the format of the complex number is important.

Arrays of complex numbers can have two different formats. The more commonly found format is the interleaved format. In a buffer in the interleaved format, the complex numbers are stored behind each other with the real part first, followed by the imaginary part of the number. The second format is the planar format where the real and imaginary part of the numbers are stored in two separate arrays.

For the speed up it is vitally important that the buffer is in a planar format. While complex multiplication in the interleaved format is possible it requires a lot of shuffling of elements in the vectors. This requires to much time to be a viable optimization.

8.3 Uniformly-Partitioned Convolution Engine

The implementation of the partitioned algorithm follows the description in chapter 4.2.2. The algorithm starts with copying the last I/O buffer into the first half of a transform buffer and the current I/O buffer into the second half (Fig. 8.5, line 1-2). The transform buffer is transformed into the frequency domain and its content is put into a Frequency-domain delay-line (FDL) (Fig. 8.5, line 4-5). The convolution is carried out by multiplying the filter partitions with the entries in the FDL and summing the results up in an accumulation buffer (Fig. 8.5, line 7-12). The last step is to transform the accumulation buffer into the frequency domain and copy the second half into the I/O buffer (Fig. 8.5, line 14-15).

A particular optimization is the access to the FDL. The FDL is a ring buffer, since shifting every entry in the domain line would need a lot of time. The required modulo operation could be replaced by an AND operation if the size of the FDL would be increased to a power of two, but instead, the size of the FDL is doubled. The first half of the FDL are pointers to the buffers storing the frequency data. The second are a copy of the pointers. Since the implementation iterates backward over the entries the implementation starts

in the second half. When the wrap around would happen the implementations simply enters the first half.

A version using AVX for the multiplication of the complex numbers exists as well.

```
copy(transform_time, last_I/O_Buffer)
copy(&transform_time[transform_time/2], I/O_Buffer)

fft(transform_time, transform_freq)
copy(fdl[currentPartition], transform_freq)

int n = currentPartition + nrOfPartition
for(int j = 0; j < nrOfPartition; j++)
  for (int i = 0; i < transform_freq.size; i++){
    freq_accumulation[i] = filter_freq[j][i] * fdl[n - j][i];
  }
}

ifft(freq_accumulation, transform_time)
copy( I/O_Buffer, &transform_time[transform_time/2]);

currentPartition = (currentPartition + 1) % nrOfPartition
last_I/O_Buffer = I/O_Buffer
```

Figure 8.5: Pseudocode for the fast convolution for partitioned convolution

8.4 Multithreading

All modern CPUs have multiple independent cores. Using them is an effective approach to increase the sample throughput, but additional time is needed for the synchronization of the threads. The multithreading in the convolution engines was designed to minimize the synchronization overhead. The overhead was kept at a minimum by assigning the convolution of an audio channel to a single thread. The channels are equally distributed to the threads with a maximum difference in the number of channels of one. Because each thread fully utilizes the computing power of a core, using more threads than the CPU has cores does not increase the sample throughput.

The advantage of this approach is that the threads are completely independent of other threads during the convolution. Synchronization is only needed to start the threads and to wait until all threads have completed their task.

The disadvantage of the approach is that each channel is only convolved by one thread. When the number of channels is not a multiple of the number of threads some threads, and with them the cores, idle while others have to process the remaining channels. The total time needed for the convolution in an ideal environment is:

$$time_{total} = \lceil \frac{no_channels}{no_threads} \rceil \cdot time_{channel} \tag{8.2}$$

Using multiple threads to convolve one channel would be possible, but the threads would need to synchronize more often and not every part of the convolution can be executed in parallel. For this reason, one thread per channel seemed to be the better approach, even if this means that some CPU Cores have to idle if the number of convolution channels is not a multiple of the number CPU cores.

8.5 OpenCL Implementations

Convolution engines using the discrete and the partitioned convolution have been implemented for OpenCL. The OpenCL implementations try to reduce the involvement of the CPU to a minimum. The only task of the CPU is the memory transfer and calling of the kernels. The convolution itself is only carried out on the GPU.

In OpenCL, the optimizations can be categorized into two categories: optimization of the kernel code and optimization of the memory transfer between the devices. OpenCL is supported by a range of devices, but the optimizations of OpenCL code were applied to maximize the performance on a GPU. The applied optimization of the code may lead to worse performance on other device types.

8.5.1 Discrete Convolution

The simple implementation of the discrete convolution with the GPU is similar to the implementation of the discrete convolution on the CPU (Fig. 8.6). Like the CPU version the GPU version implements overlap save. The major difference is that the loops of the samples over the I/O buffer is missing. The loop is implemented by spawning a GPU thread for every iteration of the loop. The amount of threads spawned is controlled by the host device (Fig. 8.7, line 8).

```
convolve(I/OBuffer, Save_Buffer, Filter, n){
int channel_id = get_global_id(0);
int sample_id = get_global_id(1);

channel_save = SaveBuffer[channel_id]
channel_filter = Filter[channel_id]

 float result;
 for (int m = 0; m < I/OBuffer.Size; m++){
   result += channel_save[(n + sample_id - m) % channel_save.size]
                * channel_filter[i];
 }
 I/OBuffer[channel_id][sample_id] = result;
}
```

Figure 8.6: Pseudocode for the computation device for the discrete convolution with OpenCL

```
cl_cmdQueue.writeBuffer(I/OBuffer, I/OBufferCL)

cl_kernel.setArg(0, I/OBufferCL);
cl_kernel.setArg(1, CL_Save_Buffer);
cl_kernel.setArg(2, CL_Filter);
cl_kernel.setArg(3, currentRingBufferPosition);

cl::NDRange global(Number of Channel, I/OBuffer.Size);
cl_cmdQueue.callKernel(convolve, global);

cl_cmdQueue.readBuffer(I/OBuffer, I/OBufferCL)
```

Figure 8.7: Pseudocode for the host for executing the discrete convolution with OpenCL

The kernel is the implementation of the equation for the discrete convolution (eq. 8.1). The equation calculates a single value for the result. The thread gets the information which channel and sample they have to calculate by the ID of the thread. In OpenCL a thread has a three-dimensional ID. In this case, the first dimension is the index of the audio channel and the second the sample in the result that the thread has to calculate (Fig. 8.6, line 2-3). The last dimension is unused.

8.5.2 Kernel Optimization

The main difference between the standard discrete convolution kernel and the optimized version is the usage of tiling. Tiling is a technique to improve the sample throughput by dividing a calculation into smaller tiles to make use of the local memory of the device.

In OpenCL each thread is part of a work group. A work group on the GPU consists of multiple GPU cores for parallel code execution and shared local memory. The local memory is smaller, but faster than the global memory of the GPU. The number of cores in a working group varies from one device to another. On a GPUs a working group has at least 32 cores. [9, 15, 1]

Because the local memory is limited, larger problems like the convolution have to be divided into multiple tiles. In practice, the processing of the I/O buffer is divided into multiple tiles. This is generally like splitting the I/O buffer into multiple smaller buffers. Each working group fully processes one tile (Fig. 8.9) and, like the unoptimized version, every thread calculates one sample for the result. [9, 15, 1]

For the fast transfer between local memory and global memory the loading process of the thread in a working groups needs to be aligned (Fig. 8.9, line 8 - 13). Aligned means that when a thread with the id x transfers element x from local to global memory the thread with id (x + 1) has to do same for the element (x + 1). If the transfer is aligned, the transfer is a single instruction if not, the GPU needs one instruction for every single transferred value. [9, 15, 1]

The convolution process starts with loading two memory tiles from the save buffer and one from the filter (Fig. 8.9, line 12, 18, 19). The convolution is than partially calculated. When the threads are finished with processing the tiles new data is loaded (Fig. 8.9, 15).

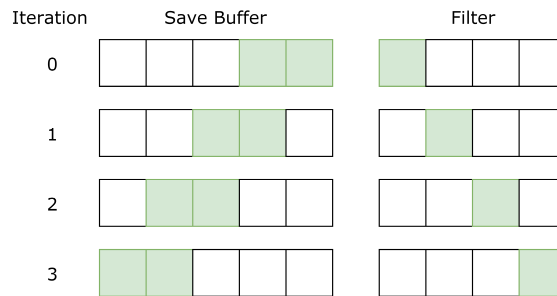

Figure 8.8: Depicts the data blocks that are loaded in the local memory for the iterations of the loop (Fig. 8.9, 15). The currently loaded blocks are colored green.

```
1  convolve(I/OBuffer, Save_Buffer, Filter, n,
2    local save, local filter, tile_size){
3    float result = 0;
4    int channel_id = get_global_id(0);
5    int tile_id = get_global_id(1);
6    int sample_id = get_global_id(2);
7
8    saveIndex = n + sample_id + tile_id * tile_size;
9    filterIndex = sample_id;
10   bufferPointer = 0;
11
12   save[bufferPointer] = SaveBuffer[channel_id][saveIndex];
13   bufferPointer = (bufferPointer + tile_size) % save.size;
14
15   for(int i; = 0; i < Filter.Size / tile.Size; i++){
16     saveIndex = (saveIndex - tileSize) % SaveBuffer[channel_id].Size;
17
18     save[bufferPointer] = SaveBuffer[channel_id][saveIndex];
19     filter[bufferPointer] = Filter[channel_id][filterIndex];
20     bufferPointer = (bufferPointer + tile_size) % save.size;
21
22     for (int m = 0; m < tile_Size; m++){
23       result += save[(bufferPointer + sample_id - m) % tile_size]
24                 * filter[m];
25     }
26     filterIndex += tileSize;
27   }
28   I/OBuffer[channel_id][sample_id + tile_id * tile_size] = result;
29 }
```

Figure 8.9: Pseudocode for an optimized version of the discrete convolution with OpenCL. For this code to work all buffers have to have a size that is a multiple of the tile_size

8.5.3 Memory Transfer

The convolution can be optimized by better usage of the involved hardware, mainly the PCI-E bus. The execution time required to process a I/O buffer can be divided into three parts (Fig. 8.11): Transfer of the data to the OpenCL device, execution of the convolution, and data transfer to the host. This means that during the time the device processes the current I/O buffer, the memory bus is idling and vice versa.

By changing the host code the available hardware can be better used by implementing a pipeline (Fig. 8.10). Processing the I/O buffer needs three processing cycles. During the first cycle, the input from the I/O buffer is transferred to the device, in the second cycle the buffer is processed, and in the last cycle, the processed data is transferred from the device to the host. This means that at any given time three I/O buffers are in the pipeline.

The advantage of the pipeline is that the time for processing an I/O buffer depends only on the longest execution time for one of its components and thus increasing the sample throughput. This is only the case when the memory bus is either full duplex or dual simplex, like PCI-E.

The disadvantages are a latency between in- and output of three full I/O buffers and more memory is needed on the computing device. (Fig. 8.10, 1-4).

```
1  temp = outputBuffer;
2  outputBuffer = processingBuffer;
3  processingBuffer = inputBuffer;
4  inputBuffer = outputBuffer;
5
6  cl_cmdQueue.writeBuffer(I/OBufferCL[inputBuffer], I/OBuffer)
7  cl_cmdQueue.readBuffer(I/OBufferTempOut, I/OBufferCL[outputBuffer])
8
9  cl_kernel.setArg(0, I/OBufferCL[processingBuffer});
10 cl_kernel.setArg(1, CL_Save_Buffer);
11 cl_kernel.setArg(2, CL_Filter);
12 cl_kernel.setArg(3, currentRingBufferPosition);
13
14 cl::NDRange global(Number of Channel, I/OBuffer.Size);
15 cl_cmdQueue.callKernel(convolve, global);
16
17 copy(I/OBufferTempOut, I/OBuffer)
```

Figure 8.10: Pseudocode for the host for executing the discrete convolution with a pipeline aproach for memory transfer and device computation

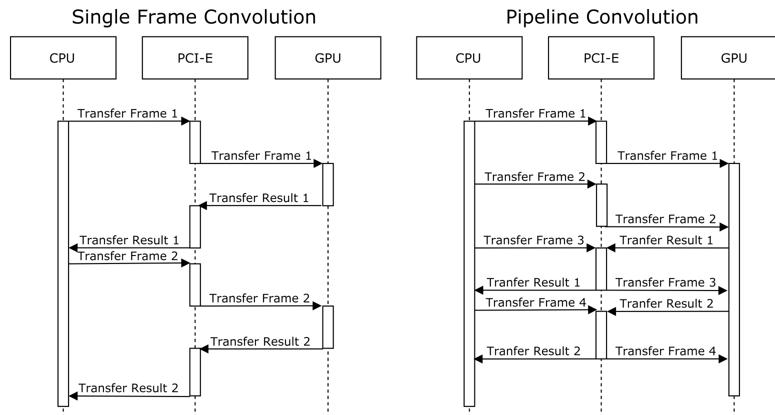

Figure 8.11: Pipline Models for the Open CL Implementation

8.5.4 Uniformly-Partitioned Convolution

The implementation of the partitioned convolution is similar to the CPU version. The two notable differences are that the clFFT library is used and that all channels are convolved at once rather than sequentially like the CPU version. For the latter, the memory for the partition has to have a specific layout. The memory is a logical four-dimensional array which is due to the limitation of OpenCL stored in a physical one-dimensional array. The first dimension is the partitions of either the FDL or the filter partitions. The second dimension are the different channels. The third dimension are the frequencies of the transformed signals, and the fourth dimension is the real and imaginary part of the complex number (Fig. 8.12).

The main kernel for the partitioned convolution is the multiplication of the entries in the FDL with the filter partitions. The kernel itself has two dimensions. The first being the channel on which the thread works, the second the complex number in the transform. The thread than iterates over all partitions and multiplies each time his complex number from the FDL with the corresponding number in the filter partitions and adds them to the accumulation result.

Two versions of the partitioned convolution were implemented. Both versions only differ in the kernel. One uses a modulo operation to access the entries in the FDL, the other replaces the operation.

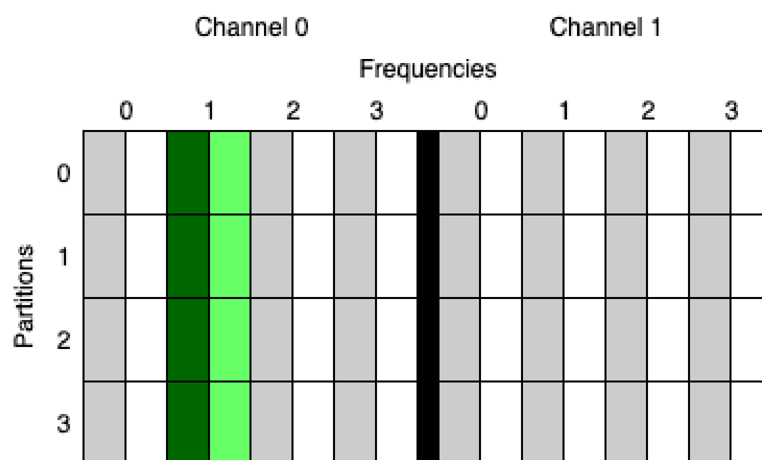

Figure 8.12: Memory layout of the arrays for the implementation of the partitioned convolution with OpenCL. Real part of the complex number in dark the imaginary part in brighter color. The marked green part are the complex numbers a single thread handles during the convolution

9 Test Environment

The evaluation of the convolution engines requires the testing of the engines. To simplify the testing process a test environment was created. The purpose of the test environment is to test the convolution engines under the same conditions, or at least as similar as possible, on an operating system.

9.1 Test Design

The aim of the tests is to allow a rating of the different convolutions based on their performance. How well an engine performs can be determined by three performance indicators.

- Throughput: How many samples can be convolved in a second
- I/O Latency: Time between data input and corresponding data output
- Memory Consumption: The amount of memory allocated by the engine.

How well an engine performs in these indicators depends on a number of parameters that can be divided into hardware parameters, how powerful the platform the engine is tested on is, and convolution parameters, how much work the engine has to do. The main parameters are:

- Hardware parameters
  - CPU
  - GPU (if used by the implementation)
  - RAM

- Convolution parameters
  - number of parallel convolutions
  - size of the I/O buffer
  - length of the filter

From the three indicators, the throughput and the I/O latency are to some extent linked. The I/O latency depends on the I/O buffer size. The size of the I/O buffer should be as low as possible, but in the case of the fast convolution, this reduces the throughput. The size of the I/O buffer should be just large enough to have an above real-time sample throughput. The memory consumption, on the other hand, is nearly irrelevant, since on current platforms the bottleneck is the processing buffer, not the memory.

From the two important performance indicators, the I/O latency can be easily calculated while the throughput can only be measured. Therefore the test case the environment executes is designed to measure the throughput. The throughput is measured by measuring the processing time of the engine. The processing time is the time between calling the processAudio function of an engine and the return from that function. Since the test is executed on an operating system the resulting measurements differs, caused by other interfering processes. To limit the influence of other processes on the processing time measurement the processing time is not only measured once, but rather several hundred times. The sample throughput is calculated by dividing the I/O buffer size by the processing time (Eq. 9.1). The sample throughput is the maximum sample rate the convolution engine could support in real-time at the given parameters.

$$throughput = \frac{1}{processingTime} \cdot I/OBufferSize = \frac{I/OBufferSize}{processingTime} \tag{9.1}$$

9.2 Test Procedure File

The testing procedure in the test environment is automated by script files. The structure of the test procedure file is relatively simplistic. Each line can at maximum only contain one command and comments starts with # and end at the line break. There are only three different commands that the test environment accepts: setting a variable in the test environment, creating a new test case, and opening of another test file. The first word in the command always determines the type.

9.2.1 Setting of an Environment Variable

The command identifier for the “set variable“ command depends on the type of the variable. the identifier can be “bool “, “int “, “float “or “string “. The identifier is followed by the name of the variable and its value.

```
typeId global_settings::varName = value
```

9.2.2 Creation of a new Test Case

The creation of a new test case starts by the identifier “test “ followed by the location of a dynamic library (path/name) and the name of a factory function for creating the engine. The next three parameters are integers for setting the convolution parameters. The order is first the number of parallel convolutions, second the size of the I/O buffer, and lastly the minimum size of the filter. Additional integers can be appended. Their function differs from engine to engine. Their function is either to set the number of threads the engine uses, or the devices used with OpenCL.

```
test filename factoryFunction channel bufferSize impulseSize
additionalParameters*
```

9.2.3 Opening of another Script File

Another script file can be opened by first writing the command identifier “testfile “ followed by a name for the test and the path of the file. The purpose of opening other testfiles is to group specific test together. All results of the test cases in a testfile are saved in their own folder with the name in the command. The folder structure represents the order of the testfiles, the start folder at the top where each opened testfile has its own subfolder.

```
testfile testName = filePath
```

9.3 Test Data

The test environment creates a CSV file for each test case containing the data and a log entry in the summary. The summary is a log file off all test cases in a test file. The entry contains all parameter of the test and some key performance indicators. In the CSV file each line contains first the number of the measurement and second the time needed for processing in milliseconds. The CSV header looks like this:

```
Buffer Number;Buffer Calculation Time(ms)
```

The character for the separation of two fields in the CSV file is a semicolon to avoid the problem with the decimal separator.

10 Measuring Inaccuracy and Platform Setup

The problem with measuring execution time on any given OS is that the result will rarely be the same, as seen in Figure 10.1. For more reliable time measurement the platform the engine is tested on should be properly configured. The main reason for the inaccuracy during the measurement is caused by other processes that the CPU executes in the background, at minimum the thread scheduler. The most obvious way to increase the accuracy is to terminate all unnecessary threads and run the threads with the highest priority. A more unique problem face Intel CPUs with hyper-threading.

Hyper-threading is Intels version of simultaneous multithreading. Other vendors have their own versions, but the term hyper-threading is more commonly used. Intel CPUs with hyper-threading have two virtual cores for each physical core. The virtual cores share the physical core, but each has its own register sets and the cache is partitioned between them. The aim of hyper-threading is to increase the performance by switching between the virtual cores. When the currently working virtual core has to wait for data of the RAM or to fill the command pipeline after a jump in the code the other core can use the waiting time to execute its own task. For real-time applications hyper-threading has some problems.

The question if hyper-threading should be enabled for measurements or during runtime can not easily be answered. To find a conclusion if it should be enabled for the measurements three tests were conducted. Each test used the same engine and the processing time of the engine was measured 1000 times. The test was conducted for hyper-threading and normal threading, so that the total number of test runs is six. The CPU was a Quadcore, meaning the number of virtual cores is eight.

The first comparison between normal threading and hyper-threading was with four convolution channels and a filter length of 44100 (Fig. 10.1). Half the cores for hyper-threading, full usage for normal threading. The average processing time for normal

threading with 6.8 milliseconds was shorter than the average of 8 milliseconds when executed with hyper-threading. Figure 10.1 shows that there is more variation in the processing time when executed with hyper-treading causing the higher average, while the lowest calculation time is similar. This is probably caused by the scheduler when it allocates threads to the same physical core.

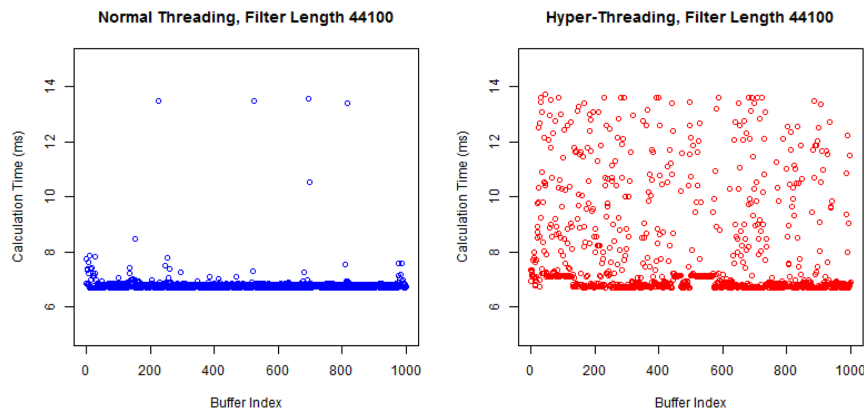

Figure 10.1: Comparison between the engine processing times for a filter length of 44100 samples and 4 convolution channels

This is contrary to the result of the second test. In the second test the engine was tested with the same number of channels, but only a tenth of the filter length (Fig. 10.2). The average for normal threading here is 0.98ms against 0.64ms with activated hyper-threading. When looking at the diagram a large number of measurements took twice as long as the lowest. The most likely explanation is that the scheduler scheduled two threads on the same core, convolving the channels sequentially instead of parallel. Hyper-threading seems to cope better with the short activities of threads. Probably, because it can suspend the lower priority threads from other processes more easily by switching the active core. A costly context switch is not necessary.

The last test was to check if there is a performance difference when all virtual cores are used. The engine was tested with a filter length of 44100 samplesand eight convolution channels (Fig. 10.3). Here the variation of the processing time is similar between normal and hyper-threading, but normal threading is faster, with an average of 12.7ms against an average of 13.9ms.

In conclusion hyper-threading is deactivated for the measurements for better performance and more accurate measurements. Hyper-threading only performances better for shorter

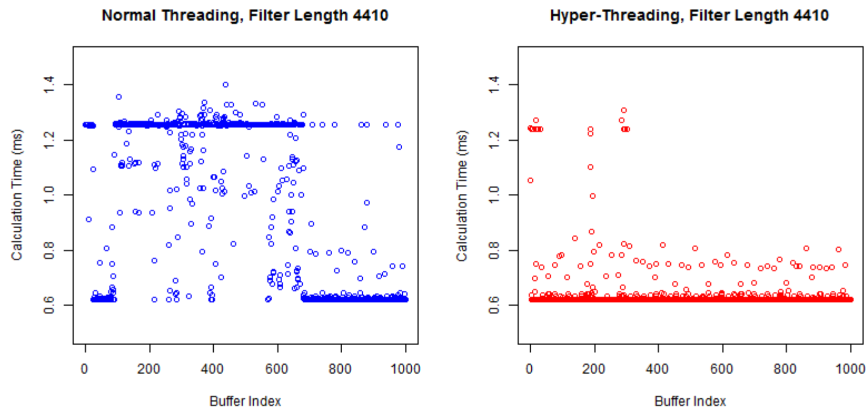

Figure 10.2: Comparison between the engine processing times for a filter length of 4410 samples and 4 convolution channels

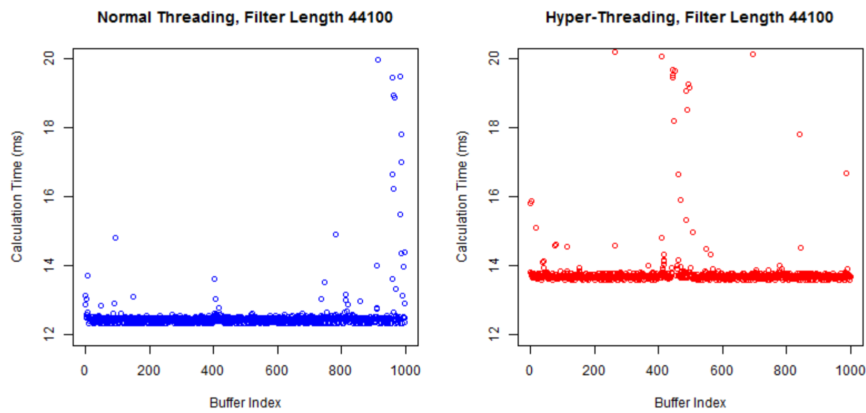

Figure 10.3: Comparison between the engine processing times for a filter length of 44100 samples and 8 convolution channels

filter lengths, which is not the primary test case. The probable cause for the poorer performance of hyper-threading are the optimizations. Reducing the number of cache misses and jumps means less idle time for the cores. Therefore the cores rarely have to wait, which is where hyper-threading would usually be beneficial.

11 Algorithmic Behavior

While the different implementations of the convolution algorithms perform differently, the implementations behave in a similar manner.

11.1 Discrete and Fast Convolution

The plot in Figure 11.1 shows the sample throughput of a discrete convolution engine and a unpartitioned convolution engine for different I/O buffer sizes for all powers of two between 32 and 8192. As seen in the plot the discrete convolution is mostly unaffected by the changes. The reason is that the discrete convolution is linear. When the number of samples in the I/O buffer is doubled the required processing time doubles as well, while the time needed to process a single sample stays the same.

On the other hand, the sample throughput of the unpartitioned convolution grows exponentially with the size of the processing buffer. The discrete convolution has a throughput of roughly 200000 Samples per Second (SpS), while the unpartitioned convolution starts with 5000 at a buffer size of 32 breaks the 44100 marks at a buffer size of 512 and overtakes the discrete convolution at a buffer size of 1024. This also means that, unlike the discrete convolution, the convolution of two signals needs more time than convolving a single signal with twice the filter length.

The reason why the unpartitioned convolution is not drawn in line in the Figure 11.1 has to do with the calculation time of the FFT and the IFFT. The time needed to calculate the transform is not linear. Only small changes can drastically affect the sample throughput (Fig. 11.2). At a transform size of 65536, a unpartitioned convolution algorithm has a sample throughput of over 80 thousand samples. Increasing the transform size slightly by six samples to 65542 samples reduces the sample throughput to under 20 thousand samples. The reason is that the unpartitioned convolution works better when the prime factors are small. The drop is explained because 65536 is a power of two while 65542

is two times a large prime. The prime factors for the transform sizes in the plot are in Table 11.1.

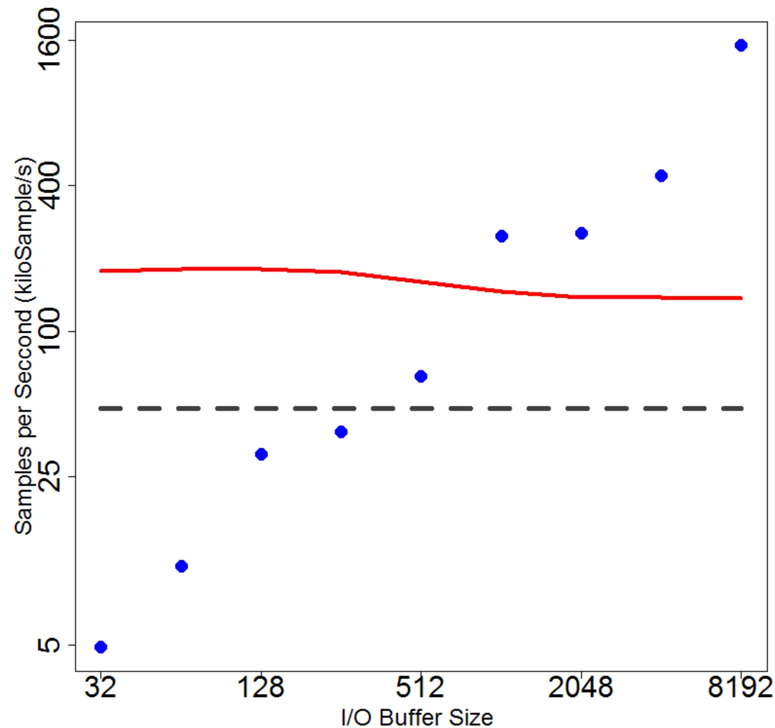

Figure 11.1: Sample throughput for the discrete convolution (red) and the unpartitioned convolution (blue) for different processing buffer sizes with a filter length of 44100 Samples. The 44100 samples per second line is marked in gray.

While it is easy to avoid large prime numbers it is not as easy to find the optimal size. As seen in Figure 11.1 the transform sizes of 65610 and 81920 roughly perform the same while one is much larger, with the largest prime in both of them being a 5. Simply rounding to the next power of two is not the ideal solution; a smaller transform size (in this case 73728) can have a higher sample throughput then the larger transform size and also requires less memory. In conclusion, the time needed for the execution of the FFT is a little bit unpredictable.

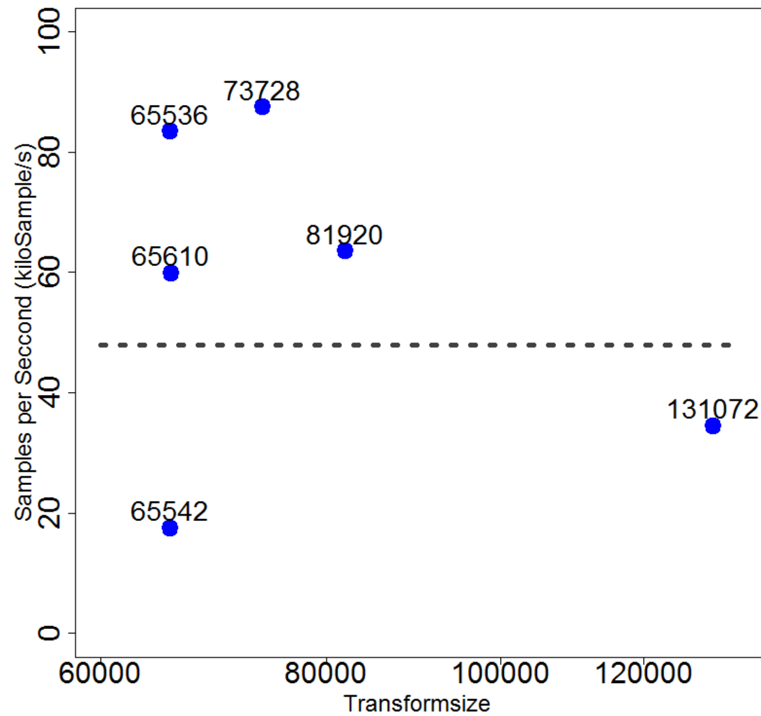

Figure 11.2: Sample throughput for the unpartitioned convolution for different sizes of the FFT transform, with a processing buffer size of 256. The 44100 samples per second line is marked in gray.

Transform size	Prime factors
65536	2^{16}
65542	$2 \cdot 32771$
65610	$2 \cdot 3^8 \cdot 5$
73728	$2^{13} \cdot 3^2$
81920	$2^{14} \cdot 5$
131072	2^{17}

Table 11.1: Prime factors of the transform sizes in Fig. 11.2 .

11.2 Multithreading

While multithreading does not lead to specific behavior the chosen implementation does. As seen in figure 11.3 the sample throughput only drops every four channel increments drastically. This is not accidentally the number of cores the processor has. As mentioned

before the multithreaded solution assigns a convolution channel only to a single core. This means the convolution is completely executed on a single core and the performance drops only if the number of threads per core rounded down changes.

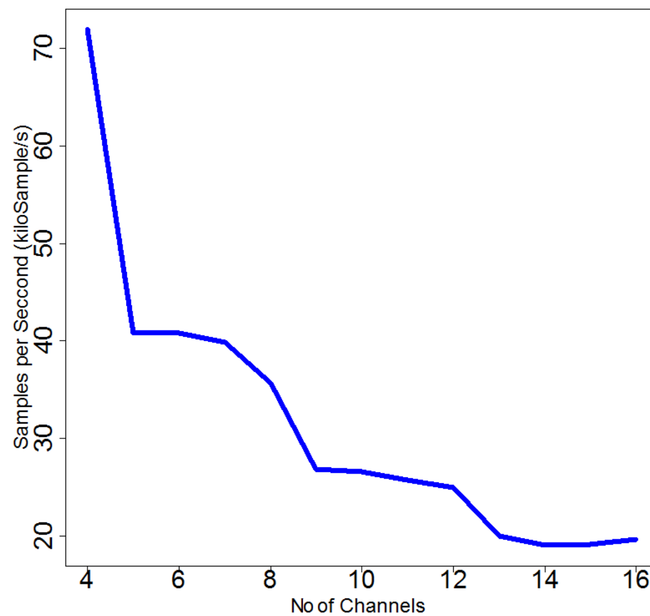

Figure 11.3: Sample throughput of a multithread engine for different numbers of channel on a Quadcore

11.3 GPU Memory Transfer

Similar to the multithreading the memory transfer optimization leads to a specific behavior.When using the pipeline approach the first two I/O buffers return filled with zeroes from the processing functions. Only after this, the engines start to return the actual result. The audio signals in Figure 11.4 show the resulting signals for two different OpenCL engines, one with a memory pipeline, one without. With a processing buffer size of 64, the latency is 128 samples compared to the non-pipeline engine and 256 samples with an I/O buffer size of 128 samples. This is additional to the one buffer size latency found in every convolution engines and all other forms of digital audio processing.

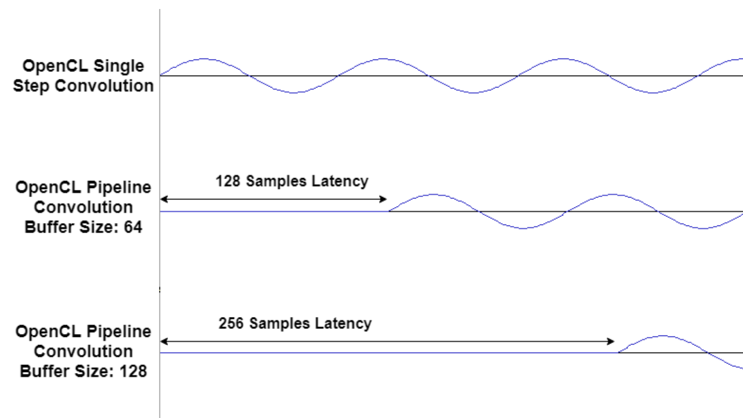

Figure 11.4: Influence of the processing buffer size on the latency of the OpenCL pipeline approach.

12 Evaluation

The evaluation of the different convolution engines consists of two different sections. The first section is the evaluation of the optimizations. The second is a test between the convolution engines to find their maximum performance and break-even points.

12.1 Hardware

The CPU used for the main test is an Intel Core i7-4770 CPU with 4 cores and a maximum clock frequency of 3.9GHz with 16 GB RAM. The GPU is a GTX 970 which has a maximum clock frequency of 1.316 GHz, 4 GB memory and, 13 computing units.

12.2 Evaluation of the Optimizations

The aim for evaluating the optimizations applied to the different convolution algorithms is to show the effect proper optimization can have on the performance and also reduce the number of tests for later tests. When an implementation of an algorithm is better than another implementation it will always be superior for all possible parameters for the convolution. This is only true for the same device since CPU and GPU scale differently.

The test case for the different implementations was to convolve a single channel with a filter with a length of 44100 samples, and an I/O buffer size of 256 samples. The measurement of the processing time was conducted 25000 times for each test. The engines were compiled with maximum optimization in favor for speed, not memory consumption and with enabling AVX instructions for the compiler, both in terms of manual usage and optimization by the compiler.

12.2.1 Discrete Convolution (CPU)

The discrete convolution on the CPU has three different versions. In this test case, the unoptimized version performed unsurprisingly the worst (Table 12.1). The unoptimized discrete convolution implementation needs roughly 70ms to process a single I/O buffer. This results in a sample throughput of 3633 samples per second which is not even a tenth of the required throughput for real-time convolution.

That the performance can be greatly increased shows the optimized version. By simple replacement of operators and loop unrolling the optimized version reaches a sample throughput of 78672.64 samples per second, a sample throughput more than 20 times greater than the unoptimized version. While this is already far better the AVX version of the discrete convolution has a sample throughput of 189325.3 samples per second, 2.4 times better than the optimized version.

The conclusion for this test is relatively straight forward. Using AVX is the best solution for the discrete convolution, and that even if the compiler is allowed to use AVX instructions for optimization purposes does not mean that thr compiler will use them.

The second test for the discrete convolution was how much multithreading, especially the synchronization effects the performance. Since the tested CPU had four cores, the number of convolution channels for the test was increased to four. This means the CPU had to solve a task four times larger with four times more resources. This means in the best case the multithreaded version requires the same amount of time for four channels than the single-threaded version for one.

The result of the test is that the optimized version with only one thread had an average processing time of 3.25 milliseconds while the multithreaded version was slightly faster with only 3.21 milliseconds. This is contradictory to the expectation since the multithreaded version does exactly the same plus synchronization overhead. In this case, minimal processing time can be interesting.

The measurements of the processing time always slightly differ because of interference of the operating system and other, processes but when a test is executed 25000 times it is quite likely that the shortest measured time is close to the shortest possible time. In the case of the single-threaded version, the shortest measured time is 3.06 ms while the multithreaded version had a time of 3.08ms. This means the multithreaded version

Engine Name	Processing Time (ms)		avg. Sample Troughput(SpS)
	average	minimum	
Discrete	70.4624	68.5641	3633.143
Discrete (opt)	3.25399	3.06125	78672.64
Discrete (opt, mul)	3.20848	3.08329	79788.56
Discrete (AVX)	1.35217	1.32188	189325.3
Discrete (AVX, mul)	1.44619	1.32581	177016.9

Table 12.1: Comparison between the different implementations for the discrete convolution on the CPU

needs roughly 0.02 ms for thread synchronizations. Similar times are found for the AVX version with a difference of 0.04 ms in favor of the single threaded version.

This shows that the engine can fully utilize the cores without major impact on the performance. Since synchronization is only performed at the beginning of the processing function and at the end the total cost is fix. The proportion amount of time for the synchronization becomes less when the problem size for the convolution increases.

12.2.2 Discrete Convolution (OpenCL)

The discrete convolution for OpenCL has two different optimizations, transfer optimization and kernel optimization. The standard version for OpenCL uses the same optimization found in the optimized version of the discrete convolution for the CPU. In this test case, the standard version of the discrete convolution has a sample throughput of 81003.69 samples per second, while the discrete convolution that uses an optimized kernel to better utilize the GPU reaches a sample throughput of 179620.13 (Table 12.2). This shows that the optimization of the kernel has a huge impact on the overall performance.

The optimization of the transfer improves the throughput of the standard version by roughly 1000 samples per second (Table 12.2). This minor improvement does not really justify the increased latency in the engine. While memory transfer optimizations are often recommended as an optimization the minor effect can be explained by the design choices. The convolution is completely executed on the GPU and only the I/O buffer is transferred during the processing. In this case, only 2048 bytes are transferred between CPU and GPU. This requires an insignificant amount of time compared to the actual convolution. The only way that the transfer optimization could be viable is to drastically shorten the length of the filter. The time for the convolution would decrease and therefore

Engine Name	Processing Time (ms)		avg. Sample Troughput(SpS)
	average	minimum	
Discrete	3.16035	2.96224	81003.69
Discrete (transfer opt)	3.11782	2.94745	82108.65
Discrete (kernel opt)	1.42523	1.35358	179620.1315

Table 12.2: Comparison between the different implementations for the discrete convolution on the GPU with OpenCL

Engine Name	Processing Time (ms)		avg. Sample Troughput(SpS)
	average	minimum	
Unpartitioned (opt)	1.69001	1.64549	151478.4
Unpartitioned (AVX)	1.64539	1.60413	155586.2
Partitioned (opt)	0.0956714	0.092372	2675826
Partitioned (AVX)	0.027986	0.026263	9147431

Table 12.3: Comparison between the different implementations of the fast convolution on the CPU

the proportion of the transfer at the total processing time would increase. But it would be far more likely that in that case, the convolution on the CPU is the better option.

12.2.3 Unpartitioned and Uniformly-Partitioned Convolution

The implementation of the unpartitioned and the partitioned convolution for the CPU show the same pattern as the discrete convolution (Table 12.3). The AVX implementation is always faster, but the gain in sample throughput is very different. While the sample throughput for the unpartitioned convolution is 151478.4 samples the AVX version has slightly better 155586.2 SpS, roughly two percent faster. The effect of the usage of AVX on the partitioned convolution is more noticeable. The sample throughput is increased from 2675826 SpS to 9147431 SpS, more than three times as much.

The reason why the unpartitioned convolution is effected by AVX to a much lesser extent is that the unpartitioned convolution uses a larger proportion of the processing time on the Fourier transforms while the partitioned convolution mostly spends its time on complex multiplication. The usage of AVX is limited to the multiplication of the complex numbers, because the Fourier transforms are provided by the FFTW library.

That the unpartitioned convolution is not worthy for further examinations can be seen when comparing the performance between the unpartitioned convolution and the par-

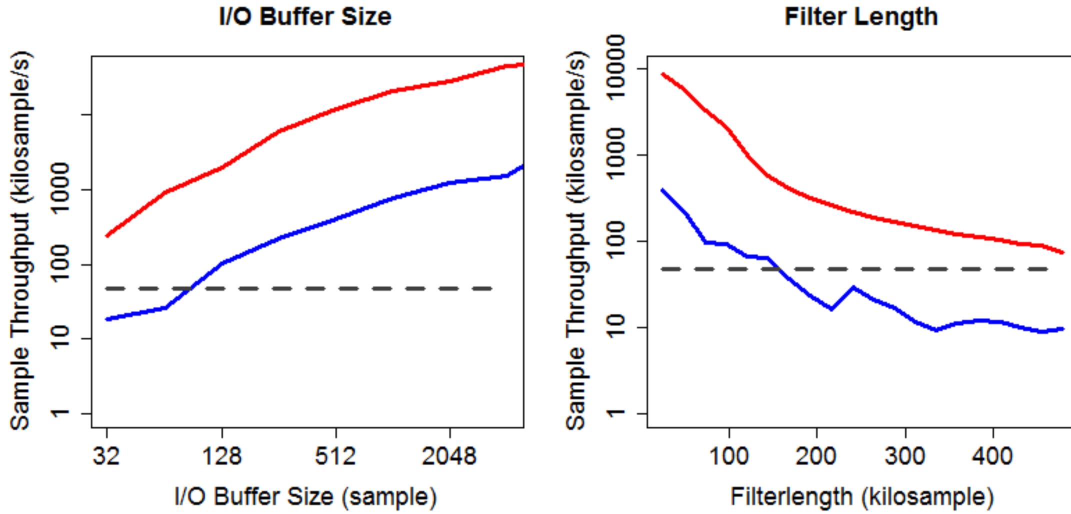

Figure 12.1: Sample throughput for the unpartitioned convolution and the partitioned convolution for different parameters for the convolution. In blue the unpartitioned convolution, in red the uniformly-partitioned convolution. The default filter length is 48000 samples and a I/O buffer size of 256 samples

titioned convolution under different parameters (Fig. 12.1). The partitioned performs always better than the unpartitioned convolution even for larger buffer sizes. This is, because the FFT transform is much lower for the partitioned convolution than for the unpartitioned convolution. The transform size for the partitioned is twice the I/O buffer size while the unpartitioned convolution has the filter length + buffer size - 1. The drawback is that the number of complex multiplications for the partitioned convolution is twice the filter length compared to again filter length + buffer size - 1. There are cases where the unpartitioned convolution performs better than the partitioned convolution, but these cases require an I/O buffer size that is greater than the filter length, but that is rarely the case.

12.2.4 Uniformly-Partitioned Convolution with OpenCL on GPU

With OpenCL only the uniformly-partitioned convolution was implemented, because test data from the CPU implementations clearly shows that the unpartitioned convolution is inferior to the uniformly-partitioned convolution. The optimizations for the code are only minimal, because the most complex part is handled by the library, meaning the FFT, and the complex multiplication cannot be optimized to a similar degree as the discrete

Engine Name	Processing Time (ms)		avg. Sample Troughput(SpS
	average	minimum	
Partitioned	0.647911	0.4398	395116
Partitioned (opt)	0.422281	0.334169	606231,4

Table 12.4: Comparison between the different implementations for the partitioned convolution on the GPU

convolution. Nonetheless one unoptimized and an optimized version were implemented. The code between them is mostly shared only the kernel code for the complex multiplication differs. The unoptimized version reaches only 2/3 of the sample throughput of the optimized version with 395116 SpS compared to 606231 SpS. The only difference between the implementations is that the unoptimized version uses a modulo operation which is replaced in the (Fig.12.4) optimized version.

12.3 Performance comparison between CPU and OpenCL/GPU Implementations

The test between the different implementations of the algorithm shows that some implementation of the same algorithm is clearly superior. Since testing a convolution engine can require a substantial amount of time only the implementation that makes best use of the computing power of their device are compared. The AVX implementation performed far better than the implementation using only code optimization, therefore the AVX multithreaded variants are selected for the CPU algorithms. From the OpenCL implementations the kernel optimized version showed the best performance for the discrete convolution, and for the partitioned convolution the optimized version was the best.

The problem for the comparison is that the convolution engine has three parameters the I/O buffer size, the length of the filter and the number of audio channels. Testing every possible configuration would require too much time. Because of this, the behavior of the different convolution engine implementations is compared when only one parameter changes while the others stay fixed. The default parameters are:

- I/O Buffer Size: 256
- Filter Length: 48000
- Number of Channels: 4

That the number of channels has a default of four is, because of the way multithreading is implemented. To make full use of the CPU the number of threads must be equal to the number of cores, in this case, four. The number of times the execution time of the processing function is measured is 10000.

12.3.1 I/O Buffer Size

The first engine parameter tested is the I/O buffer size. While theoretically, the size of the I/O buffer does not matter the performance of the discrete convolution the sample throughput on the GPU changes with the I/O buffer size (Fig. 12.2). The lowest sample throughput for the discrete convolution on the GPU is with 40 thousand SpS at an I/O buffer size of 32 samples too low for real-time processing. With increasing buffer size the sample throughput of this convolution engine increases through to better efficiency of the memory transfer. The sample throughput peaks at a buffer size of 2048 with a throughput of 1524 kSpS. On the CPU the sample throughput is in comparison relatively constant. The peak is at an I/O buffer size of 512 with 175 kSpS and at its lowest at a buffer size of 4096 with a throughput of 136 kSpS. A comparable performance between the two convolution engines is at a buffer size of 128 where the CPU version has with 172 kSpS a slightly higher throughput than the OpenCL version with 155 kSpS.

For the partitioned convolution the behavior is slightly different (Fig. 12.2). Like the discrete convolution the OpenCL implementation of the partitioned convolution starts lower than the CPU implementation with a sample throughput of 52 kSpS, but unlike the discrete convolution the OpenCL implementation is not able to surpass the CPU implementation. The CPU implementation starts with a sample throughput of 295 kSpS, a value that is surpassed by the OpenCL implementation at a buffer size of 128 samples. The throughput of both implementations increases with increases in the I/O buffer size. At the last measured buffer size of 4096 the OpenCL implementation has a throughput of 6.6 mSpS, and the CPU version an throughput of 45 mSpS.

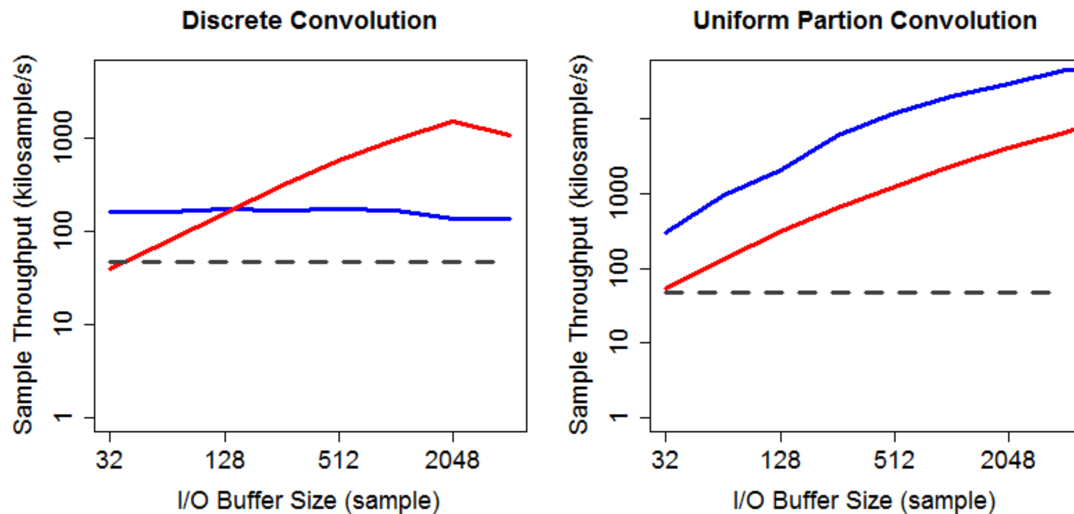

Figure 12.2: Sample throughput of the convolution engines by varying size of the I/O buffers. Start value for the buffer size is 32 samples, end 4096. Measurement points were all power of twos in between. The CPU implementations in blue, the OpenCL implementations in red. In grey the 48 kHz threshold for processing audio in real-time for the common sample rates

12.3.2 Channel Number

The next parameter to be examined is the number of parallel convolutions. The plots 12.4 show the behavior of the convolution engines when the number of channels is increased. The tests start at four channels, are incremented in four channels steps and finally ends at a channel number of 48. All convolution engines start with a higher sample rate than 48kHz. No engine with the exception of the CPU discrete convolution fall below this threshold, but the CPU partitioned convolution is at a channel number of 48 only slightly above with a throughput of 51.5 kSpS. The CPU discrete convolution falls under the 48 kHz threshold when convolving 12 channels. The sample throughput at this point 46,8 kSpS, rough than a third of the sample throughput for four channels 153.5kHz. This sample throughput is low for a sample rate of 48 kHz, but sufficient for convolving 12 channels at a sample rate of 44.1 kHz.

The sample throughput of the convolution engines decreases with increasing number of channels for the OpenCL implementations in a somewhat linear fashion. On the other hand the CPU partitioned convolution first drastically loses performance and then seemingly slows down. The sample throughput of this engine decreases fast from 5.92

mSpS to 585 kSpS at a channel size of 12, the break-even point between the CPU and the OpenCL version. This behavior is found on the other implementations as well, with the exception of partitioned convolution with OpenCL. The cause for this is that doubling of the problem size means roughly halving of the sample throughput. That this is not the case for partitioned convolution shows the strange behavior of GPUs.

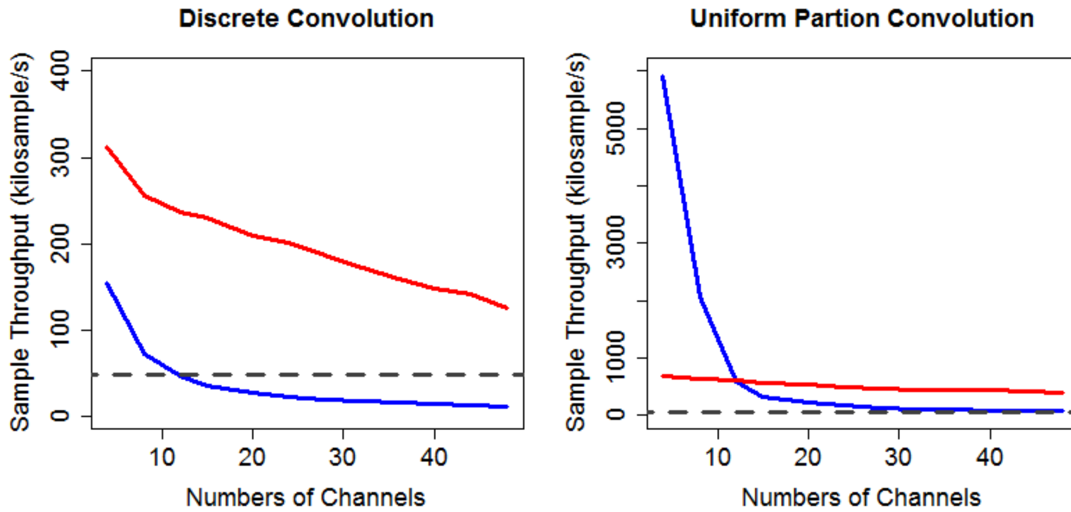

Figure 12.3: Sample throughput of the convolution engines by a varying number of channels for the convolution. The start number of channels is 4, and the end 48. Test were executed between start and end in increments of four. The CPU implementations in blue, the OpenCL implementations in red. In grey the 48 kHz threshold for processing audio in real-time for the common sample rates

12.3.3 Filter Length

The last parameter to be examined is the length of the filter. The test for the filter length is divided into two parts; one with the range of 24000 to 480000 (0.5s - 10s at 48 kHz) (Fig. 12.4), the second to find out if the partitioned convolution is better than the discrete convolution for small problem sizes. (Fig. 12.5). The former is the test between the different engines, the latter if there is a case where discrete convolution is faster than the partitioned convolution.

The plots in Figure 12.4 show that the engine behaves similarly in regard to the filer length as to the number of channels. The OpenCL implementation of the discrete

convolution is yet again always better than the CPU implementation, but this time both end up under the 48 kHz threshold. The CPU engine after a filter length of 144000 samples, the OpenCL engine at 288000 samples. Coincidentally the OpenCL discrete convolution engine reaches twice as many samples.

The CPU partitioned convolution outperforms the OpenCL version, but breaks even with it between 144000 and 168000 samples. While the OpenCL implementation outperforms the CPU implementation the CPU has still a sample throughput of 84 kSpS at a filter length of 480000. The point when the CPU version falls below it is reached far later, outside of the scope of the plot at a filter length of around 720000 samples. The limits of the OpenCL implementations lay somewhere after 5760000 samples. At this point, the sample throughput is still 50000 SpS. This filter length is equal to 120 seconds of audio at a sample rate of 48 kHz.

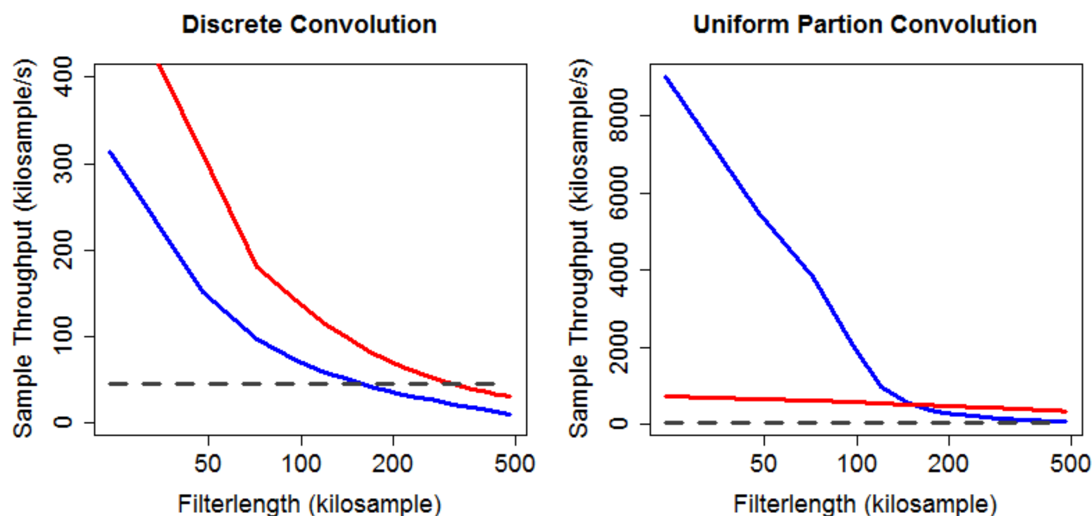

Figure 12.4: Sample throughput of the convolution engines by varying filter length. The start filter length is 24000, the end 480000. Tests were executed between start and end in increments of 24000. The CPU implementations in blue, the OpenCL implementations in red. In grey the 48 kHz threshold for processing audio in real-time for the common sample rates

That the partitioned convolution is not only good for long filter length shows another plot displaying the sample throughput for smaller filter lengths for the CPU implementations (Fig. 12.5). As is always the case the partitioned convolution outperforms the discrete convolution. The value range of the discrete convolution is between 25000 kSpS and 322 kSpS, while the value of the partitioned convolution ranges from 7250 kSpS to 1686

kSpS. This clearly shows that the partitioned convolution also handles relatively small filter lengths far better than the discrete convolution.

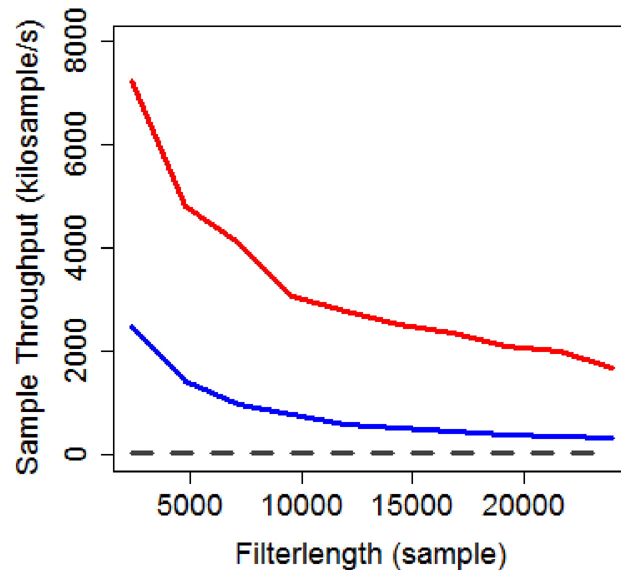

Figure 12.5: Sample throughput of the convolution engines by varying filter length. The start filter length is 2400, the end 20600. Tests were executed between start and end in increments of 2400. The size of the I/O buffers is in this case 64. The discrete convolution in blue, the partitioned convolution in red. In grey the 48 kHz threshold for processing audio in real-time for the common sample rates

12.4 Performance of Filter Changing

For an interactive environment it is necessary to change the filter during runtime. How long an engine needs to replace the filter depends on the algorithm. Each algorithm has only one implementation for the filter changed per processor. Testing involves therefore far less different implementations.

The discrete convolution does not require preprocessing of the filter. It is therefore unsurprising that the discrete convolution needs the shortest amount of time to change the filter. On the CPU the change of a single filter needs $10\mu s$ (Tab.12.5). The GPU

version need more than 10 times as long with $127\mu s$. Nearly all of the time the GPU needs longer is attributed to the transfer between the memories over the PCI-E bus.

The fast convolution algorithm require more time to replace the filter. Unpartitioned convolution needs $891\mu s$ to execute the FFT and change the filter. The partitioned convolution has to execute one transform for every partition, but despite this the replacement is faster with $231\mu s$. The main reason is that the Fourier transforms are much smaller.

The partitioned convolution is much slower with a replacement time of 32ms. The main reason is that unlike the processing changing of a filter is not optimized. In the current implementation the filter for the channel as well as the partitions are processed sequentially. A parallel solution would be better suited for the GPU. An estimation value when properly optimized would be 360μ. The estimation is based on the replacement times of the discrete GPU convolution and the partitioned CPU convolution. With proper optimization the time should be below the estimation value, but it will always take longer to replace a filter for partitioned GPU convolution than with the discrete GPU convolution.

	Filter Changing Time (ms)	
Algorithm	CPU	OpenCL/GPU
Discrete	0.01	0.127
Unpartitioned	0.891	-
Partitioned	0.231	32.5842

Table 12.5: Time needed to change a 48000 samples filter for different algorithms

13 VST Convolution Plug-In

The VST plug-in is a proof of concept, that the theoretical real-time processing of the engines works in practice. The plug-in was created with JUCE. JUCE is a cross-platform C++ framework for developing different kinds of audio plug-in types. The plug-ins in JUCE are created by implementing an interface. The framework than use wrapper for a specific plug-in type to call the functions from the interface. The convolution plug-in itself is mostly a wrapper around the convolution engine interface with a GUI (Fig 13.1). The GUI is used to set the parameters of the engine. The only parameters not controlled by the GUI are the size of the I/O buffers and the number of channels. These parameters are controlled by the DAW.

GUI Elements:

1. Selection of a convolution engine out of a list
2. Setting of the filter length used by the convolution algorithm
3. Setting of a parameter for the normalization
4. Maximum number of threads used for the convolution
5. Selection of a platform for running OpenCL
6. Selection of the device for OpenCL to run on
7. Selection of the channel for setting the filter response
8. Setting of the filter response either through browsing, entering the path or drag and drop
9. Toggle box for rendering of the filter response. Rendering of longer audio signal for the preview requires a noticeable amount of time
10. Button for applying or reverting changes of the parameters

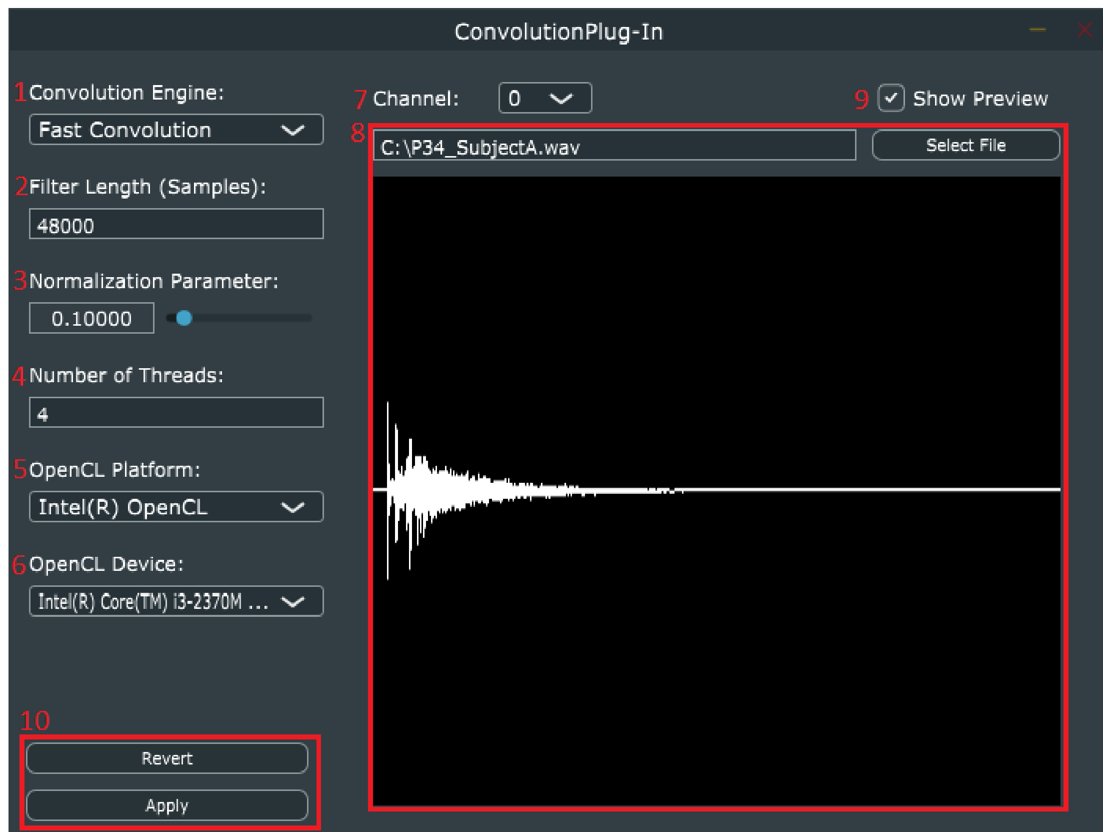

Figure 13.1: Screenshot of the plug-in GUI

14 Conclusion and Outlook

In conclusion, the discrete and the partitioned convolution can be implemented with OpenCL, but this does not necessarily mean better performance. If the OpenCL implementation is better than the CPU implementation depends on the convolution parameters, with OpenCL generally being better for larger problem sizes. For the discrete convolution, the main contributing parameter is the size of the I/O buffers. For the conducted tests the break-even point was around 128 samples. While the discrete convolution with OpenCL is better in nearly every tested case the break-even point between the two partitioned convolutions is more complex. The break-even point depends on the number of channels and the filter length and only to a minor extent on the I/O buffer size. While the partitioned convolution on the GPU with OpenCL can convolve the most channels / longest filter lengths, the actual use case may not require the performance.

A performance comparison between the implementations is always difficult. How well the convolution engine performs is highly dependent on the hardware used. A comparison between the result of other papers is hard, because similar hardware is not available and they often use different metrics to evaluate the performance. As mentioned before an additional problem is that often implementation details for the comparison algorithms are missing when comparing CPU and GPU implementations. As shown the optimization can make huge differences in the performance. In case of the discrete convolution, simple optimization of the code can reduce the processing time to 5% of the original time. When implementing reference implementation on the CPU for comparison, it is necessary to use AVX if possible. Otherwise only a fraction of the processing power of the CPU is used. The implementation of the discrete convolution as well as the partitioned convolution are more than twice as fast. The code for the CPU is not the only one that requires optimization. The performance difference an optimized kernel can make is significant. On the other hand optimization of the memory transfer can be neglected. PCI-E offers a high bandwidth and at least in this case it is unlikely that the memory transfer has a huge effect on the total execution time.

On an algorithmic basis there are two interesting aspects. First, fast convolution is nearly always worse than partitioned convolution even for larger I/O buffer sizes. Second, that partition convolution is better than discrete convolution smaller problem sizes, at least in the tested cases. The only redeeming aspect of discrete convolution is the shorter time it needs to replace the filter.

Which engine to select depends on the use cases. Discrete convolution is easy to implement but does not offer a high performance. Partitioned convolution requires third party libraries or good implementations of the FFT. The OpenCL implementation have the problem that the OpenCL 2 standard is not universally supported.

There are several steps that can be taken next. One is to implement a convolution engine using non-uniformly-partitioned convolution. Another is to make use of the portability of OpenCL to test the convolution engines on different devices like FPGAs and DSPs. A very interesting aspect would be if the optimization for the kernel code has the same effect for other devices. The test could be repeated for different platforms, but an important question is how to properly test something that is so reliable on hardware and the operating system, what metric to use, and how to handle the multitude of parameters for the test. Another option unexplored is the performance of integer arithmetic. While convolution with integer arithmetic is more complex due to possible over- and underflows, the arithmetic operations on integers is faster than on floating point.

15 Appendix

15.1 Convolution Engine Overview

A full list of all factory functions in the dynamic libraries with the name of the function to create the engine and the technologies:

DiscreteConvolutionEngines:

Function name	Overlap Add/Save	Paramters
createEngine	Save	
createOptimizedEngine	Save	
createMultithreadedEngine	Save	nrOfThreads
createAVXEngine	Save	
createMultithreadedAVXEngine	Save	nrOfThreads

DiscreteConvolutionEnginesOpenCL:

Function name	Overlap Add/Save	Paramter
createEngine	Save	platformId, deviceId
createTransferOptimizedEngine	Save	platformId, deviceId
createKernelOptimizedEngine	Save	platformId, deviceId
createFullOptimizedEngine	Save	platformId, deviceId

FastConvolutionEngines:

Function name	Overlap Add/Save	Paramter
createEngine	Add	
createAVXEngine	Add	
createMultithreadedEngine	Add	nrOfThreads
createMultithreadedEngineAVX	Add	nrOfThreads
createUniformEngine	Save	nrOfThreads
createUniformEngineAVX	Save	nrOfThreads

FastConvolutionEnginesOpenCL:

Function name	Overlap Add/Save	Paramter
createEngine	Save	platformId, deviceId
createOptimizedEngine	Save	platformId, deviceId

15.2 Test Procedure File Variables

For easier use custom highlighting for notepad++ was created. To use the highlighting it must be imported from the ScriptFileHighlighting file. Every name is case sensitive.

15.2.1 Set Variable

```
typeId global_settings::varId = value
```

typeId

- `bool`
- `int`
- `float`
- `string`

Bool Variables

Set if a given test scenario is tested. The compare bools check the result of a test scenario. This is only needed when testing a new implementation of a convolution algorithm to ascertain that the convolution was performed correctly.

- `ImpulsConvolution`
- `ImpulsConvolutionCompare`
- `SquareSignalConvolution`
- `SquareSignalConvolutionCompare`

Integer Variables

- `NrOfTestRuns`: number of test runs for the loop test

Float Variables

All float values are the normalization value for the test scenarios to prevent clipping.

- `Normalisation_Impulse`
- `Normalisation_Square`
- `Normalisation_Real`

String Variables

- `ImpulseFileName`: name prefix for the result of the impulse test
- `SquareFileName`: name prefix for the result of the square signal test
- `SummaryName`: name for the summary file

- `ImpulseInputPath`
- `ImpulseImpulsePath`
- `ImpulseOutputPath`
- `ImpulseComparePath`

- `SquareInputPath`
- `SquareImpulsePath`
- `SquareOutputPath`
- `SquareComparePath`

- `...InputPath`: audio input file for the convolution
- `...ImpulsePath`: impulse response for the convolution
- `...OutputPath`: saving location for the result of the convolution
- `...ComparePath`: audio file to validate the result of the convolution

Accepted Boolean Values

- `true`
- `True`
- `TRUE`
- `false`
- `False`
- `FALSE`

Bibliography

[1] Advanced Micro Devices Corporation (Veranst.): *OpenCL User Guide*. August 2015

[2] Belloch, Jose A. ; Gonzalez, Alberto ; Martínez-Zaldívar, F. J. ; Vidal, Antonio M.: Real-time massive convolution for audio applications on GPU. In: *The Journal of Supercomputing* 58 (2011), Dec, Nr. 3, S. 449–457. – URL https://doi.org/10.1007/s11227-011-0610-8. – ISSN 1573-0484

[3] Bosi, Marina ; Goldberg, Richard E.: *Introduction to Digital Audio Coding and Standards*. Norwell, MA, USA : Kluwer Academic Publishers, 2002. – ISBN 1402073577

[4] Brandtsegg, Øyvind ; Saue, Sigurd: Live Convolution with Time-Variant Impulse Response. (2017)

[5] Burgmeyer, Ralf: *Implementierung von Impulsantworten in die Wellenfeldsynthese-Anlage der HAW Hamburg*. 2013

[6] Fang, J. ; Varbanescu, A. L. ; Sips, H.: A Comprehensive Performance Comparison of CUDA and OpenCL. In: *2011 International Conference on Parallel Processing*, Sept 2011, S. 216–225. – ISSN 0190-3918

[7] Frigo, M. ; Johnson, S. G.: The Design and Implementation of FFTW3. In: *Proceedings of the IEEE* 93 (2005), Feb, Nr. 2, S. 216–231. – ISSN 0018-9219

[8] Gregg, C. ; Hazelwood, K.: Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In: *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. New York, NY, USA : ACM, April 2011, S. 134–144. – URL http://doi.acm.org/10.1145/2683405.2683424. – ISBN 978-1-4503-3184-5

[9] Hennessy, John L. ; Patterson, David A.: *Computer architecture: a quantitative approach*. Elsevier, 2011

[10] Intel Corporation (Veranst.): *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. May 2018

[11] Jillings, N. ; Reiss, J. D. ; Stables, R.: Zero-Delay large signal convolution using multiple processor architectures. In: *2017 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, Oct 2017, S. 339–343. – ISSN 1947-1629

[12] Lester, Michael ; Boley, Jon: The Effects of Latency on Live Sound Monitoring. In: *Audio Engineering Society Convention 123*, URL http://www.aes.org/e-lib/browse.cfm?elib=14256, Oct 2007

[13] Munshi, Aaftab ; Gaster, Benedict ; Mattson, Timothy G. ; Ginsburg, Dan: *OpenCL programming guide*. Pearson Education, 2011

[14] Nikolov, D. V. ; Mišić, M. J. ; Tomašević, M. V.: GPU-based implementation of reverb effect. In: *2015 23rd Telecommunications Forum Telfor (TELFOR)*, Nov 2015, S. 990–993

[15] Nvidia Corporation (Veranst.): *OpenCL Programming Guide for the CUDA Architecture*. August 2009

[16] Primavera, A. ; Cecchi, S. ; Piazza, F. ; Li, J. ; Yan, Y.: Hybrid Reverberator Using Multiple Impulse Responses for Audio Rendering Improvement. In: *2013 Ninth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, Oct 2013, S. 314–317

[17] Savioja, Lauri ; Välimäki, Vesa ; Smith, Julius O.: Audio Signal Processing Using Graphics Processing Units. In: *J. Audio Eng. Soc* 59 (2011), Nr. 1/2, S. 3–19. – URL http://www.aes.org/e-lib/browse.cfm?elib=15772

[18] Schoeffler, Michael ; Hess, Wolfgang: A Comparison of Highly Configurable CPU- and GPU-Based Convolution Engines. In: *Audio Engineering Society Convention 133*, URL http://www.aes.org/e-lib/browse.cfm?elib=16615, Oct 2012

[19] Smith, Steven W. u. a.: The scientist and engineer's guide to digital signal processing. (1997)

[20] Valimaki, V. ; Parker, J. D. ; Savioja, L. ; Smith, J. O. ; Abel, J. S.: Fifty Years of Artificial Reverberation. In: *IEEE Transactions on Audio, Speech, and Language Processing* 20 (2012), July, Nr. 5, S. 1421–1448. – ISSN 1558-7916

[21] Wefers, F.: *Partitioned convolution algorithms for real-time auralization.* Logos Verlag Berlin, 2015 (Aachener Beitrage zur Technischen Akustik). – URL https://books.google.de/books?id=IA-bCgAAQBAJ. – ISBN 9783832539436

[22] Wefers, Frank ; Berg, Jan: High-performance real-time FIR-filtering using fast convolution on graphics hardware. In: *Proc. of the 13th Conference on Digital Audio Effects*, 2010

[23] Wefers, Frank ; Vorlaender, Michael: Optimal filter partitions for real-time FIR filtering using uniformly-partitioned FFT-based convolution in the frequency-domain. In: *Proceedings of the 14th International Conference on Digital Audio Effects, DAFx 2011* (2011), 01

[24] Zölzer, Udo: *DAFX: digital audio effects.* John Wiley & Sons, 2011

Glossary

clFFT The clFFt library is a well optimized library for executing the fast Fourier transform with OpenCL. https://github.com/clMathLibraries/clFFT.

FFTW The FFTW (Fastest Fourier Transform in the West) is a cross-platform C library to compute the discrete Fourier transform. It was developed by Matteo Frigo and Steven G. Johnson at the Massachusetts Institute of Technology and is published under the GNU General Public License. http://www.fftw.org/.

JUCE JUCE is a cross-plaform C++ application framework, developed by ROLI, mainly used for creating audio plug-ins. It is not only cross-platform but also allows to build different kinds of audio plug-ins, like VST and AZ. https://juce.com/.

libsndfile libsndfile is a cross-platform C library for reading or writing audio data into different audio formats through a uniform interface. Developt by Erik de Castro Lopo and published under the GNU Lesser General Public License. http://www.mega-nerd.com/libsndfile/.

OpenCL Open Computing Language is a cross-platform framework for executing code on CPU, GPU, DSP and FPGA. The API is developed by the Khronos Group https://www.khronos.org/opencl/.

PortAudio PortAudio is an open-source cross-platform C library for accessing the audio inputs and outputs of a computer. The library support audio driver like Jack or ASIO. http://www.portaudio.com/.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: ______________________________

Vorname: ______________________________

dass ich die vorliegende Mastertarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Implementierung und Evaluierung von Optimierungsstrategien für die Faltung von Audio Signalen

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original