Evaluation of Data Transfer Methods for Block-based Realtime Audio Processing with CUDA

Christoph Kuhr*, Alexander Carôt† Department of Computer Sciences and Languages, Anhalt University of Applied Sciences
Köthen
Email: *christoph.kuhr@hs-anhalt.de, †alexander.carot@hs-anhalt.de

***Abstract*—Realtime audio production environments generally do not use GPUs, as long as they are not involved in 3D rendering or video production processes. Thus, the GPU is idle most of the time and can be utilized as an audio co-processor. The block-based streaming nature and floating point representation of computer audio hardware are very well suited for GPGPU programming techniques. In this paper we line out the data transfers as the most expensive part in the processing of realtime audio data and evaluate different data transfer methods and positively evaluate different data transfer methods with respect to future audio DSP applications.**

I. Introduction

Modern computer systems are equipped with a CPU and a GPU. CPUs control the peripheral hardware and perform calculations unrelated to 3D graphics or video decoding. A GPU in contrast is concerned with rendering 3D graphics or utilizing special hardware codecs to decode nowadays video codes like H264 [1].
If a computer system is used for any kind of audio production, that excludes 3D rendering and video decoding, the GPU is mostly idle. Additionally, GPUs are designed to handle multiple floating point operations at the same time in a threaded fashion.
These considerations promote the idea to use a GPU as an audio co-processor for signal processing purposes.
Computation intensive audio signal processing of realtime data has already been done, e.g. Wefers and Berg have used a GPU to process FIR and IIR filters [2], Jedrzejewski and Marasek have used the GPU to do impulse response computations for virtual room acoustics [3].

In this paper we will investigate the lower limit for the usage of a GPU for such signal processing tasks in a realtime audio production environment. The limit is given as the combination of channel count and sample buffer size in use. The bottlenecks in the communication between CPU and GPU are evaluated and discussed. Further, possible workarounds to increase the performance aspects under investigation are proposed and evaluated.

CUDA (Compute Unified Device Architecture) is a programming langauge designed for high-performance computing [4]. The idea is to make use of thousands of threads running in parallel, which is not possible with x86/x86_64 CPUs. Such parallel programs are called kernel in the CUDA domain.
When a kernel is executed on the GPU, the kernel launches a grid of several blocks, the limit is depending on GPU features. Inside each block on the grid, multiple threads execute the actual computations at runtime. The same computation runs on each thread, but with different data. Threads can be handled in a synchronous or an asynchronous way. The latter requires the concept of streams for a destinct mapping of the data shared between the threads of one block. The structure of CUDA computing grids is shown in fig. 1.

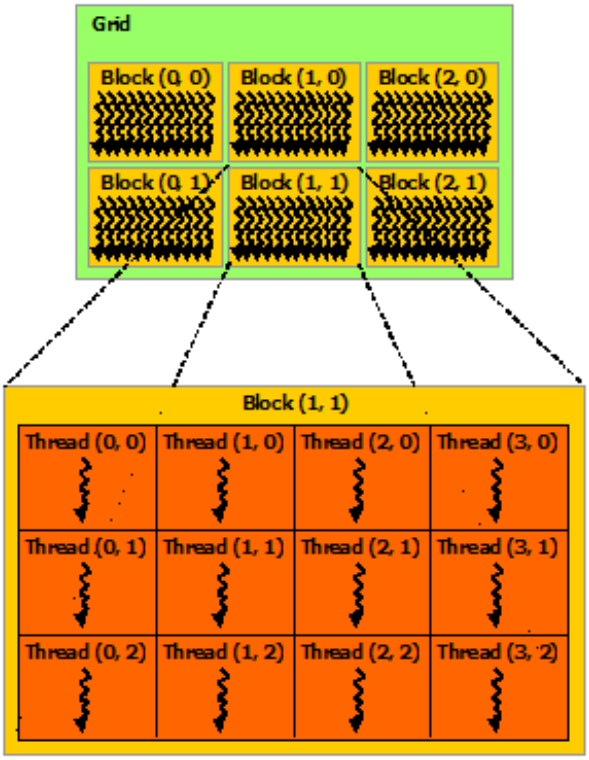

Figure 1: *CUDA Computing Grids [5]*

The concept of CUDA streams [6] is very convenient for the problem at hand.

Different audio streams can be treated asynchronously, which is a better representation of their orthogonal nature then a matrix with an appropriate amount of rows and columns. This way the orthogonality may also be represented appropriately, but access to the matrix would be centralized and would experience possible racing conditions. Beyond, using a dimension (x, y or z) for the representation of the different audio channels, reduces the available dimensionality that is useable for calculations at runtime.

This paper is part of the research project fast-music [7]. The project has the goal to enable symphonic orchestras to rehearse via the public internet, by using the realtime communication software Soundjack [8] [9]. Research in the field of packet loss concealtment will use GPUs for complex signal processing based on machine learning algorithms.

II. Architecture

The work of Wefers and Berg [2] has also shown, that realtime processing of audio data with a GPU is possible.
The communication between CPU and GPU is realized via driver calls and shared memory, either DMA, GPU or CPU RAM. The CPU is also referred to as host and the GPU as device. Nowadays, system architectures where CPU and GPU share the same cache are used increasingly, albeit mainly in embedded systems. This architecture completely eleminates memory copies, since the memory is coherently accessible by the CPU and the GPU. In conventional systems which communicate via the PCIe bus, data has to be copied from CPU RAM to GPU RAM and back.
Since the API calls copying data between CPU and GPU have much overhead, it is more efficient to copy huge amounts of data. Thus, it is even more interesting to investigate the use case of small amounts of data, as generated and processed in the audio domain.

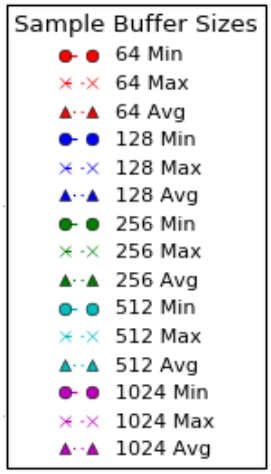

Figure 2: *Legend Data Transfer Method Measurements*

Realtime audio data is represented as a two dimensional vector field. At any sample point in time some analog digital converter process generates a sample, with typical bit depths of 16, 24 or 32 bits, either encoded as integer or floating point [10].
Computer audio hardware manages data by using buffers that consist of a predefined amount of samples. The audio driver repeatedly accesses the memory of the audio hardware and copies the sample buffers to the CPU RAM for further usage. The responsivness of such an audio system depends on the size of the sample buffers, while the response time reduces with an increasing sample buffer size. Typical sample buffer sizes are 64, 128, 256, 512, 1024 samples [11].

$$AudioDataBlock = SampleDepth \cdot SampleBufferSize \cdot ChannelCount$$

$$AudioDataBlock = 32bit \cdot \{64, 128, 512, 1024\} \frac{Samples}{s} \cdot \{2, 8, 16, 32, 64\}$$

Due to this block-based streaming nature, the data transfer and processing of audio data between CPU and GPU might reduce the impact of the data copying overhead, particularly if multiple audio channels are used.
The audio data, that we will transfer and process with the GPU, is provided by a professional audio driver and server combination called Jack Audio Connection Kit [12]. On top of a Linux ALSA [13] driver, Jack provides the means to interconnecting jack-aware audio software to the audio interface with 32 bit floating point precision. The floating point format requires the development of a prototype, because the Soundjack clients use an integer format instead of floating point and would require additional conversion.

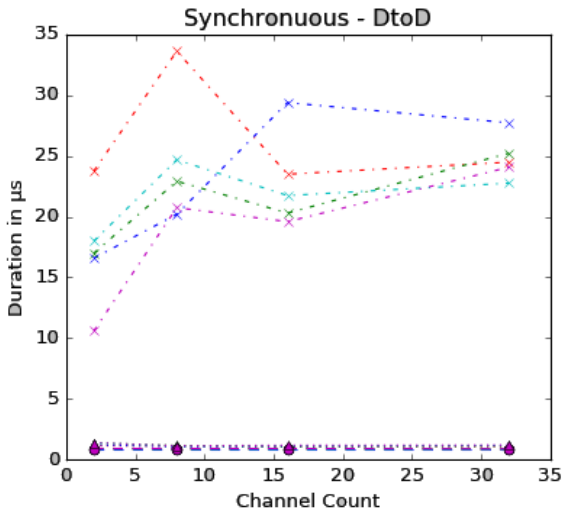

Figure 3: *Device to Device Copy Duration Synchronous Data Transfer Method*

We developed a most simple Jack client for testing purposes with varying channel counts and sample buffer sizes. The Jack client is linked against a shared library that provides the CUDA Kernel [4]. This way CUDA computations can be integrated in arbitrary C programs. The Jack Server configures the audio interface by utilizing the ALSA driver infrastructure. The most

important configuration parameters for our investigations are the channel count and sample buffer size, called frame or period in the Jack domain. At runtime, the Jack Server requests our Jack client to process a frame with a callback function. If the callback function is not done with its computations in time, the Jack Server reports a buffer underrun, also called xrun in the Jack domain.

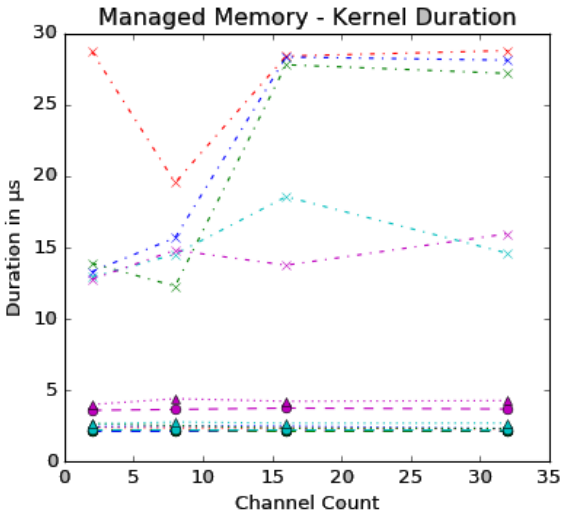

Figure 4: *Kernel Execution Duration Managed Memory Data Transfer Method*

A Nvidia Geforce GT940mx GPU with 2 GB of DDR3 RAM is connected to an Intel i7-6870 4-Core CPU with 16 GB DDR3 RAM via a PCIe x16 2.0 bus [14], in the system under test. Thus, the transfer rate between CPU and GPU is limited to the bus bandwidth of 8 GBps simplex.
The Nvidia Geforce GT940mx has a compute capability of 5.0 ($\geq$ 2.0), which allows it to use managed memory.

III. CUDA Memory Organization and Management

The data structure and data transfer between CPU and GPU are the bottlenecks for the entire signal processing. Three different data transfer methods can be used:

1) Synchronous data transfer

   A synchronous data transfer returns as soon as the memory operation on the GPU memory is done, with a success or failure result. For the GPU integration of synchronous data transfers, it is irrelevant whether the memory is pagable or pinned. Either type can be accessed. Pagable memory is memory from the virtual address space of CPU or the operating system.

2) Asynchronous data transfer

   An asynchronous data transfer returns immediately after invoking the data transfer, regardless of the result. The result of the operation has to be checked seperately. It requires the additional concept of streams for the integration on the GPU. Further, the host memory has to be pinned. Pinned memory addresses are allocated in the DMA address space of the host system.

3) Managed memory with coherent caches on CPU and GPU

   With managed memory, the requirement of memory copy operations is eliminated. The GPU driver allocates memory on the CPU and GPU respectively, manages any data access onto these memory segments implicitly and thus keeps the data in both memory locations coherent by small caching operations.

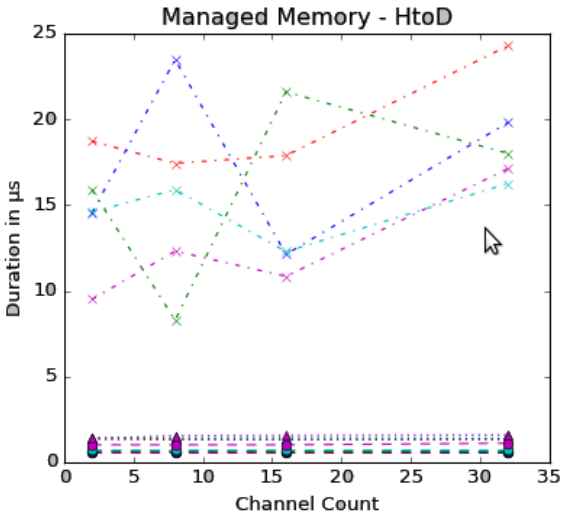

Figure 5: *Host to Device Transfer Duration Data Transfer Method*

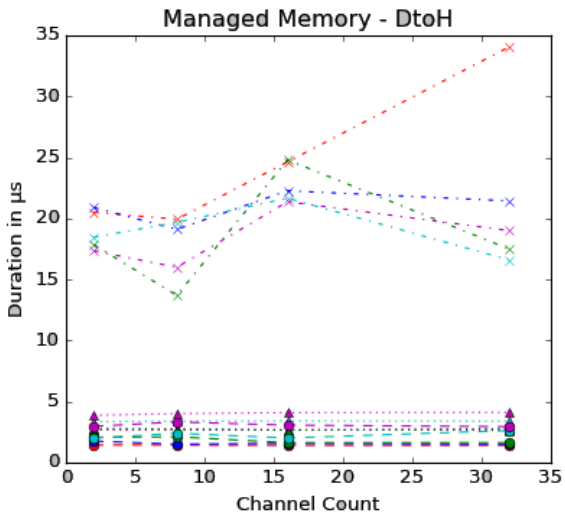

Figure 6: *Device to Host Transfer Duration Data Transfer Method*

The direction for data transfers is crucial as well. Three different directions are distinguished:

1) HostToDevice (HtoD or H2D)

   The *HostToDevice* mode utilizes the Direct Memory Access (DMA) memory of the host system. This enables the CPU to offload the data transfer operations to the GPU without waiting for the completion or result.

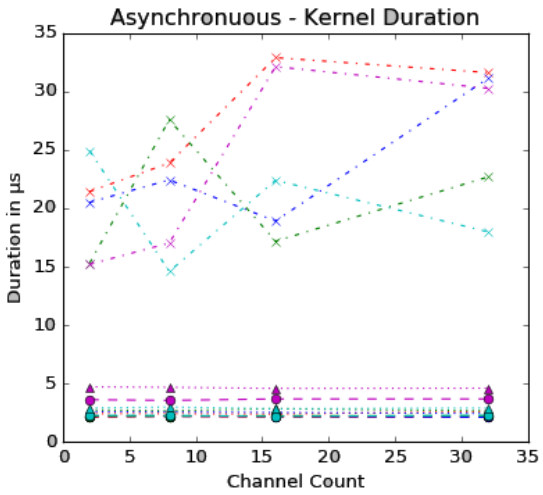

Figure 7: *Kernel Execution Duration Asynchronous Data Transfer Method*

2) DeviceToDevice (DtoD or D2D)

   Invoking CUDA memcpy between two GPUs uses memory copy operations between the RAM of both GPUs. If a D2D memory copy operation is issued on a single device however, the GPUs' internal cache is used for the data transfer.

3) DeviceToHost (DtoH or D2H)

   Although the *DeviceToHost* mode does not utilize the DMA memory it may also operate asynchronously, but slower since it is copied from GPU to CPU RAM.

IV. Experiments

We investigated the influence that the sample buffer size and channel count had on the data transfer rates. The audio channel count was varied between 2, 8, 16 and 32 channels, while each channel count was tested with each common sample buffer size of 64, 128, 256, 512 and 1024 samples per buffer. The samples were formatted as 32 bit floating point. A simple CUDA kernel is provided for an exemplary computation. Each thread in a block handles exactly one sample, copies it from the input to the output buffer. This way 64 up to 1024 threads run in parallel in a single block. The worstcase for the data transfer times, is given by the Jack servers buffersize and sample rate, which in this case is 48kHz (Sample Duration = $\frac{1}{48kHz} = 20.833\mu s$):

Sample Buffer Size	Worst Case Latency
64	1.334ms
128	2.667ms
256	5.334ms
512	10.667ms
1024	21.334ms

Table I: *Tolerable Worst Cast Latencies for Realtime Audio*

The profiling overhead of the NVidia Visual Profiler (NVVP) for 32 channels with 64 samples per buffer pushed the host machine to its limits. Thus, tests with 64 audio channels were omitted.

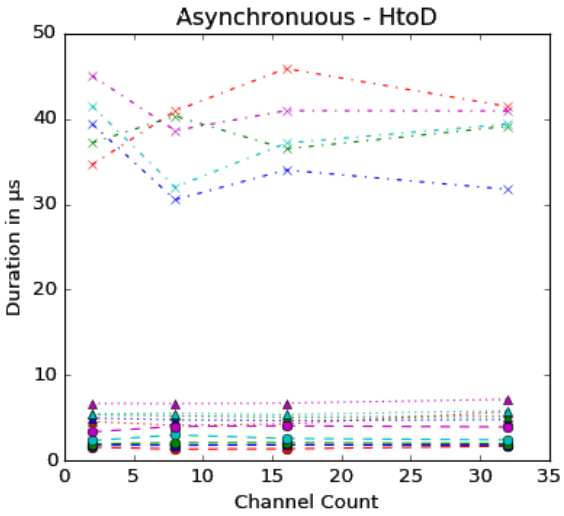

Figure 8: *Host to Device Transfer Duration Asynchronous Data Transfer Method*

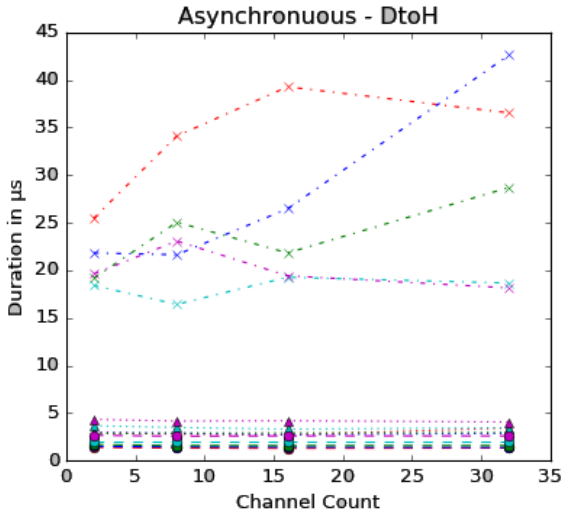

Figure 10: *Device to Host Transfer Duration Asynchronous Data Transfer Method*

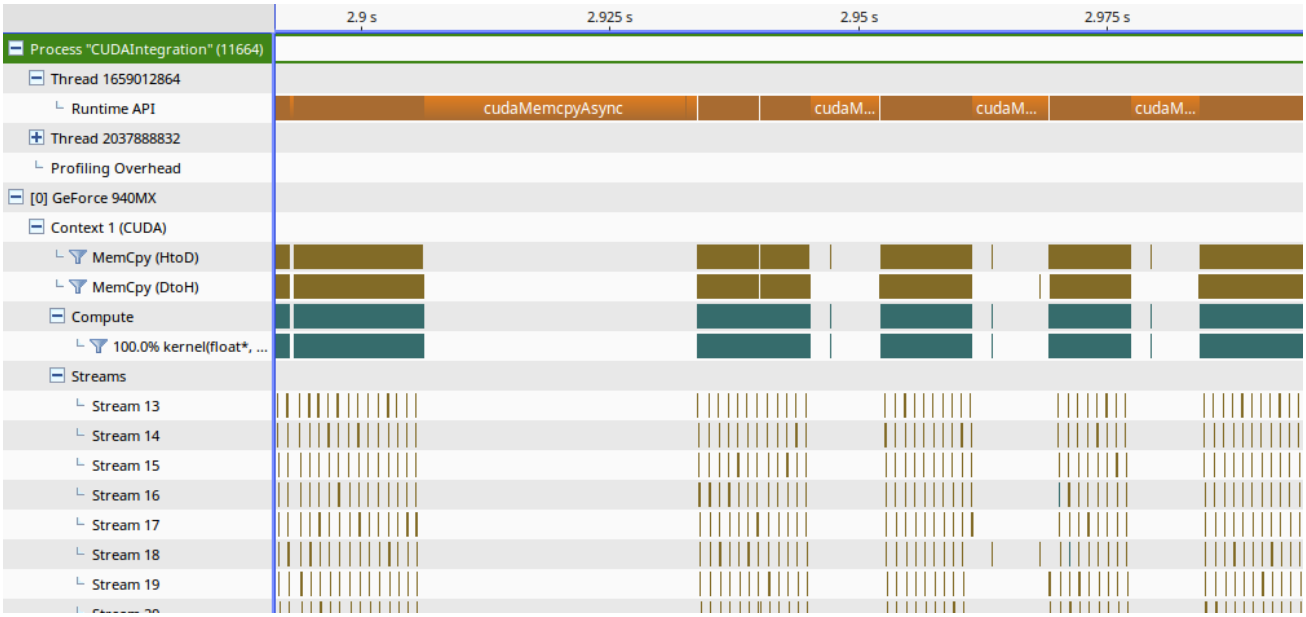

Figure 9: *NVVP Screenshot showing CUDA API Overhead*

V. Discussion

All combinations of transfer methods and modes, sample buffer sizes and channel counts take in average less then $10\mu s$ and show peaks of up to $46\mu s$, as visualized in fig. 4 to fig. 10. The visualized durations neglect the CUDA API and driver calls, they represent the execution on hardware only. The legend in fig. 2 is common for all figures.

A comparisson of fig. 5 and fig. 8 shows that the memory mapped H2D mode takes less time, at minimum, average and maximum then the asynchronous copy mode.
The kernel execution times for the two other transfer methods shown in fig. 4 and fig. 7, exhibit no significant difference. In fig. 3 only the device to device copy operation is shown, which does not involve any kernel launch. These findings suggest that the synchronous memory transfer method would also be suitable for the H2D copy mode. Since a kernel has to wait until all data is present in the GPU memory, it is of no consequence at this point, if the data is transfered synchronously or asynchronously. In contrast to the D2H mode, where a non blocking data transfer allows the processing chain to finish sooner. The magnitude of these savings is much lower then of the overhead introduced by the CUDA API and driver calls. This is observable in the rows below the CUDA Context in fig. 9, the three smaller gaps ($\approx 7ms$) on the right side and a larger gap ($\approx 28ms$) on the left side relate to the small chunks in the rows for the respective streams. These chunks are the hardware based memory operations as mentioned above and take only a few micorseconds in average.

All three memory organization modes exhibit a common problem of cyclic nature. At a given interval ($\approx 11s$ for pagable memory, $\approx 5s$ for pinned memory and $\approx 2.5s$ for managed memory) memory operations last approximately four times longer, resulting in the larger gap on the left side in fig. 9. These API and driver calls introduce jitter to the tested audio signal.

The turning point from where the CUDA API overhead is neglectable, can be quantified:

Channel Count	Sample Buffer Size
2	512
4	512
8	1024
16	1024
32	1024

Table II: *Channel Count and Sample Buffersize Limit for Realtime Audio Processing*

VI. Conclusions

All three memory transfer methods are able to operate on realtime audio data. Managed memory however is most convenient, because host and device pointers do not require any special handling and integrate smoothly into C code as well as CUDA code. For the usage with Jack however, two memory copy operations are still required, because Jack provides preallocated pointers to its buffer interface.
Low sample buffer sizes increase jitter, but no buffer underruns were detected. Although the duration of the CUDA API and driver calls suggest that underruns should occur with sample buffer sizes below 512 samples.

VII. Future Work

The evaluation of the data transfer method has been a feasability study for further goals. In the future, machine learning algorithms will be investigated in this environment as well as common signal processing algorithms, with respect to error concealment techniques and the generation of audio effects.

VIII. Acknowledgements

fast-music is part of the fast-project cluster (fast actuators sensors & transceivers), which is funded by the BMBF (Bundesministerium für Bildung und Forschung).

References

[1] *H.264: Advanced video coding for generic audiovisual services*, ITU-T Std. H.264, 2003.

[2] F. Wefers and J. Berg, "High-performance real-time fir-filtering using fast convolution on graphics hardware," in *Proc. of the 13th Int. Conference on Digital Audio Effects (DAFx-10)*. Graz, Austria: Institute of Technical Acoustics, RWTH Aachen University, Sep. 6–10, 2010, pp. DAFX–1 – DAFX–8.

[3] M. Jedrzejewski and K. Marasek, "Computation of room acoustics using programmable video hardware," in *Computer Vision and Graphics, Springer-Verlag Netherlands*. PJWSTK, 2006.

[4] *Getting Started with CUDA*, NVidia Corporation, 2008.

[5] *CUDA C PROGRAMMING GUIDE*, NVidia Corporation, 2013.

[6] *CUDA Streams, Best Practices and Common Pitfalls*, NVidia Corporation, Year unknown.

[7] (2017, Jun.) fast actuators, sensors and transceivers. [Online]. Available: https://de.fast-zwanzig20.de/

[8] (2017, Jun.) Soundjack - a realtime communication solution. [Online]. Available: http://http://www.soundjack.eu

[9] A. Carôt, "Musical telepresence - a comprehensive analysis towards new cognitive and technical approaches," Ph.D. dissertation, University of Lübeck, Germany, May 2009.

[10] A. V. Oppenheim and R. W. Schaefer, *Discrete-time signal processing*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, Inc., 1989.

[11] K. C. Pohlmann, *Principles of Digital Audio*, 5th ed. The Mcgraw-Hill Companies, 2005.

[12] (2017, Jun.) Jack audio connection kit. [Online]. Available: https://jackaudio.org

[13] (2017, Jun.) Advanced linux sound architecture. [Online]. Available: https://alsa-project.org/main/index.php/Main_Page/

[14] (2006, Dec.) Pci express base specification revision 2.0. PCI-SIG. [Online]. Available: https://members.pcisig.com/wg/PCI-SIG/document/download/8246