HIGH-PERFORMANCE REAL-TIME FIR-FILTERING USING FAST CONVOLUTION ON GRAPHICS HARDWARE

Frank Wefers,
Institute of Technical Acoustics,
RWTH Aachen University
Aachen, Germany
fwe@akustik.rwth-aachen.de

Jan Berg,
Institute of Technical Acoustics,
RWTH Aachen University
Aachen, Germany

ABSTRACT

In this paper we examine how graphic hardware can be used for real-time FIR filtering. We implement uniformly-partitioned fast convolution in the frequency-domain and evaluate its performance on a NVIDIA GTX 285 graphics card. Motivated by audio rendering for virtual reality, our focus lies on large-scale real-time filtering with a multitude of channels, long impulse responses and low latencies. Graphics hardware has already been used for audio signal processing — including FIR and IIR filtering with respect to offline and real-time processing. However, the combination of GPU computing and real-time conditions leads to a number of challenges that have not been reviewed in detail. The new contribution of this paper is an implementation and detailled analysis of a frequency-domain fast convolution method on GPUs. We discuss specific problems that emerge under real-time conditions. Our method allows to achieve an outstanding real-time filtering performance. In this work, we do not only regard a time-invariant filtering, but also time-varying filtering, where filters are exchanged during runtime. Furthermore, we examine the opportunities of distributed computation — using CPU and GPU — in order to maximize the performance. Finally, we identify bottlenecks and explain their impact on filter exchange latencies and update rates.

1. INTRODUCTION

Real-time filtering is a fundamental component in many audio applications. It is a part of audio effects software plugins (VST, DirectX) used for professional audio production, like EQs or convolution reverbs. It can be found in hardware controllers for speaker equalization. Room acoustics of auditoriums can be improved using real-time digital room correction, which adds an artificial reverb. But an application that really pushes real-time filtering to its limits is interactive audio rendering for virtual acoustic reality [1]. Here, virtual scenes which consist of a multitude of sound sources are auralized in real-time. This is done by filtering the audio signals of the sound sources with individual impulse responses (filters) that model the sound propagation through the scene. The objective is a high-quality acoustic image of the scene. Room acoustic effects — like reverberation — need to be simulated precisely. Users (listeners) shall be able to interact with the presented scene. Consequently, the sound propagation changes over time and filters need to be exchanged. Furthermore, reactions on user input (e.g. movement, rotation) must be reproduced instantaneously. Only very short processing delays are acceptable.

Signal processing in this domain is a massive task: A large number of channels (typically 10-100) has to be filtered with individual filters which are long (typically room impulse responses of 30.000 - 300.000 filter coefficients). Input-to-output latencies as well as latencies for the exchange of filters need to be very low (<20 ms). Despite the fact that recursive filters (IIR filters) require significantly less computational effort, they are not favored for interactive audio rendering including room acoustics. Today efficient fast convolution algorithms exist and very powerful hardware is available, which allows to perform FIR filtering for the auralization. While in former times such an extensive filtering could only be realized using specialized DSP boards, nowadays computer hardware is often used. Boosted by parallel hardware, like multi-CPU/multi-core machines, current computer systems are able to deliver the computation power for scenarios like introduced above. However, there is always the desire to create more realistic scenes, with more sound sources and more complex scene topologies, demanding more computational power for the real-time filtering. And of course the filtering might not be the only task running on the system. This is where graphic cards become interesting.

Modern graphic processing units (GPUs) easily outperform current multi-core CPUs by means of sheer floating-point performance. This is achieved by a massive level of inherent parallelism (several hundred individual stream processors (SPs) on a single graphic chip) and a more focussed and thereby simpler hardware architecture compared to generalpurpose CPUs. Using the GPU for general purpose calculations (popular by the synonym GPGPU) became possible with the arrival of programmable shaders [2]. Back then software development for graphic hardware was a tedious process. Today high-level programming interfaces (APIs), like NVIDIA's Compute Unified Device Architecture (CUDA) [3] and ATI's Stream technology [4] (formerly Close-To-Metal), make the development much easier. However, GPUs do not make CPU computing dispensable. The enormous computation power can only be unleashed, if algorithms meet the specific characteristics of the graphic hardware. Most suited for GPGPU computing are so-called *stream algorithms*, which perform the same set of operations on large data sets. Real-time FIR filtering falls into this class of algrithms. It has a high potential for parallelization and a low level of data interdependency. This makes it an ideal candidate for GPGPU computation.

2. RELATED WORK

GPGPU computing has been successfully applied to computational intensive problems in acoustics and audio processing: This ranges from acoustic simulation methods, like wave-based finite-difference

methods (FDMs) [5] as well as geometric acoustics modelling like ray-tracing [6] [7]. It has been applied to sound synthesis [8], spatial sound reproduction systems like wave-field synthesis (WFS) [9] and also music processing [10]. Tsingos [11] provides a good overview on applications. Not too many publications on audio signal processing using GPU can be found in literature. Only a few authors address audio filtering on graphic hardware: Most publications deal with straight-forward time-domain implementations of FIR filtering on GPUs. In 2005, Smirnov and Chiueh [12] implemented a tapped delay-line (TDL) using fragment shaders on a NVIDIA GeForce 6600 card and compare the performance to an SSE-optimized CPU variant. They conclude that GPU-processing is more efficient for long filters only (>60000 taps). In 2004, Gallo and Tsingos [13] introduce techniques for 3D rendering of virtual scenes using graphic hardware. They use head-related transfer functions (HRTFs) for spatial audio rendering and realize the filtering using simple 4-band equalizers. Their method also allows to realize doppler effects by texture scaling. A more recent time-domain implementation by Kowan and Krapalos [14] in 2008, also deals with HRTF-based audio rendering of virtual scenes. They convolve audio signals with short HRTFs filters (200 taps) using integer-based OpenGL shaders. On a NVIDIA GeForce 8800 GTX card their GPU solution outperforms a CPU convolution significantly and indicates enough performance to meet real-time conditions. However, they report subtile problems that come along with integer-processing. In a follow-up paper [15] they remeasured on a more current NVIDIA GTX 280 card. A recent publication by Trebien and Oliviera [16] deals with the implementation of 1D recursive filters on GPUs. The only paper we could find on GPU-based fast convolution in the frequency-domain, is an unpublished course work by Rush [17]. He implements a uniform partitioning on a NVIDIA G80 CPU and considers offline filtering. Unfortunately, no performance values are presented and the results are not compared to CPU-implementations.

Combining GPGPU computing with real-time conditions is an interesting problem in computer science. Especially real-time FIR filtering pairs intesive computation with tight timing dependencies in the range of milliseconds. After a careful literature review, we come the conclusion that GPU implementations of fast convolution in the frequency-domain have not been studied so far—with the exception of [17]. The special issues of real-time processing have not been reviewed in detail. None of the contributions deals with time-varying filtering.

3. FAST CONVOLUTION ALGORITHM

Fast convolution as a method for efficient FIR filtering has been researched for more than four decades. Several fast convolution algorithms are known today. We found that the choice of algorithm is even more important when considering GPGPU computation. Therefore we first give a brief overview on the methods and discuss their pros and cons. Afterwards we introduce our chosen algorithm.

3.1. Brief overview of fast convolution techniques

All fast convolution algorithms have in common, that they calculate *linear convolution* efficiently in the frequency-domain, by simple multiplication of discrete Fourier spectra, known as *circular convolution*. The term *fast* is reasoned by the Fast Fourier Transform (FFT) used to convert between the time- and frequency-domain. The original idea was proposed by Stockham [18] in 1966. His algorithm uses *one* FFT to convolve two signals (M,N samples). The length of the FFT is choosen so that the convolution result ($M + N - 1$ samples) does not exceed it and time-aliasing is avoided. This algorithm outperforms time-domain filtering (direct-form FIR filters, TDLs) by several magnitudes. However it has the disadvantage of an input-to-output latency that equals the FFT-length. Moreover its efficiency drops when long signals are convolved with short ones (many ineffective zeros are processed). These problems can be tackled by choosing a shorter FFT-length and by processing the *input data* in several steps — either in *overlap-add* or *overlap-save* fashion [19]. Still, the FFT-length is at least as long as the filter impulse response and so is the latency. For real-time filtering with long filters, the filter also needs to be *partitioned*. This allows to freely choose FFT-lengths and thereby to adjust the latency. Two variants are known: In *uniformly partitioned* fast convolution, filters are subdivided into several subfilters of *equal lengths*. The overall output is assembled from all subfilter outputs, which need to be delayed accordingly. Kulp [20] demonstrates, how the number of required FFTs/IFFTs can be reduced to one, when delays and sums are implemented directly in the frequency-domain. An DSP-implementation of the algorithm can be found in [21], [22]. The uniformly partitioned fast convolution is also most efficient for offline processing. Here, the FFT-lengths can be optimized in order to minimize the computational effort. The concept of *non-uniformly partitioned* fast convolution is relative new [23]. This algorithm is designed for efficient convolution of long filters (> 1000 filter coefficients) with a short input-to-output delay. Short subfilters are used to minimize the latency, whereas longer subfilters reduce the overall computational effort. It can be shown that this algorithm is even significantly more efficient than the uniformly partitioned variant [24]. Details on the implementation can be found in the famous paper by Gardner [25]. But a non-uniform partitioning has also drawbacks: As opposed to a uniform partitioning, the complete filter cannot be exchanged with every processed block. This introduces additional filter exchange latencies. Strategies exist to reduce this problem [26]. The filter partitioning itself can be optimized for maximum efficiency [24].

3.2. Basics of audio streaming

In this paper we concern *real-time* FIR filtering of *continuous* audio signals. Continuous recording, processing and playback of audio data using computer hardware is performed by *audio streaming*. Here, samples are processed and exchanged in units of *blocks*. All blocks consists of a fixed number of samples, referred to as the streaming block length—or just *block length*. In this work it is denoted by B. Furthermore we consider multiple channels. C is the

Length [samples]	Duration [ms]	Data size [bytes]
128	2.9	512
256	5.8	1024
512	11.6	2048

Table 1: *Typical properties of sample blocks in real-time audio streaming. Here, a sampling rate of 44.1 kHz and 32-bit floating point samples (4 bytes/sample) are considered.*

number of channels. The recurrent elementary step of processing the blocks—including multiple channels—is referred to as stream processing step — or short *stream processing*. Block-based processing introduces latency, which is determined by the *duration* of the streaming blocks. Additional latencies are contributed by electronics and intermediate buffering. In order to reduce the latency, very short block lengths are chosen for real-time applications—typically in the range of 128-512 samples. Furthermore, the block length determines the amount of time that can be spent on processing the samples — only a few milliseconds in practice. Table 1 gives an overview on typical values. Less latency implies less time for computations. If the processing exceeds the block duration, the audio stream is interrupted and dropouts occur. This must be prevented under any circumstances.

3.3. Convolution algorithm

For audio rendering of complex scenes, non-uniformly partitioned fast convolution is the algorithm of choice. But implementing it on graphic cards can be difficult: The algorithm heavily relies on the ability to process subfilters concurrently. Computation tasks must be priorized in order to ensure flawless operation. We circumvent these issues by choosing uniformly partitioned fast convolution for our examinations. It has a constant load balance and does not require asynchronous computations, but is less efficient.

The general principle of the algorithm for one channel is illustrated in figure 1. It consists of two main parts: stream processing and filter processing. Before a filter impulse response can be used for convolution, it has to be transformed into a frequency-domain representation. Therefore, it is uniformly partitioned into filter parts of the block length B. Each filter part is then padded with additional B zeros. We refer to this process as *filter packing*. Afterwards, each padded part is FFT-transformed into a discrete Fourier spectrum. All together, these are then used for the convolution. The stream processing computes the convolution. A time-domain input buffer acts as a sliding window of $2B$ samples over the input stream. With each new input block, the right half is copied to the left and the new block is then stored in the right half (called *input packing*). The buffer is then FFT-transformed and the result is stored in a frequency-domain delay-line (FDL). Before, the FDL is shifted by one slot. Now the DFT spectra in the FDL are point-wise complex-valued multiplied with the DFT spectra of the prepared filter parts. All results are summed up in a frequency-domain accumulation buffer. The next its contents are IFFT-transformed back into the time-domain. From the result, only the right half of samples is selected (named *output packing*). The left is discarded (overlap-save).
The algorithm can be split into four major parts:

1. Packing and FFT transformation of the input data
2. Packing and FFT transformation of the filter data
3. Spectral convolution
   (complex-valued multiplication and accumulation)
4. IFFT transformation and unpacking of the output data

4. GPU IMPLEMENTATION

For each of the algorithm's parts can be parallelized and is thereby a candidate for GPU computation. But we also consider the CPU for computations and will only employ the GPU if it benefits the performance. The question is, where to perform the computations – on the host (CPU) or on the grapics card (GPU). One decision is fixed: The multiplication and addition of DFT spectra the demands the major share of computation and will therefore be computed on the graphics hardware. But we will later see, that for the other parts the choice is not trivial.

When implementing the fast convolution algorithm for GPGPU computation, we are faced with some fundamental questions:

- How to parallelize the spectrum multiply-adds on the GPU?
- Where to perform the FFT/IFFT-transforms? Host or GPU?
- How to organize the data structures for maximum efficiency?

In the following we present our parallelization approach and address each of the questions in detail. Our parallelization founds on two key principles:

- Avoid thread synchronization by avoiding mutual write access on memory locations
- Keep data reordering operations to a minimum by using well-arranged data structures

4.1. Data structures

Optimization in high-performance computing begins with a close look at data structures. An efficient memory access pattern, especially with respect to the cache-hierarchy, is usually the biggest leap forwards in increasing performance. This also holds for GPGPU computing, but with different circumstances: Graphic cards do not have a cache-hierarchy, but they benefit from a faster memory link. But in the face of the GPUs massive parallelism, data-structures must be carefully chosen to result in an optimal performance.

A driving factor for our data structure design turned out to be the option to perform FFTs/IFFTs on the graphic hardware. This is appealing, because CUDA not only allows to calculate a single FFT/IFFT, but a large number of equally-sized transforms in parallel on the graphics hardware. Throughout this work, we refer to this feature by the term *Batch-FFT*. However, the input and output data for such Batch-FFTs must be aligned accordingly. Input blocks (time-domain) and output spectrums (frequency-domain) must directly succeed each other in a linear sequence. Destination DFT coefficients cannot be shuffled in a clever way, so that later calculations would benefit from a better memory access pattern. In order to allow the computation of Batch-FFTs on the GPU we maintain this alignment and arrange the input/output data, filter data and the frequency-domain delay-lines (FDLs) in a compatible fashion. Generally, we use real-to-complex Fast Fourier Transforms (R2C-FFTs) and complex-to-real Inverse Fast Fourier Transforms (C2R-IFFTs). Each sample or filter block of B values corresponds to a complex-conjugate symetric DFT spectrum of $B + 1$ values.

The input and output data transferred between host and graphic device consists of the input and output blocks for all channels. If the FFT-/IFFT-transforms are calculated on the GPU, we transfer C blocks of $2B$ samples (time-domain). When the sliding window is implemented on the GPU, just $C \times B$ samples need to be transferred. However, this requires additional data reordering. Since transferred stream data is rather small, we do not consider this beneficial. When the FFT-/IFFT-transforms are calculated on the CPU, we transfer C complex-conjugate symetric DFT spectra

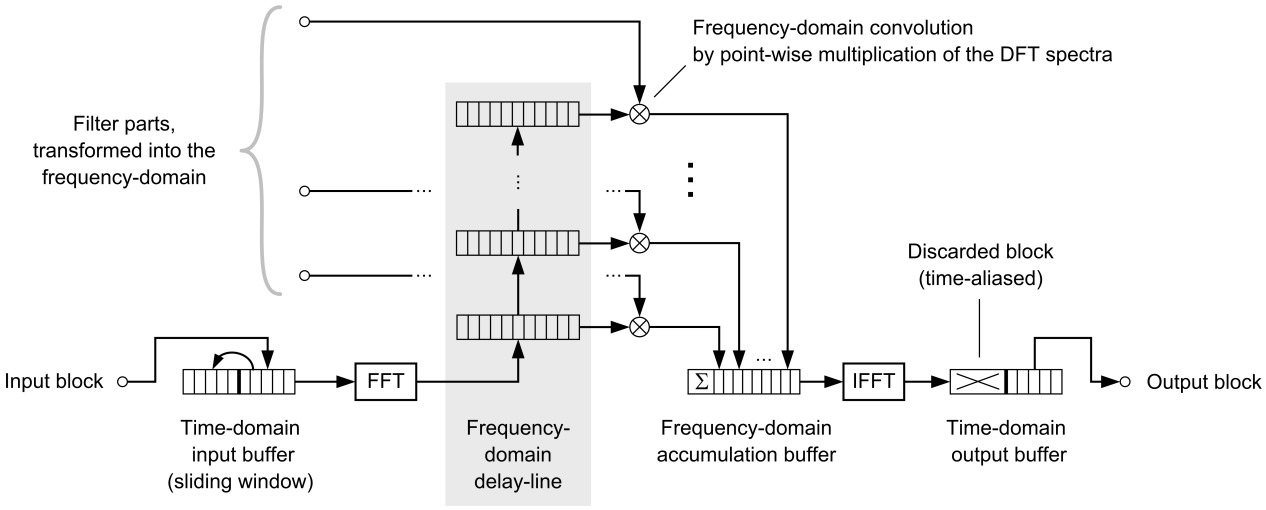

Figure 1: Uniformly-partitioned fast convolution algorithm using the overlap-save scheme. By realizing subfilter delays and the accumulation of subfilter outputs in the frequency-domain, only one FFT/IFFT-transform is required per processed input block.

of $B+1$ complex-valued coefficients each (frequency-domain). With every stream processing step the data has to be transferred twice — from host-to-device and after calculation from device-to-host. Important to mention here is, that stream processing enforces a synchronous data transfer.

The filter data has to be transferred just once (host-to-device), but is usually significantly larger. We consider filters to be partitioned into K parts, each of B filter coefficients. For one channel it consists of K blocks of B filter coefficients in the time-domain or —alternatively—K complex-conjugate symetric DFT spectra of $B+1$ coefficients each. The amount of filter data increases linear with the number of channels.

The frequency-domain delay-line is entirely stored within the graphic memory. Figure 2 depicts the layout of the complete FDLs for all C channels. It consists of K FDL lines, corresponding to K individual subfilters. Each FDL line covers delayed DFT spectra for all C channels. Note that even if they succeed each other, DFT spectra within one line belong to independent channels and filters. What they share is a common delay. We avoid shifting of the actual FDL contents (cp. figure 1) in the stream processing, by using an FDL cursor and accessing the FDL in the sense of a cyclic buffer. The FDL cursor denotes the line that has been written last and is incremented with stream processing step. This type of FDL layout support direct write the input data into the destination FDL line and prevents unnecessary data reordering.

4.2. Kernel

Every GPU thread (CUDA thread) performs the complex-valued multiplication of DFT coefficents within one 'column' of the FDL and adds up the result in an accumulation buffer. Figure 2 depicts the responsibilities of one GPU thread, illustrated by the grayed-out vertical bar. This parallelization strategy does not require thread synchronization, because every DFT coefficient in the accumulation buffers underlies exclusive write access by just one CUDA thread. Listing 1 shows the pseudocode of the GPU convolution kernel. The accumulation is performed locally, in the registers of the SPs. Afterwards, the result is written into the graphic memory. The number of loop iterations equals the number of filter parts K. Since there are plenty of filter parts (cp. table 2), we achieve a good balance between computation and memory access on the SPs. This is executed in the GPU in thread blocks of 32 threads each. The number of allocated thread blocks therefore is $C{\cdot}(B+1)/32$. Threads assigned to a streaming multiprocessor (SM) work on neighbouring elements and we can benefit from coalescing.

```
k = GPUThreadID;    // Element-index
c = channelof(k);   // Channel number
n = FDLCursor;      // Last inserted FDL-line

// Initialize the local accumulator
accu = 0;

// Iterate over filter parts
for i=0 to NumFilterParts-1 {
  // Complex-valued multiply-add
  x = FDL[line=n][element=k];
  y = FilterSpectrum[channel=c, part=i];

  ComplexMulAdd(src=x, src=y, dest=accu);

  n = (n+1) mod NumFilterParts;
}

// Write the result into the global memory
output[k] = accu;
```

Listing 1: Convolution kernel (pseudocode)

5. PERFORMANCE

In this section we analyze the performance of our method for two applications: Time-invariant real-time filtering without the exchange of filters and time-varying filtering. Firstly, we introduce our test system and regard important measures individually — the

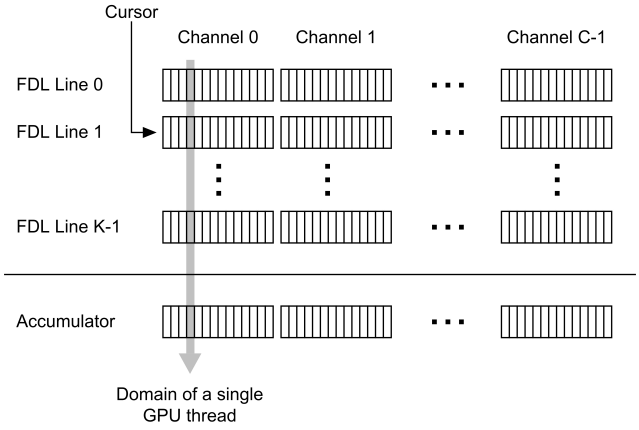

Figure 2: Structure of the frequency-domain delay-lines (FDLs) and the frequency-domain accumulation buffers. The grayed-out vertical bar illustrates the range of DFT coefficients, a single GPU thread is responsible for.

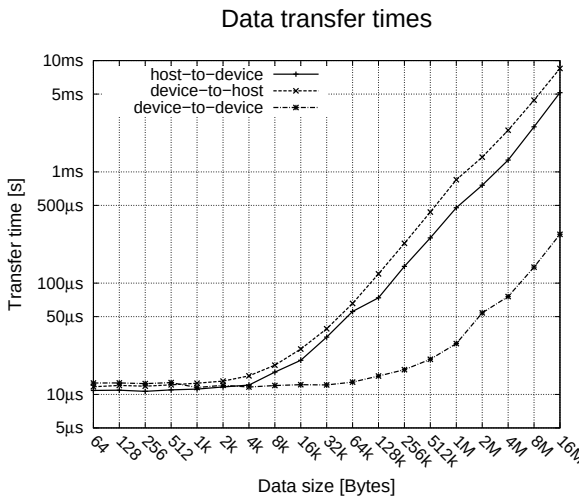

Figure 3: Data transfer times measured on the test system.

data transfer and computation of Fast Fourier Transforms. Afterwards, we present the performance measures for static filtering. Dynamic filtering can not be expressed by a single performance value, because it depends on opposing parameters: The number of channels and filter length versus the desired filter update rates. Here we consider the performance by means of an example scenario and discuss how these parameters relate.

5.1. Test system

The test system used is a dual quad-core machine, with two Intel Xeon X5570 (Gainestown) processors [27] running at 2.93 GHz. Each processor has 8 MB of shared L3-Cache available. The machine has 4 GB of DDR3-1333 memory. The graphic card is an NVIDIA GeForce GTX 285 [28]. It features the NVIDIA GT200b graphic chip, which has 240 SPs in total, each clocked at 1476 MHz. They are arranged in 30 SMs. Our card has 1024 MB GDDR3 video memory, clocked at 1242 MHz and linked via 512-bit memory interface. It uses a PCI-Express 2.0 BUS interface with 16 lanes (x16), resulting in a theoretical bus bandwith of 8 GB/s. The operating system is Microsoft Windows XP Professional (32-bit). We built our software using the Microsoft Visual Studio 2005 (SP1) C++ compiler. We use Streaming SIMD Extensions (SSE) along with a 16-byte structure alignment. The CUDA version used is 2.3. For FFTs on the host we employ the FFTW library [29] version 3.2.2. All tests were carried out on 32-bit single precision floating points. High-precision timing was realized using the Win32-function *QueryPerformanceCounter*. We used an RME Hammerfall audio device and Steinberg's ASIO interface [30] for low-latency audio streaming.

5.2. Data transfer

In advance of any calculation on the GPU, the required data must first be copied onto the graphic card and after the calculation is finished the results must be read back to the host. Data transfer is fundamental — especially for real-time processing. Every microsecond that is spent on data transfer, cannot be used for GPU calculations. Asynchronous data transfers can be used to mask this problem. But it complicates matters much and is not considered here. For our later considerations, we measured the transfer times and bandwidths for three important types of data transfers:

1. Host memory to graphic memory
2. Graphic memory to host memory (readback)
3. Graphic memory to graphic memory

All transfers we executed using the *cudaMemcpy* function. The results on the test system are shown in figure 3. Each operation (function call) is loaded with a nearly constant time overhead of 10-12 us. We presume that this falls back to bus latencies and buffering. Only for larger data sizes, runtimes vs. data size scale nearly linear. For host-to-device and device-to-host transfers this starts at $\geq$32-64k and for on-device memory access for $\geq$1-2 MB. On our test system, the readback from device-to-host turns out to be slower than in the opposite direction. Transferring 128 MB host-to-device takes 36 μs, whereas the device-to-host transfer accounts to 68 μs. he maximum bandwidths measured are 5,36 GB/s for host-to-device and 3,20 GB/s for device-to-host transfer. Despite the function call overhead, the on-device data transfer is impressively fast and peaks at bandwidth of 65,9 GB/s.

5.3. Fast Fourier Transforms

As explained in the introduction, the Fast Fourier Transform (FFT) is a keystone for frequency-domain FIR filtering. Large-scale real-time filtering applications with a multitude of individual channels and with long filters, account to a considerable number of FFTs and Inverse Fast Fourier Transforms (IFFTs) . But all of these transforms have the small transform size $2{\cdot}B$, of two times the block length. A typical 256-point FFT is a rather simple operation and does not include too many arithmetical operations. Its single-threaded execution on one CPU-core/-thread of the test system is very fast and takes just 1,2 μs. Such single small FFTs cannot be efficiently parallelized on the GPU. But CUDA supports the calculation of multiple equally-sized FFTs, we refer to as Batch-FFTs. These are exactly what is needed in the uniformly-partitioned convolution algorithm. Hence it is very interesting to see where they can be carried out the fastest — on the CPU or GPU.

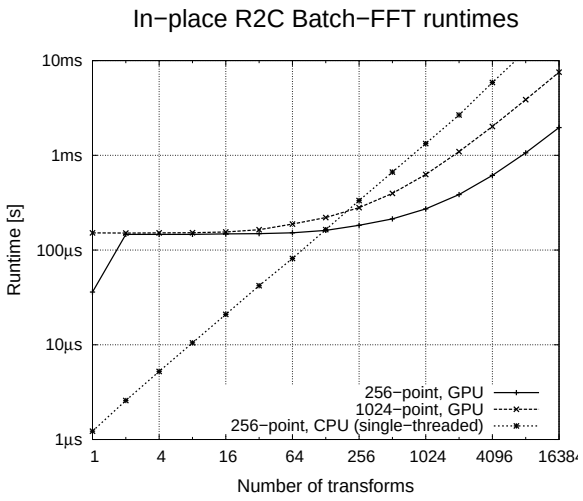

Figure 4: Comparison of real-to-complex FFT between the CUDA Batch-FFT executed in parallel on the GPU and an iterated single-threaded calculation on the CPU

Num channels	Block length	Filter coeffs.	Filter parts per channel	Filter parts in total
32	128	44.100	345	11.040
64	256	44.100	173	11.072
128	128	88.200	690	88.320

Table 2: Several filtering scenarious with different numbers of channels and filter lengths. Again we consider a sampling rate of 44.1 kHz.

Figure 4 shows the measured runtimes of the CUDA Batch-FFT, for in-place real-to-complex forward transforms (R2C-FFT). We have also measured the runtimes for single-threaded (serial) calculation on one CPU-core with the FFTW library. Plotted against the abscissa is the number of transforms — referred to as the *batch size*. For small batch sizes (<128), there is not enough level of parallelism in order to occupy all 240 stream processors on the G200b graphic chip. Runtimes are almost constant and do not increase with the batch size. Regarding 256-point FFTs, the graphic card first gets fully utilized for more than 128 transforms. For 1024-points FFTs — which have more potential for parallelization — this begins earlier. Once the graphic card is fully utilized, we observe a linear runtime progression with the batch size. Interestingly, all CUDA Batch-FFTs are loaded with an overhead of 146 μs. The only exception can be found for 1-2 smaller transforms <512 points.
Alternatively, Batch-FFTs can be calculated on the host, using a loop. This can either be processed in serial (by a single thread) or in parallel (multiple threads, loop parallelization). Thanks to machine-tailored assembly code (called codelets [29]), small-size transforms can be executed very fast on the CPU. Considering 256-point FFTs, calculation on the CPU is much quicker, by several magnitudes. The break-even point is reached for a batch size of nearly 128 transforms. For parallel execution on the host with eight CPU-threads/CPU-cores we estimate a break-even point of 3.200 transforms.

From these results we conclude that, for a reasonable number of channels (<128), the FFT-/IFFT-transforms for the streamed input and output data should be calculated on the CPU. The filter processing usually involves much more transforms. Computation on the GPU is beneficial here (cp. figure 4). These rules only apply for filtering tasks with long filters and consequently a large number of filter parts. In general, it is adviced to consider the given filtering circumstances.

5.4. Static filtering

The simplest case is time-invariant filtering, in which filters are not changed during runtime. We refer to this as *static filtering*. Here, filters can be preloaded into the graphic memory, before the audio streaming starts. Data transfer during runtime reduces just on the input and output data. We measured the achievable performance by means of the maximum possible number of channels. Every channel is convolved with an individual filter lengths of a fixed length. In theory, the limit of channels is reached, as soon as the available time budget for stream processing — the block duration (cp. table 1) — is exhausted. In practice we cannot spend the full block duration for computations. Other resources also employ the CPU, host memory, PCI bus and of course also the graphic hardware. There is also still a video signal that needs to be transferred to the graphics card. For a seamless operation without dropouts we need to include a safety margin. In our measurements, we observed that runtimes of the stream processing are in general quite constant. However, for some stream processings they vary and the maximum stream processing time is 1 ms longer. This 1 ms disturbance seems to stem from the system and is constant along all tests. Consequently, small block length samples suffer this problem most. For $B = 128$, just 70% of the available time budget should be used. The problem relaxes for longer block lengths. For $B = 512$ a stable operation is achieved when 90% of the budget are exploited.

The results of our benchmark are shown in table 3. Again, a sampling rate of 44.1 kHz is used. For a block length of $B = 128$ samples, we found that the filtering runs smoothly for 118 channels, each with a filter of 1s or 44100 coefficients. Here 1,77 ms account for the stream processing in total. The whole 2,9 ms time budget gets fully exhausted for 192 channels, but then the streaming has dropouts. For filters of 2 s or 88.200 coefficients, we achieved 58 channels. The theoretical limit here is reached for 112 channels. We considered filtering with very long impulse responses of 5 s or 220.500 coefficients. Here we still managed to run 7 channels smoothly. For longer block lengths and thus longer input-to-output latencies, stability is achieved more easily and the performance significantly increases. We managed for example to convolve 256 channels with 1 s filters at block length of $B = 256$ samples. If we accept a latency of 11,9 ms ($B = 512$) even more channels can be realized. For filters of 1.0 s we reached an impressive 352 channels.

Reasoned by the timing issues as well as the mandatory synchronous data transfer, we did not manage to utilize the full processing power of the graphic chip in any case.

5.5. Dynamic filtering

For interactive audio rendering the exchange of filters during runtime is required. Several issues complicate matters in this case:

- The filter data must be transferred to the graphic card

- The filter parts need to be FFT-transformed
- The stream processing demands more computation

Consequently, we expect the overall performance to be less than for the case of static filtering. In the following we will analyze this in detail. We approach this in the way, that we examine how the filter exchange can be added to static filtering, without causing any disturbances. All additional processing must therefore be performed within the remaining available processing time.

Direct switching of filters is rarely applied in practice, because it mostly results in audible artefacts (e.g. clicks). A smooth exchange of filters is usually realized by cross-fading in the time-domain. For the time of the cross-fade, two output branches need to be processed — one with the old filter and another with the new filter. Therefore, the spectrum multiplications and accumulations and the IFFTs have to be performed twice, for every filter that is exchanged.

Our performance considerations found on the following example scenario: 64 channels, a sampling rate of 44,1 kHz, a block length $B = 256$ samples and filter lengths of 1,0 s (44.100 filter coefficients). We exhaust 80% of the block duration for computations — that is 4,8 ms. The measured average stream processing runtime for this case is 2,26 ms. Consequently, we have 2,54 ms left over for the filter exchange. For the filter transformation 173 FFT have to be computed for a single channel and accordingly 11072 FFT for all 64 channels together (cp. table 2).

In the simplest of cases the filter of just *one* of the 64 channels is exchanged at a time. The filter data for one channel comsumes 356 kB. Transferring this portion of data to the graphics card takes 182 μs. For a single channel, the filter transformation (173×512-point R2C-FFTs) is faster on the CPU and demands approximately 280 μsfor computation. The stream processing is extended by one additional spectrum multiply-add, one additional 512-point IFFT and slightly larger output data. Including the filter transfer, we calculate an increase in stream processing runtime by 216 μsor 9,6%, compared to static filtering. Both, data transfer and additional processing fit into the time budget and the filter can be exchanged instantaneously.

The worst-case scenario is that the filters of *all* 64 channels are exchanged synchronously at a time. A complete filter set for all 64 channels consumes 22,7 MB. The data transfer in this case takes 6,88 ms. This exceeds our time budget of 2,54 ms significantly. Thus the transfer cannot be done in one step. Considering an asynchronous data transfer here, we would still have to wait three blocks. The computation of FFT-transforms for all 11072 filter parts demands 2,24 ms on the GPU. This introduces another block of delay. For the stream processing the number of spectrum multiply-add and IFFTs doubles, so does the size of the output data. This increases the an overall stream processing runtime to 4,13 ms, which marks an increase of 82,7%, compared to static convolution. The good news is that the 4,13 ms are still within our set time budget of 4.8 ms. Apart from the transfer of the filter data, a seamless streaming operation is threreby still possible. Unfortunately, we have to accept a filter exchange latency of at least four block durations or 23,6 ms. Thereby we achieve a worst-case filter update rate of 42,4 Hz.

Block length [samples]	Filter length [s]	Maximum number of channels	Average stream processing runtime [ms]	GPU load [%]
128	1,0 s	118	1,77 ms	41%
128	2,0 s	58	1,47 ms	40%
128	5,0 s	7	2,06 ms	68%
256	1,0 s	244	4,91 ms	46%
256	2,0 s	158	4,82 ms	58%
256	5,0 s	78	4,78 ms	68%
512	1,0 s	352	10,53 ms	36%

Table 3: Static filtering performance achieved on the test system. Here, filters are time-invariant and not changed during runtime. A sampling rate of 44.1 kHz was used, resulting in filters of 44.100, 88.200 and 220.500 coefficients. All values have been measured under stable operation.

For time-varying filtering significant amounts of filter data need to be transferred. Consequently the data transfer matters even more here. But we like to point out, that the filters can benefit from asynchronous data transfers. Nevertheless, the dynamic filtering performance is still remarkable and fulfills the needs for audio rendering. As illustrated at the example, dynamic filtering demands different considerations: For applications, first the required filter update rate must be defined. Then the two parameters, number of channels and filter lengths, can be adjusted for the given task. In practice such worst-case scenarios occur seldom, so the actual filter update rate will be greater.

6. CONCLUSIONS

We have presented an implementation real-time FIR on graphics hardware. Our method achieves an outstanding performance, which exceeds all prior implementations. On a NVIDIA GTX 285 card we successfully realized static filtering of more than 200 channels with individual filters of more than 40.000 coefficients at an input-to-output latency of less than 6 ms. We like to point out, that all measures have been benchmarked under stable operation. The theoretical peak performance is even higher, because this performance was achieved by just utilizing 44% of the computation power of our GPU. This is reasoned by subtile problems that emerge under real-time conditions. For a stable operation, variations in the processing runtimes must be tolerated. In order to prevent dropouts, not the full time budget time can be exhausted for computations. We discovered randomly occuring system latencies of approximately 1 ms. These turned out to be critical especially for very low latency applications.

We also analyzed time-varying filtering in detail, where filters are exchanged during runtime. Dynamic filtering demands more computation and relies even more an fast data transfers. Nevertheless, for 64 channels and 1,0 s filters, the full filter set (all 64 channels at once) can still be updated with over 40 Hz. This is impressive, but we like to state the PCI-Express BUS can still be a bottleneck. When many long filters are exchanged synchronously, additional filter exchange latencies occur. However, this has minor relevance for applications in practice. Here, typical filter update rates are in the range of 50-100 Hz. Moreover, long filters are usually not entirely exchanged with such high rates [1].

We summarize, that graphic hardware is very well-suited for real-time FIR filtering and pushes the possible filtering performance a big leap forward. Thereby, our technique can excellently be applied for audio rendering of complex virtual scenes. As an outlook, it will be interesting to see the potential of non-uniformly partitioned fast convolution for GPU computation.

7. REFERENCES

[1] Tobias Lentz, Dirk Schröder, Michael Vorländer, and Ingo Assenmacher, "Virtual reality system with integrated sound field simulation and reproduction," *EURASIP Journal on Advances in Signal Processing*, vol. 2007, available at http://downloads.hindawi.com/journals/asp/2007/070540.pdf.

[2] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A.E. Lefohn, and T.J. Purcell, "A survey of general-purpose computation on graphics hardware," in *proceedings of EUROGRAPHICS*, 2005, available at http://www.idav.ucdavis.edu/func/return_pdf?pub_id=907.

[3] "NVIDIA CUDA Programming Guide Version 2.3," available at http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf.

[4] "ATI Close-To-Metal (CTM) Guide Version 1.01," available at http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf.

[5] Nikunj Raghuvanshi, Nico Galoppo, and Ming C. Lin, "Accelerated wave-based acoustics simulation," in *SPM '08: Proceedings of the 2008 ACM symposium on Solid and physical modeling*, New York, USA, 2008, pp. 91–102, ACM.

[6] Marcin Jedrzejewski and Krzysztof Marasek, "Computation of room acoustics using programmable video hardware," in *International Conference on Computer Vision and Graphics ICCVG'2004*, Warsaw, Poland, 2004.

[7] Niklas Röber, Ulrich Kaminski, and Maic Masuch, "Ray acoustics using computer graphics technology," in *Conference on Digital Audio Effects (DAFx-07) proceedings*, Bordeaux, France, 2007.

[8] Qiong Zhang, Lu Ye, and Zhigeng Pan, "Physically-based sound synthesis on GPUs," *Entertainment Computing-ICEC 2005*, pp. 328–333.

[9] Dimitris Theodoropoulos, Catalin Bogdan Ciobanu, and Georgi Kuzmanov, "Wave field synthesis for 3d audio: architectural prospectives," in *CF '09: Proceedings of the 6th ACM conference on Computing frontiers*, Ischia, Italy, 2009.

[10] Eric Battenberg and David Wessel, "Accelerating Non-Negative Matrix Factorization for Audio Source Separation on Multi-Core and Many-Core Architectures," in *10th International Society for Music Information Retrieval Conference (ISMIR 2009)*, Kobe, Japan, 2009.

[11] Nicolas Tsingos, "Using programmable graphics hardware for acoustics and audio rendering," in *proceedings of the 127th AES Convention*, New York, USA, 2009.

[12] Alexey Smirnov and Tzi-cker Chiueh, "An Implementation of a FIR Filter on a GPU," 2005, available at http://www.ecsl.cs.sunysb.edu/fir/fir.ps.

[13] Emmanuel Gallo and Nicolas Tsingos, "Efficient 3D audio processing with the GPU," in *GP2, ACM Workshop on General Purpose Computing on Graphics Processors*, 2004.

[14] Brent Cowan and Bill Kapralos, "Spatial sound for video games and virtual environments utilizing real-time gpu-based convolution," in *Proceedings of the 2008 Conference on Future Play*, New York, NY, USA, 2009, ACM.

[15] Brent Cowan and Bill Kapralos, "Real-time gpu-based convolution: a follow-up," in *Proceedings of the 2009 Conference on Future Play on @ GDC Canada*, Vancouver, Canada, 2009.

[16] F. Trebien and M.M. Oliveira, "Realistic real-time sound resynthesis and processing for interactive virtual worlds," *The Visual Computer*, vol. 25, no. 5, pp. 469–477, 2009.

[17] Michael Rush, "Convolution engine utilizing NVIDIA's G80 processor," available at http://www.ece.ucdavis.edu/m̃mrush/mmrush_eec277_final_writeup.pdf.

[18] T.G. Stockham Jr, "High-speed convolution and correlation," in *Proceedings of the April 26-28, 1966, Spring joint computer conference*. ACM, 1966.

[19] Alan V. Oppenheim and Ronald W. Schafer, *Discrete-Time Signal Processing*, Prentice Hall Signal Processing Series. Prentice Hall, 1989.

[20] Barry D. Kulp, "Digital equalization using fourier transform techniques," *Journal of the Audio Engineering Society*, 1988.

[21] Anders Torger and Angelo Farina, "Real-time partitioned convolution for ambiophonics surround sound," *IEEE Workshop on the Applications of Signal Processing to Audio and Acoustics*, pp. 195–198, 2001.

[22] E. Armelloni, C. Giottoli, and A. Farina, "Implementation of real-time partitioned convolution on a DSP board," pp. 71–74, 2003.

[23] Gerald P. M. Egelmeers and Piet C. W. Sommen, "A new method for efficient convolution in frequency domain by nonuniform partitioning for adaptive filtering," *IEEE Transactions on signal processing*, vol. vol 44, 1996.

[24] Guillermo García, "Optimal filter partition for efficient convolution with short input/output delay," in *Audio Engineering Society, Convention Paper 5660*, 2002.

[25] William G. Gardner, "Efficient convolution without input-output delay," *Journal of the Audio Engineering Society*, vol. vol 43, pp. 127–136, 1995.

[26] Christian Müller-Tomfelde, "Time-varying filter in non-uniform block convolution," in *Conference on Digital Audio Effects (DAFX-01) proceedings*, 2001.

[27] "Intel® Xeon® Processor X5570," available at http://ark.intel.com/Product.aspx?id=37111.

[28] "NVIDIA GeForce GTX 285 Specifications," available at http://www.nvidia.com/object/product_geforce_gtx_285_us.html.

[29] "Fastest Fourier Transform in the West (FFTW)," available at http://www.fftw.org.

[30] Steinberg Media Technologies GmbH, *Audio Streaming Input Output (ASIO) 2.2 Specification*, Steinberg Media Technologies GmbH, 2006.

All online documents last accessed June 25, 2010.