Realtime Computation of a VST Audio Effect Plugin on the Graphics Processor

Wolfgang Fohl
HAW Hamburg
University of Applied Sciences
Hamburg, Germany
Email: fohl@informatik.haw-hamburg.de

Julian Dessecker
Steinberg Media Technologies GmbH
Hamburg, Germany
Email: J.Dessecker@steinberg.de

***Abstract*—A plugin system for GPGPU real time audio effect calculation on the graphics processing unit of the computer system is presented. The prototype application is the rendering of mono audio material with head-related transfer functions (HRTFs) to create the impression of a sound source located in a certain direction relative to the listener's head. The virtual source location can be controlled in realtime. Since HRTFs are measured only for certain incident angles, a interpolation for intermediate angles has to be performed in realtime. Plugins are implemented using the VST software development kit offered by Steinberg Media Technologies. Two GPU processing frameworks for a NVIDIA graphics processor were evaluated: CUDA and OpenCL. The overall processing speed can be increased by the factor 2.2 with the GPGPU modules. When calculating the FIR filter outputs by fast convolution on the GPU, the processing speed can even be increased by the factor ten.**

***Keywords*-GPGPU computing; VST plugin; Spatial audio; Head-related transfer functions;**

I. INTRODUCTION

This article presents a case study for the application of GPGPU techniques to the realtime audio signal processing. GPGPU computing (i.e., execution of general purpose computation on the graphics processor) has raised considerable interest with the availability of software development kits (SDKs) that offer an access to the massive parallel computing capabilities of modern graphics processors. The two most common frameworks are OpenCL, which provides an vendor-independent access to GPGPU computing, and CUDA-C, which is an extension to the C language for GPGPU on NVIDIA graphics processors [1] [2]. While CUDA is a proprietary framework restricted to NVIDIA GPUs, OpenCL is an open standard maintained and published by the Khronos Group [3]. The actual OpenCL development framework is provided by the GPU manufacturers.

The initial motivation for this work was the promise of GPGPU to drastically accelerate tasks that can be parallelised. This is the case for realtime audio processing. Input and output data of audio processing units are buffered, which offers the opportunity to calculate all the output samples of a buffer simultaneously from the input samples. Furthermore, the calculation of a single output sample offers opportunities for parallel execution: Most audio processing tasks consist in evaluating a finite difference equation for the N output buffer elements y_{out}:

$$\{y_{out}\} = \left\{ y_i | y_i = \sum_{k=0}^{N} f_k(x_{i-k}) - \sum_{k=1}^{N} g_k(y_{i-k}) \right\}_{(i=0\ldots N-1)} \quad (1)$$

Here, $\{y_{out}\}$ is the array of output samples, i is the sample index, x is the input signal, and k is a summation index.

In the case of a linear and time-invariant audio processor, the functions of eq. 1 are merely multiplications with constant values $\{a_k\}$ and $\{b_k\}$:

$$\{y_{out}\} = \left\{ y_i | y_i = \sum_{k=0}^{N} b_k \cdot x_{i-k} - \sum_{k=1}^{N} a_k \cdot y_{i-k} \right\}_{(i=0\ldots N-1)} \quad (2)$$

Very often in audio processing, filters with a finite impulse response (FIR filters) are employed , for which the difference equation simplifies to

$$\{y_{out}\} = \left\{ y_i | y_i = \sum_{k=0}^{N} b_k \cdot x_{i-k} \right\}_{(i=0\ldots N-1)} \quad (3)$$

which is the *convolution* of the impulse response $\{b_i\}$ of the filter with the input signal $\{x_i\}$. So for each value of the output buffer, the summation of eq. 3 has to be performed, where the computation for each output value as well as the computation of each summand can be calculated simultaneously. The convolution operation of eq. 3 can be considerably sped up by using the *fast convolution* algorithm. The basic idea is given by:

$$\{b_i\} \overset{\text{FFT}}{\Rightarrow} \{B_j\} \quad (4)$$

$$\{x_i\} \overset{\text{FFT}}{\Rightarrow} \{X_j\} \quad (5)$$

$$\{y_{out}\} \overset{\text{IFFT}}{\Leftarrow} \{B_j \cdot X_j\} \quad (6)$$

A FFT is applied to the filter coefficients and the input signal block, the resulting arrays are multiplied element-wise, and the product is transformed back. The two arrays $\{b_i\}$ and $\{x_i\}$ have to be brought to twice the buffer length by zero-padding, which in turn will give an output signal of twice the buffer length. The first half is added to the second half of the previous processing step and transferred to the output buffer.

The audio processing task that shall be executed as prototype application is the spatial rendering of mono audio signals by head-related transfer functions (HRTFs) according to the procedure described in [4].

In the following sections, the algorithm for spatial rendering is described, followed by an overview of the GPGPU and VST plugin architectures. Finally, our solution is described together with some implementation details, and the results of performance measurements are presented and discussed.

 ISBN: 978-1-61208-157-1

II. Overview of Technical Concepts

A. Spatial Audio Rendering

In order to make a mono audio signal appear to be coming from a certain direction of incidence, the signal is filtered by head-related transfer functions (HRTFs), that mimic the inter-aural time delays, level differences and differences in frequency response. If the resulting stereo signal is replayed via headphones, the perceived signals are similar to real signals emanating from a source at the corresponding location.

Since the coefficients of the filters could only be measured at certain discrete angles as indicated in Fig. 1, an interpolation has to be performed for intermediate angles. The input signal is routed to four filters, each representing one corner in the interpolation grid of Fig. 1, then a linear interpolation of the output signals is performed. Details are given in [4].

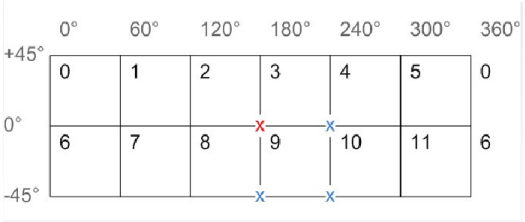

Figure 1. Grid of measured HRTF data

B. GPGPU Computing

The parallel processing architecture of the graphical processing unit (GPU) found on modern hardware suits in an optimal way the needs of many numerical processing tasks. The release of the CUDA SDK by NVIDIA in 2007 and the approval of the OpenCL standard in 2008 opened the field of GPGPU computing to the community of developers and researchers.

GPGPU computing offers a performance boost for algorithms that can be parallelised, as is shown in the code snippets in figure 2. Both programs perform an array addition. While in conventional CPU processing the program iterates over all array elements, the GPU program starts one thread for each array element, provided there are enough processing units. The threads on the GPU are programmed as so-called “kernels”.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}

int main(void)
{
  // CPU operation
   for (int i = 0; i < N, i++)
      C[i] = A[i] + B[i];
  ...
  // Kernel invocation with N threads
  VecAdd<<<1, N>>>(A, B, C);
  ...
}
```

Figure 2. Array addition on the GPU with the CUDA SDK

One drawback of GPGPU computing on consumer-grade GPUs has to be mentioned. Since these GPUs are designed for optimum video game performance, there is no need for checking the integrity of the GPU memory, since memory errors would only affect the currently displayed video frame. High reliability can be either attained by software means as described in [5], or by using GPU hardware dedicated to GPGPU computing, which are equipped with ECC-protected memory [6].

C. VST Plugin Architecture

Steinberg Media Technologies developed a plugin system for the extension of audio workstation software with external effects and with external virtual instruments. Developers can obtain a SDK after registering on the Steinberg website [7].

A plugin consists of two parts, the *processor* and the *edit controller*. The processor does the audio signal processing, the edit controller provides the GUI for parameter visualisation and modification.

Data transfer between host and plugin is performed by means of the VST plugin interface methods: One block of audio data is provided by the host in the input buffer of the plugin. Then the host calls the `process` method of the plugin, the plugin then has to compute the audio samples and transfer them to the output buffer, where the host will fetch it.

III. System Implementation

A. Technical Details

The software was implemented on a PC with an Intel Core2Quad Q9400 processor operated at 2.66 GHz, a NVIDIA GeForce 9600 graphics processor with 650 MHz core clock, and a Creative ES 1371 sound card. The operating system was Windows XP SP 3, the VST host application was Steinberg Cubase 5. The VST SDK was version 3.1.0.

B. Filter Module Architecture

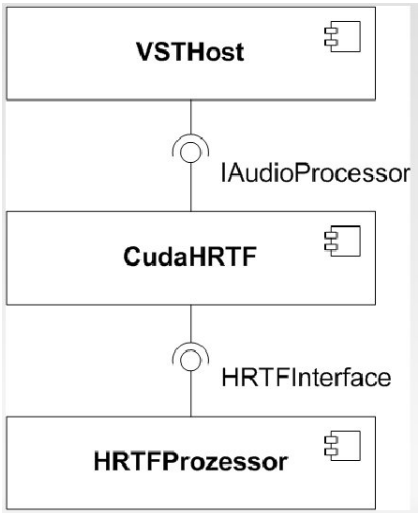

Figure 3. System architecture with VST Host (Cubase), VST plugin, and HRTF processor

In this project, the `CudaHRTF` VST plugin was developed as a wrapper for various filter modules with various implementation of

the HRTF rendering algorithm. The filters modules were created as independent dll files. Four filter dlls were developed:

filt_cuda.dll
: Filter implementation using the CUDA-C SDK and fast convolution in the frequency domain

filt_ocl.dll
: Filter implementation using the OpenCL SDK and time-domain convolution

filt_sse.dll
: Filter implementation on the CPU using the SSE2 (Streaming Single-Instruction-Multiple-Data) extension, time-domain convolution

filt_fpu.dll
: Filter implementation using standard FPU code and time-domain convolution

Fig. 3 shows the component design, showing the VST host, the VST plugin, and one filter module.

Note on Time-Domain Convolution: The computation steps for time-domain convolution of the input signal x_i with the FIR filter coefficients $\{b_i\}_{i=0\ldots L-1}$, where L is the filter length according to

$$y_i = \sum_{k=0}^{L} x_{i-k} \cdot b_k \tag{7}$$

can only be parallelised for the *products* in the sum. The summation itself has to be either performed in a sequential loop, or by successively cutting the summand array into halves and adding these halves in parallel operations. With the abbreviation $s_{i,k} = x_{i-k} \cdot b_k$ the sum of eq. 7 can be rewritten as

$$y_i = \sum_{k=0}^{L/2} s_{i,k} + s_{i,k+L/2+1} \tag{8}$$

This procedure of cutting the summand array into halves can be repeated $\log_2 L$ times.

The plugin accepts mono audio data from the VST host, and provides stereo audio data.

The class `HRTFModule` connects the VST plugin and the filter component. The class diagram is given in Fig. 4.

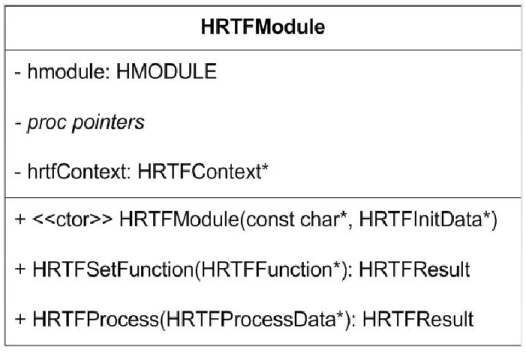

Figure 4. HRTF filter component class diagram

The data structure `HRTFProcessData` contains the necessary data for the `HRTFModule` to perform the HRTF rendering.

Four filters are referenced by their ID, containing the coefficients for the interpolation limits for azimuth and elevation angles. Two

```
typedef struct HRTFProcessData {
  int       numSamples;       // I/O buffer size
  int       filterID[4];      // Filters for interpolation
  float     weightStart[4];   // Weights at block start
  float     weightEnd[4];     // Weights at block end
  float     *input;           // Mono input signal
  float     *leftOutput;      // Stereo output: left
  float     *rightOutput;     // Stereo output: right
} HRTFProcessData;
```

Figure 5. HRTFProcessData data structure

sets of weight factors contain the interpolation weights at block start and block end, so that a smooth movement of the source can be rendered. A problem occurs when the source position crosses the limit of the current interpolation cell. Possible solutions were either to extrapolate beyond the limits or to limit the source movement to the end of the interpolation interval and continue with a new interpolation interval in the next blocks (see Fig. 6). Prototypes for both approaches were implemented in Matlab. It turned out, that the extrapolation approach resulted in audible clicks at block limits, whereas there were no audible clicks in the second approach.

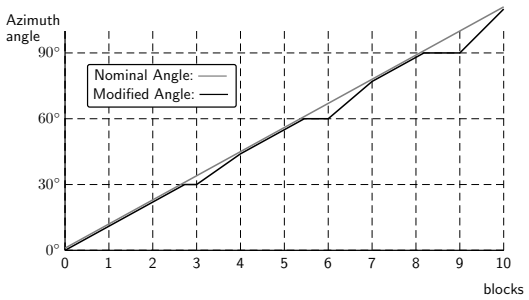

Figure 6. Dynamic angle interpolation at interval limits. The gray line shows the desired values of the azimuth angle of the virtual source, the black line shows the actual rendered angles due to block-wise data processing.

C. Filter Design

HRTF filters were designed in Matlab as FIR filters using previously measured data of a dummy-head system. Frequency and phase responses were measured at azimuth angles from 0° to 180° in steps of 30°. The data for the interval from 180° to 360° were obtained by interchanging the left and right channels. Elevation angles were -45°(below), 0°(plane), and 45°(above). In order to easily configure the plugin filter components, a XML DTD was defined, and the Matlab filter design program produced a corresponding XML file as output, which in turn could be read by the filter component. For XML parsing the TinyXML library is used [8].

D. Code Instrumentation

An audio plugin is a realtime application and as such very sensitive to modifications of the runtime environment of the plugin. This makes it difficult or impossible to use standard tools like debuggers and profilers. The approach used in this project is code instrumentation. Each call to the `process` methode of the filter module is framed with calls to the start and stop methods of

 ISBN: 978-1-61208-157-1

the timer object which in turn calls the Windows API function `QueryPerformanceCounter`, to get high resolution timing information. The stop method calculates the elapsed time and updates

```
HRTFResult HRTFProcess(HRTFProcessData* data)
{
  REQUIRE(hrtfProcessPtr);
  timer.start();
  HRTFResult result = hrtfProcessPtr(hrtfContext, data);
  timer.stop();
  return result;
}

// .. The timer methods
inline void start()
{
  QueryPerformanceCounter(&t0);
}

inline double stop()
{
  QueryPerformanceCounter(&t1);
  double cur = (t1.QuadPart - t0.QuadPart) * factor;
  avg = (avg + cur) * 0.5;
  max = max > cur ? max : cur;
  min = min < cur ? min : cur;

  return cur;
}
```

Figure 7. Code Instrumentation for Performance Measurement

the log. This approach implements a performance measurement with minimum interference with the normal plugin operation. It has to be noted tough, that the measured time is "wall-clock time", not CPU time, so to a small extent the measured results depend on the scheduling of the plugin by the operating system.

IV. Results and Discussion

A. Listening Impression

All filter components were tested with pieces of solo vocal music. The filters produced naturally sounding position rendering of the input audio material without clicks and other artefacts when moving the controls to change the virtual source location.

B. Performance Measurements

All performance measurements were executed with 20 seconds of audio playback. During this time the azimuth and elevation angles of the virtual source are varied according to a path that has been recorded once and was replayed by the automation functionality of the VST host.

Module	t_{min}/ms	t_{max}/ms	t_{avg}/ms
CUDA-C			
Fast Convolution	2	56	2
OpenCL			
Time Domain Convolution	8	18	10
CPU-SSE2			
Time Domain Convolution	12	36	17
CPU.dll			
Time Domain Convolution	12	31	22

It can be seen, that the GPU algorithms perform significantly faster than the FPU algorithms, which was to be expected. An irritating observation is the large maximum value of the execution time for the CUDA filter module, while the average execution time is equal to the minimum execution time. This indicates, that this long time has occurred very few times, probably only once. Unfortunately the way of code instrumentation does not give any information, when and how often such a large execution time occurs. It can be assumed, that this large time occurs during the setup phase of the FFT algorithm (during setup a *plan* is created). If this assumption is confirmed by further experiments, the creation of the FFT plan can be moved to the plugin constructor, so it would not cause audio dropouts.

V. Conclusion

In this article, a GPGPU based realtime audio effects processor was presented. In particular, spatial audio rendering by filtering the mono input signal with the head related transfer functions for the corresponding angle of sound incidence has been performed. The FIR filtering algorithm has been moderately customised to exploit the benefits of GPGPU computing, leading to an increase in computation speed by a factor of 2.2.

During the listening and performance measurement test no observable memory errors occurred. A systematic test for memory error problems has to be conducted. For high reliability requirements a graphics card dedicated to GPGPU computing must be employed. These cards have ECC protected memory, which consumer level cards do not have.

The reason for the large maximum execution time for the CUDA fast convolution implementation has to be identified and removed.

VI. Acknowledgements

This article is the result of a project course on professional audio application development held at the Hamburg University of Applied Sciences (HAW) together with Steinberg Media Technologies. Thanks to the Steinberg developers, especially Ralf Kürschner, Dr. Nico Becherer and Yvan Grabit for their support and thanks to the students Leonhard Dahl, Tobias Hassenklöver, Ines Ouanes, Marcus Rohwer, Areg Siradeghyan, Alexander Vette, and Özhan Yavuz for their contributions to the software.

References

[1] *OpenCL Programming Guide for the CUDA Architecture*, 3rd ed., NVIDIA Corp., 2010.

[2] *NVIDIA CUDA C Programming Guide*, 3rd ed., NVIDIA Corp., 2010.

[3] "Khronos opencl api registry," Khronos Group, Accessed 2011-05-05. [Online]. Available: http://www.khronos.org/opencl/

[4] W. Fohl, J. Reichardt, and J. Kuhr, "A System-On-Chip Platform for HRTF-Based Realtime Spatial Audio Rendering," in *Proc. of the Second International Conference on Creative Content Technologies (Content10)*, Lisbon, Portugal, 2010.

[5] M. Dimitrov, M. Mantor, and H. Zhou, "Understanding software approaches for GPGPU reliability," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 94 –104, ACM ID: 1513907.

[6] "Why choose tesla," NVIDIA Corp., Accessed 2011-05-05. [Online]. Available: http://www.nvidia.com/object/why-choose-tesla.html

 ISBN: 978-1-61208-157-1

[7] “Vst plugin sdk,” Accessed 2011-05-02. [Online]. Available: http://www.steinberg.net/en/company/developer.html

[8] L. Thomason, Y. Berquin, and A. Ellerton, “TinyXML Project Page,” Accessed 2011-04-28. [Online]. Available: http://www.grinninglizard.com/tinyxml/index.html

 ISBN: 978-1-61208-157-1