GPU-BASED SPECTRAL MODEL SYNTHESIS FOR REAL-TIME SOUND RENDERING

Pei-Yin Tsai, Tien-Ming Wang, and Alvin Su
SCREAM Lab
National Cheng Kung University
Tainan, Taiwan
vivian7684@gmail.com

ABSTRACT

The timbre of an instrument is usually represented by sinusoids plus noise. Spectral modeling synthesis (SMS) is an audio synthesis technique which can create musical timbre and give control over the frequency and amplitude. Additive synthesis and LPC synthesis are usually applied for synthesizing sinusoids and residuals, respectively. However, it takes fairly large computing power while implementing the algorithms. The purpose of this paper is to present GPU-based techniques of implementing SMS for real-time audio processing by using parallelism and programmability in graphics pipeline. The performance is compared to CPU-based implementations.

1. INTRODUCTION

In recent years, audio synthesizer is a fundamental component in most of multimedia systems. The software version of synthesizer presents significant advantages such as great flexibility for porting. However, some algorithms are computational intensive when synthesizing lots of channels is required. It is limited by the computational capacity of CPU. Even though current CPUs are so powerful that they can handle most common audio processing tasks, it is hard for them to accomplish such real-time missions. Therefore, parallel architectures are likely to improve the efficiency.

On the other hand, modern video cards have presented theoretical throughput capabilities that highly exceed most CPUs. Graphics hardware companies have recently developed technologies such as CUDA (NVIDIA) [1] and CTM (AMD) [2] which are oriented toward general-purpose processing in order to reflect a demand for offloading compute-intensive processes to GPUs. Unlike CPUs, however, GPUs have parallel many-core architecture, and each core is capable of handling thousands of threads simultaneously. In CUDA, a GPU belongs to one abstract object named *grid*. The grid consists of a number of abstract objects named *blocks*. The minimum process unit called *thread* is capable of executing user-specified *kernel function* in parallel. The flexibility is obviously taken into account while user can specify the number of threads and blocks for a certain application.

Some researchers presented the performance of video cards by implementing several common algorithms such as matrix multiplication [3, 4], fast Fourier transform [5, 6], Viterbi algorithm [7], and digital filter design [8, 9]. Moreover, a number of techniques about audio processing such as sound spatialization [10, 11] and modal synthesis [12] were proposed by using GPU-based implementation. The parallel programming model for prevalent algorithms is the most important topic discussed by above literatures. In this paper, techniques for implementing spectral modeling synthesis (SMS) [13] are presented for real-time audio processing by using parallelism and programmability in graphics pipeline.

The paper is organized as follows. The introduction of spectral modeling representation is given in Section 2. The detail implementation of spectral modeling synthesis on GPUs is described in Section 3. Section 4 presents GPU performances comparing to CPU. Finally, Section 5 concludes with a brief summary.

2. DESCRIPTION OF SPECTRAL MODELING REPRESENTATION

As mentioned in [13], sounds are modeled as stable sinusoids (deterministic component) plus noise (residual component). The deterministic component is assumed to be represented by harmonics, s_h, while the residual component is represented by a filtered noise, s_n. Therefore, the input sound can be represented by sinusoidal model formulation as

$$\begin{aligned} s(t) &= s_h(t) + s_n(t) \\ &= \sum_{p=1}^{P(t)} A_p(t)\cos\left[\theta_p(t)\right] + s_n(t) \end{aligned} \tag{1}$$

, where $A_p(t)$, and $\theta_p(t)$ are the instantaneous amplitude and phase of p^{th} harmonic, respectively. $P(t)$ denotes the number of harmonics included in the harmonic part. If the estimation the fundamental frequency ω_0 is achieved, equation 1 can be replaced by

$$s(t) = \sum_{p=1}^{P(t)} A_p(t)\cos\left[p\omega_0 t + \phi_p\right] + s_n(t) \tag{2}$$

, where ϕ_p the initial phase of p^{th} harmonic.

By using time-domain additive synthesis, the deterministic component is able to be generated with neglecting phase information. Therefore, the fundamental frequency and the amplitude of each harmonic are recorded at each time unit which is set as 10 milliseconds. On the other hand, the synthesis of the stochastic component can be understood as the generation of a noise signal that has the spectral envelopes of the stochastic representation. A low-order LPC filter can completely characterize the residual by encoding its amplitude and spectral features. The prediction model can be represented as

$$\hat{s}_n(n) = -\sum_{i=1}^{P} a_i s_n(n-i), \tag{3}$$

where $\hat{s}_n(n)$ is the predicted noise signal, $s_n(n-i)$ the previous observed noise samples, and a_i the predictor coefficients. By

taking the white noise as the excitation, we can generate the stochastic component by an all-pole filter with the coefficients a_i.

3. SPECTRAL MODELING SYNTHESIS ON GPUS

Several algorithms are presented on GPUs in this section. In the simulation, hundreds of instruments are synthesized using SMS. Each of them comprises 50 partials. The sampling rate is 44100 Hz. Furthermore, for real-time purpose, application programs manage concurrency through *streams*. A stream is a sequence of commands that execute in order. On the other hand, different streams may execute their commands out of order concurrently or with respect to one another. The length of a stream here is defined as one time unit (10ms).

3.1. Deterministic part

Two algorithms are implemented for the generation of sinusoids to test the speedup. Let there be N instruments to be synthesized. The other method is designed for residual part.

Algorithm 1

Here one stream represents a frame of synthetic audio samples. The synthetic sound is stereo; sinusoids of N/2 instruments are synthesized in each channel. Based on CUDA's structure, we set 2 blocks per grid, each of them deals with one channel. Then 441 threads are constructed for each block because the length of a stream is one time unit (10 ms). A straightforward implementation using C is shown in Appendix A. Figure 1 shows the memory model of this algorithm.

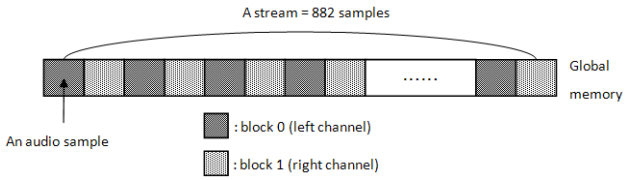


Figure 1: *The memory model of Algorithm 1.*

Algorithm 2

In *Algorithm 1*, threads deal with (N/2)*50 sinusoids to generate an audio sample simultaneously. In this case, *Algorithm 1* is re-designed with two kernel functions by using more GPU resources to shorten the execution time. The first one allocates an **N/2-by-882** memory space in order to store the samples of one stream. There are 441 blocks being included in one grid and N threads in one block. Every thread calculates the value of one instrument. Figure 2 shows the memory model of the first kernel function.

The second kernel function is designed for calculating the samples. We use 2 blocks to construct one grid for stereo. Each block contains 441 threads which sum up the values of the N/2 instruments at each time instant. Figure 3 shows the memory model of the second kernel function. The kernel functions are shown in Appendix B.

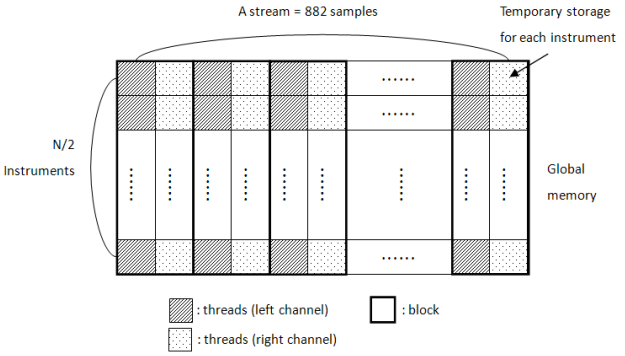


Figure 2: *The memory model of the first kernel function in Algorithm 2.*

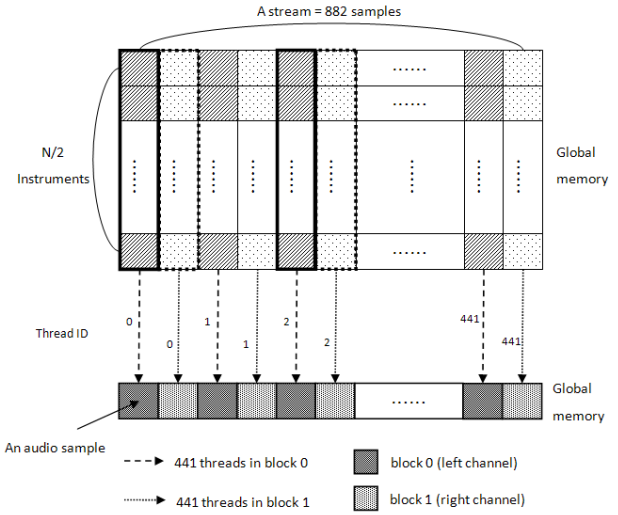


Figure 3: *The memory model of the second kernel function in Algorithm 2.*

3.2. Residuals

In this part, we apply an all-pole filter to generate residuals. In a recursive filter, however, the value of an output sample depends on the values of previous ones. Such values may not be available in parallel computation architecture.

Two kernel functions are presented in this stage. The first one allocates a 2-D memory with **441*n-by-N**, where **n** is the number of time units in a granularity which stands for the latency of updating parameters when synthesizing sounds. Then one block is set in a grid and N threads in a block. Each thread is in charge of generating residual samples of some instrument in a certain granularity. The difference between deterministic part and residual part is that the samples of one synthesized instrument cannot be computed in parallel in residual part because the dependency exists in such a prediction model. Figure 4 shows the memory model of the first kernel function.

The second kernel function adds the residuals to the sinusoids. The parameters are set as **n** block per grid, 441 threads per block. Each block here is responsible for one time unit. Each thread in a certain block is going to add the residual values to the corresponding sinusoidal samples. Figure 5 shows the memory model of the second kernel function. The kernel functions are shown in Appendix C.

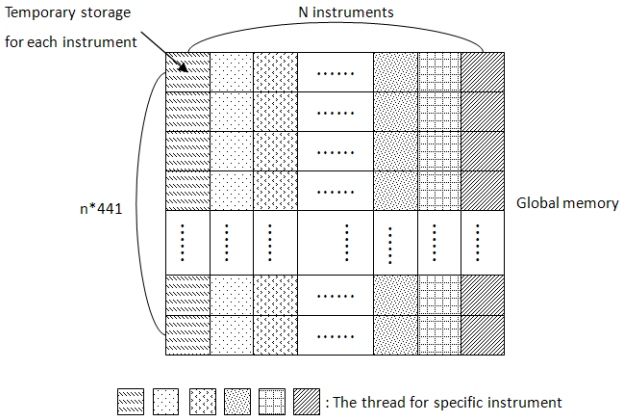


Figure 4: *The memory model of the first kernel function of residual part.*

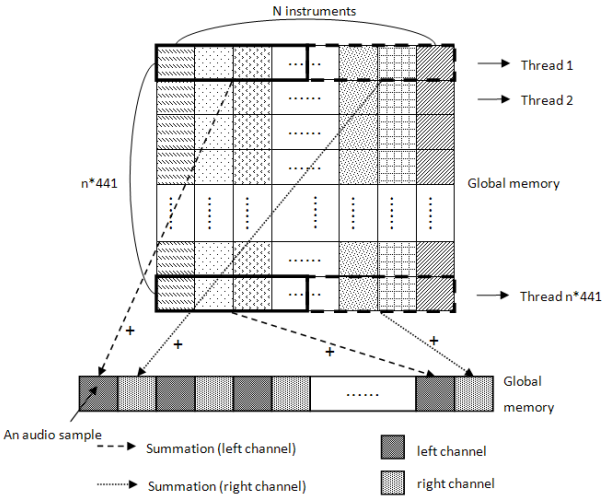


Figure 5: *The memory model of the second kernel function of residual part.*

4. RESULTS AND DISCUSSION

The environment of the SMS CUDA implementation is presented in Table 1. Table 2 shows the execution time of SMS in the experiment. These GPUs are able to achieve the task (synthesizing 5000 sinusoids + residual for each sample) in real time when alternative GPU implementations are used. CPU implementations fail in all cases. Though Intel Q6600 CPU is more powerful then T8300, the CPU execution time of personal computer is still quite similar to notebook. One reason is that the testing program executed on CPU is implemented on single thread, which is independent of the number of CPU cores. Figure 6 shows the line charts of Table 2. Table 3 shows that *Algorithm 2* has the most significant speed up.

	Machine 1 (Notebook)	***Machine 2 (PC)***
OS	Windows 7	Windows XP
CPU	Intel® Core™ Duo Mobile Processor T8300 (2.4 GHz)	Intel® Core™ Quad Processor Q6600 (2.4 GHz)
GPU	GeForce 9500M GS	Tesla C1060
CUDA Capability revision number	1.1	1.3
Number of multiprocessors	4	30
Number of cores	32	240
Clock rate	0.95 GHz	1.30 GHz

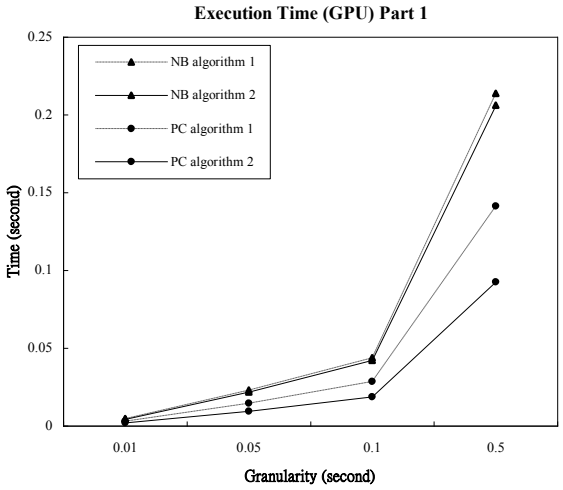
Table 1: *The specifications of two machines.*

Granularity (sec)	***NB(sec)***			***PC(sec)***		
	CPU	***Alg 1***	***Alg 2***	***CPU***	***Alg 1***	***Alg 2***
0.01	**0.160**	0.0047	0.0044	**0.152**	0.003	0.002
0.05	**0.799**	0.023	0.022	**0.761**	0.015	0.010
0.1	**1.599**	0.044	0.042	**1.522**	0.029	0.019
0.5	**7.993**	0.214	0.206	**7.611**	0.141	0.093
1	**15.99**	0.420	0.409	**15.222**	0.282	0.185
5	**79.93**	2.134	2.050	**76.108**	1.406	0.924
10	**159.86**	4.238	4.098	**152.217**	2.805	1.850

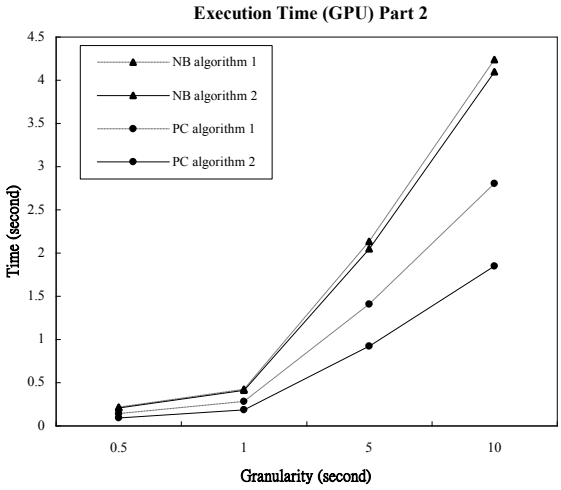
Table 2: *The execution time of SMS with different granularities. (N=100)*

Granularity (sec)	***NB(sec)***		***PC(sec)***	
	Alg 1	***Alg 2***	***Alg 1***	***Alg 2***
0.01	34.012	36.332	50.739	76.108
0.05	34.752	36.665	52.129	80.114
0.1	36.414	37.881	53.409	81.399
0.5	37.403	38.782	53.863	82.191
1	38.026	39.057	53.997	82.146
5	37.454	38.992	54.116	82.404
10	37.719	39.008	54.272	82.293

Table 3: *The speed up of GPUs comparing to CPUs. (N=100)*



(a)



(b)

Figure 6: *The line charts of table 1. Granularity is presented by two parts depending on (a) smaller and (b) larger than 0.5 second.*

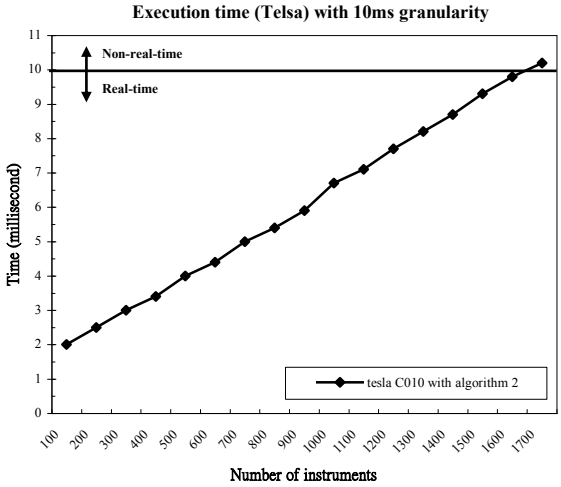


Figure 7: *The execution time of synthesizing different numbers of instruments on Tesla C1060 by using Algorithm 2.*

The performance limitation of Tesla is also tested by increasing the number of synthetic instruments. The execution time of implementing *Algorithm 2* by synthesizing different number of instruments is shown in Figure 7. It is executed on Tesla and the granularity is 10 ms. The execution time is growing intuitively and almost linearly with the number of instruments. When the number is larger than 1700, the computing power will be so huge that *Algorithm 2* may not be performed in real time.

5. CONCLUSION

In this paper, we present efficient implementations of SMS on GPUs. The alternative solutions are presented and discussed based on CUDA architecture, and particularly designed for additive synthesis. In addition, we compare the methods on two different hardware platforms. This may enable new sound rendering applications that require real-time processing with a huge number of target sources.

6. APPENDIX

Appendix A. Kernel function of algorithm 1

```
__global__ void sinusoidal_mehod1(
        float* freq ,           // pointer to frequency array in global memory
        float* amplitude,    // pointer to amplitude array in global memory
        short* samples,      // pointer to sample array in global memory
        int streamNo,        // the stream index
        float timeOffset     // the time offset of current granularity
) {
        float t = (float)(threadIdx.x) / sampleRate            // sampleRate : 44100Hz
            + (float)streamNo*UnitTime + timeOffset ;  // UnitTime : 0.01s
        float sum = 0;
        // blockIdx.x: 0 for left channel, 1 for right channel.
        if ( blockIdx.x == 0 ) {
            // block 0 computes the instrument index 0 – 49 for left channel
            // d_nInstrument : number of instruments. (N=100)
             for ( int i = 0 ; i < d_nInstrument/2 ; i ++) {
               // each instrument has 50 partials
               // d_ nPartials: number of partials of one instrument.
               for ( int k = 0 ; k < d_nPartials ; k ++) {
               // texPartials : the look-up table of normalized partial energy
                    sum += amplitude [ i ] *
                            tex2D( texPartials, i*d_nPartials + (k-1), streamNo ) *
                            __sinf(2 * Pi * freq[i] * k * t) ;
               }
            }
            samples [ threadIdx.x*2 ] = (short) sum ;
      }
      else{
            // block 1 compute the instrument index 50 – 99 for right channel
            for ( int i = d_nInstrument/2 ; i < d_nInstrument ; i ++) {
                for ( int k = 1 ; k <= d_nPartials ; k ++) {
                    sum += amplitude [ i ] *
                        tex2D( texPartials, i*d_nPartials + (k-1), streamNo ) *
                        __sinf(2 * Pi * freq[i] * k * t) ;
                }
            }
            samples[ threadIdx.x*2 +1 ] = (short) sum ;
       }
}
```

Appendix B. Kernel function of algorithm 2

```
/********** first kernel **********/
__global__ void sinusoidal_method2_1 (
        float* freq ,           // pointer to frequency array in global memory
        float* amplitude,    // pointer to amplitude array in global memory
        short* samples,      // pointer to sample array in global memory
        int streamNo,        // the stream index
        float timeOffset     // the time offset of current granularity
){
        float t = (float)(threadIdx.x) / sampleRate            // sampleRate : 44100Hz
            + (float)streamNo*UnitTime + timeOffset ;  // UnitTime : 0.01s
        float sum = 0;
        // The frequency and amplitude in one block is the same.
        __shared__ float f;
        __shared__ float a;
        If ( threadIdx.x == 0 ) {
                        f = freq [ blockIdx.x ];
                        a = amplitude [ blockIdx.x ];
        }
        __syncthreads();
        // one instrument has 50 partials
        for ( int k = 1 ; k <= d_nPartials ; k ++) {
            // texPartials : the look-up table of normalized partial energy
            sum += a *
                tex2D( texPartials, blockIdx.x*d_nPartials + (k-1), streamNo )
                * __sinf(2 * Pi * f * k * t) ;
        }
        // assign sum register to temporary storing matrix in global memory
```

```
        stored_matrix[ blockIdx.x * blockDim.x + threadIdx.x ] = sum;
}
/********** second kernel **********/
__global__ void sinusoidal_method2_2 ( short* samples ){
            float sum = 0;
            // blockIdx.x: 0 for left channel, 1 for right channel.
            if ( blockIdx.x == 0 )      {
               // block 0 compute the instrument index 0 – 49 for left channel
               for ( int i = 0 ; i < d_nInstrument/2 ; i ++)
                           sum += stored_matrix[ threadIdx.x + i * blockDim.x ];
               // assign sum register to sample array in global memory
               samples[ threadIdx.x*2 ] = (short) sum;
            }
            else {
                // block 1 compute the instrument index 50 – 99 for right channel
                // d_nInstrument : number of instruments. (N=100)
               for ( int i = d_nInstrument/2 ; i < d_nInstrument ; i ++)
                           sum += stored_matrix[ threadIdx.x + i*blockDim.x ];
               // assign sum register to sample array in global memory
               samples[ threadIdx.x*2 +1 ] = (short) sum;
            }
}
```

Appendix C. Kernel function of residual part

```
/********** first kernel **********/
__global__ void noise_1(
        int* noise          // pointer to noise array in global memory
){
      int index = 0;
      for( int i = 0 ; i < fsize ; i++ ) {  // fsize: the frame size (= granularity size)
         // index : the index of temporary matrix in global memory
         // FILTER_ORDER : the order of LPC filter (5)
         index = (i+ FILTER_ORDER)*blockDim.x + threadIdx.x ;
         // (i%n) is equivalent to (i&(n-1)) if the second number is a power of two.
         // WHITE_NOISE_LENGTH : 10240
         noise[ index ] =
               tex1Dfetch( texWhiteNoise, i&(WHITE_NOISE_LENGTH-1) );
         for ( int k = 1 ; k <= FILTER_ORDER ; k++ )
               //d_a : the array for storing the preceding noise samples
               noise[ index ] -= (int)( d_a[ k-1 ] *
                           (float)noise[ index - k * blockDim.x ] );
      }
}
/********** second kernel **********/
__global__ void noise_2(
        float* d_g,         // pointer to filter gains in global memory
        short* samples,   // pointer to synthetic sample array in global memory
        int* noise          // pointer to noise array in global memory
){
        int index = blockIdx.x * blockDim.x + threadIdx.x ;
        int noiseL = 0, noiseR = 0;
        int halfInstrumentNum = N_INSTRUMENT/2;
        // halfInstrumentNum : 100/2 = 50

        for( int i = 0 ; i < halfInstrumentNum; i++ ) {
            // left channel
            noiseL += (int)((float)
                  noise[ (index+FILTER_ORDER)*N_INSTRUMENT + i ] *
                  d_g [ i ] / (float)halfInstrumentNum ) ; // d_g : filter gain
            // right channel
            noiseR += (int)((float)
                  noise[ (index+FILTER_ORDER)*N_INSTRUMENT + i +
                  halfInstrumentNum ] *
                  d_g [ i + halfInstrumentNum ] / (float)halfInstrumentNum ) ;
        }
        // truncation of left channel
        if( noiseL > SHRT_MAX)              noiseL = SHRT_MAX;
        else if ( noiseL < SHRT_MIN)        noiseL = SHRT_MIN;
        //add noise to the correspond sample value
        samples[ index*2 ] += (short) noiseL;

        // truncation of right channel
        if( noiseR > SHRT_MAX)              noiseR = SHRT_MAX;
        else if ( noiseR < SHRT_MIN)        noiseR = SHRT_MIN;
        //add noise to the correspond sample value
        samples[ index*2+1 ] += (short)noiseR ;
}
```

7. REFERENCES

[1] NVIDIA. "NVIDIA GPU Computing Developer Home Page," http://developer.nvidia.com/object/gpucomputing.html.

[2] J. Hensley, “AMD CTM overview,” in *International Conference on Computer Graphics and Interactive Techniques*, San Diego, California, 2007, pp. 7.

[3] K. Fatahalian, J. Sugerman, and P. Hanrahan, “Understanding the efficiency of GPU algorithms for matrix-matrix multiplication,” *in Proceedings of the ACM SIGGRAPH /EUROGRAPHICS conference on Graphics hardware*, Grenoble, France, 2004, pp. 137.

[4] E. Larsen, and D. McAllister, “Fast matrix multiplies using graphics hardware,” in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, Denver, Colorado, 2001, pp. 55-55.

[5] K. Moreland, and E. Angel, “The FFT on a GPU,” in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, San Diego, California, USA, 2003, pp. 112-119.

[6] T. Jansen, B. von Rymon-Lipinski, N. Hanssen et al., "Fourier volume rendering on the GPU using a split-stream-FFT," in *Proceedings of the Vision, Modeling, and Visualization Conference*, Stanford, California, USA, 2004, pp. 395–403.

[7] D. Horn, M. Houston, and P. Hanrahan, “ClawHMMER: A Streaming HMMer-Search Implementatio,” in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Seattle, Washinton, 2005, pp. 11.

[8] A. Smirnov, and T. Chiueh, "An Implementation of a FIR Filter on a GPU," Tech. rep., Experimental Computer Systems Lab, Stony Brook University, 2005. http://www. ecsl. cs. sunysb. edu/fir, 2005.

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

[10] N. Tsingos, E. Gallo, and G. Drettakis, “Breaking the 64 spatialized sources barrier,” Gamasutra Audio Resource Guide 2003, 2003.

[11] E. Gallo, and N. Tsingos, "Efficient 3D audio processing with the GPU," in *ACM Workshop on General Purpose Computing on Graphics Processors*, Los Angeles, California, 2004.

[12] Q. Zhang, L. Ye, and Z. Pan, “Physically-based sound synthesis on GPUs,” Lecture notes in computer science, vol. 3711, pp. 328, 2005.

[13] X. Serra, and J. Smith III, “Spectral modeling synthesis: A sound analysis/synthesis system based on a deterministic plus stochastic decomposition,” *Computer Music Journal*, vol. 14, no. 4, pp. 12-24, 1990.