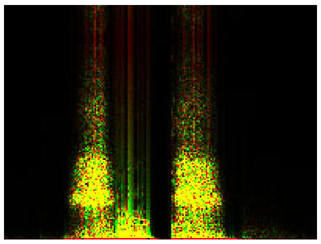
GPUSpectrogram Sample

Collapse All

This sample demonstrates how to perform data processing on the GPU without creating a window or swapchain. In this case, it constructs a spectrogram from wav data using the GPU.



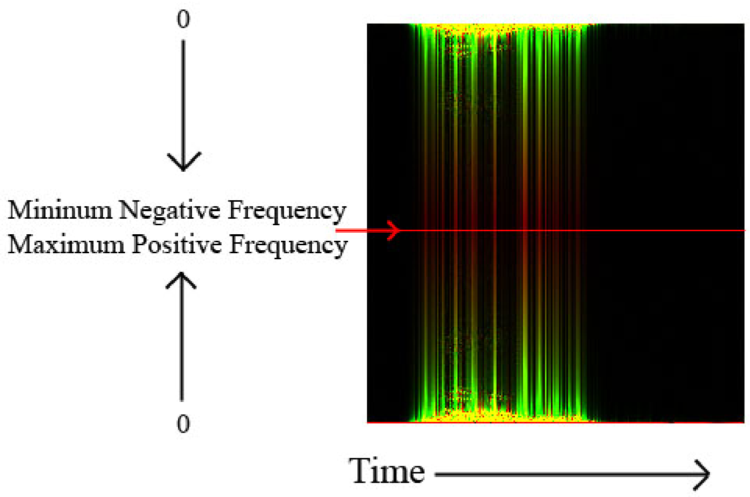
Path

Source	*SDK root*\Samples\C++\Direct3D10\GPUSpectrogram
Executable	*SDK root*\Samples\C++\Direct3D10\Bin\x86 or x64\GPUSpectrogram.exe

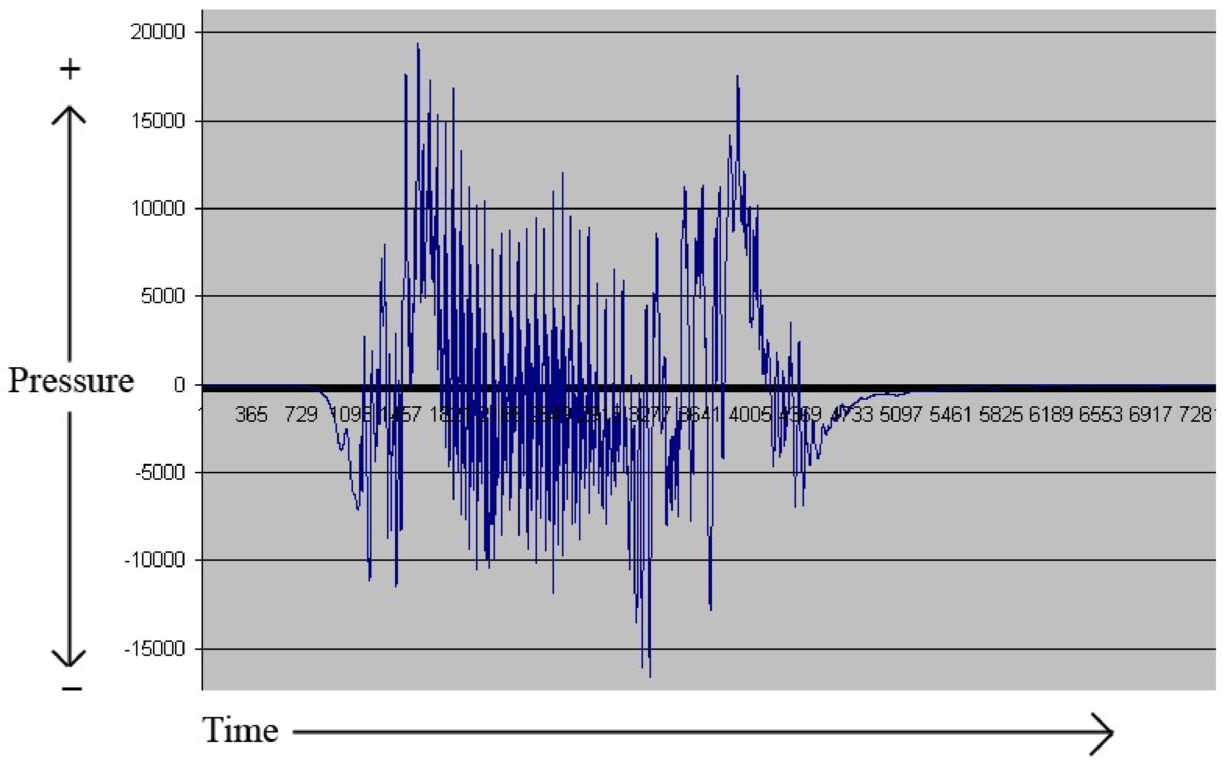
General-Purpose Processing on the GPU

GPUSpectrogram demonstrates how to use the power of the GPU for non-graphics related tasks. In this case, the GPU creates a spectrogram from data in a wave file. The application does not create a window or swap chain and makes use of Render-to-Texture type operations for computation.

A spectrogram represents sound data as a graph. The vertical axis represents the frequency components of the sound, while the horizontal axis represents time. The colors represent the intensity and phase shift of that particular frequency.



In a wave file, the data is stored as a one dimensional array, where each value represents air pressure (or amplitude) and each index represents a distinct time step.



In order to convert from the amplitude versus time domain to the frequency versus time domain, the sample employs the use of the Discrete Fourier Transforms (DFTs). Specifically, the sample uses a subset of Discrete Fourier Transforms known as Fast Fourier Transforms (FFTs). They are considered fast because their O(N*log(N)) complexity is much lower than the O(N*N) complexity of general DFTs.

FFTs only convert amplitude versus time data into frequency data. In order to get frequency data over time, we must split the incoming audio data into *windows*. Each *window* consists of a subsequent set of N audio samples, where N is the *window* size. For example, if the *window* size is 16 samples, then the first *window* consists of samples 0 to 15. The second consists of samples 16-31, and so on.

Because each *window* represents the amplitudes encountered during a specific time interval, the FFT of any particular *window* will represent the frequencies encountered during that specific time interval. Putting these FFTs next to each other on a graph gives a plot of frequency components versus time.

How the Sample Works

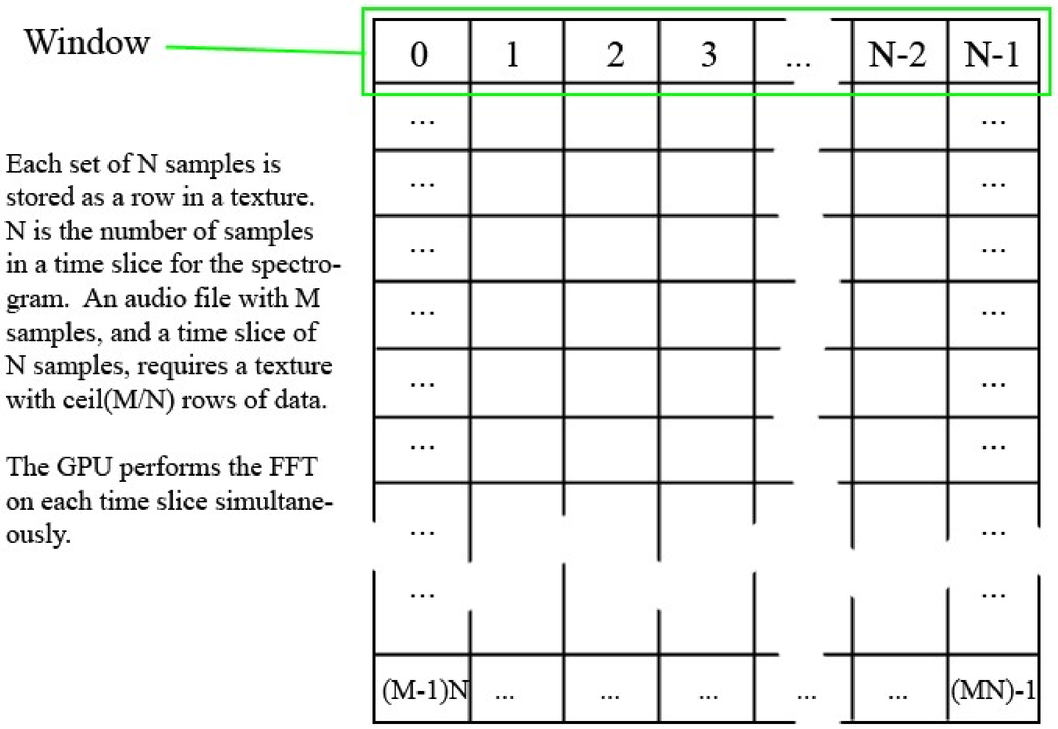
Loading Audio Data

The CAudioData class handles loading the sound data from the wave file. Internally it utilizes the CWaveFile sample class to load the actual data from the wave file. The CAudioData class splits the wave data into channels and stores the data as floating point numbers regardless of the internal format of the wave file.

Getting Audio Data onto the GPU

The audio data loaded from the CAudioData class is placed into a texture. Because the FFT works on complex numbers, the texture contains two color channels, red and green. Red represents the real components corresponding to the amplitudes in the wave file. Green represents the imaginary components and is set to 0 when the texture is populated with wave data. For GPUSpectrogram, two such textures are used. One is used as the render target while the other is bound as a texture input. When the operation is complete, the two are switched.

The texture is exactly N elements wide, where N is the size of the *window*. Each row of the texture contains exactly N audio samples. For a wave file that contains M audio samples, the texture will be N by ceil(M/N) texels in size. This allows the GPU to perform an FFT on each row of the texture in parallel.



Each set of N samples is stored as a row in a texture. N is the number of samples in a time slice for the spectrogram. An audio file with M samples, and a time slice of N samples, requires a texture with ceil(M/N) rows of data.

The GPU performs the FFT on each time slice simultaneously.

GPU FFT Algorithm

The bulk of the GPUSpectrogram calculations are performed using Render-to-Texture type operations. In these operations, the viewport is set to the exact size of the texture. The texture with the most recent data is bound as the input texture, while the other texture is bound as the render target. A quad that has texture coordinates that correspond exactly with the input texture is draw to exactly cover the render target. When the operation completes, the render target texture contains the most recent data.

GPUSpectrogram leverages the Danielson-Lanczos FFT algorithm. This document will not go into the details of the algorithm, only the GPU implementation of it. The first part of the algorithm sorts the audio data according to the reverse bit order of the column index. This is handled in the PSReverse pixel shader. This shader uses the subroutine ReverseBits to find the location of the data that will fill the current position after the sort.

```
uint ReverseBits( uint x )
{
         //uses the SWAR algorithm for bit reversal of 16bits
         x = (((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1));
         x = (((x & 0xcccccccc) >> 2) | ((x & 0x33333333) << 2));
         x = (((x & 0xf0f0f0f0) >> 4) | ((x & 0x0f0f0f0f) << 4));      //8bits

         //uncomment for 9 bit reversal
         //x = (((x & 0xff00ff00) >> 8) | ((x & 0x00ff00ff) << 8));    //16bits
         //push us back down into 9 bits
         //return (x >> (16-9) & 0x000001ff;

         return x;
}

//
// PSReverse
//
float4 PSReverse( PSQuadIn input ) : COLOR
{
         uint iCurrentIndex = input.tex.x;
         uint iRevIndex = ReverseBits( iCurrentIndex );

         float2 fTex = float2( (float)iRevIndex, (float)input.tex.y );
         fTex /= g_TextureSize;
         return tex2D( g_samPointClamp, g_txSource, fTex ).xyyy;
}
```

After the reverse sort, the actual FFT takes place. The outer two loops of the Daneilson-Lanczos algorithm are performed on the CPU and consist of setting various shader constants. The inner-loop is performed on the GPU. The final result is that the most recent render target contains the results of the FFT on the data.

```
         //outer two loops
         UINT iterations = 0;
         float wtemp,wr,wpr,wpi,wi,theta;
         UINT n = g_uiTexX;
```

```
UINT mmax = 1;
while( n > mmax )
{
        UINT istep = mmax << 1;
        theta = 6.28318530717959f / ((float)mmax*2.0f);
        wtemp = sin( 0.5f*theta );
        wpr = -2.0f*wtemp*wtemp;
        wpi = sin( theta );
        wr = 1.0f;
        wi = 0.0f;

        for( UINT m=0; m < mmax; m++ )
        {
                //Inner loop is handled on the GPU
                {
                        g_pWR->AsScalar()->SetFloat(wr);
                        g_pWI->AsScalar()->SetFloat(wi);
                        g_pMMAX->AsScalar()->SetInt(mmax);
                        g_pM->AsScalar()->SetInt(m);
                        g_pISTEP->AsScalar()->SetInt(istep);

                        if( 0 == iterations%2 )
                        {
                                g_ptxSource->AsShaderResource()->SetResource( g_pDestTexRV );
                                RenderToTexture( pd3dDevice, g_pSourceRTV, false, g_pFFTInner );
                        }
                        else
                        {
                                g_ptxSource->AsShaderResource()->SetResource( g_pSourceTexRV );
                                RenderToTexture( pd3dDevice, g_pDestRTV, false, g_pFFTInner );
                        }
                        pd3dDevice->OMSetRenderTargets( 1, apOldRTVs, pOldDS );

                        iterations++;
                }

                wtemp = wr;
                wr = wtemp*wpr-wi*wpi+wr;
                wi = wi*wpr+wtemp*wpi+wi;
        }
        mmax = istep;
}
```

Complexity Considerations

A look at the implementation of the FFT shows that the inner loop gets executed N times, where N is the size of the *window*. Because the pixel shader is run for every pixel in the row, and a row is N samples wide, the time complexity could be said to be O(N*N). This is no better than the general DFT. However, taking a closer look at the PSFFTInner pixel shader reveals that most of the work is only done when special cases are met. For each execution of the shader the following happens, texture load, modulus operation, and two if statements. However, if any if the if statements are true, which they are 2*Log(N) times out of N, you have many more operations. Because of this you can state that this implementation is actually an O(M*N*N + 2*L*N*Log(N)) operation, where M < L. This is still not O(N*log(N)), but does perform fewer operations than the O(N*N) DFT.