AMD TrueAudio Next API

Version History

Version	Date	Author	Change
1.0	Aug 16 2016	G. Park	GPU Open release

Table of Contents

1 INTRODUCTION 5

1.1 OVERVIEW 5

1.2 DEFINITIONS, ACRONYMS AND ABBREVIATIONS 5

1.3 DESIGN CONSIDERATIONS 5

1.3.1 Floating point format used for processing 5

1.3.2 Transparent OpenCL context 5

1.3.3 C++ implementation with C wrapper 5

2 API SPECIFICATIONS 6

2.1 LOW LEVEL APIS 6

2.1.1 Status codes 6

2.1.2 Library initialization 7

2.1.3 Basic stream processing functions 8

2.1.4 Fast convolution 10

2.1.5 Response curve generation 12

2.1.6 Music specific functions 14

2.2 HIGH LEVEL APIS 15

3 SAMPLE CODE 15

3.1 ROOM ACOUSTICS DEMO 15

3.2 VST PLUGINS 16

3.3 WWISE PLUGINS 16

1 Introduction

Today, most real-time audio processing on PCs is done on the CPU, or fixed purpose DSPs. AMD APUs and graphics cards have GPUs that could, with some driver changes, be used for realtime audio and other digital signal processing purposes.

1.1 Overview

This document describes an Application Programming Interface (API) for a GPU accelerated audio processing library.

1.2 Definitions, Acronyms and Abbreviations

PCM – *Pulse Code Modulation* a stream of signed integers representing a digitized analog audio signal.

FFT – *Fast Fourier Transform* a fast algorithm implementing the Fourier Transform.

1.3 Design Considerations

1.3.1 Floating point format used for processing

To minimize unnecessary conversions, all signal processing functions in ths API operate on non-interleaved, 1D arrays of floats. Conversion functions are provided to de-interleave and convert interleaved integer arrays to 1D float arrays for input from PCM streams, and convert and re-interleave floats to integer PCM streams for output to PCM streams.

1.3.2 Transparent OpenCL context

OpenCL contexts and queues used by functions in this API are accessible by application code. The developer using this library have control of creation and management of OpenCL contexts and queues.

1.3.3 C++ implementation interface

C++ headers are provided that expose public classes and methods.

2 API Specifications

For clarity, API functions and associated structures will be divided into two broad categories, Low Level, and High Level APIs, Each of these will contain groups of related functions.

2.1 Low Level APIs

2.1.1 Status codes

This section describes status return codes used by all AMD True Audio functions.

```
/*******************************************************************************
Return codes are valid AMD Media Framework (AMF) codes:

errors are positive, OK is zero.

So error check should take the form:

if(status != AMF_OK) {
       //handle error ...
}
*******************************************************************************/
enum AMF_RESULT
{
    AMF_OK                     = 0,
    AMF_FAIL                      ,

// common errors
    AMF_UNEXPECTED                ,

    AMF_ACCESS_DENIED             ,
    AMF_INVALID_ARG               ,
    AMF_OUT_OF_RANGE              ,

    AMF_OUT_OF_MEMORY             ,
    AMF_INVALID_POINTER           ,

    AMF_NO_INTERFACE              ,
    AMF_NOT_IMPLEMENTED           ,
    AMF_NOT_SUPPORTED             ,
    AMF_NOT_FOUND                 ,

    AMF_ALREADY_INITIALIZED       ,
    AMF_NOT_INITIALIZED           ,

    AMF_INVALID_FORMAT            ,// invalid data format

    AMF_WRONG_STATE               ,
    AMF_FILE_NOT_OPEN             ,// cannot open file

// device common codes
    AMF_NO_DEVICE                 ,

// component common codes

    //result codes
    AMF_EOF                       ,
    AMF_REPEAT                    ,
    AMF_INPUT_FULL                ,//returned by AMFComponent::SubmitInput if input queue is full
```

```
//error codes
    AMF_INVALID_DATA_TYPE        ,//invalid data type

// component TAN
    AMF_TAN_CLIPPING_WAS_REQUIRED, // Resulting data was truncated to meet output type's value limits.
    AMF_TAN_UNSUPPORTED_VERSION  , // Not supported version requested, solely for TANCreateContext().

    AMF_NEED_MORE_INPUT          ,//returned by AMFComponent::SubmitInput did not produce buffer
}
```

2.1.2 Library initialization

This section describes True Audio Next library initialization functions. An instance of the AMDTrueAudio class may be initialized for GPU or for CPU. Once initialized, it cannot be re-initialized. If both GPU and CPU operations are required, two AMDTrueAudio class instances may be used, initialized for GPU and CPU respectively.

Note: the CPU mode refers to non-OpenCL code optimized for CPU. It may also be possible to select an OpenCL CPU context, and initialize the class using InitializeForGPU, but not all functions are guaranteed to work, as they may use OpenCL extensions not available in a CPU context.

```
/**************************************************************************************
 TAN object creation functions:
 **************************************************************************************/
// TAN objects creation functions.
extern "C"
{
   // Creates a True Audio Next context. After the Context is initialized, it can be passed to creation
   // functions for Convolution, Converter, FFT, and Math objects.
    TAN_SDK_LINK AMF_RESULT         AMF_CDECL_CALL TANCreateContext(amf_uint64 version,
                                                        amf::TANContext** ppContext);
   // Create a TANConvolution object:
    TAN_SDK_LINK AMF_RESULT         AMF_CDECL_CALL TANCreateConvolution(
                                                        amf::TANContext* pContext,
                                                        amf::TANConvolution** ppConvolution);

    // Create a TANConverter object:
    TAN_SDK_LINK AMF_RESULT         AMF_CDECL_CALL TANCreateConverter(
                                                        amf::TANContext* pContext,
                                                        amf::TANConverter** ppConverter);

    //Create an TANFFT object:
    TAN_SDK_LINK AMF_RESULT         AMF_CDECL_CALL TANCreateFFT(
                                                        amf::TANContext* pContext,
                                                        amf::TANFFT** ppFFT);

    // Create a TANMath object:
    TAN_SDK_LINK AMF_RESULT         AMF_CDECL_CALL TANCreateMath(
                                                        amf::TANContext* pContext,
                                                        amf::TANMath** ppFFT);

    // Set folder to cache compiled OpenCL kernels:
    TAN_SDK_LINK AMF_RESULT         AMF_CDECL_CALL TANSetCacheFolder(const wchar_t* path);
    TAN_SDK_LINK const wchar_t*     AMF_CDECL_CALL TANGetCacheFolder();
}

   //---------------------------------------------------------------------------------------------
   // TANContext interface:
```

```
// TANContext may be initialized for OpenCL using either a cl_context, or one or two
// cl_command_queues.
// the general queue may be shared by application kernels, the convolution queue is meant to be
// dedicated for a convolution object.
//
// NOTE: If TANContext::InitOpenCL is not called, objects initialized with the context will
// use CPU processing only.
//-----------------------------------------------------------------------------------------------
class TANContext : virtual public AMFPropertyStorage
{
 public:

     virtual AMF_RESULT  AMF_STD_CALL    InitOpenCL(
                                             cl_context pContext);

     virtual AMF_RESULT  AMF_STD_CALL    InitOpenCL(
                                             cl_command_queue pGeneralQueue = nullptr,
                                             cl_command_queue pConvolutionQueue = nullptr);

     virtual cl_context   AMF_STD_CALL   GetOpenCLContext();
           virtual       cl_command_queue     AMF_STD_CALL  GetOpenCLGeneralQueue();
           virtual       cl_command_queue     AMF_STD_CALL  GetOpenCLConvQueue();

 };
```

2.1.3 Basic stream processing functions

This section describes utility functions that may be useful in and digital audio signal processing application.

```
//-----------------------------------------------------------------------------------------------
// TANConverter interface
//
// Provides conversion between normalized FLOAT and SHORT representations.
//
// Converts an array of floats int the range - 1.0 -> + 1.0
//    to or from
//    an array of shorts int the range - 32767 -> + 32767
//
//    inputStep      interleave step size for inputBuffer.
//    outputStep     interleave step size for outputBuffer.
//
//    conversionGain = 1.0 gives standard - 1.0 -> + 1.0 to / from - 32768 -> + 32768
//
// NOTE: to interleave or deinterleave data : Use step = 1 for mono data, 2 for stereo, etc.
//-----------------------------------------------------------------------------------------------

AMF_RESULT  AMF_STD_CALL    Convert(short* inputBuffer, amf_size inputStep,
                                    amf_size numOfSamplesToProcess,
                                    float* outputBuffer, amf_size outputStep,
                                    float conversionGain);

AMF_RESULT  AMF_STD_CALL    Convert(float* inputBuffer, amf_size inputStep,
                                    amf_size numOfSamplesToProcess,
                                    short* outputBuffer, amf_size outputStep,
                                    float conversionGain);
```

```
// Method for batch processing
AMF_RESULT  AMF_STD_CALL    Convert(short** inputBuffers, amf_size inputStep,
                                    amf_size numOfSamplesToProcess,
                                    float** outputBuffers, amf_size outputStep,
                                    float conversionGain,
                                    int channels);
AMF_RESULT  AMF_STD_CALL    Convert(float** inputBuffers, amf_size inputStep,
                                    amf_size numOfSamplesToProcess,
                                    short** outputBuffers, amf_size outputStep,
                                    float conversionGain,
                                    int channels);
// methods for GPU memory buffers:
//
AMF_RESULT  AMF_STD_CALL    Convert(cl_mem inputBuffer,
                                    amf_size inputStep,
                                    amf_size inputOffset,
                                    TAN_SAMPLE_TYPE inputType,
                                    cl_mem outputBuffer,
                                    amf_size outputStep,
                                    amf_size outputOffset,
                                    TAN_SAMPLE_TYPE outputType,
                                    amf_size numOfSamplesToProcess,
                                    float conversionGain);

// Method for batch processing
AMF_RESULT  AMF_STD_CALL    Convert(
                                  cl_mem* inputBuffers,
                                  amf_size inputStep,
                                  amf_size* inputOffsets,
                                  TAN_SAMPLE_TYPE inputType,
                                  cl_mem* outputBuffers,
                                  amf_size outputStep,
                                  amf_size* outputOffsets,
                                  TAN_SAMPLE_TYPE outputType,
                                  amf_size numOfSamplesToProcess,
                                  float conversionGain,
                                  int count);


//-----------------------------------------------------------------------------------------------
// TANMath interface
//
// Provides mathematical utility functions.
//
// buffers are arrays of channels pointers to floats, each at least numOfSamplesToProcess long.
//-----------------------------------------------------------------------------------------------

AMF_RESULT ComplexMultiplication(const float* const inputBuffers1[],
                                 const float* const inputBuffers2[],
                                 float *outputBuffers[],
                                 amf_uint32 channels,
                                 amf_size numOfSamplesToProcess);

AMF_RESULT ComplexDivision(const float* const inputBuffers1[],
                           const float* const inputBuffers2[],
                           float *outputBuffers[],
                           amf_uint32 channels,
                           amf_size numOfSamplesToProcess);
```

```
// methods for GPU memory
AMF_RESULT ComplexMultiplication(const cl_mem inputBuffers1[],
                                 const amf_size buffers1OffsetInSamples[],
                                 const cl_mem inputBuffers2[],
                                 const amf_size buffers2OffsetInSamples[],
                                                       cl_mem outputBuffers[],
                                 const amf_size outputBuffersOffsetInSamples[],
                                 amf_uint32 channels,
                                 amf_size numOfSamplesToProcess);

AMF_RESULT ComplexDivision(const cl_mem inputBuffers1[],
                           const amf_size buffers1OffsetInSamples[],
                           const cl_mem inputBuffers2[],
                           const amf_size buffers2OffsetInSamples[],
                                             cl_mem outputBuffers[],
                           const amf_size outputBuffersOffsetInSamples[],
                           amf_uint32 channels,
                           amf_size numOfSamplesToProcess);

//-------------------------------------------------------------------------------------------------
// TANFFT interface
//
//-------------------------------------------------------------------------------------------------
enum TAN_FFT_TRANSFORM_DIRECTION
{
    TAN_FFT_TRANSFORM_DIRECTION_FORWARD = 0,
    TAN_FFT_TRANSFORM_DIRECTION_BACKWARD = 1,
};

// FFT function.
//
// Note: input and output arrays consist of pairs (real, imag).
// Note: 'log2len' sets the length of the FFT's data, which is 2 ^ log2len * 2 (complex).
// Note: CPU implementation currently returns unscaled results for backward transformation
//       (multiplyed by 2 ^ log2len).
// Note: Position and count functionality of TANAudioBuffer isn't supported.
// pBufferInput     pointer to channels input vectors of floats, (complex R, I pairs), to be converted
// pBufferOutput    pointer to channels output vectors of floats, (complex R, I pairs), result

AMF_RESULT  AMF_STD_CALL    Transform(TAN_FFT_TRANSFORM_DIRECTION direction,
                                                  amf_uint32 log2len,
                                                  amf_uint32 channels,
                                                  float* pBufferInput[],
                                                  float* pBufferOutput[]);

AMF_RESULT  AMF_STD_CALL    Transform(TAN_FFT_TRANSFORM_DIRECTION direction,
                                      amf_uint32 log2len,
                                      amf_uint32 channels,
                                      cl_mem pBufferInput[],
                                      cl_mem pBufferOutput[]);
```

2.1.4 Fast convolution

This section describes convolution methods of the AmdTrueAudioConvolution class.

Note: partioned FFT methods are most efficient for long convolution lengths and relatively small buffer sizes, but very small buffer sizes work better with time domain mode.

```
//-------------------------------------------------------------------------------------------------
// TANConvolution interface
//-------------------------------------------------------------------------------------------------
```

```
// Initialization function.
//
// Note: this method allocates internal buffers and initializes internal structures. Should
// only be called once.
AMF_RESULT    AMF_STD_CALL  Init(TAN_CONVOLUTION_METHOD convolutionMethod,
                                  amf_uint32 responseLengthInSamples,
                                  amf_uint32 bufferSizeInSamples,
                                  amf_uint32 channels);

TANContext* AMF_STD_CALL    GetContext();

 // Time domain float data update responce functions.
 //
 // Note: kernel is time domain data, and if shorter or longer than length specified in
 // Init(), it will be truncated or zero padded to fit.
 // Note: buffer contains 'channels' arrays of impulse response data for each channel.
 // Note: there should be as many 'states' and 'flagMasks' as channels in the buffer (set in
 // Init() method).
AMF_RESULT AMF_STD_CALL UpdateResponseTD(float* ppBuffer[],
                                         amf_size numOfSamplesToProcess,
                                         const amf_uint32 flagMasks[],
                                         const amf_uint32 operationFlags);

AMF_RESULT AMF_STD_CALL UpdateResponseTD(cl_mem ppBuffer[],
                                         amf_size numOfSamplesToProcess,
                                          const amf_uint32 flagMasks[],
                                          const amf_uint32 operationFlags);

 // Frequency domain float data update responce functions.
 //
 // Note: kernel is frequency domain complex float data, must be 2 * length specified in
 // Init().
 // Note: buffer contains 'channels' arrays of impulse response data for each channel.
 // Note: there should be as many 'flags' as channels in the buffer (set in Init() method).
 // Note: not currently implemented.
 AMF_RESULT  AMF_STD_CALL    UpdateResponseFD(float* ppBuffer[],
     amf_size numOfSamplesToProcess,
     const amf_uint32 flagMasks[],   // Masks of flags from enum TAN_CONVOLUTION_CHANNEL_FLAG.
     const amf_uint32 operationFlags // Mask of flags from enum TAN_CONVOLUTION_OPERATION_FLAG.
     );

// Convolution process functions.
//
// ppBufferInput            - pointer to a channels long array of arrays of floats to be processed
// ppBufferOutput            - pointer to a channels long array of arrays of floats to take output
// numOfSamplesToProcess    - number of samples, from each array, of input samples to process
// pNumOfSamplesProcessed    - number of samples, from each array, actually processed.
//
// On success:
// returns AMF_OK and pNumOfSamplesProcessed will contain number of samples actually processed. This
// will be numOfSamplesToProcess, rounded down to next lower integral number of bufSize samples.
// On failure: returns appropriate AMF_RESULT value.
// Process system memory buffers:
AMF_RESULT   AMF_STD_CALL    Process(float* ppBufferInput[],
                                     float* ppBufferOutput[],
                                     amf_size numOfSamplesToProcess,
                                     const amf_uint32 flagMasks[],    //
                                     amf_size *pNumOfSamplesProcessed //
                                     );
```

```
// Process OpenCL cl_mem buffers:
AMF_RESULT  AMF_STD_CALL    Process(cl_mem pBufferInput[],
                                    cl_mem pBufferOutput[],
                                    amf_size numOfSamplesToProcess,
                                    const amf_uint32 flagMasks[],    //
                                    amf_size *pNumOfSamplesProcessed //
                                    );
```

2.1.5 [future feature] Response curve generation

Most, though not all, audio processing can be implemented using a convolution of the audio stream data with a short data array or curve. Passing a single 1 value followed by a stream of zeros (an “impulse”) through any convolution reproduces this curve or “impulse response”. This also works in the physical world. For example one way to reproduce the reverberation of a natural cave or architectural space, is to fire a starter’s pistol (the “impulse”) in the space, and record the resulting sound. This recording becomes the “impuse response” for the space, and can be used to recreate the echo’s and reverberations of that space for any sound stream by convolution.

Many traditional analog audio filter processes can also be done using an appropriate impuse response. This section describes AMD True Audio functions for generating impuse responses.

2.1.5.1 Impulse response generators for common audio filters.

This section describes functions and classes for generation of impulse responses which may be used to implement common audio filter functions with AmdTrueAudio::Convolution class.

```
class AmdTrueAudio_Filters {
private:
public:
    AmdTrueAudio_Filters(AmdTrueAudio *ata);
    ~AmdTrueAudio_Filters();

    /*********************************************************************************
    AmdTrueAudio::generateLowPassResponse

    generate a response function to implement a low pass filter.

    response  1D array of floats to receive the generated filter response
    length          length of response
    cornerFreq      Frequency in Hz that above which response begins to decrease
    Q               controls sharpness of frequency roll off

    *********************************************************************************/
    int generateLowPassResponse(float *response, int length, float cornerFreq, float Q);

    /*********************************************************************************
    AmdTrueAudio::generateHighPassResponse

    generate a response function to implement a high pass filter.

    response        1D array of floats to receive the generated filter response
    length          length of response
    cornerFreq      Frequency in Hz that below which response begins to decrease
    Q               controls sharpness of frequency roll off

    *********************************************************************************/
    int generateHighPassResponse(float *response, int length, float cornerFreq, float Q);
```

```
    /******************************************************************************
    AmdTrueAudio::generateParametricEQResponse
    generate a response function to implement a parametric filter.

    response         1D array of floats to receive the generated filter response
    length           length of response
    centerFreq       Frequency in Hz of center of pass/reject band
    value            sets boost or cut amount. 1.0 means flat response.
    Q                controls width of filter band

    ******************************************************************************/
    int generateParametricEQResponse(float *response, int length, float centerFreq, float value,
                                     float Q);

    /******************************************************************************
    AmdTrueAudio::generateNBandEQResponse
    generate a response function to implement an N-band equalizer.

    response         1D array of floats to receive the generated filter response
    length           length of response
    nBands           number of filter bands
    bandCenters      nBand length array of center frequencies
    bandValues       nBand length array values to set boost or cut amount of each band.
                     1.0 means flat response.

    ******************************************************************************/
    int generateNBandEQResponse(float *response, int length, int nBands, float *bandCenters,
                                float *bandValues);

};
```

2.1.5.2 [future feature, sample provided] Simple room reverb stereo response generator.

This section describes functions that can be used to generate impulse responses that simulate the reverberations, stereo amplitude and phase information, for up to two mono sources and two listeners (“ears”) on a simulated human head in a rectangular room.

```
class AmdTrueAudio_VR {
private:
public:
    AmdTrueAudio_VR(AmdTrueAudio *ata);
    ~AmdTrueAudio_VR();

/**********************************************************************************************
AmdTrueAudio::AmdTrueAudio_VR::generateRoomResponse:

Generates an impulse response for a rectangular room, given room dimensions, damping factors for
each of the six walls, source and microphone positions in the room.

**********************************************************************************************/
static void generateRoomResponse( RoomDefinition room, MonoSource source, StereoListener ear,
       int inSampRate, int responseLength, float *responseLeft, float *responseRight, int flags = 0);

static float estimateReverbTime(RoomDefinition room, float finaldB, int *nReflections);

/**********************************************************************************************
AmdTrueAudio::AmdTrueAudio_VR::generateSimpleHeadRelatedTransform:
```

```
Generates a simple head related transfer function (HRTF) for acoustic shadowing as heard by
a human ear on a human head, as a function of angle to a sound source.

This function models the head as a sphere of diameter earSpacing * 1.10, and generates a table of
180 impulse response curves for 1 degree increments from the direction the ear points.

*****************************************************************************************************/
static void generateSimpleHeadRelatedTransform(HeadModel * pHead, float earSpacing);
static void applyHRTF(HeadModel * pHead, float scale, float *response, int length, float earVX,
                      float earVY, float earVZ, float srcVX, float srcVY, float srcZ);
};
```

2.1.6 [future feature] Music specific functions

This section describes functions for music synthesis, that may be useful for efficient game background music generation, or for interactive music related games.

2.1.6.1 Musical waveform synthesis

```
/**************************************************************************************
class AmdTrueAudio::WaveTableSynth

implements a simple wave table synthesizer
**************************************************************************************/
class WaveTableSynth {
private:
public:
       WaveTableSynth(AmdTrueAudio *ata, float samplesPerSecond);
       ~WaveTableSynth();

       // waveform types
       enum TA_WAVE_FORM {
              SINE,
              SQUARE,
              TRIANGLE,
              SAW,
              ARBITRARY
       };

       /**************************************************************************************
       class AmdTrueAudio::WaveTableSynth:WaveTable

       table of samples representing one cycle of a waveform
       synthesizer will interpolate wave table to generate require pitch
       **************************************************************************************/
       class WaveTable {
       private:
       public:

       /**************************************************************************************
       AmdTrueAudio::WaveTableSynth:WaveTable constructor

       if wavfrm == ARBITRARY, samples must point to wave form array of count floats,
       otherwise samples is ignored.
       **************************************************************************************/
              WaveTable(count, TA_WAVE_FORM wavfrm, float *samples = NULL);
```

```
        ~WaveTable();
        int count;
        float *samples;
    };

    static float NoteToPitch(int note);

    // Instrument Description:
    typedef struct _Instrument {
        WaveTable *wavtab;   // pointer to wave table to be interpolated to generate note.
        float attack;        // rate of initial amplitude rise. logarithmic.
        float decay;         // rate of amplitude fall after attack. logarithmic.
        float sustain;       // hold level until note duration is over.
        float release;       // final decay rate, after duration is over. logarithmic.
    } Instrument;


    int scheduleNote( float pitch, float amplitude,
                      int duration, Instrument instrument, float startTime);

    int scheduleNote(float pitch1, float pitch2, float glissando, float amplitude,
                      int duration, Instrument instrument, float startTime);

    int synthesizeNextBlock(float *output, int length);

    int resetClock(float time);
};
```

2.1.6.2 Musical instrument Physical modelling synthesis

Functions for physical modeling of real instruments. TBD.

References:

Karplus–Strong string synthesis http://en.wikipedia.org/wiki/Karplus-Strong_string_synthesis

Digital waveguide synthesis http://en.wikipedia.org/wiki/Digital_waveguide_synthesis

2.2 [future feature, sample provided] High Level APIs

Functions that combine low level APIs to simplify common audio tasks.

May include, for example, functions to create, manage and synchronize stream processing threads, response generator threads, graphic threads, UI threads, etc., in an application.

3 Sample code

Sample application code to demonstrate TrueAudio APIs, and showcase GPU audio processing performance.

3.1 Room Acoustics Demo

Simple room acoustics simulation that uses “simple room reverb stereo response generator” to simulate two sources and a stereo listener in a room with parameters to set room dimensions and wall absorptions. The sources can be moved and the listener moved and rotated, interactively.

3.2 Oculus Room Tiny

Demonstrates simple room reverb and directional sound with several sound sources integrated with Oculus Room Tiny sample for Oculus DK 2 HMD.

3.3 [future feature] VST plugins

A VST reverb plugin using GPU accelerated Fast Convolution. Can import reverb responses generated using **Room Acoustics Demo**.

A VST N-band equalizer plugin.

A VST noise reduction plugin.

more …

3.4 [future feature] Wwise plugins

A Wwise plugin that demonstrates functionality of **Room Acoustics Demo** in a game. Details TBD.