ISSN (Print) : 0974-6846
Indian Journal of Science and Technology, *Vol 8(33), DOI: 10.17485/ijst/2015/v8i33/80399, December 2015* ISSN (Online) : 0974-5645

Implementation Analysis of Binaural Audio Crosstalk Cancellation on Heterogeneous Parallel Computing platforms using Mixed Non-Uniform Partitioned Convolution

Chunduri Sreenivasa Rao[1*], Dhulipalla Venkata Rao[2] and Somayajula Lakshminarayana[1]

[1]ECE Department, KL University, Vijayawada - 522502, Andhra Pradesh, India;
chsrinivas19800305@rediffmail.com, drslakshminarayana@gmail.com
[2]Narasaraopet Institute. of Tech, Narasaraopet, Guntur - 522601, Andhra Pradesh, India;
dvenky221101@rediffmail.com

Abstract

As general DSP processors don't have massive parallel architecture, they are not suitable to implement 3D audio virtual techniques at very long filters due to computational problems. To address these implementation issues of very long filters, an efficient method called *Mixed Non-uniform Partitioned Convolution* is proposed in this paper for implementing binaural audio crosstalk cancellation on heterogeneous parallel computing platforms. By using massive parallel architecture of heterogeneous platforms, the proposed approach is able to solve computational problems even at filter lengths of 65536 (32-bit floating point). The partitioning scheme followed in this paper is explained in detail to schedule partitions on various compute units of GPU device. The proposed approach was implemented on AMD GPUs using task parallel concept. The instruction level optimization was also provided for complex frequency multiplication and addition using OpenCL. The performance of this approach is compared against the existing techniques proposed by Garcia and Gardener. The cost vs. computational performance tradeoff comparison was given between proposed approach and existing methods. The comparison clearly shows that proposed approach is very efficient at very long filters and requires reasonable cost of implementation in terms of number of compute units. The combination of instruction level and algorithmic level optimizations make the proposed approach more suitable for implementation of not only stereo inputs based audio CTC but also multichannel inputs, particularly at very long filter lengths on parallel computing platforms.

Keywords: Crosstalk Cancellation, Heterogeneous Parallel Computing, Mixed Filtering, OpenCL, Partitioned Convolution

1. Introduction

3D audio systems have several audio applications in home theatre entertainment, gaming, teleconference and remote control. With an aim of reproducing spatial reverberation characteristics and spatial audio pattern at the desired locations, this technology has undergone tremendous changes over the past three decades. Started with Head Related Transfer Function (HRTF) in 1983, this technology was extended to conventional loudspeaker systems and Optimum Source Distribution (OPSODIS). Headphones are difficult to wear in all kind of environments and inconvenient. Conventional loudspeaker systems suffer from spectral coloration in obtaining the inverse of acoustic transfer matrix. This disadvantage is rectified in OPSODIS by making the condition number of system inverse matrix as unity in operating bandwidth. In all these techniques, the

* *Author for correspondence*

intended audio sounds should be retained at the listener's ears and unwanted sounds should be cancelled out, which is generally referred to as audio crosstalk cancellation. This is the existing technology so far[1–4].

To reproduce spatial audio pattern at the listener's ears, it is required to have long impulse responses in binaural audio crosstalk cancellation structure as shown in Figure 1. Due to long filter coefficients, more computational power and memory are required to implement these on real-time DSP processors. Also latency is major factor that needs to be addressed in implementation[5]. To address all implementation problems, an efficient and sophisticated technique needs to be developed for improving computational performance.

This paper explains the drawbacks of existing techniques and proposes a new method called *Mixed Non-uniform partition* to implement audio CTC system on heterogeneous parallel computing platforms. It describes the importance of parallel computing platforms for implementation and the scheduling mechanism of optimum non-uniform partitions and finally compares the performance with that of existing techniques.

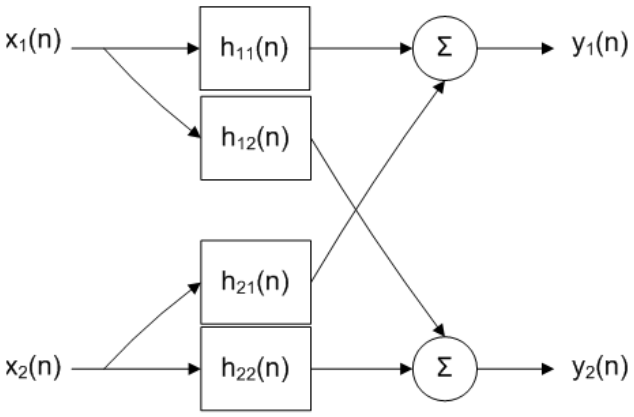

Figure 1. Binaural audio Crosstalk Cancellation (CTC) with stereo inputs.

This paper is organized as follows. Section 2 explains the review of existing techniques and their drawbacks. Section 3 provides objectives of current work. Section 4 describes the mathematical approach of proposed technique. Section 5 describes the architecture of parallel computing platforms. Section 6 details about partitioning scheme to schedule the partitions on various compute units and Section 7 concludes the results and observations and provides future extension of this work.

2. Review of Previous Work

Time domain convolution is the original method from which all other methods are derived. But it is not computational efficient and hence not preferable. On other hand, frequency domain based techniques like overlap save and overlap add methods are computational efficient in real-time implementations. But at long filter lengths, the appending of zeros caused more output latency and unreasonable to use for long filters due to the fact that more FFT size is required for processing of less frame length audio samples[5–7].

The method proposed by Vetterli in 1988 is multi-rate running convolution. In this, impulse response is decimated into bi-orthonormal filter banks so that minimum FFT Size equal to 2*L*. Followed by decimation, interpolation technique is applied to get filtered output. Though latency issue is solved with this upto certain extent, this method is not computationally efficient because every filter bank needs filtering process[8,9].

These techniques are mainly meant for low end applications; usually the filter order is around 1024 or 2048. Beyond this, these techniques require more computational power and more on-chip memory. The below approaches are basically used at long filter lengths.

2.1 Mixed Filtering

The name implies that this method is able to perform all filtering operations of CTC at a time. Reference[5] explains this approach where a complex sequence is formed with real-time outputs $y_1(n)$, $y_2(n)$ and by simplifying the frequency domain equivalent of output complex sequence, one can arrive at:

$$Y(k) = Y_1(k) + jT_2(k) -= X_1(k)H_1(k) + X_2(k)H_2(k) \quad (1)$$

Where

$$H_1(k) = H_1(k) + jH_2(k)$$

$$H_2(k) = H_2(k) + jH_2(k)$$

Equation 1 requires computations of one FFT computation with decomposition, complex frequency multiplication and one IFFT. The CTC outputs, $y_1(n)$ and $y_2(n)$ are given by real and imaginary components of IFFT output respectively[5]. This method is suitable for moderate filter lengths as it uses overlap save method. It suffers from more computational complexity at long filter lengths.

2.2 Uniform Partitioned Convolution

This method divides the impulse response into segments or partitions of equal length. Each partition size is equal to frame size so that overall *M/L* partitions would be obtained. For each partition, overlap save method will be applied and added the outputs to get the overall filter output. The mathematical analysis involved in this is explained below[10–17].

In short, the impulse response, h (n) is expressed in z-domain as:

$$H(z)=\sum_{n=0}^{M-1}h(n)z^{-n}=\sum_{n=0}^{L-1}\left[h_0(n)+z^{-L}h_1(n)+...+z^{-(M-L)}h_{m-1}(n)\right]z^{-n} \quad (2)$$

Where

$$\left.\begin{aligned}&h_0(n)=h(n),\\&h_1(n)=h(n+L),\\&...\\&h_{m-1}(n)=h(n+M-L),\end{aligned}\right\}\ n=0,1,..L-1\ \&\quad m=M/L \quad (3)$$

Equation 3 gives partitioned impulse responses. Each partition size is *L* and total partitions are *M/L*. The output, y (n) in z-domain, is given by:

$$Y(z)=H(z)X(z)=X(z)H_0(z)+\underline{z^{-L}X(z)}H_1(z)+...+\underline{z^{-(M-L)}X(z)}H_{m-1}(z) \quad (4)$$

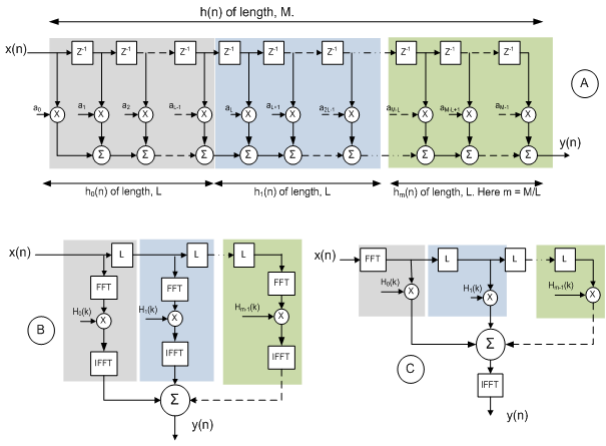

Figure 2. Block diagram of uniform partitioned convolution.

Figure 2 and Equation 4 provide the process involved in uniform partitioned convolution. The latency in this algorithm depends on the partition size, which is same as frame length. So latency is obtained as frame length divided by sampling frequency. The FFT size is twice the frame length as overlap save approach is used for each partition. The total computations are given by *M/L* times the complexity that would be needed for overlap save approach. But this can be further simplified to have single FFT and IFFT, complex multiplications and addition for all partitions. This method solves the difficulties faced by overlap save method but when filter lengths are very long i.e. in the order of 16384, this is inconvenient because the number of complex frequency multiplications become increase. Also this method suffers from more memory constraints at very long filter lengths.

In References[23–26], mixed uniform partitioned convolution was explained to implement audio CTC system with stereo inputs and multi-channel inputs. This is restricted to moderate filter lengths and it was implemented on DSP processors[23–26]. The current work is concentrating on very long filter lengths. For implementation of long filters, the number of complex frequency multiplication blocks are more and hence DSP processor architecture is not suitable to handle complex frequency multiplications in partitioned convolution efficiently.

2.3 Non-Uniform Partitioned Convolution

This method is same as uniform partitioned convolution but partitions are not uniform. This technique was initially developed by Gardener and later improved by Garcia[12]. They suggested that partitioned convolution can be optimized arithmetically by using non-uniform partitions and scheduling them on parallel processing units. Partitioning can be started with small length to obtain low output latency and increase in partition lengths for further partitions to reduce the computational complexity. As all partitions are executed in parallel, the latency depends on the initial partition and hence low latency can be obtained due to small partition length[12–17]. In this method, impulse response, *h (n)* is expressed as:

$$\begin{aligned}H(z)&=\sum_{n=0}^{M-1}h(n)z^{-n}=\sum_{n=0}^{L_1-1}h(n)z^{-n}+\sum_{n=L_1}^{L_2-1}h(n)z^{-n}+...+\sum_{n=L_{l-1}}^{L_l-1}h(n)z^{-n}\\&=\sum_{n=0}^{L_1-1}h(n)z^{-n}+z^{-L_1}\sum_{n=0}^{L_2-1}h(n+L_1)z^{-n}+...+z^{-L_{l-1}}\sum_{n=0}^{L_l-1}h(n+L_{l-1})z^{-n}\\&=H_0(z)+z^{-L_1}H_1(z)+...+z^{-L_{l-1}}H_{l-1}(z)\end{aligned} \quad (5)$$

where $l_1 = L_1$, $l_2 = L_2 - L_1$,..$l_l = L_l - L_{l-1}$ and l_1, l_2,..l_l are non-uniform partitions. The condition $l_1 + l_2 +... + l_l =$

M should be satisfied. Finally the system output, y (n) in z-domain is given by:

$$Y(z) = H(z)X(z)$$
$$= X(z)H_0(z) + \underline{z^{-L_1}X(z)}H_1(z) + \ldots + \underline{z^{-L_{l-1}}X(z)}H_{l-1}(z) \quad (6)$$

Though l_1, l_2,..l_l are the non-uniform partition lengths, they will be chosen as integer multiple of frame length i.e. L so that uniform partitioned convolution will be applied for each non-uniform partitioned filter. All non-uniform partitions are processed using uniform partitioned convolution in parallel and outputs of all are added to yield the final output.

To implement this kind of algorithm, it is required to execute all non-uniform partitions in parallel to save computational power. Obviously more number of cores is required and hence DSP platforms are not suitable for implementing this solution. Though DSP platforms have SIMD feature, this won't serve the requirement as the number of non-uniform partitions are more and these are dependent on the filter length. To address this, the massive parallel system architectures like heterogeneous parallel computing platforms are needed where the architecture contains more number of compute units and SIMDs. This architecture is explained in Section 5. The implementation approach is more complex, in particular, if each filter in audio CTC is implemented separately. Hence more computations are needed even on parallel computing platforms. To overcome this, an efficient method is proposed in this paper.

3. Objecives of Current Work

In this paper, new algorithm is obtained by combining the existing techniques to implement audio CTC system with stereo inputs aiming at the reduction of computational complexity, particularly for very long filters. The mathematical analysis of proposed algorithm, *Mixed Non-uniform Partitioned Convolution* is provided in detail. In this work, maximum of 65536 is considered in each CTC filter.

As the architecture of general DSP processors is not suitable to implement parallel partitioned filters, the proposed technique was developed on heterogeneous parallel computing platforms like AMD GPUs. The partitioning scheme followed in this work was described.

The performance of proposed algorithm was compared with that of existing methods. AMD Radeon HD 7900 series GPUs were used as the platform for implementation.

4. Proposed Algorithm - Mixed Non-Uniform Partitioned Convolution

As the name implies, the proposed algorithm is the combination of mixed filtering and non-uniform partitioned convolution. From Equation 1, note that the time domain equivalents of frequency responses, $H_1(k)$ & $H_2(k)$ are given by $h_{11}(n) + j\,h_{12}(n)$ & $h_{21}(n) + j\,h_{22}(n)$ respectively. Let us assume that these impulse responses are partitioned non-uniformly and these non-uniform partitions are given by:

$$h_1(n) = \{h_{1,0}(n), h_{1,1}(n), h_{1,2}(n), \ldots, h_{1,l-1}(n)\}$$
$$= \{h_{11,0}(n) + jh_{12,0}(n)\ \tilde{h}_{11,1}(n) + jh_{12,1}(n) \quad h_{11,l-1}(n) + jh_{12,l-1}(n)\}$$
$$h_2(n) = \{h_{2,0}(n), h_{2,1}(n), h_{2,2}(n), \ldots, h_{2,l-1}(n)\}$$
$$= \{h_{21,0}(n) + jh_{22,0}(n)\ \tilde{h}_{21,1}(n) + jh_{22,1}(n) \quad h_{21,m-1}(n) + jh_{22,l-1}(n)\}$$

Here the lengths of partitions are in the order l_1, l_2,...l_l. The z-domain equivalent of Equation 1 is given by:

$$Y(z) = Y_1(z) + jY_2(z) = X_1(z)H_1(z) + X_2(z)H_2(z)$$
$$= X_1(z)\left[H_{1,0}(z) + z^{-L_1}H_{1,1}(z) + \ldots + z^{-L_{l-1}}H_{1,l-1}(z)\right] +$$
$$X_2(z)\left[H_{2,0}(z) + z^{-L_1}H_{2,1}(z) + \ldots + z^{-L_{l-1}}H_{2,l-1}(z)\right]$$
$$= X_1(z)H_{1,0}(z) + \underline{z^{-L_1}X_1(z)}H_{1,1}(z) + \ldots + \underline{z^{-L_{l-1}}X_1(z)}H_{1,l-1}(z) +$$
$$X_2(z)H_{2,0}(z) + \underline{z^{-L_1}X_2(z)}H_{2,1}(z) + \ldots + \underline{z^{-L_{l-1}}X_2(z)}H_{2,l-1}(z)$$
$$= UPC(X_1, H_{1,0}, 0, L_1) + UPC(X_1, H_{1,1}, L_1, L_2) + \ldots +$$
$$UPC(X_1, H_{1,l-1}, L_{l-1}, L_l) + UPC(X_2, H_{2,0}, 0, L_1) + UPC(X_2, H_{2,1}, L_1, L_2) +$$
$$\ldots + \tilde{\ }\tilde{\ }\tilde{\ }\ (\ _2,\ _{2,l-1},\ _{l-1},\ _l) \quad (7)$$

The outputs of CTC system in complex signal form are given by:

$$\tilde{\ }(\tilde{\ }) = \ _1(\ ) + \ _2(\ )$$
$$= z^{-1}\{UPC(X_1, H_{1,0}, 0, L_1) + UPC(X_2, H_{2,0}, 0, L_1)\} +$$
$$z^{-1}\{UPC(X_1, H_{1,1}, L_1, L_2) + UPC(X_2, H_{2,1}, L_1, L_2)\} + \ldots +$$
$$z^{-1}\{UPC(X_1, H_{1,l-1}, L_{l-1}, L_l) + UPC(X_2, H_{2,l-1}, L_{l-1}, L_l)\} \quad (8)$$

Here the term $UPC\ (X_i,\ H_{i,j},\ D,\ B)$ has meaning that uniform partitioned convolution can be applied whose input sequence is $X_i(z)$ with delay z^{-D} and whose z-transform coefficients are given by $H_{i,j}(z)$ where $i=1,2$ and value of j depends on the number of non-uniform partitions. The value B is the non-uniform partition

length. This term can be represented as:

$$UPC(X_i, H_{i,j}, D, B) = z^{-D}\left[\begin{array}{l} X_i(z)H_{i,j,0}(z) + \underline{z^{-L}X_i(z)H_{i,j,1}(z) + \ldots + z^{-(m_k-1)L}X(z)H_{i,j,m_k-1}(z)} \\ m_k = B/L \end{array}\right], \quad (9)$$

Where Equation 4 was reused here to represent Equation 9. Clearly, Equations 8 and 9 give the impression that each non-uniform partition in turn can be implemented using uniform partitioned convolution. For example, if $l_1 = 512$, l_2=512, l_3=1024, l_4=1024 and $L = 128$, then 1st and 2nd non-uniform partitions can get 512/128 = 4 uniform partitions. Similarly, 3rd and 4th non-uniform partitions can get 1024/128 = 8 uniform partitions and each non-uniform partition can be executed independent to each other on parallel platforms. Hence 4 independent cores are needed to run them in parallel for this case.

A special case that corresponds to uniform partitioned convolution can be obtained if $l_1 = l_2 = .. = l_l = L$. In this case, all partitions can become uniform which is same as that described in Reference[10].

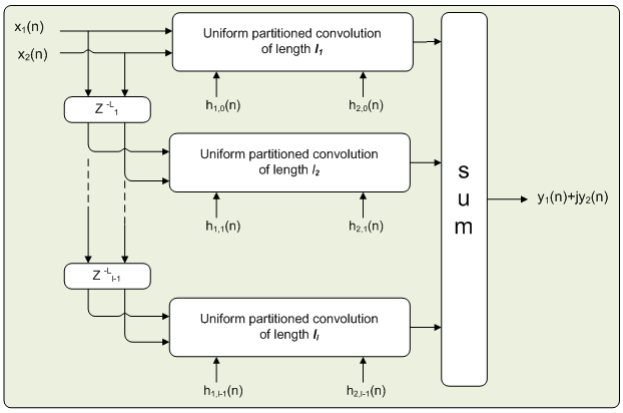

Figure 3. Block diagram of mixed non-uniform partitioned convolution to obtain CTC outputs.

As explained in Section 2.2, uniform partitioned convolution contains single FFT, single IFFT and $m=M/L$ number of complex frequency multiplication blocks. When Equation 8 is developed on parallel computing platform, each term under z^{-l} requires a separate core and this core is responsible for implementing all uniform partitioned tasks. The total number of cores is depending on the filter length and how the non-uniform partitioning was done. Figure 3 shows the block diagram representation of this approach. The 1st UPC block processes input samples upto length L_1. These L_1 samples will be automatically copied into delay buffer that is needed for 2nd UPC block. Whenever the sample count is more than L_1, 2nd UPC block starts its processing of delayed buffer contents. When sample count is $L_1 + L_2$, 1st UPC block processes recently received L_1 samples and 2nd UPC processes rest of L_2 samples. The outputs of both blocks are summed to yield the CTC outputs. This process will be continued till input stream is completed. Note that all UPC blocks are operated in parallel to reduce the computational power. This is the main aim of non-uniform partitioned convolution. Since DSPs suffer from more computations as they don't have more parallel computing cores, it is necessary to switch to suitable platforms that support compute units to achieve computational savings.

5. Parallel Computing Platforms and Scheduling of Parallel Tasks

This section describes the architecture of parallel computing platforms, the methods available for scheduling of parallel tasks.

5.1 Heterogeneous Parallel Computing

Heterogeneous computing involves the use of various computational units, usually a general purpose processing unit such as CPU or GPU or DSP processors. The advances in CPU technology proved insufficient to cope with requirements of modern computer applications that require interactions with various systems. To achieve greater performance gains, specialized hardware is required so that entire system becomes heterogeneous and application designers have the option to choose on which compute unit the corresponding tasks can be scheduled to run.

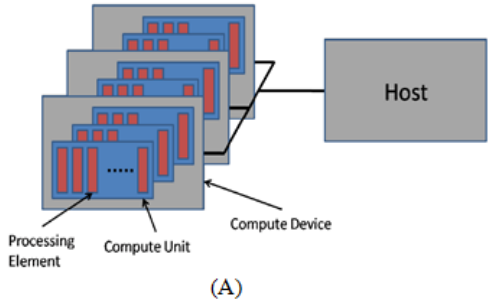

Figure 4A. OpenCL (A) Platform model

OpenCL (Open Computing Language) is an open and royalty free parallel computing API designed to enable GPUs to work in tandem with CPU, providing additional raw computing power. This was first developed by Khronos Group as an independent standard. The platform model of OpenCL (shown in Figure 4A) contains host connected to one or more OpenCL devices. Host is generally a CPU and device can be GPU or DSP or multi-core CPU. OpenCL device consist of a collection of one or more compute units, in turn, each compute unit is composed of one or more SIMD (Single Instruction Multiple Data) processing elements[18–22].

5.2 OpenCL Execution Model

This model comprises kernels and host programs as two main components. Host program is basically responsible for setting up and managing execution of kernels on OpenCL device through the use of context. Host can manipulate devices, program objects, kernels and memory objects. After context is created, command queues are created to manage the execution of kernels. Commands are placed into the command queue in-order and execute either in in-order or out-of-order. In former case, commands are executed in serial whereas in out-of-order case, commands execution is based on synchronization constraints placed on the command.

Kernels are basic unit of executable code that runs on one or more OpenCL devices and are similar to C function that can be either data or task parallel[18–22].

5.3 OpenCL Memory Model

There is no common memory space available to host and OpenCL device. OpenCL memory is divided into 4 regions (shown in Figure 4B). Global memory is accessible to both host and OpenCL device and can be allocated by host during runtime. Constant memory is a part of global memory, which is constant throughout the execution of kernel and kernel has only read access but host has both read and write access as host allocates this memory. Local memory is a region of memory used for data sharing across work items in work group. Private memory is a region that is accessible to only one work item. Generally, data must be explicitly move from host to global to local and back. This process works by enqueuing read/write commands in command queue and these commands can be either blocking or non-blocking. Blocking commands leads the host to wait until memory transaction is completed whereas non-blocking leads the host to put the command in command queue and need not to wait for memory transaction to complete[18–22].

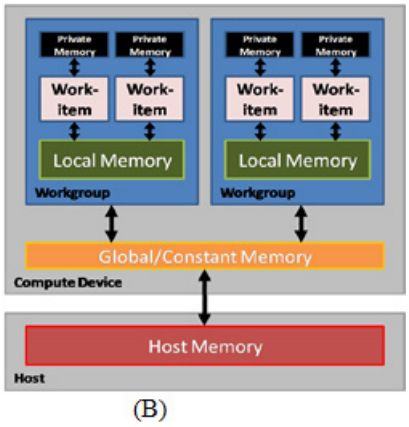

Figure 4B. OpenCL (B) Memory model

5.4 Data Parallelism vs. Task Parallelism

The main characteristics of the data parallel method are that programming is relatively simple since multiple processors are all running the same program and that all processors finish their tasks at about the same time. This method is efficient when the dependencies between the data being processed by each processor are minimal.

The main characteristic of the task parallel method is that each processor executes different commands. This increases the programming difficulty when compared to the data parallel method. The task parallel method requires a way of balancing the tasks to take full advantage of all the cores. One way is to implement a load balancing function on one of the processors. Another method for task parallelism is known as pipelining. Pipelining is usually in reference to the "instruction pipeline" where multiple instructions, such as instruction decoding, arithmetic operation and register fetch, are executed in an overlapped fashion over multiple stages. This concept can be used in parallel programming as well[18–22].

6. Scheduling of Partitions and Experimental Results

This section explains the partitioning scheme and experimental results.

6.1 Partitioning Scheme

The aim of the partitioning is to keep computational power

as low as possible without compromising for increase in latency. To achieve this, the partitioning should be done in systematic way. Existing methods proposed by Gardener and Garcia follows some systematic way in that the partitioning is done in the manner:

128*2, 256*2,512*2,1024*2,2048*2,4096*2,

and

128*14, 1024*14

for filter length of 16128[12]. The former one is proposed by Gardener and latter one is proposed by Garcia. In both cases, initial partition length is 128 and this ensures that the latency to be minimal. But Gardener solution requires 12 CUs (Compute Units) and Garcia solution requires 28 CUs in order to implement these partitions on GPUs. Though Gardener requires 12 CUs, this solution needs more computations because the computational complexity is obtained by maximum of that needed for all partitions. In this case, partition 4096 requires more computations than what is needed for 1024 in Garcia solution but Garcia requires more CUs. Obviously there must be some tradeoff between cost (in number of CUs) and computational complexity.

Table 1. Partitioning of impulse response length in non-uniform partitioned convolution algorithm

Filter Length, M	Partitions x1024	Partitions x2048	Partitions x4096	Partitions x8192	Total Partitions
4096	4	0	0	0	4
8192	4	2	0	0	6
12288	4	4	0	0	8
16384	4	6	0	0	10
20480	4	6	1	0	11
24576	4	6	2	0	12
28672	4	6	3	0	13
32768	4	6	4	0	14
36864	4	6	5	0	15
40960	4	6	6	0	16
45056	4	6	7	0	17
49152	4	6	8	0	18
53248	4	6	8	1	19
57344	4	6	8	1	19
61440	4	6	8	2	20
65536	4	6	8	2	20

The proposed partitioning follows linear increments in partition lengths as well as number of CUs. Table 1 gives partitioning scheme followed for filter lengths from 4096 to 65536 with step size in filter length of 4096. For example, filter length of 16384 requires 4 UPCs of 1024 each and 6 UPCs of 2048 length each (16384 = 4*1024 + 6*2048). This was designed like this to keep in mind that each non-uniform partition is still partitioned into uniform partition with filter length of L = 128 during implementation. This ensures that the latency of the system is obtained as 128/44.1 = 2.9msec at sampling frequency of 44.1kHz. Also with this approach, the cost of the system is low as required CUs are low.

For the case of 16384 length, 10 CUs are required. The computational complexity is equal to maximum of what is required for 1024 and 2048. It is not necessary to consider the number of CUs while calculating the computations as all CUs are running in parallel. For lengths of 57344 and 61440, sufficient zeroes are appended at the end to make sure partitions are done as integer multiples of 8192. This does not create any latency in output because latency is depending on uniform partition length i.e. 128.

6.2 Design of Proposed Method

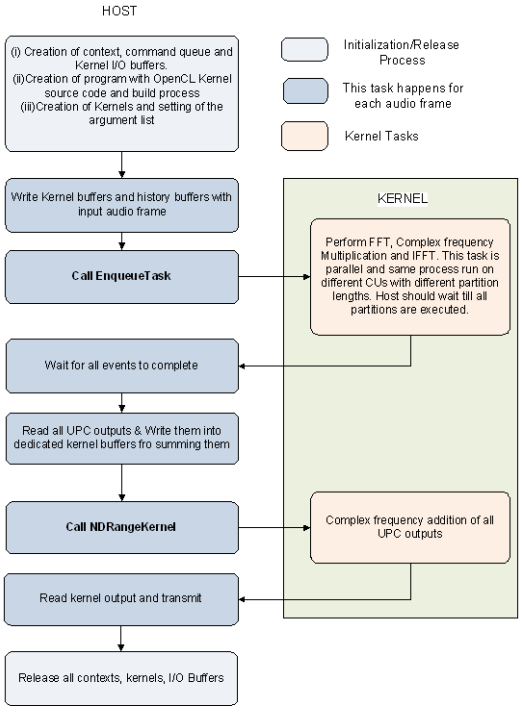

Figure 5. Flow chart for execution of proposed method.

Figure 5 shows the flow diagram of high level design of the proposed approach. In this, Host basically initializes all the kernels, required OpenCL kernel I/O buffers, the OpenCL context and the builds the kernel source code.

```
void ComplexFreqMult(__global float   *pXL_Real, __global float   *pXL_Imag,
                     __global float   *pXR_Real, __global float   *pXR_Imag,
                     __global float   *pHA_Real, __global float   *pHA_Imag,
                     __global float   *pHB_Real, __global float   *pHB_Imag,
                     __global float   *pY_Real,  __global float   *pY_Imag,
                       int     bufSize,          int     partitionCount)
{
      int i,j;
      __global float *pTempXL_Real = pXL_Real;
      __global float *pTempXL_Imag = pXL_Imag;
      __global float *pTempXR_Real = pXR_Real;
      __global float *pTempXR_Imag = pXR_Imag;
      __global float *pTempHA_Real = pHA_Real;
      __global float *pTempHA_Imag = pHA_Imag;
      __global float *pTempHB_Real = pHB_Real;
      __global float *pTempHB_Imag = pHB_Imag;

      for(i=0;i<partitionCount;i++)
      {
           for(j=0;j<(bufSize/16);j++)
           {
               float16 XL_Real = vload16(j,pTempXL_Real);  float16 XL_Imag = vload16(j,pTempXL_Imag);
               float16 XR_Real = vload16(j,pTempXR_Real);  float16 XR_Imag = vload16(j,pTempXR_Imag);

               float16 HA_Real = vload16(j,pTempHA_Real);  float16 HA_Imag = vload16(j,pTempHA_Imag);
               float16 HB_Real = vload16(j,pTempHB_Real);  float16 HB_Imag = vload16(j,pTempHB_Imag);

               float16 Y_Real = vload16(j,pY_Real);        float16 Y_Imag = vload16(j,pY_Imag);

               Y_Real = Y_Real + XL_Real*HA_Real - XL_Imag*HA_Imag + XR_Real*HB_Real - XR_Imag*HB_Imag;
               Y_Imag = Y_Imag + XL_Real*HA_Imag + XL_Imag*HA_Real + XR_Real*HB_Imag + XR_Imag*HB_Real;

               vstore16(Y_Real, j, pY_Real);               vstore16(Y_Imag, j, pY_Imag);
           }

           pTempXL_Real = pTempXL_Real + bufSize;          pTempXL_Imag = pTempXL_Imag + bufSize;
           pTempXR_Real = pTempXR_Real + bufSize;          pTempXR_Imag = pTempXR_Imag + bufSize;

           pTempHA_Real = pTempHA_Real + bufSize;          pTempHA_Imag = pTempHA_Imag + bufSize;
           pTempHB_Real = pTempHB_Real + bufSize;          pTempHB_Imag = pTempHB_Imag + bufSize;
      }
}
```

```
__kernel void ComplexAdd(__global float   *pInputReal, __global float   *pInputImag,
                         __global float   *pOutputReal, __global float   *pOutputImag,
                         int     bufSize,          int     partitionCount)
{
       int idx = get_global_id(0);
       __global float *pTempReal = pInputReal;
       __global float *pTempImag = pInputImag;

       for(i=0;i<partitionCount;i++)
       {
            pOutputReal[idx] = pOutputReal[idx] +pTempReal[idx];
            pOutputImag[idx] = pOutputImag[idx] +pTempImag[idx];

            pTempReal = pTempReal + bufSize;
            pTempImag = pTempImag + bufSize;
       }
}
```

Then it sets all the required kernel argument list. After this, it writes the input frame contents into kernel Input buffers and calls the clEnqueueTask function. Upon calling this function, the OpenCL kernel runs on GPU. On GPU, the kernel function is implemented basically to perform FFT of input audio frame, complex frequency multiplication and IFFT. These blocks are part of uniform partitioned convolution. The below function ComplexFreqMult shows the ability of OpenCL kernel in vectored form to utilize the SIMD units effectively.

As per Equation (8), the count of non-uniform partitions are based on length of impulse response and the partitioning scheme followed as per Table 1. So all non-uniform partitions are executed on different CUs in parallel within GPU. The execution time of kernel function running on each CU depends on the partition length. When the execution of all CUs is completed, the kernel function returns to Host. To ensure this, Host waits using clFinish function till execution of all kernels are completed.

The clEnqueueTask basically performs the task parallelism operation and after completion of this, it

is required to add all outputs of kernel functions i.e. all UPC outputs. To do this, Host writes all UPC outputs in one dedicated kernel buffer and calls clNDRangeKernel, basically meant for data parallelism operation. This kernel function does the summing of all UPC outputs and produces the final outputs, y_1 (n) and y_2 (n) in complex form. The below function ComplexAdd gives the summing of all UPC outputs. Here the group size is set to bufSize so that when operated on *get_global_id* (0), the processing of single instruction leads to happen for all samples of bufSize.

In this case, Host does not need to wait for kernel completion because data parallel kernel returns after completion of kernel execution only. Then Host transmits the CTC outputs for rendering. Once all the audio frames are processed successfully, Host releases all the kernels, allocated kernel I/O buffers and various contexts.

6.3 Experimental Details

To proceed with experiments, initially the audio room was simulated to measure the impulse responses at different lengths ranging from 4096 to 65536 at step size of 4096. These impulse responses were used in measuring the performance evaluation of proposed approach with that of existing techniques. The hardware platform details are as follows:

Processor: AMD FX-4100 Quad-core processor, 3.6GHz
RAM Size: 4GB
GPU: TAHITI (AMD Radeon HD 7900 series)
Active CUs: 28

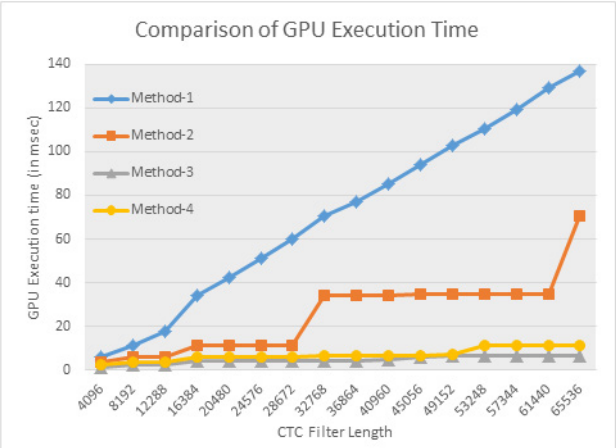

Figure 6. Comparison of GPU Execution Time in msec.

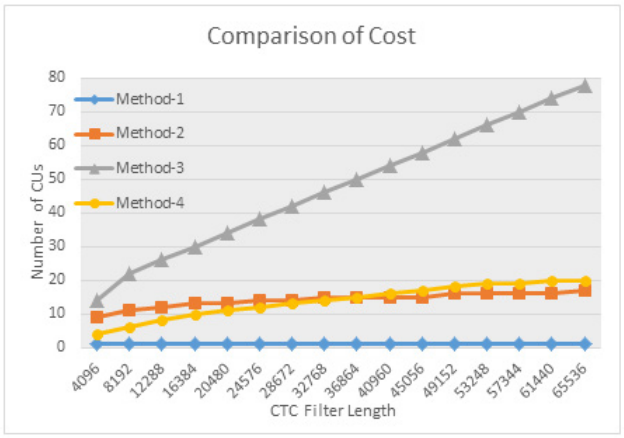

Figure 7. Comparison of system cost.

The proposed technique was implemented on AMD Radeon 7900 series based TAHITI GPU, which has 28 Active CUs per GPU. Win 10 operating system was used for experiments. Microsoft Visual C++ and CodeXL were used for Host and kernel developments respectively. To test this, 0dB audio signal of 44.1kHz was used. The implementation was done as described in Section 6.2. To compare the performance of proposed technique, the existing approaches were also implemented on the same hardware. The comparison was provided in Figures 6 and 7 for various filter lengths. The former explains the GPU kernel execution time in msec whereas Figure7 provides the comparison of system cost in terms of number of CUs required for each approach. The execution profiling was computed using clGetEventProfilingInfo with attributes.

Cl_Profiling_Command_Start and Cl_Profiling_Command_End

The 1st method (Method-1) is uniform partitioned convolution. In this method, impulse response is partitioned uniformly and implemented on single CU. The advantage of this method is system cost because only one CU is used for implementation but the performance drops drastically as filter length is increased (as shown in Figure 7). Hence this method is best suited for short audio CTC filter lengths. If filter lengths are long, this method is not good for developers as performance is major factor in deciding the efficiency of the best approach at long CTC filter lengths.

The 2nd method (Method-2) is Gardener's approach[12]. This approach is given more importance to latency and

execution time. So the partitioning scheme is basically contains lower partitions initially and then increases. Because of this, although the best system cost can be attained (17 CUs at 65536 length), the computational cost is still a major issue for long filter lengths (70.318msec at 65536 filter length). The GPU execution time follows staircase approximation for this approach. This is basically depending on the partitioning scheme that Gardener was proposed.

The 3rd method (Method-3) is Garcia's approach. The partitioning scheme of Garcia basically contains many lower partitions initially and then many higher partitions[12]. Because of this approach, more number of CUs are required at long filter lengths (78 CUs at filter length of 65536) though the execution time (6.562 at 65536 filter length) is better. This becomes a big issue when required number of CUs is not available on particular GPU. In this case, GPU simply performs the parallel operation till the available CUs and again repeats the same process for the rest of CUs.

The 4th method (Method-4) is the proposed approach i.e. *Mixed Non-uniform partitioned convolution*. This method provides the best approximation in terms of system cost and GPU execution time. As is indicated in Table 1, this approach requires 20 CUs at maximum and needs 11.324msec at 65536 filter length. On considering the tradeoff between cost and performance, the proposed approach is best suited for implementing audio CTC system at very long filter lengths.

7. Conclusion and Future Scope

To obtain low latency and best performance in computational complexity, the existing implementation techniques for audio CTC suffer from more computations at very long filter lengths. For this, an optimum method called *Mixed Non-uniform partitioned convolution* was proposed to implement audio CTC on heterogeneous parallel computing platforms. A best partition scheme was provided based on proposed method. The advantage of this approach is that though the impulse responses of CTC filters are partitioned non-uniformly, uniform partitioned convolution can be used for implementing each non-uniform partition. Due to this, the latency still depends on the frame length used in uniform partitioned convolution. i.e frame length is low and hence the latency is low. Computations point of view, this method is better compared to existing methods because of better partitioning scheme. The comparison of computational complexity with respect to existing methods reveals that this is best suitable at long filter lengths for audio CTC implementation.

This approach can be extended for multi-channel inputs audio CTC where the computational complexity variation can be analyzed for different multi-channel cases. Also the lower partitions can be implemented with time domain implementation in order to get zero latency using the same algorithm.

8. References

1. Otani M, Ise S. Fast calculation system specialized for head-related transfer function based on boundary element method. Journal of Acoustical Society of America. 2006 May; 119(5):2589–98.
2. Ole K, Per R, Philip AN, Angelo F. Design of crosstalk cancellation networks by using fast deconvolution. In AES 15. 1999 May; p. 9900–5.
3. Tobias L, Oliver S. Adaptive cross-talk cancellation system for a moving listener. AES 21st International Conference Proc; 2002 Jun. Available from: http://www.aes.org/e-lib/browse.cfm?elib=11175
4. Wang L, Yin F, Chen Z. A stereo crosstalk cancellation system based on common-acoustical pole/zero model. Eurasip Journal on Advances in Signal Processing – Special issue on digital audio effect. 2010,. Available from: http://www.asp.eurasipjournals.com/content/pdf/1687-6180-2010-719197.pdf
5. Rao S. Udayalakshmi CR, Jeyasingh P. Fast implementation of audio crosstalk cancellation of audio crosstalk cancellation on DSP processors. AES 45 Conference Proc. 2012 Mar 1-4. p. 56–63.
6. Proakis JG, Manolakis DG. Digital signal processing principles, algorithms and applications. 3rd ed. Prentice Hall Publications Inc. USA: p. 430–76.
7. Lyons RG. Understanding digital signal processing. 3rd ed. Prentice Hall Publications Inc: New Jersey; 2010 Nov.
8. Vande Kieft JS. Computational improvements to linear convolution with multi-rate filtering methods. 1998 Apr. Available from: http://mue.music.miami.edu/thesis/jvandekieft/jvtitle.htm
9. Vetterli M. Running FIR and IIR Filters using Multi-rate Filter Banks. IEEE transactions on Acoustics, Speech and Signal Processing. 1988 May; 36(5):730–8.
10. Battenbaerg E, Avizienis R. Implementing real-time partitioned convolution algorithms on conventional operating systems. Proc of 14th Int Conference on Digital Audio Effects: Paris, France; 2011 Sep 19-23. Available from: http://www.ericbattenberg.com/school/partconvDAFx2011.pdf
11. Torger A, Farina A. Real-time partitioned convolution for ambiophonic surround sound. IEEE Workshop on applications of Digital Signal Processing to Audio and Acoustics; New Paltz, New York. 2001. p. 195–8.

12. Guillermo G. Optimal filter partition for efficient convolution with short input/output delay. AES 113th International Conference Proc; 2002 Oct. p. 2660.
13. Gardiner WG. Efficient convolution without input-output delay. Journal of AES. 1995 Mar; 43(3):127–36.
14. Hurchalla J. A time distributed FFT for efficient low latency convolution. AES Convention 129: 2010 Nov. Available from: http://www.aes.org/e-lib/browse.cfm?elib=15679
15. Hurchalla J. Low latency convolution in one dimension via two dimensional convolutions - An intuitive approach. AES Convention 125: 2008 Oct. Available from: http://www.aes.org/e-lib/browse.cfm?elib=14785
16. Armelloni E, Giottoli C, Farina A. Implementation of real-time partitioned convolution on a DSP board. IEEE Workshop on Applications of Signal processing to Audio and Acoustics; New Paltz, NY; 2003 Oct 19-22. p. 71–4.
17. Battenberg E, Wessel D, Colmenares J. Advances in the parallelization of music and audio applications. Proceedings of the International Computer Music Conference: 2010. Available from: http://parlab.eecs.berkeley.edu/sites/all/parlab/files/ParLabCnmatICMC_submitted.pdf
18. Gaster BR, Howes L, Kaeli D, Mistry P, Schaa D. Heterogeneous computing with OpenCL. USA: Advanced Micro Devices Inc, Elsevier Publications; 2012.
19. Kim D, Lee J, Lee J. Scheduling in heterogeneous computing environments for proximity queries. IEEE Transactions on Visualization and Computer Graphics. 2013 Sep; 19(9):1513–25.
20. Lee C, Ro WW, Gaudiot JL. Cooperative heterogeneous computing for parallel processing on CPU/GPU hybrids. Proceedings of 2012, 16th Workshop on Interaction between compilers and computer architectures: Interact; New Orleans, LA. 2012. p. 33–40.
21. Advanced Micro Devices Inc, Introduction to OpenCL Programming. Rev. A, issue Publication #137-41768-10, Sunnyvale and California; 2010 May.
22. Nakamura T, Izuka T, Asahara A. The OpenCL programming book revised for OpenCL 1.2. Fixstars Publishers: Sunnyvale and California; 2012.
23. Rao CS, Rao DV, Lakshminarayana S. Design and implementation analysis of OSD based audio crosstalk cancellation with multi-channel inputs on DSP processors. Indian Journal of Science and Technology. 2015 Mar; 8(5):419–31.
24. Rao CS, Rao DV, Lakshminarayana S. An efficient implementing solution for three channel OSD based audio crosstalk cancellation with stereo inputs. International Journal of Applied Engineering Research. 2015 Jan; 10(1):1995–2011.
25. Rao CS, Rao DV. Real-time implementation of multi-channel audio crosstalk cancellation using mixed single frequency delay line filtering algorithm. International Journal of Modern Engineering Research. 2013 Mar; 3(2):1088–96.
26. Rao CS, Mahalakshmi NVK, Rao DV. Real-time DSP implementation of audio crosstalk cancellation using mixed uniform partitioned convolution. Signal Processing: An International Journal. 2012 Oct; 6(4):118–27.