IMPLEMENTATION OF DIGITAL DOWN CONVERTER IN GPU

Submitted in Partial Fulfillment of the Requirements of the Degree of

Bachelor of Technology

in

Avionics Engineering

by

AMIT UPADHYAY

YAWATKAR SHAKUN RAJAN

Department of Avionics

Indian Institute of Space Science and Technology

Thiruvananthpuram

April 2012

BONAFIDE CERTIFICATE

This is to certify that this project report entitled **"IMPLEMENTATION OF DIGITAL DOWN CONVERSION IN GPU"** submitted to **Indian Institute of Space Science and Technology, Thiruvananthapuram,** is a bonafide record of work done by **Amit Upadhyay and Yawatkar Shakun Rajan** under my supervision at the Giant Metrewave Radio Telescope, NCRA-TIFR from "**9th January 2012**" to "**27th April 2012**".

Mr. Ajith Kumar
Head, Digital Backend,
GMRT, NCRA-TIFR

Dr. Yashwant Gupta
Chief Scientist,
GMRT,NCRA-TIFR

Place: GMRT, Khodad

Date: 27/04/2012

DECLARATION BY AUTHORS

This is to declare that this report has been written by us. No part of the report is plagiarized from other sources. All information included from other sources has been duly acknowledged. We aver that if any part of the report is found to be plagiarized, we are shall take full responsibility for it.

Amit Upadhyay

Roll No.:SC08B058

Yawatkar Shakun Rajan

Roll No.:SC08B040

Place: GMRT, Khodad

Date: 27/04/2012

ACKNOWLEDGEMENT

We thank Dr. Yashwant Gupta, Chief Scientist, GMRT, for sparing his valuable time to help us understand various aspects of this project and for guiding us to the successful completion of the project.

We express our gratitude to Mr. Ajith Kumar, Head, Digital Backend Systems, GMRT, for facilitating this project and also for all help in support rendered.

We wholeheartedly thank Mr. Harshvardhan Reddy, Engineer C [FTA], GMRT for spending his quality time helping us to complete this project.

We also place on record our thanks to Dr. S. K. Ghosh**,** Centre Director NCRA and Dr. K.S. Dasgupta, Director IIST for giving us such a great opportunity by arranging this internship program.

Finally, we sincerely thank all people, who have helped us during the project, directly or indirectly.

April 2012 **Amit Upadhyay**

Yawatkar Shakun Rajan

ABSTRACT

Giant Metrewave Radio Telescope is undergoing an upgradation. GMRT is mainly used for pulsar, continuum and spectral line observations. Spectral Line observations require more resolution which can be achieved by narrowband mode. Thus to utilize the GMRT correlator resources efficiently and to speed up the further signal processing, Digital Down Converter is of great use. Digital Down Conversion down-converts the desired band to baseband resulting into the increased resolution of signal. The limited FFT size can be utilized to get more resolution of signal. Graphic Processing Unit (GPU) decreases the time of various operations in DDC significantly. Hence, the DDC design is done in GPU The project work includes the design of DDC block in GPU using a programming tool Compute Unified Device Architecture C (CUDA-C). All the sub blocks of DDC were designed and combined to form DDC. The block was tested for the CW signal and also the channel noise signal; it showed the desired down-conversion of the band to baseband. DDC block also increases resolution of the signal. It decimates the output and spreads it over all the channels. It reduces the data rate in the GMRT correlator program to a great extent. The timing analysis of the DDC block was done with and without using shared memory of the GPU. Using shared memory is beneficial as far as the convolution time is considered. When the same code for DDC was run on CPU, it was observed that GPU gives about 250 times enhancement in the time consumed. The design of the DDC block was integrated with the existing GMRT Correlator program. It was placed just before the FFT block. Local GMRT software 'TAX' was used to visualize the outputs obtained from the program. The desired down converted band was observed with the increased resolution. The GMRT Correlator program along with the DDC block has been further modified as per the requirements of Phase Correction and MAC block of the GMRT Correlator design. It enabled us for the sky testing of the GMRT Correlator program with the DDC feature. The sky test results were obtained and verified successfully.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ADC	Analog to Digital Converter
ALU	Arithmetic Logic Unit
ANSI	American National Standards Institute
API	Application Program Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CW	Continuous Wave
DDC	Digital Down Converter
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
GMRT	Giant Metrewave Radio Telescope
GPU	Graphic Processing Unit
HPC	High Performance Computing
LO	Local Oscillator
LPF	Low Pass Filter
MAC	Multiplication And Accumulation
NCRA	National Centre for Radio Astrophysics
NVCC	NVIDIA CUDA C Compiler
RF	Radio Frequency
TAX	TK Assisted Xstract

1. INTRODUCTION

1.1. Organization Profile:

National Centre for Radio Astrophysics has set up a unique facility for radio astronomical research using the metre wavelengths range of the radio spectrum, known as the Giant Metrewave Radio Telescope (GMRT), it is located at a site about 80 km north of Pune. GMRT consists of 30 fully steerable gigantic parabolic dishes of 45m diameter each spread over distances of up to 25 km. GMRT is one of the most challenging experimental programs in basic sciences undertaken by Indian scientists and engineers. GMRT is an interferometer which uses a technique named Aperture Synthesis[2] to make images of radio sources in sky.

The number and configuration of the dishes was optimized to meet the principal astrophysical objectives which require sensitivity at high angular resolution as well as ability to image radio emission from diffuse extended regions[8]. Fourteen of the thirty dishes are located more or less randomly in a compact central array in a region of about 1 sq. km. The remaining sixteen dishes are spread out along the 3 arms of an approximately `Y'-shaped_configuration over a much larger region, with the longest interferometry baseline of about 25 km. The multiplication or correlation of radio signals from all the 435 possible pairs of antennas or interferometers over several hours will thus enable radio images of celestial objects to be synthesized with a resolution equivalent to that obtainable with a single gigantic dish 25 kilometers in diameter!

- Telescope works at long wavelengths 21cm and longer.
- Each antenna has 4 different receivers mounted at focus and each individual receiver assembly can rotate so that the user can select the frequency at which to observe.
- GMRT operates in frequency bands centered at:153 MHz, 233 MHz, 327 MHz, 610 MHz and 1420 MHz
- The GMRT Correlator is an FX[2] correlator.

1.2 GMRT correlator system upgrade:

The goal of GMRT correlator upgrade is to provide continuous spectral coverage and better spectral resolution. The new upgrade of GMRT correlator system aims at continuous coverage from 100MHz-1600MHz. The current system has only 32MHz of maximum bandwidth which is aimed to be improved up to 400MHz.

The GMRT correlator will provide wideband coverage for radio astronomy applications namely, continuum observation, pulsar observation and spectral line observation. This wideband coverage upgrade also leads to the need for using a DDC system because if narrowband observations are to be made the 400MHz bandwidth should be down converted to required baseband frequency. The addition of DDC will also reduce the data rate because of the decimation performed by DDC block. The narrowband observations are usually done for spectral line analysis of any radio source.

The FFT size used in current GMRT correlator system is 2048 or 2K i.e. 1024 frequency channels. Thus the current spectral resolution is 32/1024 MHz. The GMRT correlator will be upgraded to perform 4K FFT i.e. 2048 frequency channels which will make the spectral resolution as 400/4096 MHz which is enhanced as compared to current GMRT correlator system.

1.3 Need for Digital Down Converter in GPU

Digital Down Conversion is a technique that takes a band limited high sample rate digitized signal, mixes the signal to a lower frequency and reduces the sample rate while retaining all the information.

A fundamental part of many communications systems is Digital Down Conversion (DDC). Digital radio receivers often have fast ADC converters to digitize the band limited RF signal generating high data rates; but in many cases, the signal of interest represents a small proportion of that bandwidth. To extract the band of interest at this high sample rate would require a prohibitively large filter. A DDC allows the frequency band of interest to be moved down the spectrum so the sample rate can be reduced, filter requirements and further processing on the signal of interest become more easily realizable.

In GMRT, when the observed signal is in a narrow band or it is a spectral signal, we can utilize the whole FFT size to analyze the narrowband signal. It gives a good resolution of data. To achieve this, we should down convert the received signal. Earlier, GMRT used to use analog blocks for the down conversion which included very large filters. As a consequence, this system was not efficient and had many drawbacks. Digital Down Conversion has many advantages over the analog one.

After DDC, further operations can be done. As the data received at the receiver through the antennas is very large, a large number of computations are to be done at a very fast rate. CPUs are inefficient in doing this similar kind of work. Graphic Processing Unit can work parallel with CPU so that the computations can be done at a fast rate. This will result into reduced processing time and increased efficiency. So it is better to implement DDC in GPU.

2. GPU AND CUDA-C

The high performance computing (HPC) industry's need for computation is increasing, as large and complex computational problems become commonplace across many industry segments. Traditional CPU technology, however, is no longer capable of scaling in performance sufficiently to address this demand. The parallel processing capability of the GPU allows it to divide complex computing task into thousands of smaller tasks that can be run concurrently. This ability is enabling computational scientists and researchers to address some of the world's most challenging computational problems up to several orders of magnitude faster.

2.1 Graphic Processing Unit (GPU)

A simple way to define GPU is to say that it is a "*parallel numeric computing engine*" [1]. GPUs can be used in case when we encounter very large number of computations but of similar kind. The performance of multi-core microprocessors has not increased much over the years whereas GPUs have showed tremendous increase in performance throughput. The reason for this enlarging performance throughput between GPUs and CPUs lies in fundamental design philosophies between them, design for CPUs are optimized for sequential code performance whereas the motive of GPUs design is to optimize for the execution of massive number of threads. Actually GPUs are designed as a numeric computing engine and it will not perform well on some tasks that CPUs are designed to perform better. Thus, one must expect that most applications will use both CPUs and GPUs, executing the sequential oriented parts on CPU and numerical intensive parts on GPU.

NVIDIA GPU devices:

- Each line of GPU chips comes packaged as "gaming" (**GeForce/GTX**[3]) and "computing" (**Tesla**[3]) boards.
- In **Fermi**[3] arch, compute-specific boards have fast double-precision floating point enabled.

Figure 2-1 Graphic Processing Unit

Following figure shows the fundamental difference in architecture design of CPU and GPU.

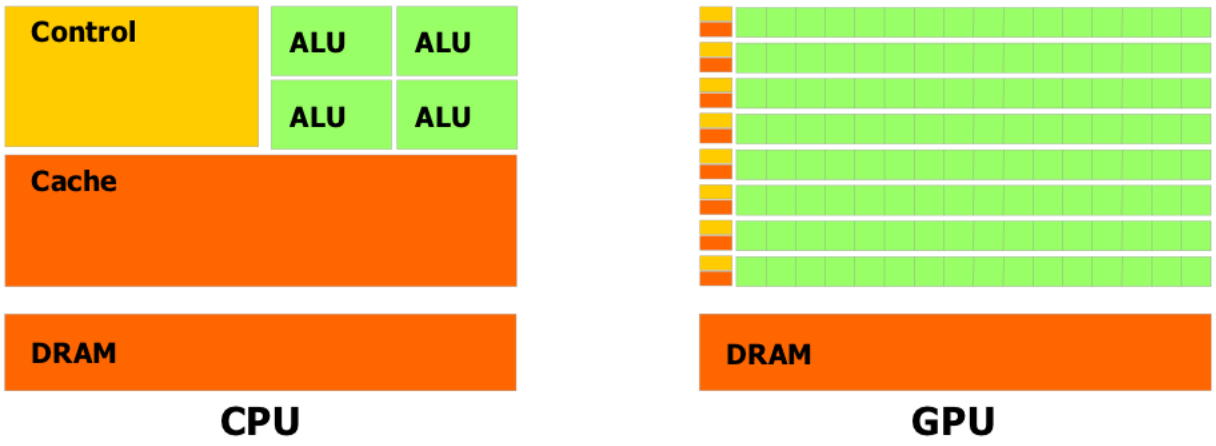

Figure 2-2 Fundamental architecture design difference between CPUs and GPUs[3]

Various GPU programming tools are[3]:

- **OpenGL**
- **OpenCL**
- **NVIDIA CUDA C**(we have used this tool for programming our GPU device)

How GPUs can facilitate Radio Astronomy:

To observe radio universe, signals from multiple telescopes can be correlated to form an interferometer array. Data collected from these telescopes is used to obtain images with high angular resolution as compared to images obtained from single antenna. However, increasing the size of array also increases the amount of computation necessary to correlate the signals.

2.2 CUDA C (Compute Unified Device Architecture)

CUDA C consists of one or more phases that are executed on either the host (CPU) or a device such as GPU. No data parallelism phase are executed in CPU whereas data parallelism phase needs to be executed in GPU. The program supplies a single source code combining both host and device programming. **NVCC**[4] (NVIDIA CUDA C Compiler) separates both the phases.

The host code is straight ANSI C code and is compiled with the standard C compilers on host and runs as an ordinary process. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called Kernels, and their associated data structures. Device code is compiled by NVCC and executed on GPU device.

C for CUDA extends C by allowing the programmer to define C functions, called **Kernels**[4], that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. A kernel is defined using the __global__ declaration specifier and the number of CUDA threads for each call is specified using a new **<<<…>>>**[4] syntax:

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],float  C[N][N])
{
        int i = threadIdx.x;
        int j = threadIdx.y;
        C[i][j] = A[i][j] + B[i][j];
}
int main()
{
        // Kernel invocation
        dim3dimBlock(N, N);
        MatAdd<<<1, dimBlock>>>(A, B, C);
}
```

Each of the threads that execute a kernel is given a unique thread ID that is accessible within the kernel through the built-in **threadIdx**[3]variable.The **threadIdx** is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional threadindex, forming a one-dimensional, two-dimensional, or three-dimensional threadblock.

On current GPUs, a thread block may contain up to ***512***threads.However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. These multiple blocks are organized into a one-dimensional or two-dimensional grid of thread blocks. The dimension of the grid is specified by the first parameter of the <<<…>>>syntax. Each block within the grid can be identified by a one-dimensional or two-dimensional index accessible within the kernel through the built-in **blockIdx**[3]variable. The dimension of the thread block is accessible within the kernel through the built-in **blockDim**[3] variable.

Process involved in execution of parallel computing:

- Identifying the part of application programs that need to be parallelized.
- Using an API function to allocate memory on GPU device for performing parallel computations.
- Using an API function to transfer the data to GPU device.
- Developing a Kernel function that will be executed by individual threads and launching this Kernel function to perform parallel computing.
- Eventually transferring the output data back to host CPU using an API function call.

Overview of CUDA device memory model:

Device Code can –

- R/W per-thread registers.
- R/W per-thread Local Memory.

- R/W per-block Shared Memory.
- R/W per-grid Global Memory.
- Read only per-grid Constant Memory.

Host Code can –

- R/W per-grid Constant Memory.

CUDA API Functions for Device Global Memory Model:

- **cudaMalloc()** and **cudaFree()** –

These are two most important API functions for allocating and de-allocating device Global Memory.

The first parameter for the cudaMalloc()[4]function is the address of a pointer that needs to point to the allocated object after a piece of Global Memory is allocated to it. The second parameter gives the size of the object to be allocated.

cudaFree()[4]frees the object from device global memory and it takes the pointer to the freed object as its parameter.

CUDA API Function for Data Transfer between memories:

- **cudaMemcpy()**[4]–

  Once a program has allocated device memory for the data objects, it can request that databe transferred from the host to the device memory. This is accomplished by calling cudaMemcpy(), the CUDA API functions for data transfer between memories. The cudaMemcpy() function requires four parameters. The first parameter is a pointer to the source data object to be copied. The second parameter points to the destination location for the copy operation. The third parameter specifies the number of bytes to be copied. The fourth parameter indicates the types of memory involved in the copy: from host memory to host memory, from host memory to device memory, from device memory to host memory, and from device memory to device memory.

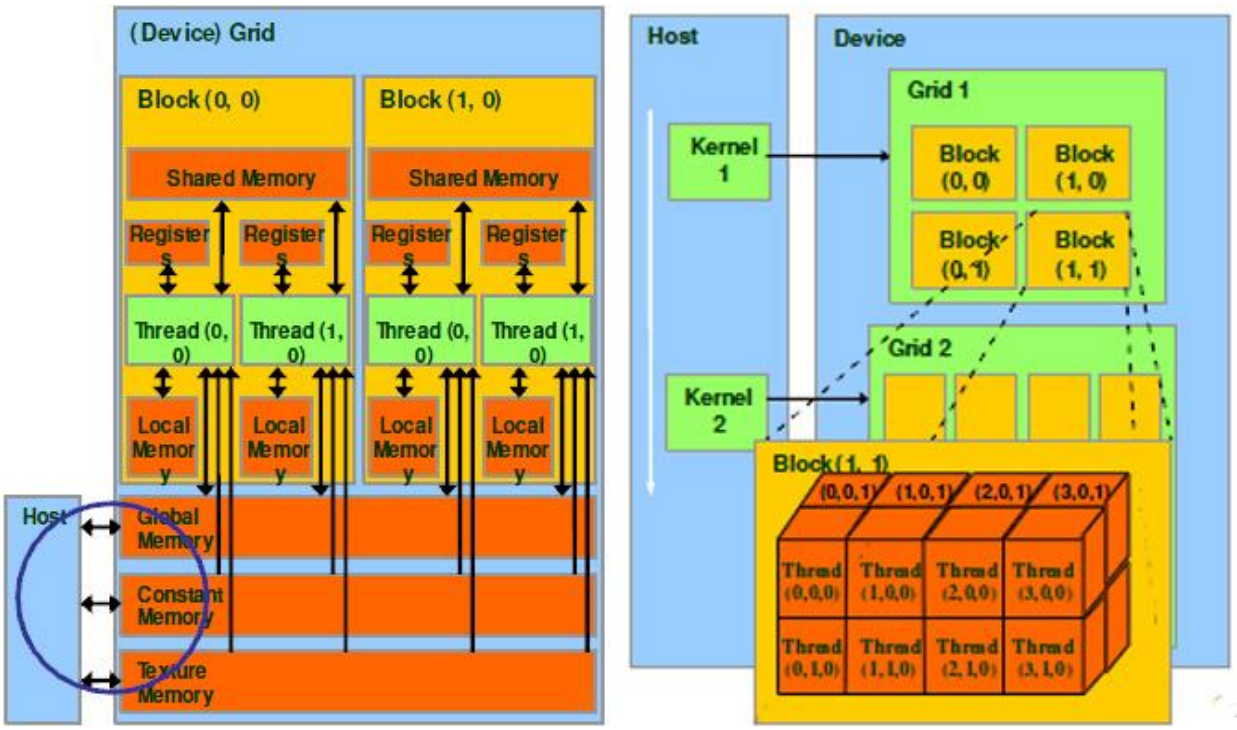

Figure 2-3 CUDA memory architecture (LEFT); CUDA Thread organization (RIGHT)[3]

3. DIGITAL DOWN CONVERTER

Digital Down Conversion is a technique that takes a band limited high sample rate digitized signal, mixes the signal to a lower frequency and reduces the sample rate while retaining all the information.

A fundamental part of many communications systems is Digital Down Conversion (DDC). Digital radio receivers often have fast ADC converters to digitize the band limited RF signal generating high data rates; but in many cases, the signal of interest represents a small proportion of that bandwidth. To extract the band of interest at this high sample rate would require a prohibitively large filter. A DDC allows the frequency band of interest to be moved down the spectrum so the sample rate can be reduced, filter requirements and further processing on the signal of interest become more easily realizable.

In GMRT, when the observed signal is in a narrow band or it is a spectral signal, we can utilize the whole FFT size to analyze the narrowband signal. It gives a good resolution of data. To achieve this, we should down convert the received signal using a DDC and then further image processing operations can be done.

3.1 How a DDC works:

The block diagram of a theoretical DDC is as shown in Figure 3-1.A Digital Down Converter is basically complex mixer, shifting the frequency band of interest to baseband. Consider the spectrum of the original continuous analogue signal prior to digitization, as shown in Figure 3-2, because it is a real signal it has both positive and negative frequency components. If this signal is sampled by a single A/D converter at a rate that is greater than twice the highest frequency the resulting spectrum is as shown in Figure 3-3. The continuous analogue spectrum repeated around all of the sample frequency spectral lines. The first stage of the DDC is to mix, or multiply, this digitized stream of samples with a digitized cosine for the phase channel and a digitized sine for the quadrature channel and so generating the sum and difference frequency components. Figure 3-4 shows the amplitude spectrum of either the phase or quadrature channel after mixing, the mixer frequency has been chosen in this example to move the signal frequency band down to baseband.

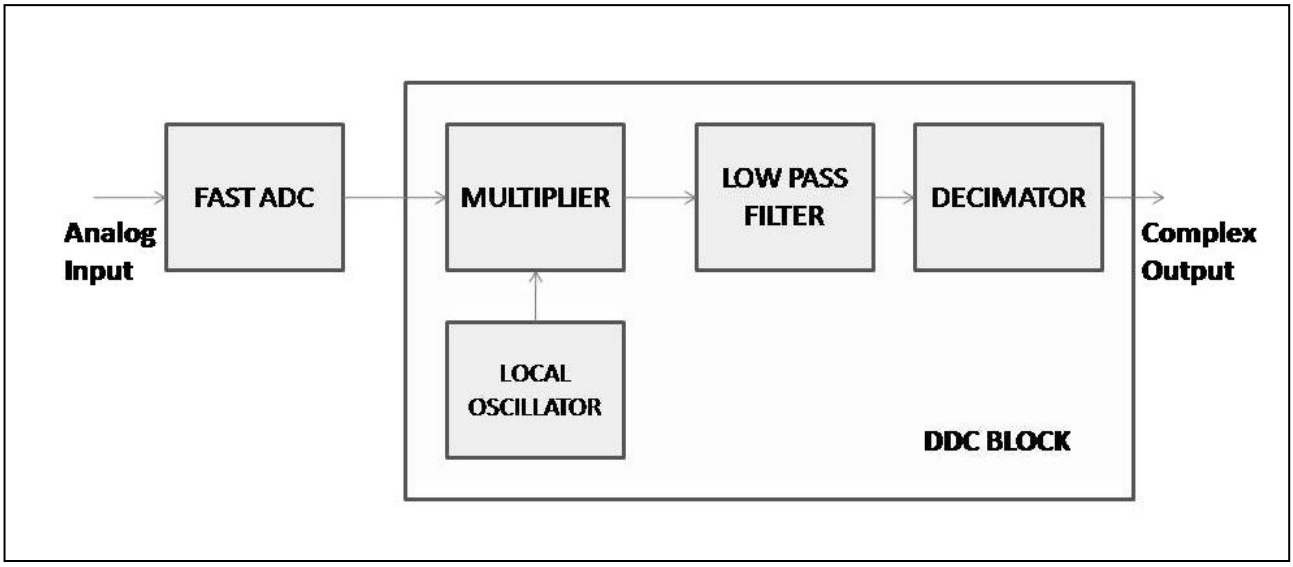

Figure 3-1 A theoretical DDC block diagram

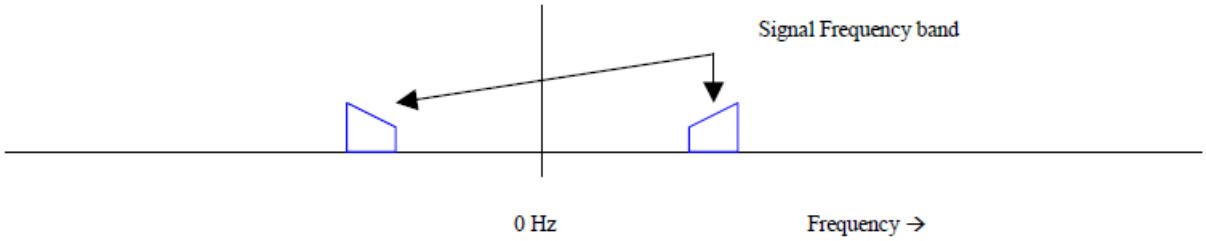

Figure 3-2 spectrum of a continuous analog signal[l]

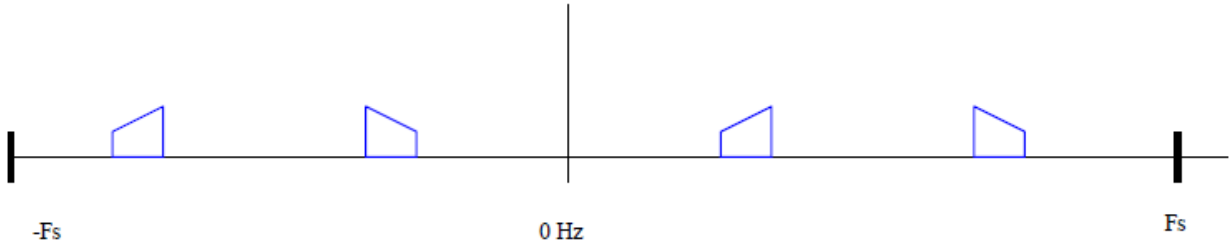

Figure 3-3 Spectrum of digitized signal Sample frequency Fs[5]

The amplitude spectrum of both phase and quadrature channels will be the same but the phase relationship of the spectral components is different. This phase relationship must be retained, which is why all the filters in the phase path must be identical to those in the quadrature path. It should also be noted that because we have quadrature signals the spectral components from both

positive and negative frequencies can be overlaid, for non-quadrature sampling the two frequency components would have to be kept separate and so requiring twice the bandwidth.

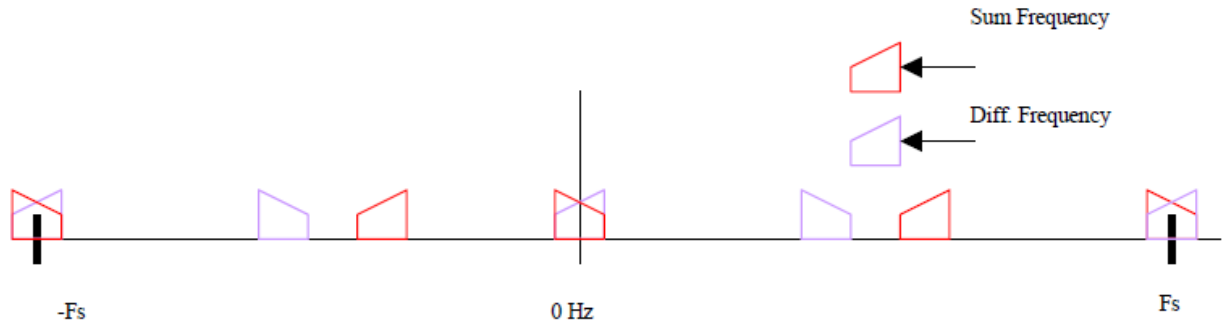

Figure 3-4 Spectrum of digitized signal after mixing sample frequency Fs[5]

This spectrum of both phase and quadrature signals can now be filtered using identical digital filters, with a response shown in Figure 3-5, to remove the unwanted frequency components. The phase and quadrature samples must be filtered with identical filters, which with digital filters are not a problem. A digital filter frequency response is always symmetrical about 0.5Fs. The unwanted frequency components fall outside the pass bands of the filter, giving the resultant spectrum for both phase and quadrature as shown in Figure 3-6.

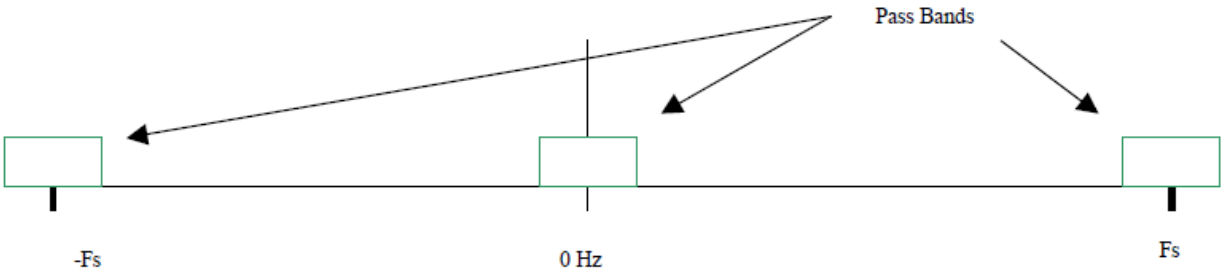

Figure 3-5 Low pass digital filter frequency response Sample frequency Fs[5]

The sample frequency is now much higher than required for the maximum frequency in our frequency band and so the sample frequency can be reduced or decimated, without any loss of information. Up to this point we have talked about using the requirements for the quadrature signals in the digital mixer and in the filters on the phase and quadrature paths, but now we can start to see the advantages of using this technique.

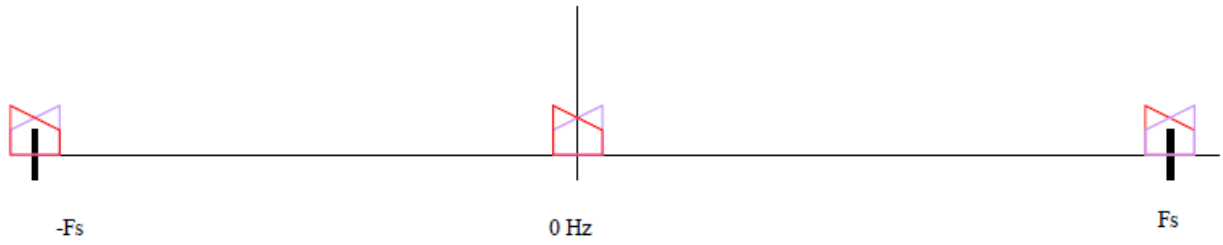

Figure 3-6 Spectrum of digitized signal after filtering Sample Frequency Fs[5]

In theory the sample frequency of the quadrature signals could be reduced to just greater than the bandwidth of our frequency band, although in practice a slightly higher frequency would be chosen to allow a guard band. The new quadrature spectrum is shown in Figure 3.7. With the sample frequency reduced to a seventh of the original value. This decimation is achieved by only taking every seventh value and ignoring the rest. This technique of decimation is limited to integer decimation rates, but there are other more complicated techniques that can give non integer decimation.

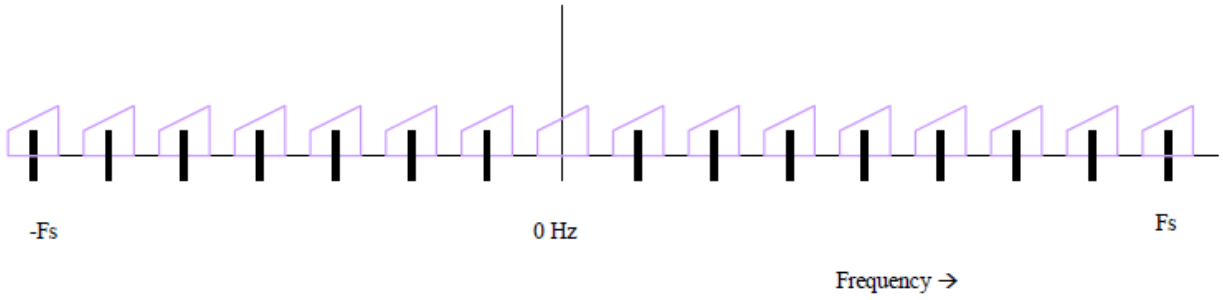

Figure 3-7 Spectrum of Quadrature signal after decimation Sample Frequency Fs/7[5]

The output from the Digital Down converter has retained all the information in our frequency band of interest but has moved it down to baseband and so allowed the sample frequency to be greatly reduced. This has the advantage of simplifying any further processing on the data; the detail of a particular converter will depend on the application. The maximum A/D sample frequency may not be high enough so that band pass sampling techniques, as discussed in more detail later, may be used. The relation between quadrature phase and in-phase can be explained from the figure below.

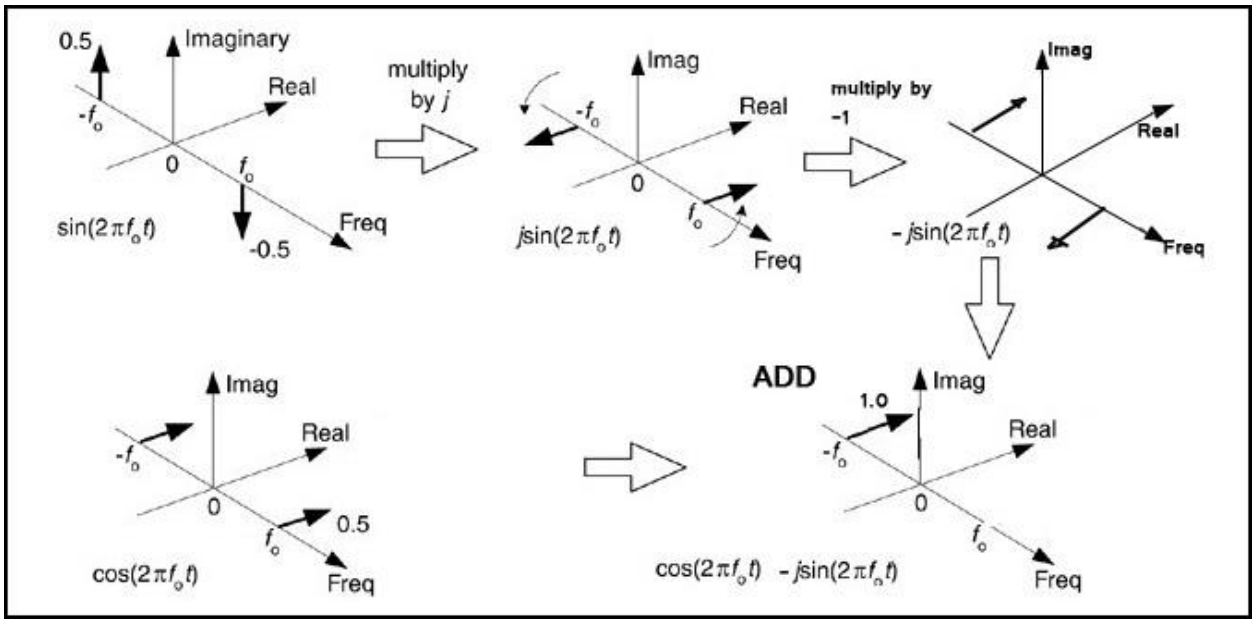

Figure 3-7 Complex Frequency domain view

By using the above method we can eliminate the positive frequency components to get the desired output. The overall operational diagram is as shown below.

3.2 Advantages of using DDC over analog techniques:

The DDC is typically used to convert an RF signal down to baseband. It does this by digitizing at a high sample rate, and then using purely digital techniques to perform the data reduction. Being digital gives many advantages, including:

- Digital stability – not affected by temperature or manufacturing processes. With a DDC, if the system operates at all, it works perfectly – there's never any tuning or component tolerance to worry about.
- Controllability – all aspects of the DDC are controlled from software. The local oscillator can change frequency very rapidly indeed – in many cases a frequency change can take place on the next sample. Additionally, that frequency hop can be large – there is no settling time for the oscillator.
- Size. A single ADC can feed many DDCs, a boon for multi-carrier applications. A single DDC can be implemented in part of GPU device, so multiple channels can be implemented – or additional circuitry could also be added.

3.3 Disadvantages of using DDC:

Mainly the disadvantages are for the digital conversion of the signal.

- ADC speeds are limited. It is not possible today to digitize high-frequency carriers directly.
- The decimation could only be performed for base two decimation factors because FFT sizes used in general are base two numerals.

4. AIMS AND OBJECTIVES

4.1 Objectives of the Project

- To design a DDC block in GPU using CUDA-C
- To do the timing analysis of the DDC block.
- Integrate the DDC block in the existing GMRT Correlator program.

4.2 Problem Definition

GMRT Correlator requires a DDC block designed in GPU which can increase the resolution of the signal received and reduce the further computation time by decreasing the amount of useful received data.

4.3 Project Plan

To achieve the objectives, the project is split up into different work packages:

1. To study the basics of Radio Astronomy and the construction and working of GMRT Correlator
2. To study the CUDA architecture and to implement some simple programs on GPU so as to get familiar with GPU programming.
3. To write separate programs for all the required blocks for the DDC design. This includes Local Oscillator, Multiplier, Low Pass FIR Filter, Decimator and Complex to Complex FFT block
4. To combine all the separate blocks to form the complete DDC block and do the timing analysis of the various operations included in DDC block.
5. To study the dataflow in the existing GMRT Correlator program
6. To integrate the designed DDC block with the existing GMRT Correlator and do the timing analysis of the block in the GMRT Correlator program.

5. DDC BLOCK DESIGN

The digital Down Converter block is to be embedded with the GMRT Correlator[2] as shown in the Figure 5-1 between unpacking block and FFT block. And then it will be followed by Phase Correction Block and MAC (Multiplication and Accumulation).

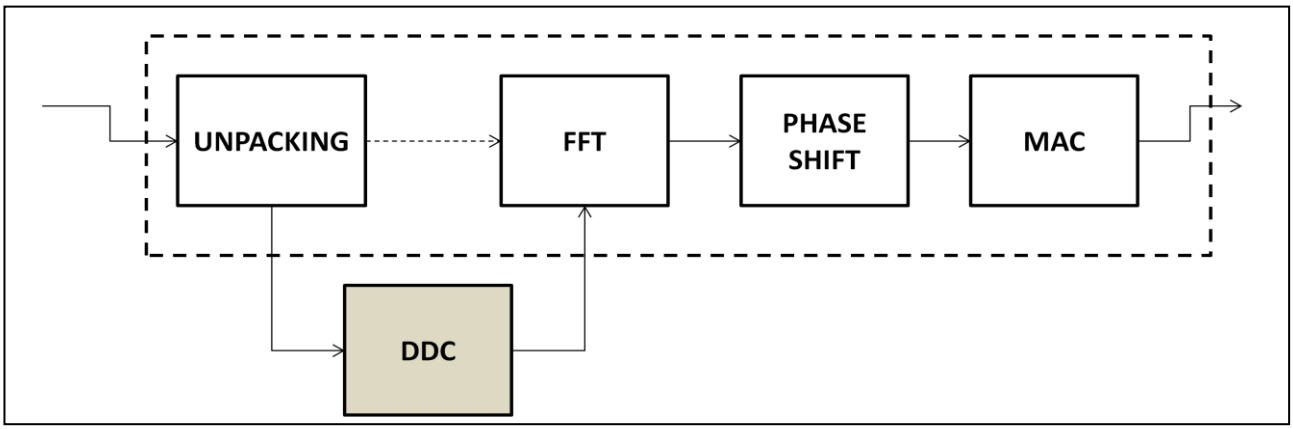

Figure 5-1Poition of DDC block in GMRT Correlator

The DDC block contains following sub-blocks as shown in Figure 5-2. They are explained one by one as follows:

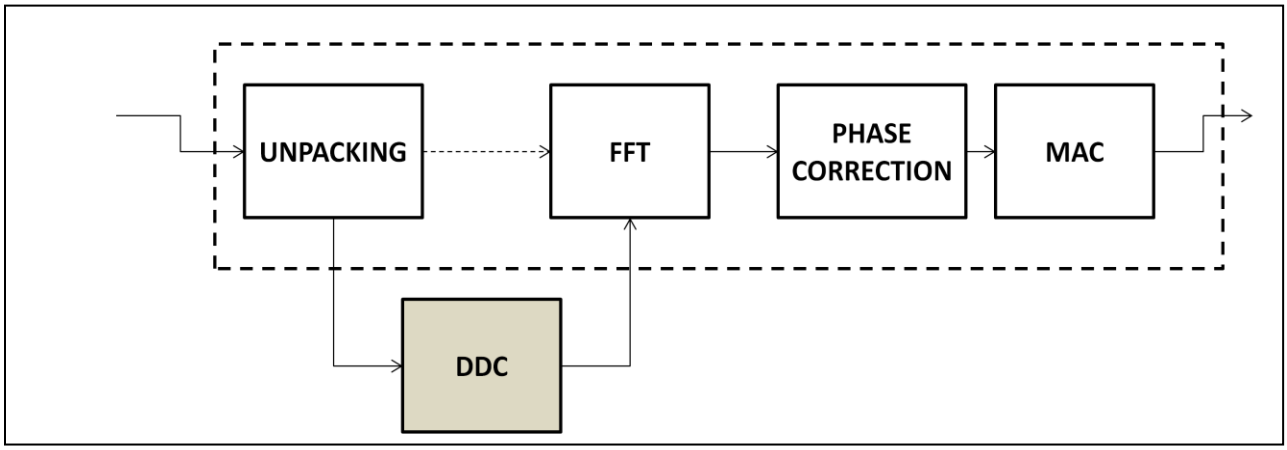

Figure 5-2 DDC Block Descriptions

5.1 Local Oscillator

A sine and cosine wave is generated with user given frequency and unit amplitude. These are continuous waves. The local oscillator frequency is chosen same as the lower cut-off frequency of the required narrowband. This local oscillator data is provided to mixer block.

5.2 Mixer

The mixer blocks perform the operation of multiplying the input data with the sine and cosine waves from local oscillator. Further, this multiplied data is stored and provided to low pass FIR filter.

5.3 Low Pass Filter

These blocks receive data from mixer and perform low pass filtering operation on the same. Both the filters are identical and have number of coefficients as user input. These blocks include multiplication with hamming window coefficients and filter coefficients followed by convolution. The filter coefficients are obtained using the standard library in C whereas window coefficients are generated using the function for the same. The cutoff frequency of the LPFs is same as the bandwidth of the required narrowband. So, the lower cutoff of the required narrowband is the local oscillator frequency and bandwidth is the cutoff frequency of LPF.

5.4 Decimator

When we are down converting the signal to baseband, the bandwidth of interest reduces. Thus the sampling frequency can be reduced to double the bandwidth of the desired band. This reduces the amount of data significantly, and consequently the time for the processing data is also reduced. Thus decimator block decimates the data by the decimation factor which is obtained as a command line user input.

5.5 Complex to Complex FFT[7]

The decimated in-phase values (obtained after multiplication with cosine wave and LPF) from the decimator are treated as real part of complex number. The negative of quadrature phase values (obtained after multiplication with sine wave and LPF) are treated as the imaginary part of the respective complex numbers. The complex to complex FFT is performed using the standard CUFFT library in CUDA-C. The FFT size is taken as a user input.

5.6 Frequency Domain Visualization of working of DDC Block

The following figures will give an idea about the working of DDC. We consider a 200MHz bandwidth signal from -100MHz to 100MHz as an input and a local oscillator of 40MHz and LPF of 30MHz.

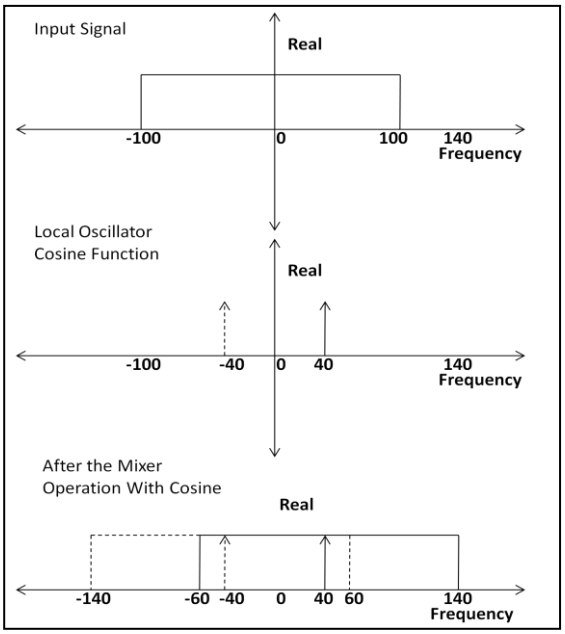

Figure 5-3 Obtaining In-Phase Signal

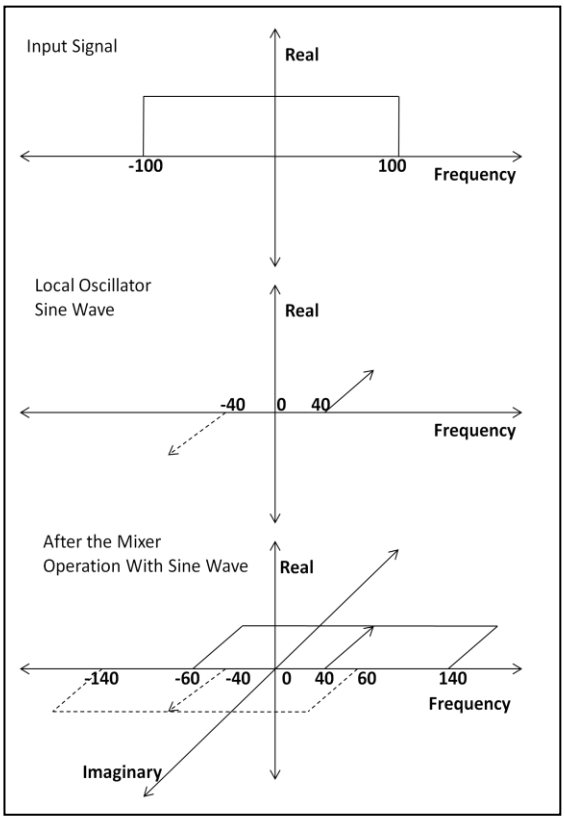

Figure 5-4 Mixing Operation of Input Signal and Sine Wave in Frquency Domain

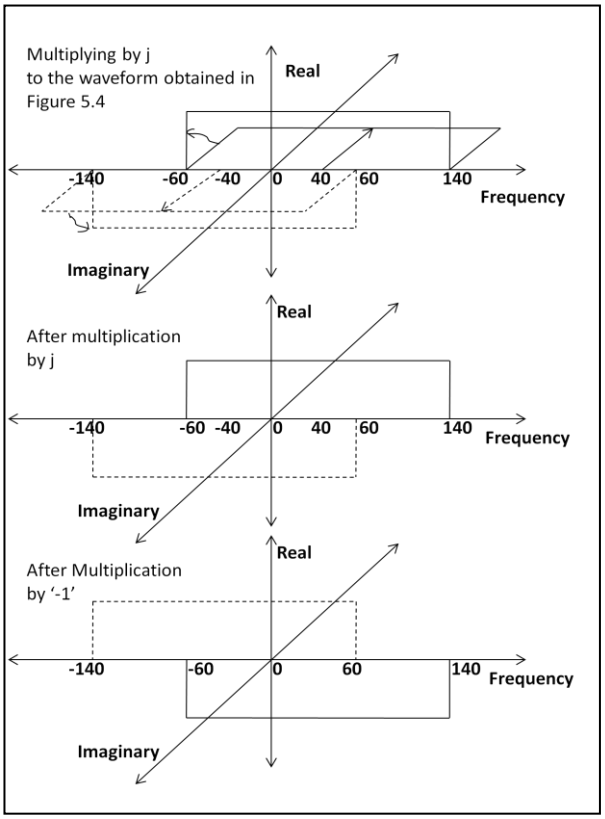

Figure 5-5Obtaining Quadrature Phase Signal

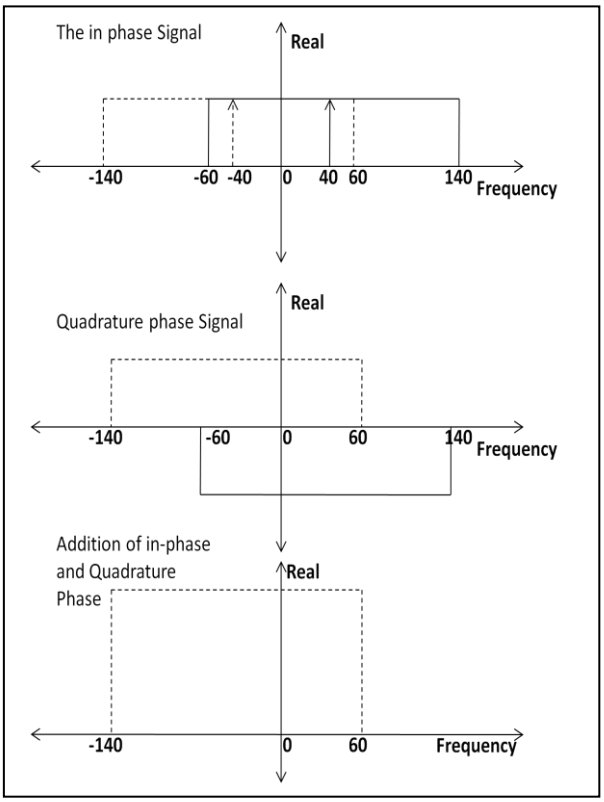

Figure 5-6 Addition of In-Phase and Quadrature Phase Signals

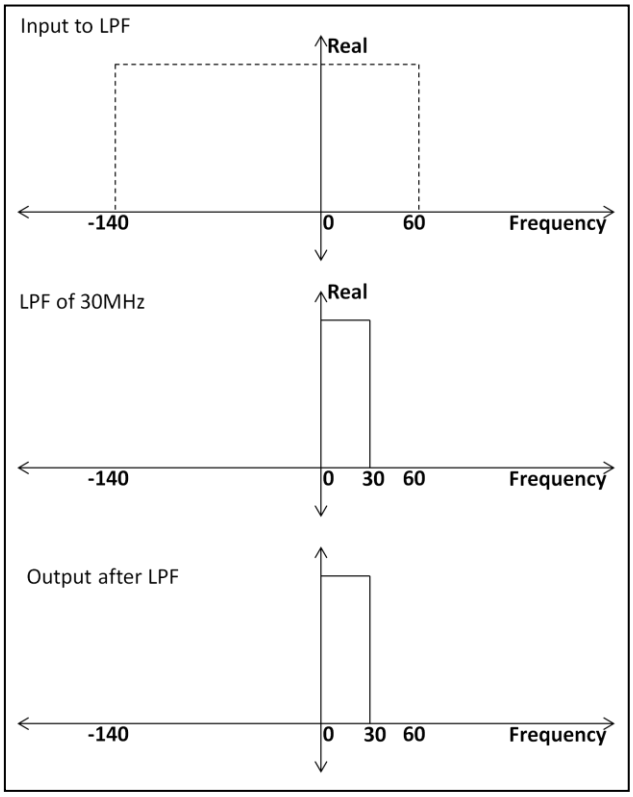

Figure 5-7 LPF Operation

After the LPF, Decimation[5] is done which enables us to see the desired band of particular bandwidth (in this case 40MHz-70MHz, Bandwidth 30MHz) spread over all the 1024 channels.

6. INTEGRATION OF DDC WITH GMRT CORRELATOR BLOCK

6.1 Brief explanation of GMRT correlator code:

The GMRT correlator block is written in CUDA-C and runs on GPU machine. It performs correlation operation on data obtained from various antennas, calculating both cross-correlation and self-correlation values. Although we have limited our study of GMRT correlator program only up to two antennas.

The input data is received as binary value and read as signed char. The input data is unpacked and converted into floats. Then FFT is performed and phase corrections are done by calling separate CUDA-C functions for all these operations. Time domain input signal once converted into frequency domain and its phase corrected is multiplied to perform correlation operation. The data is processed in large number of fixed sized input data blocks and correlation output obtained for all these are accumulated together; resulting into integration of correlated output data. The multiplication and data accumulation operations are altogether performed by MAC (Multiplication and Accumulation) block.

The GMRT correlator code contains a main program which calls all the required functions. All the required parameters are parsed as command line inputs. A separate function is called to allocate and de-allocate memories on GPU device. The correlation output is visualized using a tool named ‘TAX’ which was developed along with GMRT correlator. Our familiarity with this tool has been restricted only to its usage and complete functioning and understanding is not needed for this project.

6.2 Adding DDC feature to GMRT correlator:

The input signal after unpacking and its conversion to floats is multiplied with sine and cosine waves separately, thus performing the mixing operation. The frequency of sine and cosine wave is the LO frequency (the baseband frequency to which the signal needs to be down converted). The sine and cosine waves are generated and multiplied within the unpack function of original GMRT correlator code by adding these features to the unpack function. The decimation is also performed here itself so that amount of data processed further gets reduced. The factor by which data is decimated determines the low pass filter cutoff frequency. The GMRT correlator code was modified in such a way that it accepts only powers of 2 (1, 2, 4, 8,

16…) as decimation factor. This is because; the number of bits of data we are processing at a time is in powers of 2. If we don't give the decimation factor in powers of two, some of the samples will be missed out.

Then convolution is performed using the function used in original DDC code on outputs coming from both sine and cosine multiplication. After convolution output of sine multiplied signal is taken as imaginary part of a complex number and is also multiplied by '-1', whereas convolution output of cosine multiplication is treated as real value of the complex number. The complex number output obtained after convolution is converted into frequency domain by doing FFT. But, the FFT block in original GMRT correlator code was only enabled to perform Real to Complex FFT, thus to rectify this problem FFT block of GMRT correlator code is modified to perform Complex to Complex FFT.

The GMRT Correlator program featured with DDC was further modified according to the phase correction block and MAC block.

6.3 Sky Test of GMRT Correlator Program Featured With DDC

The GMRT correlator with DDC block integrated to it was tested for signals received from a radio source in sky. The radio source used for observation was a quasar named 3C48 at 1280MHz frequency band [9]. Two of GMRT antennas were employed for observations, one from Central Square C-11 and one from southern branch S-02. The signal received at feed of antenna is in form of raw voltage. When signal arrives at GMRT backend system it is digitized using fast ADC connected to ROACH board. *ROACH (Reconfigurable Open Architecture Computing Hardware) is a standalone FPGA processing board*[10].The output of ADC is packetized on ROACH board and passed to GPU using a 10GB Ethernet link.

The data received from sky is stored in shared memory. The GMRT correlator integrated along with DDC used to take its input from a file. Now the GMRT correlator code input flag was modified to receive the data directly from GPU shared memory instead of reading it from a file. The integration of data was done for 0.671 seconds.

The self-correlation and cross-correlation operations were performed on the signals received from the antennas and DDC operation was also performed by varying parameters namely, LO frequency, decimation factor and number of filter taps.

7. RESULTS AND DISCUSSIONS

7.1Output after decimation operation:

All results were obtained by testing the DDC blocks for two different input signals, namely:

- Sine wave of 40MHz and size of 400MB sampled at 400MHz and Channel Noise
- Source of bandwidth 200MHzand size of 64MB sampled at 400MHz

For sine wave input:

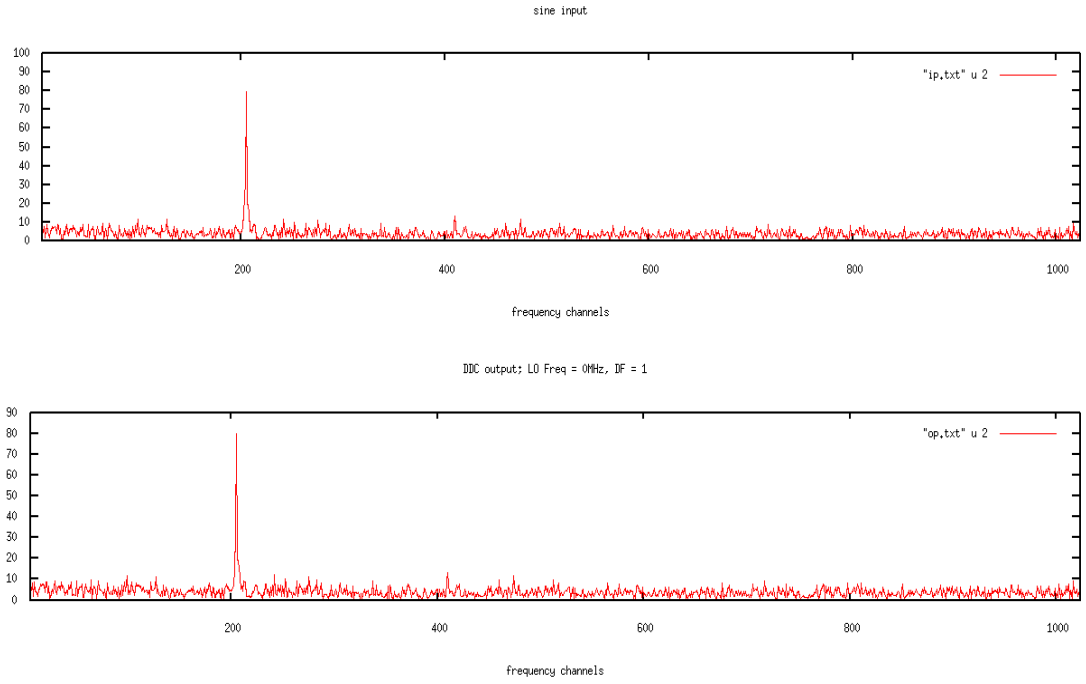

Figure 7-1 DDC Input and output; LO 0MHz, DF 1, Taps 21

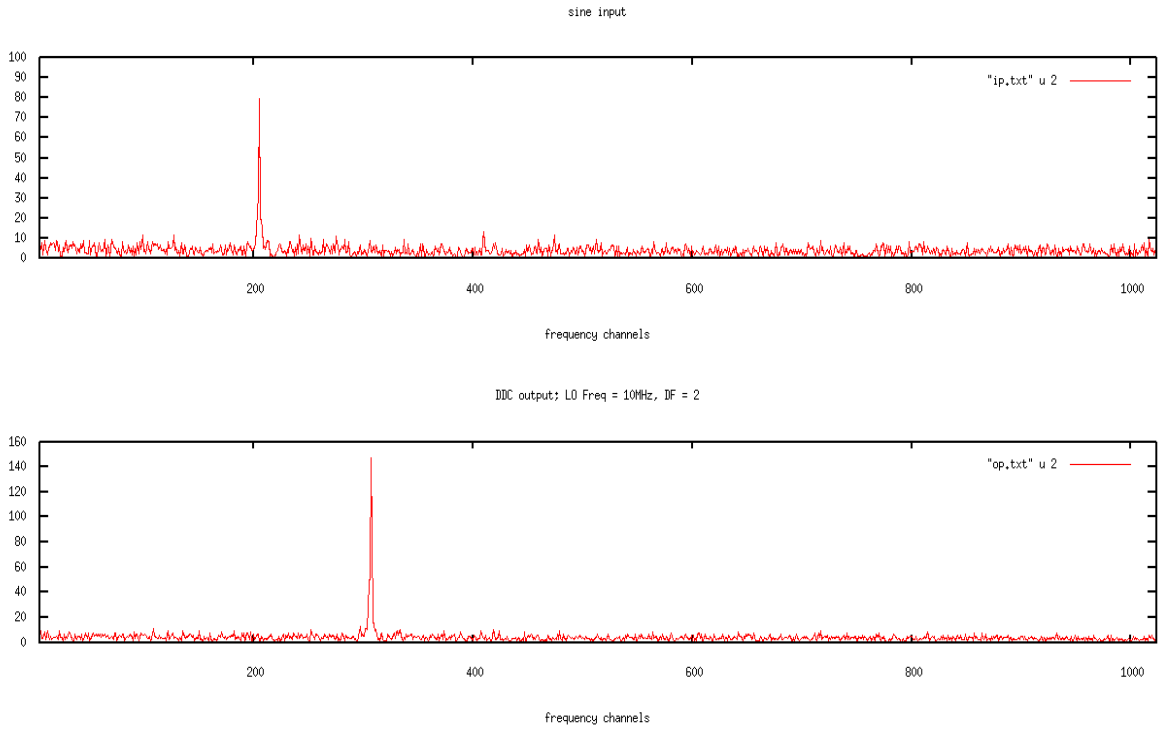

Figure 7-2 DDC Input and Output; LO 10 MHz, DF 2, Taps 21

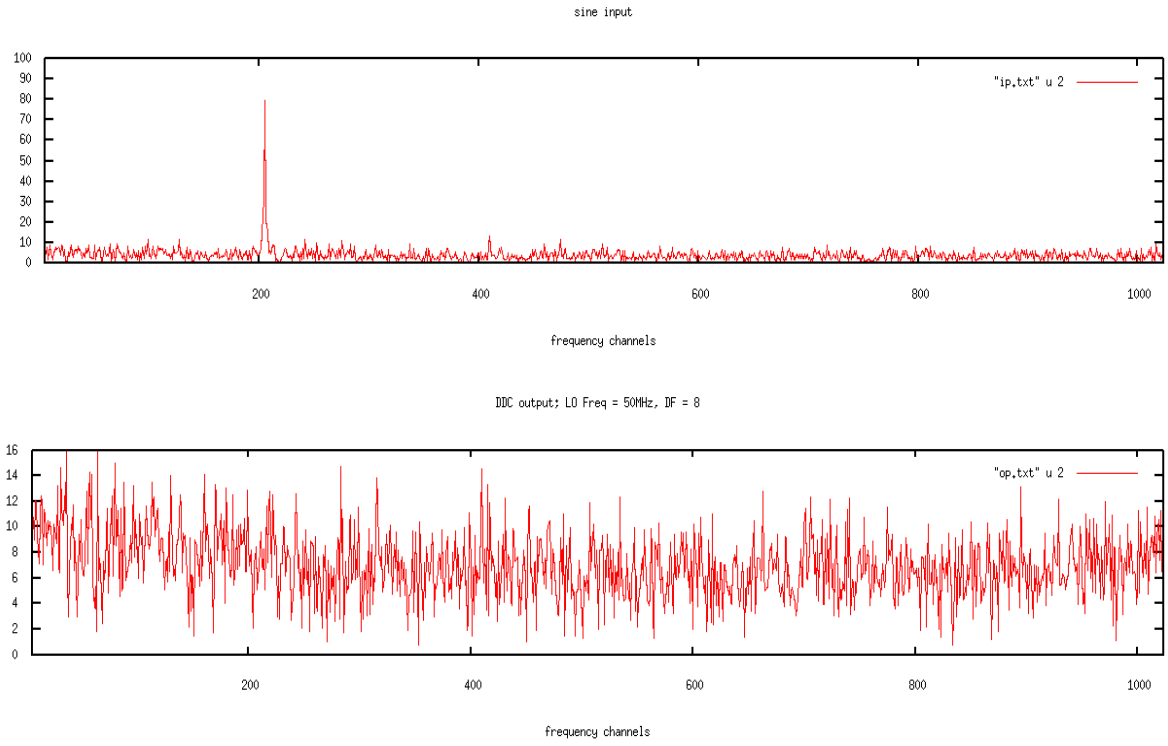

Figure 7-3 DDC Input and Output; LO 50MHz, DF 8, Taps 21

For channel noise input:

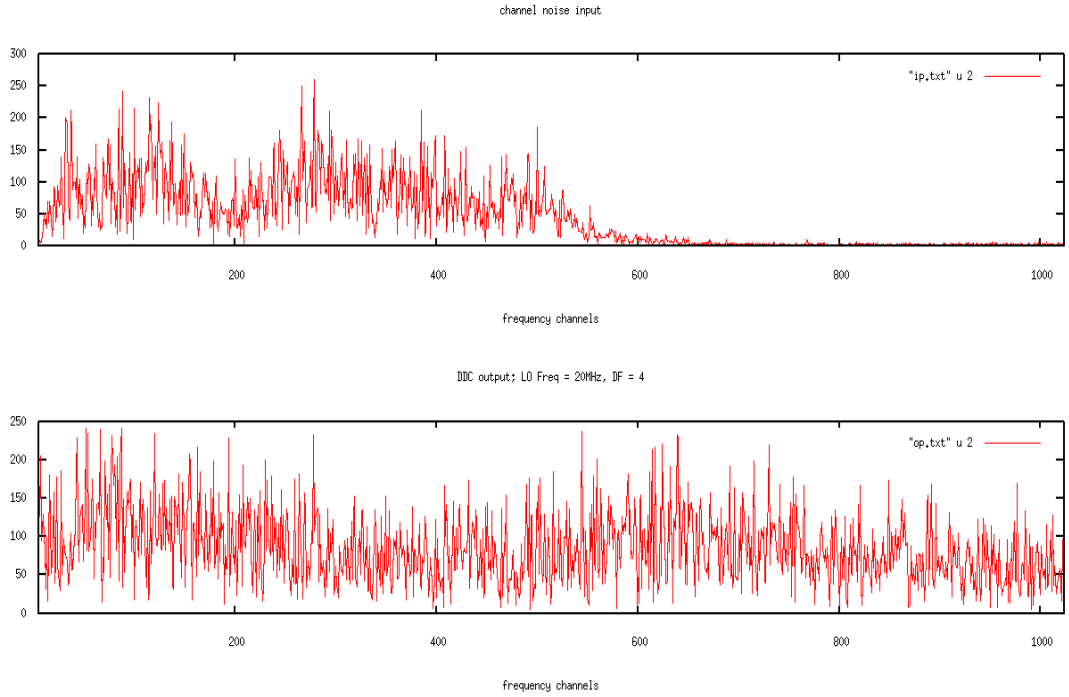

Figure 7-4 DDC Input and Output; LO 20 MHz, DF 4, Taps 21

Discussion:

- The complete band is replicated in output when LO is set at 0MHz and DF at 1 for the sine wave input as observed in Figure 7-1.Neither down conversion nor decimation takes place.
- The sine peak at 40MHz in input signal is seen at 30MHz when input signal is down converted to 10MHz by setting LO at 10MHz. Also, the entire band is not observed because the signal was decimated by a factor of 2. The band observed in Figure 7-2 is 10-110 MHz.
- When LO frequency is set at 50MHz and DF at 8, the 50MHz-75MHz band gets down converted and the sine wave peak is not observed. The decimated output is plotted in Figure 7-3 and the down converted band is spread over complete 1024 channels and thus, resulting into increased resolution.
- The DDC operation was also verified with the channel noise input as observed in Figure 7-4

7.2 GMRT correlator output including DDC operation

To visualize the output from GMRT correlator we used a tool called 'TAX'. Its knowledge was only restricted to its usage and not its complete understanding.

Note: All plots have frequency channels on x-axis ranging from 0-1023 a total of 1024 channels as FFT size used at GMRT is 2048.

The value of frequency corresponding to every channel is depicted as follows:

$$F = \frac{N}{2F_c}\left(F_p - F_{lo}\right)$$

- Where,
  F is frequency at which peak of input sine wave is expected.
- N is number of frequency channels, as 2K FFT is used thus N is 1024.
- F_p is frequency at which peak of sine wave lies in input signal i.e. 40MHz.
- F_{lo} is local oscillator frequency.
- F_s is sampling frequency.
- F_c is LPF cutoff frequency which is dependent of Decimation factor 'D' by following relation:

$$F_c = \frac{F_s}{2D}$$

Note: The outputs for sine wave input are plotted in channel 00 and for channel noise input in channel 01.

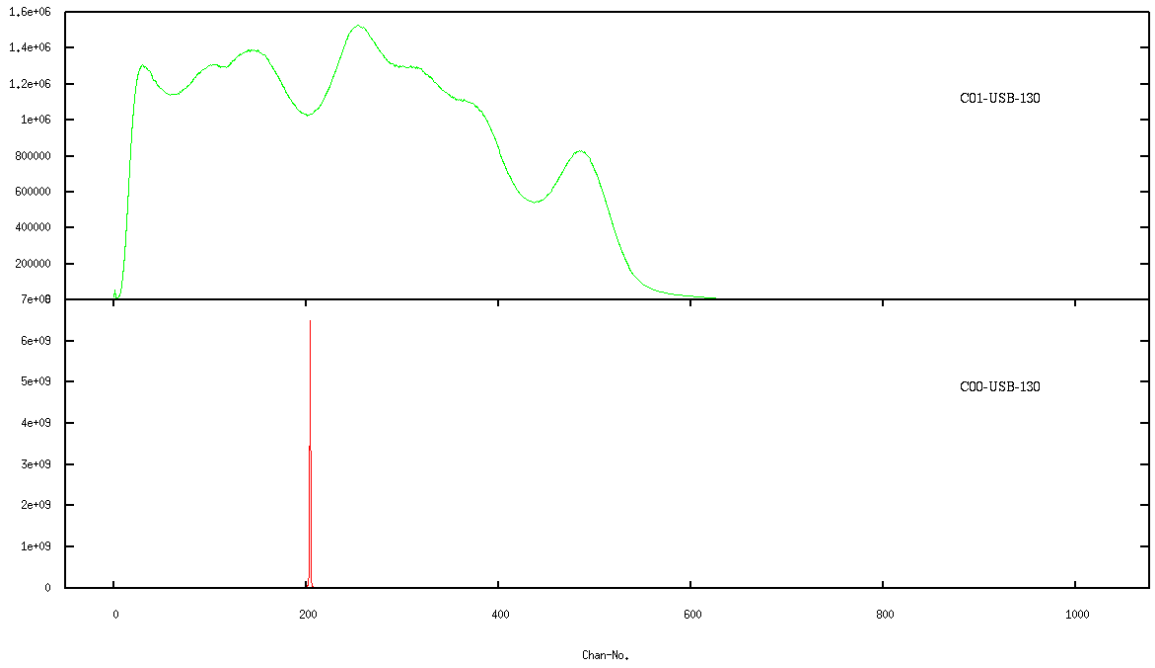

Figure 7-5 Plot showing output of GMRT correlator viewed using TAX tool; this plot shows the output performing DDC operation; LO Freq = 0MHz and Decimation Factor = 1; x-axis – frequency channels, y-axis – signal level

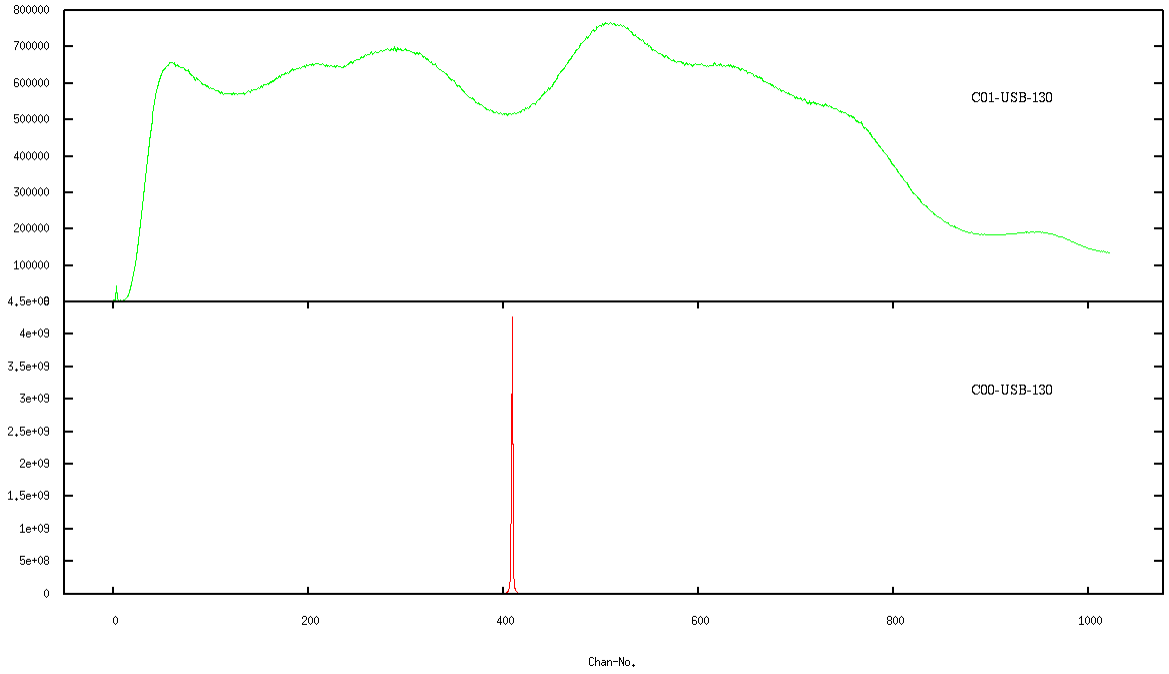

Figure 7-6 Plot showing output of GMRT correlator viewed using TAX tool; this plot shows the output performing DDC operation; LO Freq = 0MHz and Decimation Factor = 2; x-axis – frequency channels, y-axis – signal level

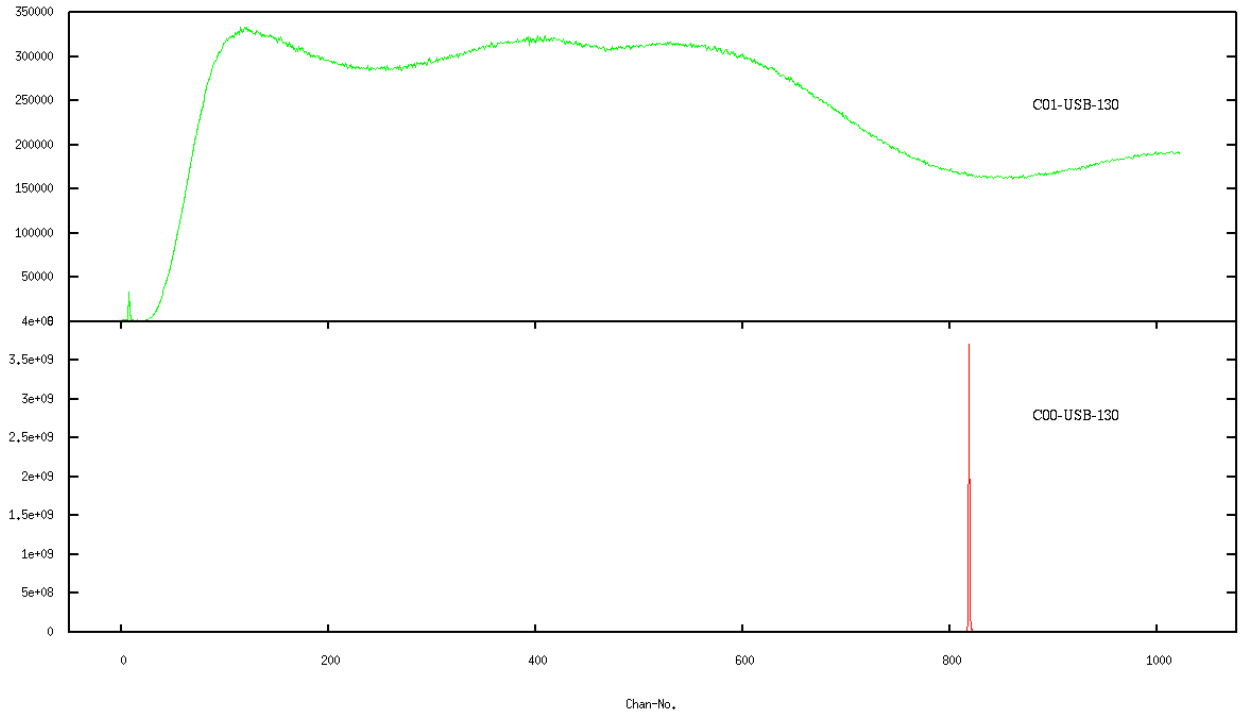

Figure 7-7 Plot showing output of GMRT correlator viewed using TAX tool; this plot shows the output performing DDC operation; LO Freq = 0MHz and Decimation Factor = 4; x-axis – frequency channels, y-axis – signal level.

Discussion:

- When LO frequency is set at 0MHz and decimation factor at 1, the input signal is neither down converted nor decimated as shown in Figure-7.9. The sine wave peak which was at 40MHz in input signal is still observed at the same frequency. The same operation takes place for channel noise input.
- When decimation factor is changed to 2 keeping the LO frequency unchanged i.e. at 0MHz, decimated output is observed in Figure-7.10 in form of bandwidth reduction to 100MHz from 200MHz. The output signal is decimated over all 1024 frequency channels, thus increasing the resolution.
- When decimation factor is increased to 4 keeping LO frequency at 0MHz, the bandwidth is further reduced to 50MHz as observed in Figure-7.11.

As the LO frequency was set at 0MHz for all the above cases the down conversion is not observed. And by varying the decimation factor we observe the bandwidth reduction, thus proving that decimation takes place.

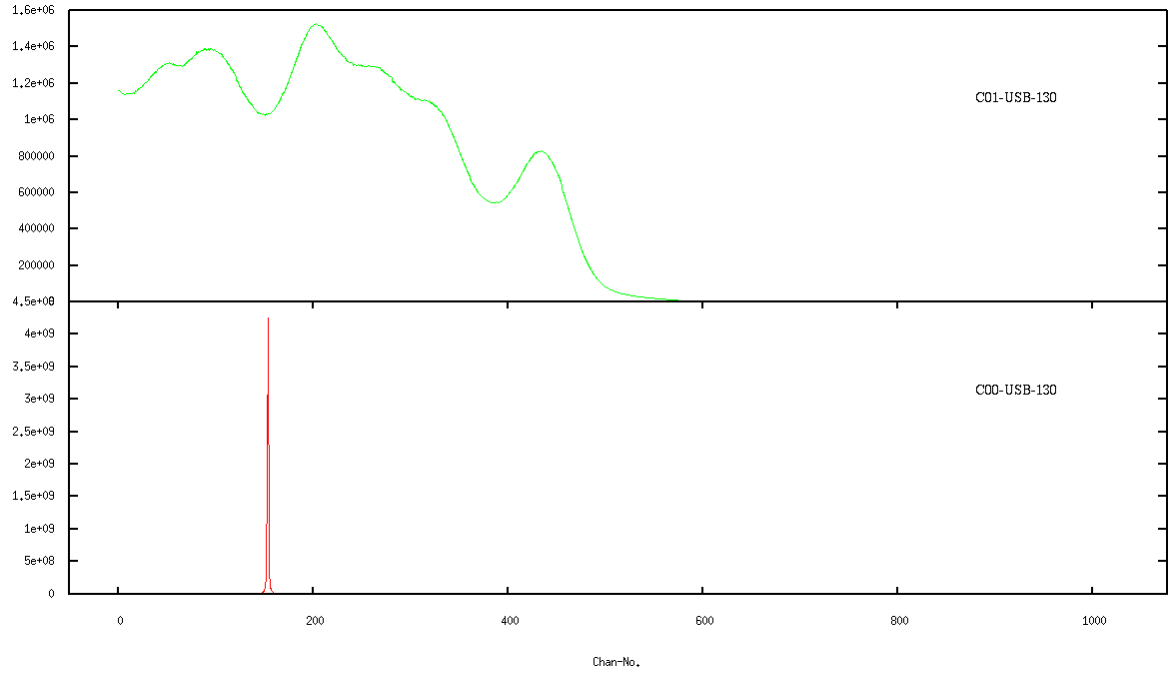

Figure 7-8 Plot showing output of GMRT correlator viewed using TAX tool; this plot shows the output performing DDC operation; LO Freq = 10MHz and Decimation Factor = 1; x-axis – frequency channels, y-axis – signal level.

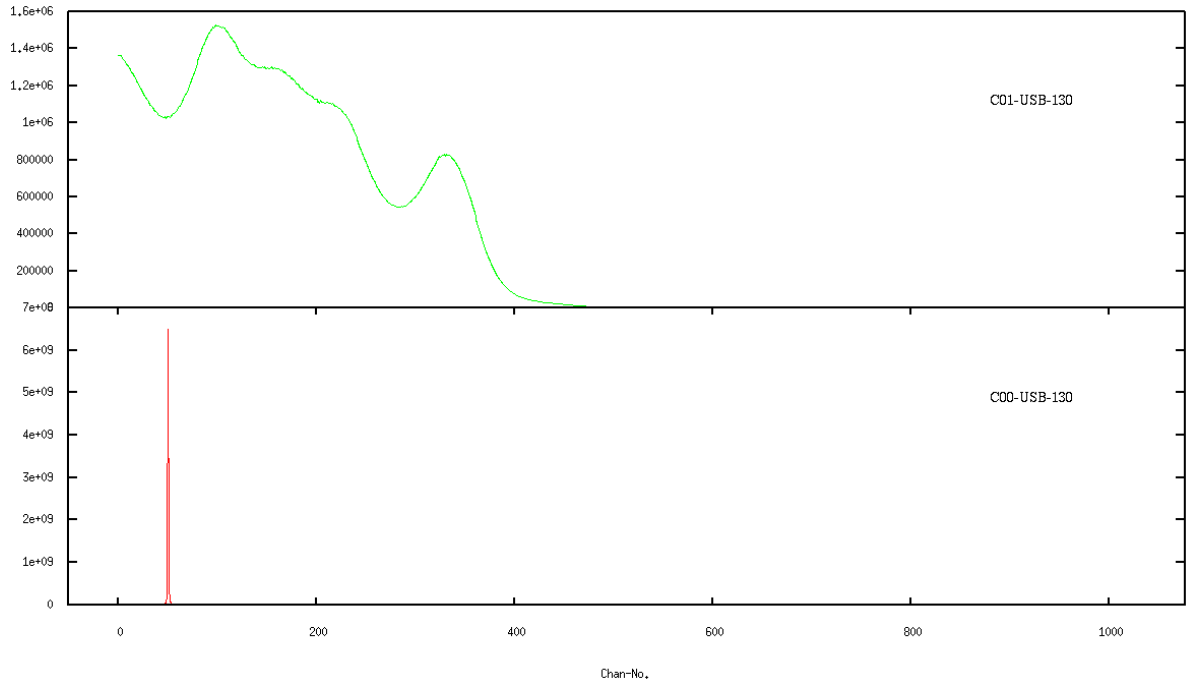

Figure 7-9 Plot showing output of GMRT correlator viewed using TAX tool; this plot shows the output performing DDC operation; LO Freq = 30MHz and Decimation Factor = 1; x-axis – frequency channels, y-axis – signal level.

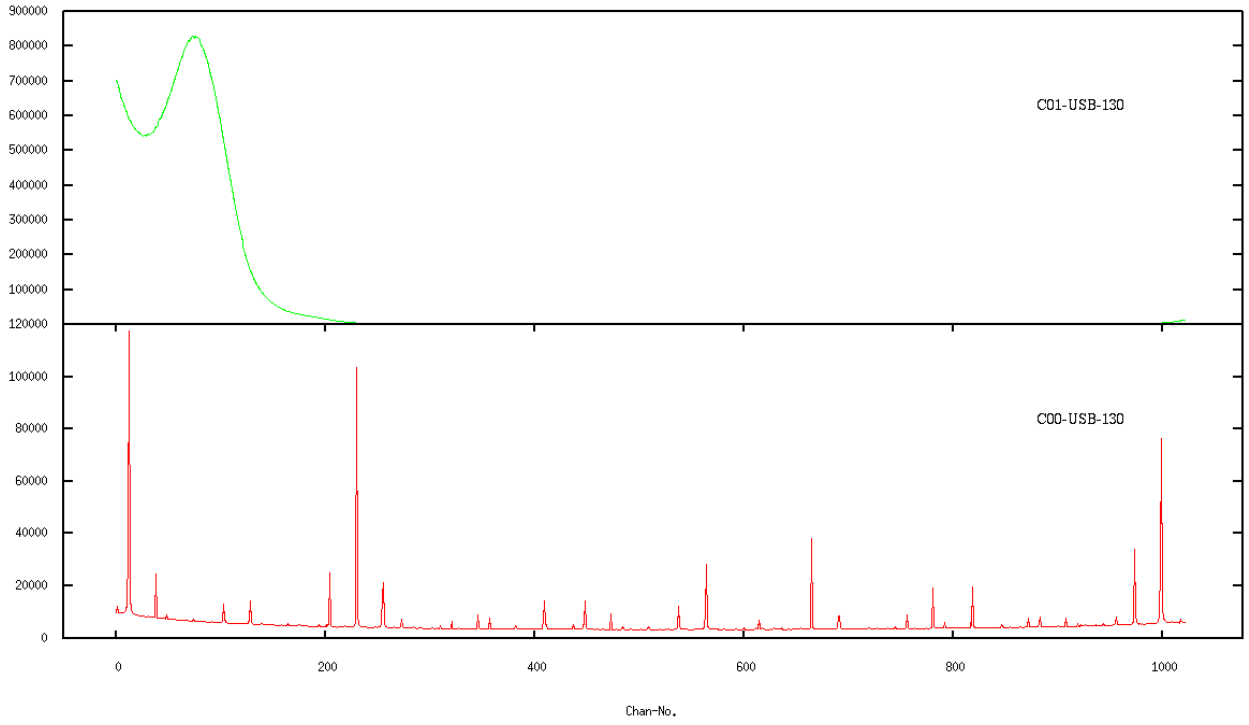

Figure 7-10 Plot showing output of GMRT correlator viewed using TAX tool; this plot shows the output performing DDC operation; LO Freq = 80MHz and Decimation Factor = 1; x-axis – frequency channels, y-axis – signal level.

Discussion:

- When LO is set at 10MHz and decimation factor is at 1, the signal is not decimated but non-zero LO frequency leads to down conversion as observed in Figure-7-12.
- The LO frequency is increased to 30MHz keeping the decimation factor unchanged, the input signal is down converted to 30MHz as observed in Figure-7-13.
- The sine wave peak which was present at 40MHz is not seen when band is down converted to 80MHz by further increasing the LO frequency to 80MHz. Instead we observe low voltage noise source as observed in Figure-7-14.
- Keeping the decimation factor 1 in all the above three cases leads to no decimation, but down conversion of input signal takes place according to the LO frequency. Thus, down conversion phenomenon is tested.

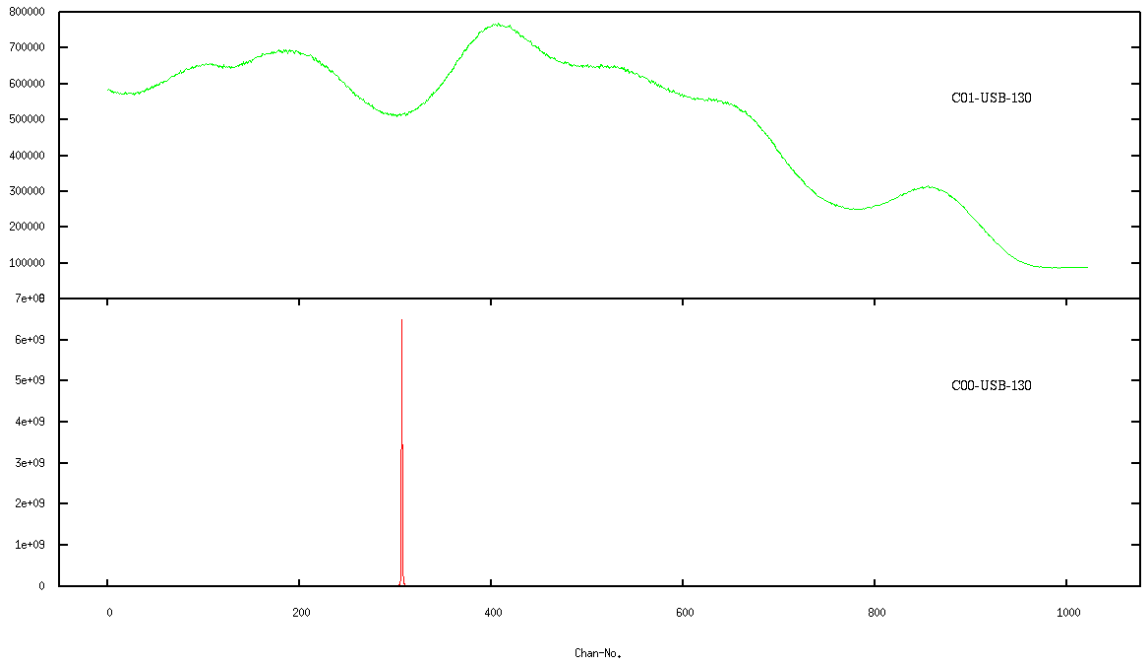

Figure 7-11 Plot showing output of GMRT correlator viewed using TAX tool; this plot shows the output performing DDC operation; LO Freq = 10MHz and Decimation Factor = 2; x-axis – frequency channels, y-axis – signal level.

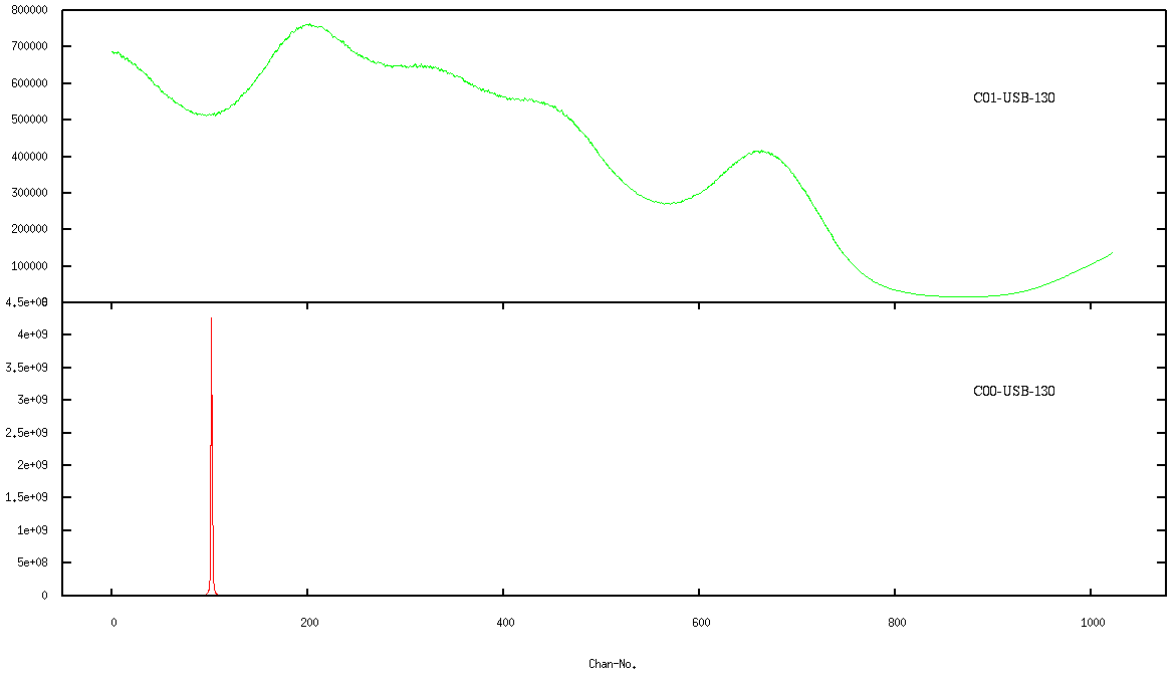

Figure 7-12 Plot showing output of GMRT correlator viewed using TAX tool; this plot shows the output performing DDC operation; LO Freq = 30MHz and Decimation Factor = 2; x-axis – frequency channels, y-axis – signal level.

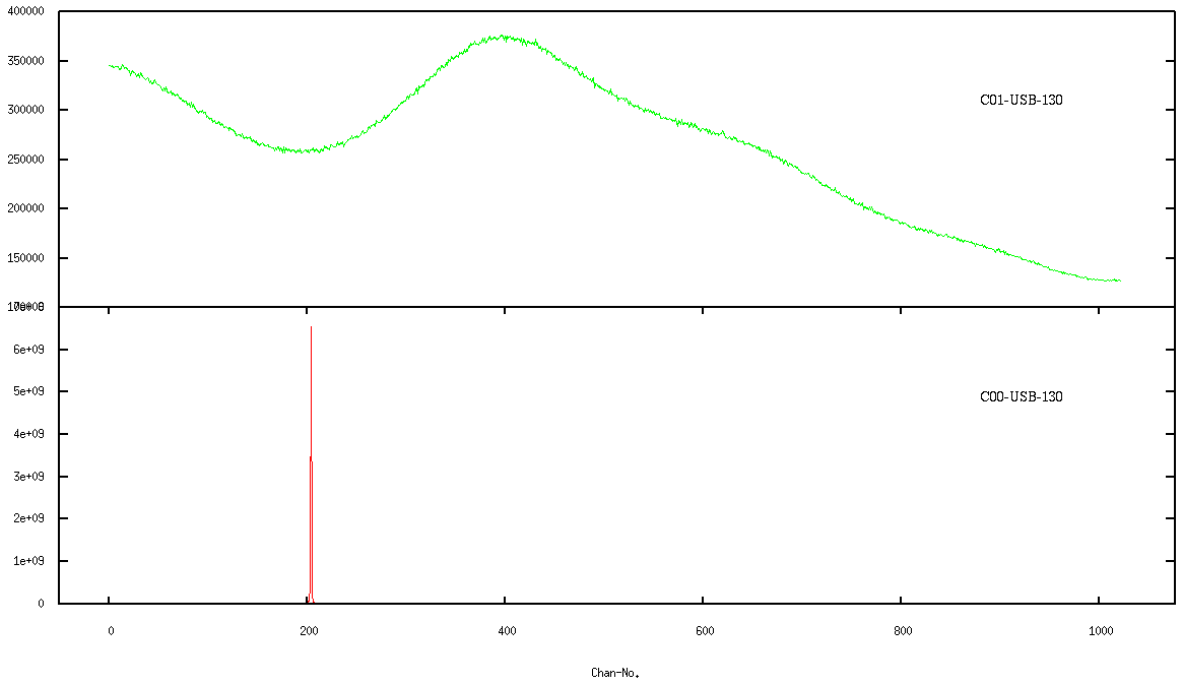

Figure 7-13 Plot showing output of GMRT correlator viewed using TAX tool; this plot shows the output performing DDC operation; LO Freq = 30MHz and Decimation Factor = 4; x-axis – frequency channels, y-axis – signal level.

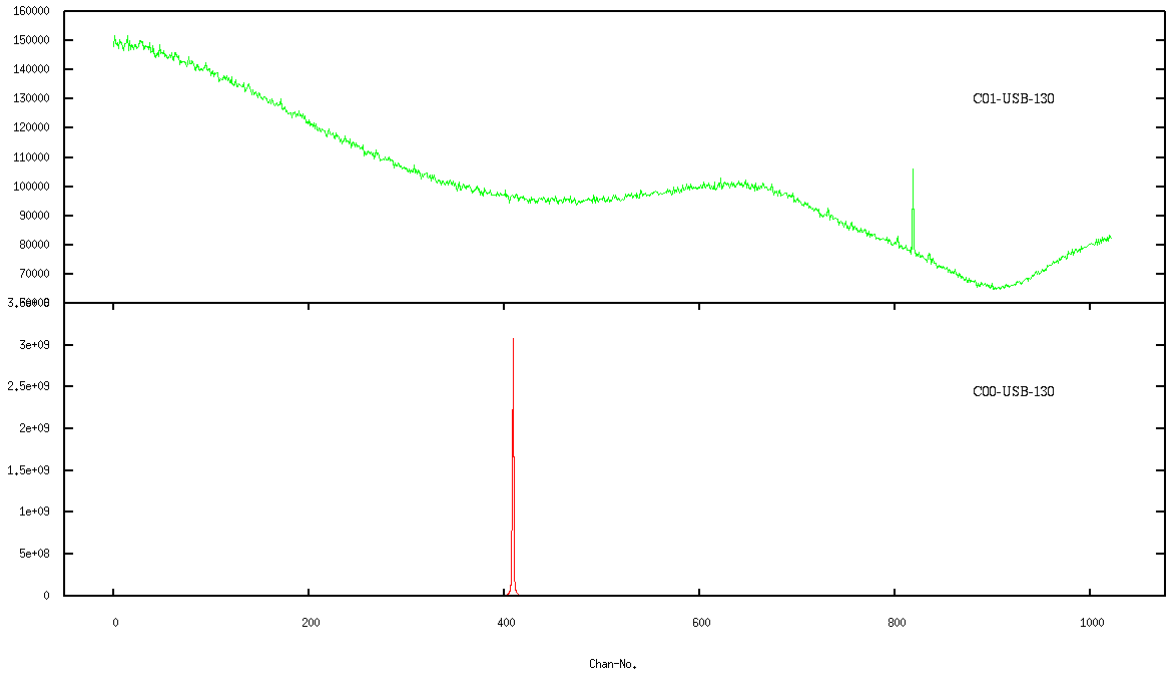

Figure 7-14 Plot showing output of GMRT correlator viewed using TAX tool; this plot shows the output performing DDC operation; LO Freq = 30MHz and Decimation Factor = 8; x-axis – frequency channels, y-axis – signal level.

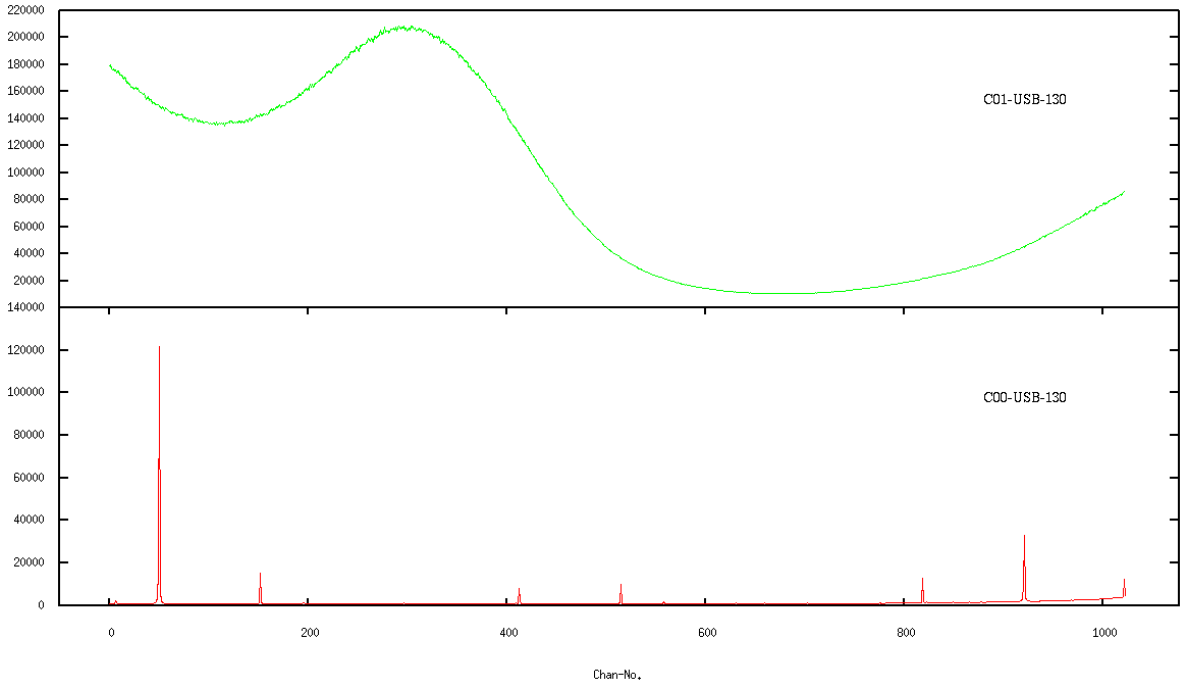

Figure 7-15 Plot showing output of GMRT correlator viewed using TAX tool; this plot shows the output performing DDC operation; LO Freq = 80MHz and Decimation Factor = 4; x-axis – frequency channels, y-axis – signal level.

Discussion:

- The LO frequency is set at 10 MHz and decimation factor at 2. The input signal is moved to baseband frequencies of 10MHz-110MHz as observed in Figure-7.15. The signal gets down converted to 10MHz and decimation factor of 2 leads to 100MHz LPF cut-off frequency. Thus the band 10MHz-110MHz is spread over all 1024 frequency channels.
- The authenticity of above module is tested by varying the LO frequency and decimation factor. The desired outputs were obtained for all cases as shown in Figure-7-16, Figure-7-17, Figure-7.18 and Figure-7.19.
- By varying decimation factor and LO frequency we can obtain desired down conversion and decimation of the input signal. Thus, verifying working of DDC block along with GMRT correlator system.

7.3 Sky Test Outputs of GMRT Correlator Featured with DDC

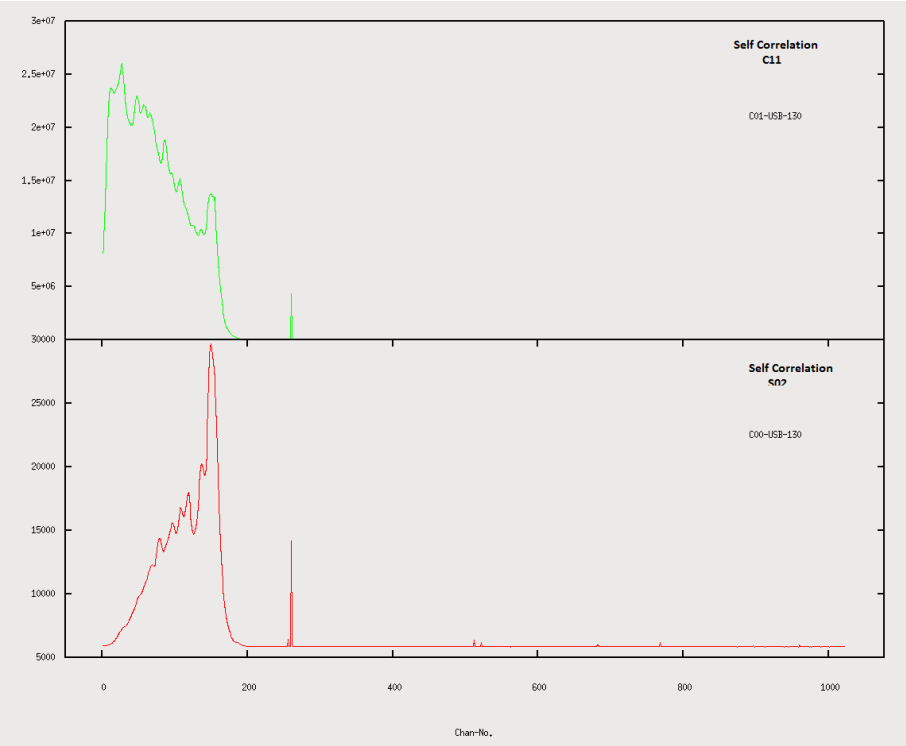

Figure 7-16 Self Correlation; DF 1, LO 0MHz, Taps 15

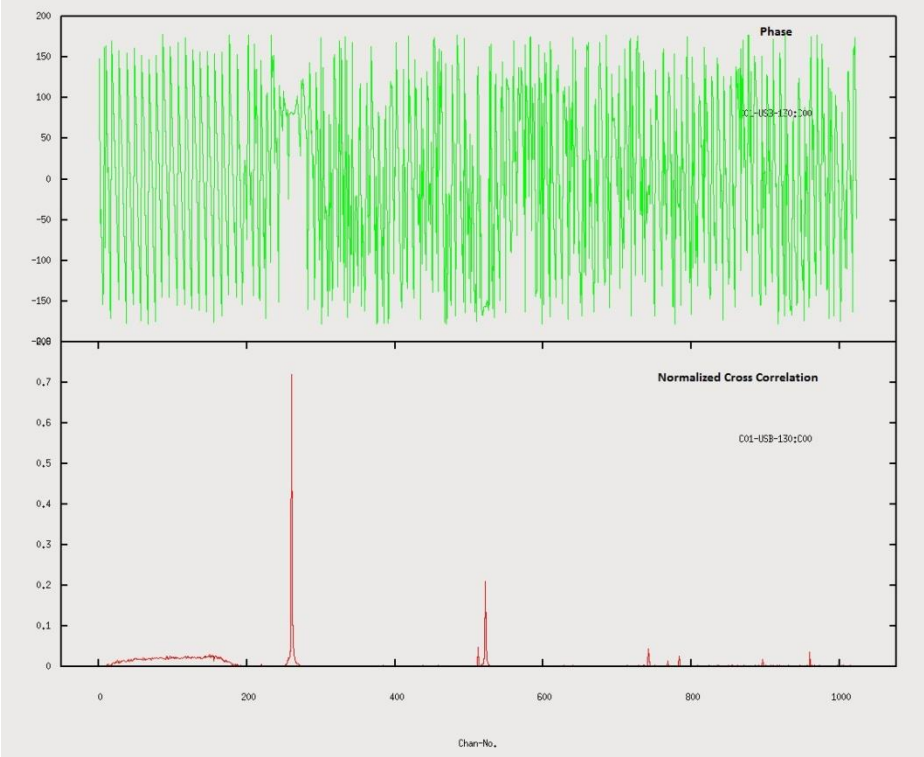

Figure 7-17 Cross Correlation; DF 1, LO 0MHz,Taps 15

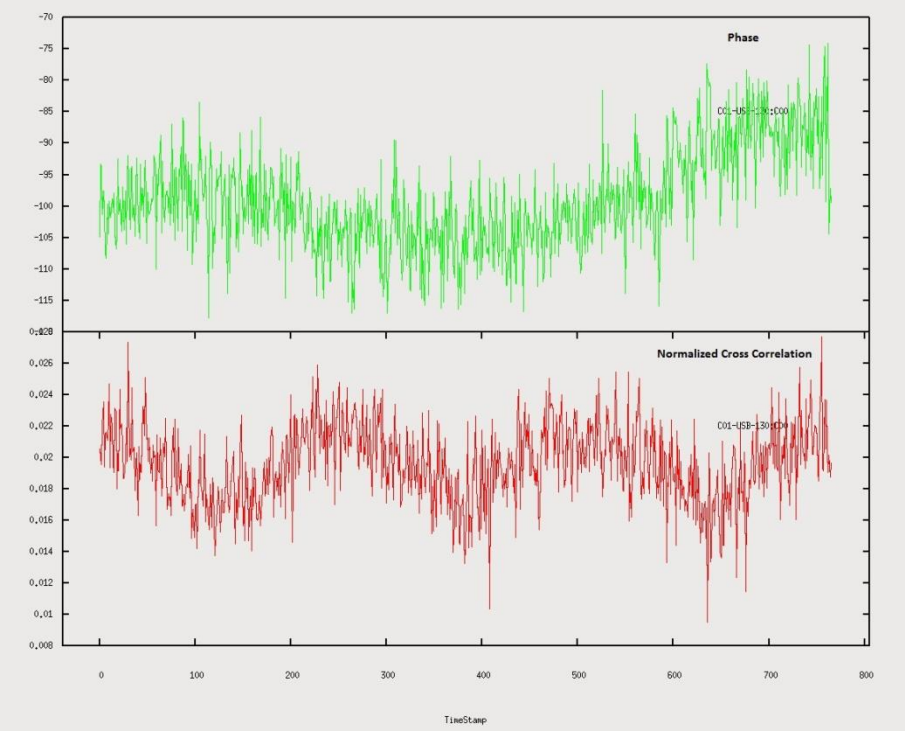

Figure 7-18 Timestamp Cross Correlation; Channel 75, DF 1, LO 0MHz, Taps 15

Discussion:

- The self-correlation of antennas C11 and S02 are plotted in CH01 and CH00 respectively as seen in Figure 7-16.
- No decimation or down conversion has been performed as LO was set to 0MHz and DF at 1.
- Figure 7-17 shows the normalized cross correlation output of the two antennas.
- The existing GMRT Correlator covers a maximum wideband frequency of 32MHz. Thus, the signal of interest lies within 0-32 MHz
- The cross correlation at frequency channel 75 is continuously recorded for a timestamp of approximately 5 minutes as plotted in Figure 7-18.
- The phase of both cross correlation plots are also shown in Figure 7-17 and Figure 7-18.

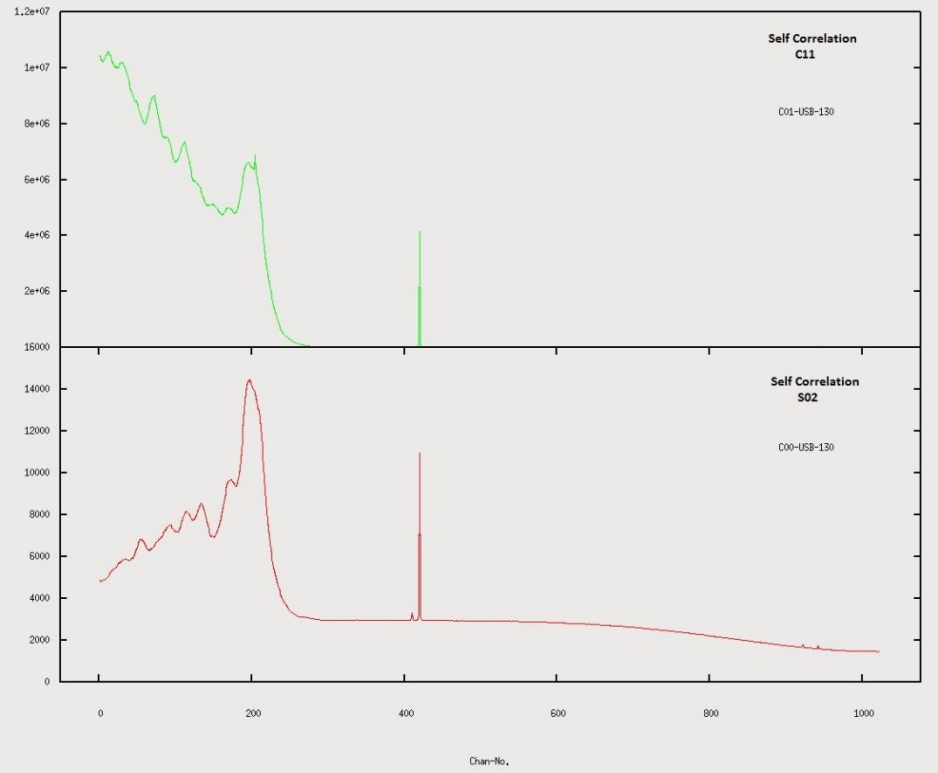

Figure 7-19 Self Correlation; DF 2, LO 10MHz, Taps 15

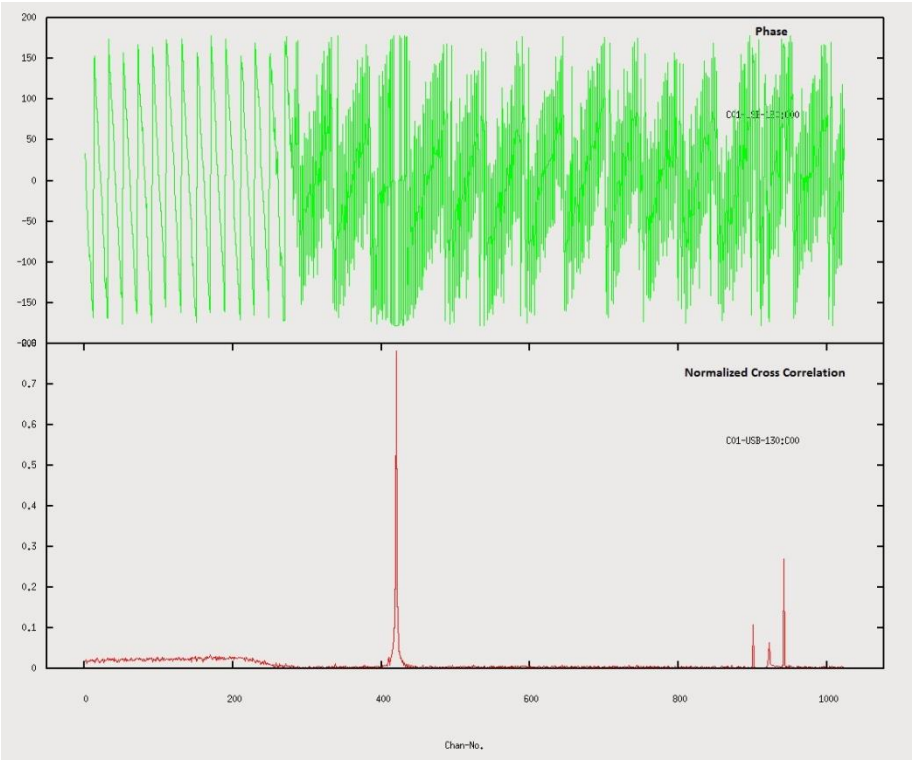

Figure 7-20 Cross Correlation; DF 2, LO 10MHz, Taps 15

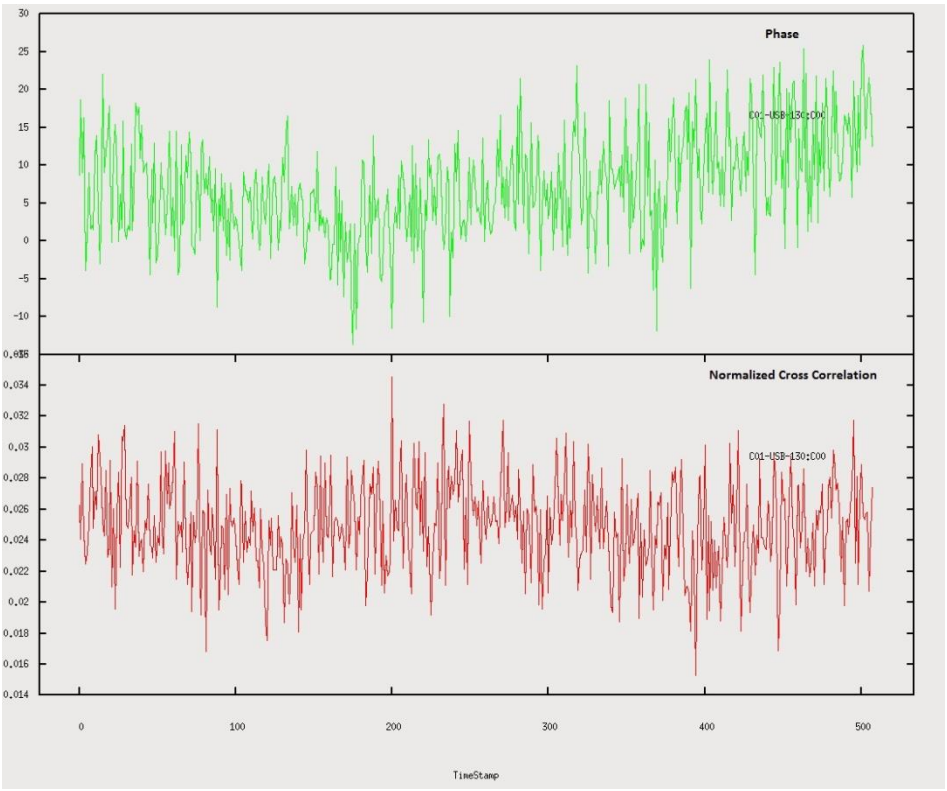

Figure 7-21 Timestamp Cross Correlation; Channel 200, DF 2, LO 10MHz, Taps 15

Discussion:

- The self-correlation plots as shown in Figure 7-19 were down converted to 10MHz and decimated such that band of 10-110 MHz is covered for all 1024 frequency channels.
- The similar down conversion and decimation operations were also performed for cross correlation and outputs were plotted as shown in Figure 7-20 and Figure 7-21.
- The above three figures thus confirm the successful working of DDC block integrated with the GMRT Correlator program tested for the real time data.

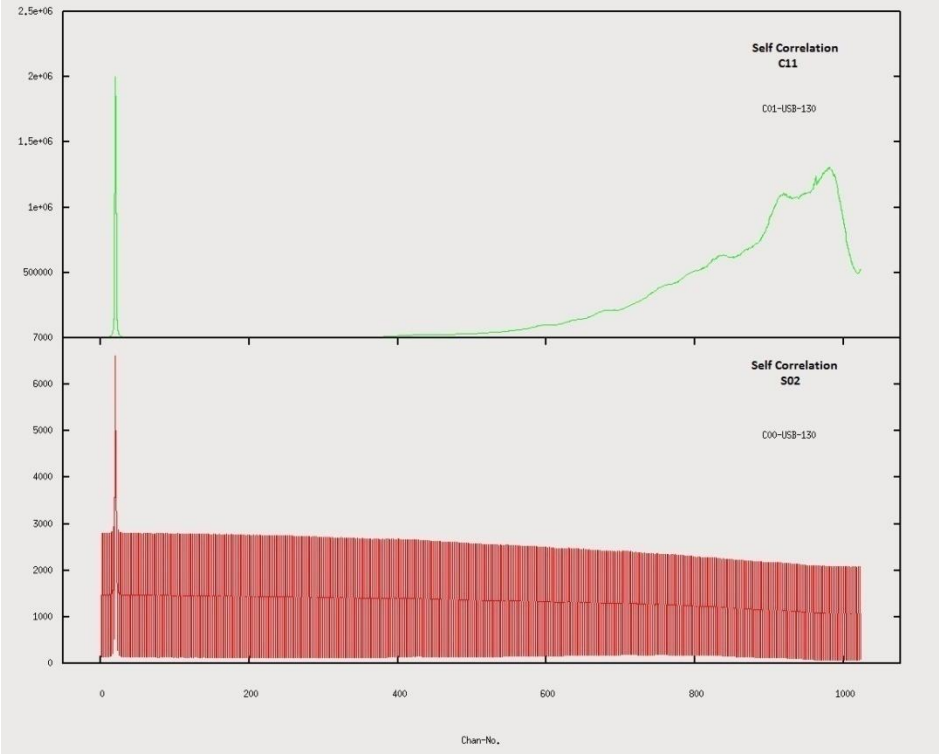

Figure 7-22 Self Correlation; DF 4, LO 50MHz, Taps 15

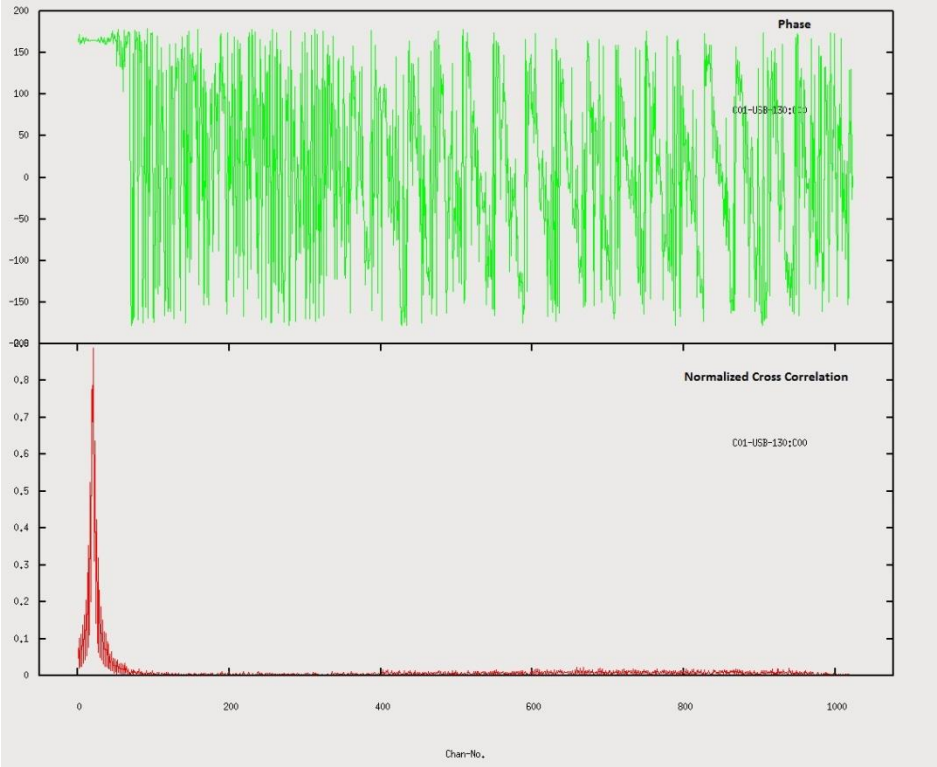

Figure 7-23 Cross Correlation; DF 4, LO 50MHz, Taps 15

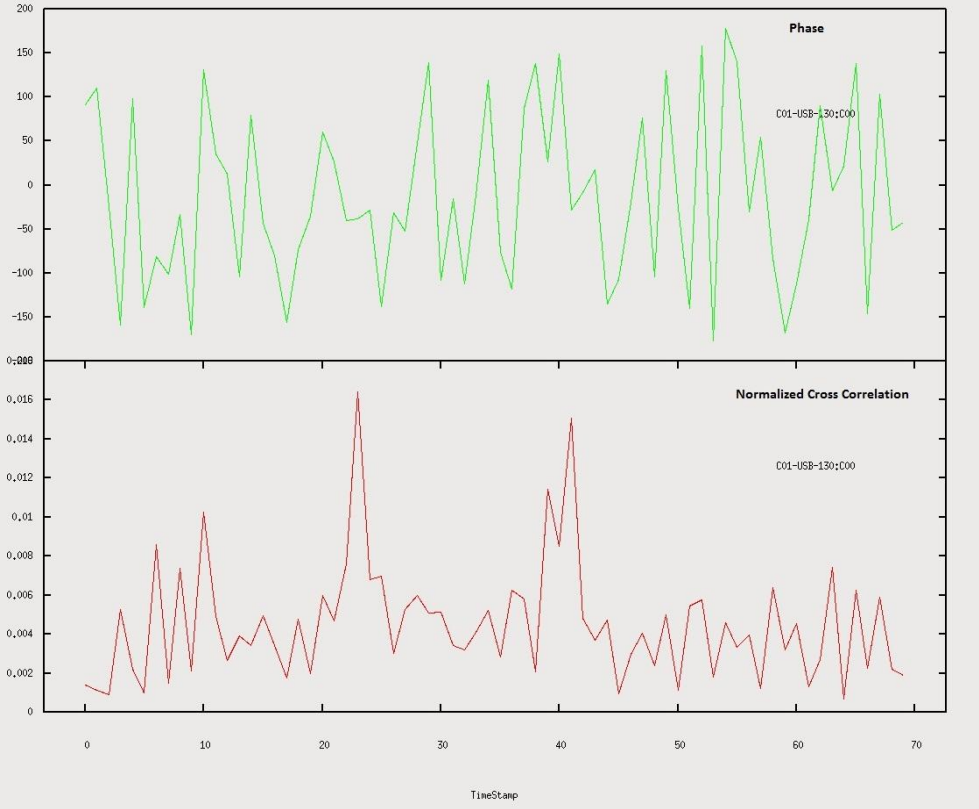

Figure 7-24 Timestamp Cross Correlation; Channel 200, DF 4, LO 50MHz, Taps 15

Discussion:

- The signal lies within 0-32 MHz. Thus, down conversion to 50MHz will lead to complete loss of signal of interest.
- The plots in Figure 7-22, Figure 7-23 and Figure 7-24 does not perform either self-correlation or cross correlation for our desired signal.
- Use of DDC with LO set at frequency greater than 32MHz will completely remove the signal from observed radio source.

8. TIMING ANALYSIS

The DDC block design code (Refer Appendix) was run in GPU with the usage of shared memory and without shared memory. Time taken to perform Convolution operation was recorded for both the cases. The convolution was also performed in CPU and the time taken was recorded for comparison purpose. The same program was run varying different parameters namely, number of filter taps, sampling frequency, decimation factor, FFT Size, integration factor, local oscillator frequency and number of data loops. Table 8.1 shows all the recorded timings. Improvement factor graphs were plotted.

S.No	Taps	Samp Freq	Decimation	FFT Size	Integrat	LO Freq	Loops	Convolution Time without using GPU shared memory (ms)	Convolution Time using GPU shared memory (ms)	Convolution Time using CPU (ms)	Improvement (b/w shared and not-shared)	Improvement (b/w CPU and GPU)
1	21	400	10	2048	50	30	20	2.2742	1.8241	354.302	1.246751823	194.2338688
2	21	400	10	2048	100	30	20	4.2702	3.451	796.0653	1.237380469	230.6767024
3	21	400	10	2048	150	30	20	6.2061	5.0563	1057.7302	1.227399482	209.1905544
4	51	400	10	2048	50	30	20	4.9211	3.6407	839.6079	1.351690609	230.6171615
5	51	400	10	2048	100	30	20	9.6383	7.1126	1681.8339	1.355102213	236.4583837
6	51	400	10	2048	150	30	20	13.9262	10.4416	2519.7189	1.333722801	241.3154019
7	51	400	10	2048	50	30	1000	247.5364	182.3027	42044.0156	1.357831782	230.6274981
8	51	400	10	2048	100	30	1000	477.4268	355.7923	92583.7109	1.3418694	260.2184221
9	51	400	10	2048	150	30	1000	698.0832	522.1184	126094.7421	1.337020875	241.5060302
10	51	400	10	2048	50	30	2500	616.3806	455.4644	112189.8671	1.353301378	246.319728
11	51	400	10	2048	100	30	2500	1194.1149	889.1081	210026.4375	1.34304805	236.221487
12	51	400	10	2048	150	30	2500	1745.8126	1306.1876	314858.2812	1.336571102	241.0513476
13	51	400	10	2048	50	30	4095	1,008.92	746.5337	172149.7343	1.351466786	230.598745
14	51	400	10	2048	100	30	4095	1955.2242	1457.1501	344032.2812	1.34181386	236.0994116
15	51	400	10	2048	150	30	4095	2858.2263	2138.8462	516370.8125	1.336340266	241.4249386

Table 8-1 Contains the Convolution Time for GPU with and without using GPU shared memory and Convolution time for CPU by varying various parameters

8.1 Convolution Timing analysis:

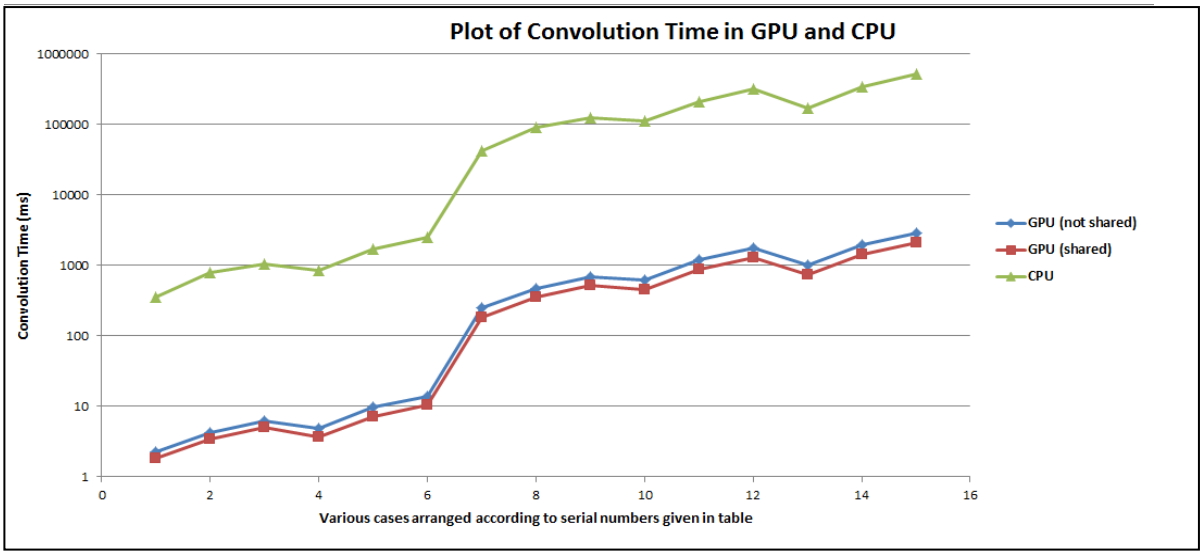

Figure 8-1 Convolution Time for GPU with and without using GPU shared memory and Convolution time for CPU

8.2 Improvement Factor between Convolution timings of GPU and CPU:

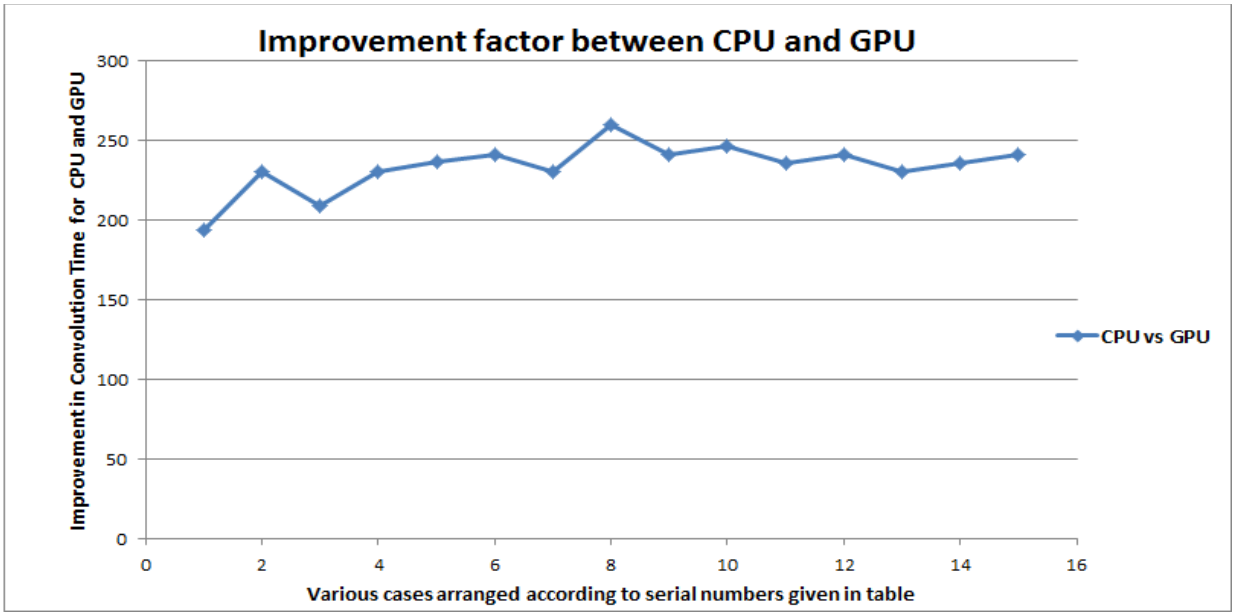

Figure 8-2 Improvement factor between GPU convolution timings and CPU convolution timings

8.3 Improvement Factor between Convolution timings of GPU when using GPU shared memory and without using GPU shared memory:

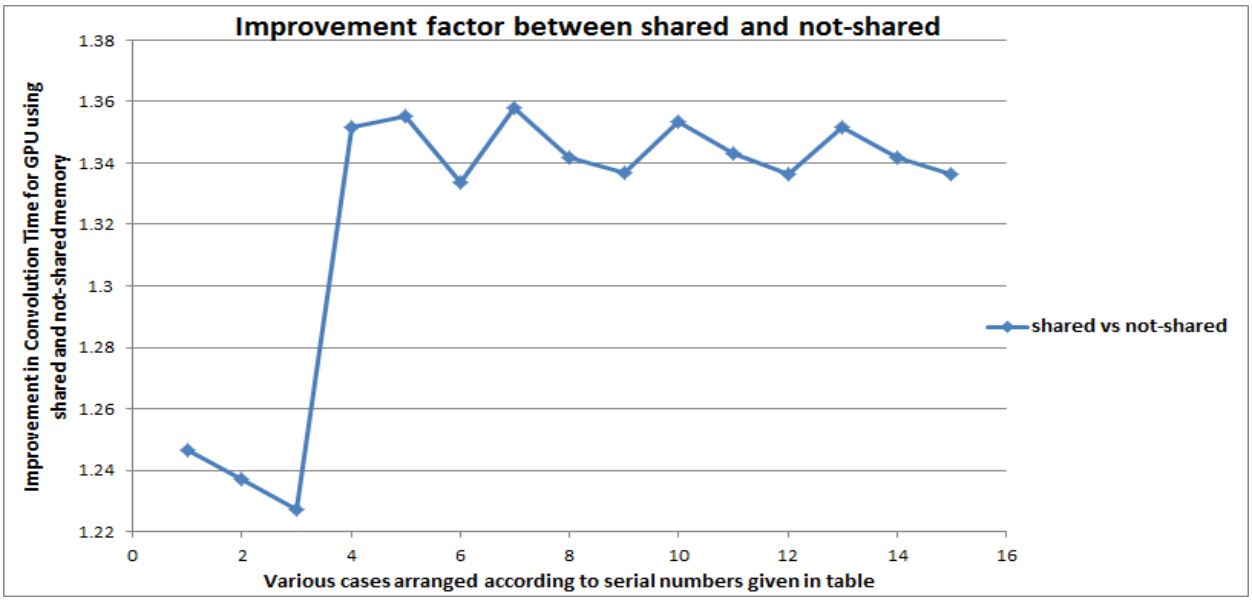

Figure 8-3 Improvement factor between GPU timings while using shared GPU memory and while not using GPU shared memory for various cases recorded in table

Discussion:

- In case of GPU more optimized results were obtained with respect to timings when GPU shared memory is used. The shared memory of GPU stores the input values, thus reducing the data transfer latency between CPU and GPU.
- From Fig-8.1 we infer that the time taken for convolution operation is least for GPU with usage of its shared memory. Whereas the convolution time for GPU without shared memory usage is slightly higher. And the improvement factor between them is plotted in Fig-8.3. Improvement factor is approximately 1.32.
- From Fig-8.1 we can also infer that the time taken for convolution using CPU is much larger as compared to GPU. The improvement factor is approximately 233.77 and it is plotted in Fig-8.2.

9. CONCLUSION

The Digital Down Converter was designed in Graphic Processing Unit and implemented successfully. The DDC block was integrated with the existing GMRT Correlator program and successfully tested with continuous wave and channel noise input signals.

The design architecture of GPU was analyzed and compared with that of CPU. CUDA-C, a programming tool for GPU was studied and practiced for small and simple programs. The memory architecture and thread management scheme were emphasized to optimize the computations involved in DDC design. The underlying principle behind DDC was understood and all operations involved in DDC were designed as separate blocks. These blocks were separately tested and then integrated together to perform DDC operation.

The existing GMRT correlator program was studied to understand the involved data flow. The DDC was integrated to work along with GMRT Correlator. The outputs of GMRT Correlator integrated with DDC were visualized using local GMRT software called 'TAX' and desired down converted and decimated results were obtained and verified.

The time required to perform the convolution operation within DDC block was analyzed for both CPU and GPU. The convolution time taken by GPU was found to be approximately 233.77 times lesser than CPU. The timing analysis was also done for the DDC block with and without using GPU shared memory. The convolution timings were improved approximately 1.32 times by usage of GPU shared memory.

The DDC as part of GMRT Correlator upgrade will result into ease in narrowband observations and reduction in data rate. The reduced data rate will lead to significant decrease in the timings involved in computations.

The sky test for the GMRT Correlator Program integrated with DDC block was done for the real time data from antennas C11 and S02. The DDC operation along with self-correlation and cross correlation was verified.

10.FUTURE SCOPE AND RECOMMENDATIONS

The DDC block produces some unexpected ambiguities in amplitude of output waveform. The output wave form is acquiring gain when LO frequency is given any non-zero value and decimation is performed. This problem might have been caused due to some scaling which would have occurred during the course of programming the DDC block, but the code was cross checked and no such scaling operation was found. So, the increase in amplitude still remains unexplained which can be considered as future scope of improvement in DDC block.

The narrowband observations often demand the visibility of dual bands on either side of LO frequency. Thus, to facilitate DDC block with this feature the code can be modified accordingly and a switching flag which can simply switch between either of the side bands can be introduced.

The windowing function used for implementing LPF filter is hamming window, but the code needs to be made more versatile so that various windowing functions could also be employed.

The decimation factor should be any number of base two (i.e. 1, 2, 4, 8, 16...) because the size of data processed while performing FFT are also base two numerals. The decimation factor decides the LPF cutoff frequency. Thus, limit on decimation factor also limits the usage of different LPF cutoff frequencies. In future some better algorithm for decimation should be thought of in order to remove the limitations on decimation factor.

9. REFERENCES

1. Christopher John Harris, "*A thesis on A Parallel Model for the Heterogeneous Computation of Radio Astronomy Signal Correlation*",
2. Jayaram N. Chengalur, Yashwant Gupta, K. S. Dwarakanath, "*Low Frequency Radio Astronomy*"
3. NVIDIA documents for CUDA Architecture
4. Jason Sanders, Edward Kandrot, "*CUDA by Example*"
5. "*The Theory of Digital Down Conversion*", Hunt Engineering
6. Richard G. Lyons, *"Understanding Digital Signal Processing*"
7. William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Wetterling "*Numerical recipes in C, The art of scientific computing*"
8. Radio Astronomy "*http://en.wikipedia.org/wiki/Radio_astronomy*"
9. 3C48 "*http://en.wikipedia.org/wiki/3C_48*"
10. ROACH-CASPER "*https://casper.berkeley.edu/wiki/ROACH*"

APPENDIX – I

(Code for parsing command line inputs)

```
// taking command line inputs and assigning them....

if(argc != 9)
    {
fprintf(stderr, "USAGE: <windowLength><sampFreq><DF in
int><FFT_Size><int><sin/cosFreq><file_name><loops>\n");
exit(1);
    }

intwindowLength = atoi(argv[1]);
floatsampFreq = atof(argv[2]);
int DF = atoi(argv[3]);
floattransFreq = (sampFreq/(2 * DF));
int NX = atoi(argv[4]);
int BATCH = atoi(argv[5]);
float f  = atof(argv[6]);
float s = atof(argv[8]);
int b = ((NX * BATCH) + windowLength);
int c = (NX * BATCH);


    FILE *file = fopen((argv[7]), "r");
if (file == NULL)
          {
      fprintf(stderr, "Can't open input file ! \n");
      exit(1);
          }
```

APPENDIX – II

(Code for allocating and de-allocating memories on CPU)

```
   // on CPU ....
signed char *input = (signed char *)malloc(b * sizeof(signed char));
      if (input == NULL)
      {
      fprintf(stderr,"Could not allocate memory to input \n");
            exit(1);
      }

    // memory allocation for saving the input array in float .....
      float *input_host = (float *)malloc(b * sizeof(float));
if (input_host == NULL)
    {
      fprintf(stderr,"Could not allocate memory to input_host \n");
            exit(1);
      }

    //allocating memory on host to store the real part of input FFT
float *real_ip_fft = (float *)malloc((NX/2 +1) * sizeof(float));
if (real_ip_fft == NULL)
    {
      fprintf(stderr,"Could not allocate memory to real_ip_fft \n");
      exit (1);
    }

    //allocating memory on host to store the img part of input FFT
float *img_ip_fft = (float *)malloc((NX/2 +1) * sizeof(float));
if (img_ip_fft == NULL)
    {
      fprintf(stderr,"Could not allocate memory to img_ip_fft \n");
      exit (1);
    }

    //allocating memory on host to store the mag of input FFT
float *mag_ip_fft = (float *)malloc((NX/2+1) * sizeof(float));
if (mag_ip_fft == NULL) {
      fprintf(stderr,"Could not allocate memory to mag_ip_fft \n");
      exit (1);
    }

    //allocating memory on host to store the phase
float *phase_ip_fft = (float *)malloc((NX/2+1) * sizeof(float));
if (phase_ip_fft == NULL) {
      fprintf(stderr,"Could not allocate memory to phase_ip_fft \n");
      exit (1);
    }


float *cosine = (float *)malloc(b * sizeof(float));
if (cosine == NULL)
    {
```

```
fprintf(stderr,"Could not allocate memory to cosine \n");
      exit(1);
    }

float *sine = (float *)malloc(b * sizeof(float));
if (sine == NULL)
    {
      fprintf(stderr,"Could not allocate memory to sine \n");
      exit(1);
    }

float *lpf = (float *)malloc(windowLength * sizeof(float));
if (lpf == NULL)
    {
      fprintf(stderr,"Could not allocate memory to lpf \n");
      exit (1);
    }

float *lpf_hamming = (float *)malloc(windowLength * sizeof(float));
if (lpf_hamming == NULL)
    {
      fprintf(stderr,"Could not allocate memory to lpf_hamming \n");
      exit (1);
    }


float *sine_conv = (float *)malloc(c * sizeof(float));
if (sine_conv == NULL)
    {
      fprintf(stderr,"Could not allocate memory to sine_conv \n");
      exit (1);
    }

float *cosine_conv = (float *)malloc(c * sizeof(float));
if (cosine_conv == NULL)
    {
      fprintf(stderr,"Could not allocate memory to cosine_conv \n");
      exit (1);
    }

float *dec_sine_conv = (float *)malloc((c/DF) * sizeof(float));
if (dec_sine_conv == NULL)
    {
      fprintf(stderr,"Could not allocate memory to dec_sine_conv \n");
      exit (1);
    }

float *dec_cosine_conv = (float *)malloc((c/DF) * sizeof(float));
if (dec_cosine_conv == NULL)
    {
      fprintf(stderr,"Could not allocate memory to dec_cosine_conv \n");
      exit (1);
    }
```

```
    //allocating memory on host to store the real
float *real = (float *)malloc((NX/2) * sizeof(float));
if (real == NULL)
    {
      fprintf(stderr,"Could not allocate memory to real \n");
      exit (1);
    }

    //allocating memory on host to store the img
float *img = (float *)malloc((NX/2) * sizeof(float));
if (img == NULL)
    {
      fprintf(stderr,"Could not allocate memory to img \n");
      exit (1);
    }

    //allocating memory on host to store the mag
float *mag = (float *)malloc((NX/2) * sizeof(float));
if (mag == NULL) {
      fprintf(stderr,"Could not allocate memory to mag \n");
      exit (1);
    }

    //allocating memory on host to store the phase
float *phase = (float *)malloc((NX/2) * sizeof(float));
if (phase == NULL) {
      fprintf(stderr,"Could not allocate memory to phase \n");
      exit (1);
    }


// freeing all memory on host (CPU) ....
free(input);
free(input_host);
free(sine);
free(cosine);
free(lpf);
free(lpf_hamming);
free(real);
free(img);
free(mag);
free(phase);
free(result);
free(ip_fft_result);
free(real_ip_fft);
free(img_ip_fft);
free(mag_ip_fft);
free(phase_ip_fft);
free(sine_conv);
free(cosine_conv);
free(dec_cosine_conv);
free(dec_sine_conv)
```

APPENDIX – III

(Code for allocating and de-allocating memories on GPU)

```
// on GPU ....
float *dev_input_host;
    HANDLE_ERROR(cudaMalloc((float **)&dev_input_host, b *
sizeof(float)));

float *dev_sine;
float *dev_cosine;
float *dev_op_cosine;
float *dev_op_sine;

    HANDLE_ERROR(cudaMalloc((float **)&dev_sine, b * sizeof(float)));
    HANDLE_ERROR(cudaMalloc((float **)&dev_cosine, b * sizeof(float)));
    HANDLE_ERROR(cudaMalloc((float **)&dev_op_sine, b * sizeof(float)));
    HANDLE_ERROR(cudaMalloc((float **)&dev_op_cosine, b * sizeof(float)));

float *dev_op_cosine_conv;
float *dev_op_sine_conv;
float *dev_lpf_hamming;

    HANDLE_ERROR(cudaMalloc((float **)&dev_op_sine_conv, b *
sizeof(float)));
    HANDLE_ERROR(cudaMalloc((float **)&dev_op_cosine_conv, b *
sizeof(float)));
    HANDLE_ERROR(cudaMalloc((float **)&dev_lpf_hamming, windowLength *
sizeof(float)));

float *dev_dec_sine_conv;
float *dev_dec_cosine_conv;

    HANDLE_ERROR(cudaMalloc((float **)&dev_dec_sine_conv, (c/DF) *
sizeof(float)));
    HANDLE_ERROR(cudaMalloc((float **)&dev_dec_cosine_conv, (c/DF) *
sizeof(float)));

    Complex *dev_comp;
    HANDLE_ERROR(cudaMalloc ((Complex **)&dev_comp, (c/DF) *
sizeof(Complex)));

// freeing all memory on device (GPU) ....
cudaFree(dev_input_host);
cudaFree(dev_cosine);
cudaFree(dev_sine);
cudaFree(dev_op_sine);
cudaFree(dev_op_cosine);
cudaFree(dev_op_sine_conv);
cudaFree(dev_op_cosine_conv);
cudaFree(dev_comp);
cudaFree(dev_lpf_hamming);
cudaFree(dev_dec_sine_conv);
cudaFree(dev_dec_cosine_conv);
```

APPENDIX – IV

(Code for creating filter coefficients and generating sine and cosine waves)

```
// Createsinc function for filter - Low and High pass filters

enumfilterType {LOW_PASS, HIGH_PASS};

float *create1TransSinc(intwindowLength, float transFreq, float sampFreq,
enumfilterType type)
{
      int n;

      // Allocate memory for the window
      float *window = (float *) malloc(windowLength * sizeof(float));
      if (window == NULL) {
            fprintf(stderr, "create1TransSinc: Could not allocate memory
for window\n");
            return NULL;
      }

      if (type != LOW_PASS && type != HIGH_PASS) {
            fprintf(stderr, "create1TransSinc: Bad filter type, should be
either LOW_PASS of HIGH_PASS\n");
            return NULL;
      }

      // Calculate the normalisedtransistion frequency. As transFreq
should be
      // less than or equal to sampFreq / 2, ft should be less than 0.5
      floatft = transFreq / sampFreq;

      float m_2 = 0.5 * (windowLength-1);
      inthalfLength = windowLength / 2;

      // Set centre tap, if present
      // This avoids a divide by zero
      if (2*halfLength != windowLength)
      {
            floatval = 2.0 * ft;

            // If we want a high pass filter, subtract sinc function from
a dirac pulse
            if (type == HIGH_PASS) val = 1.0 - val;

            window[halfLength] = val;
      }
      else if (type == HIGH_PASS)
      {
            fprintf(stderr, "create1TransSinc: For high pass filter,
window length must be odd\n");
```

```
            return NULL;
      }

      // This has the effect of inverting all weight values
      if (type == HIGH_PASS) ft = -ft;

      // Calculate taps
      // Due to symmetry, only need to calculate half the window
      for (n=0 ; n<halfLength ; n++) {
            floatval = sin(2.0 * M_PI * ft * (n-m_2)) / (M_PI * (n-m_2));

            window[n] = val;
            window[windowLength-n-1] = val;
      }

      return window;
}



// create window function ....

enumwindowType {HAMMING};

float *createWindow(float *in, float *out, intwindowLength, enumwindowType
type)
{
        // If output buffer has not been allocated, allocate memory now
if (out == NULL)
        {
out = (float *) malloc(windowLength * sizeof(float));
if (out == NULL)
                {
fprintf(stderr, "Could not allocate memory for window\n");
return NULL;
                }
        }

int n;
int m = windowLength - 1;
inthalfLength = windowLength / 2;

        // Calculate taps
        // Due to symmetry, only need to calculate half the window
switch (type)
        {
case HAMMING:
for (n=0 ; n<=halfLength ; n++)
                {
floatval = 0.54 - 0.46 * cos(2.0 * M_PI * n / m);
out[n] = val;
out[windowLength-n-1] = val;
                }
```

```
break;
          }

         // If input has been given, multiply with out
if (in != NULL)
         {
for (n = 0 ; n <windowLength ; n++)
                  {
out[n] *= in[n];
                  }
         }

return out;
}

// getting filter coefficients for a low pass filter using a sinc function
and windowing using hamming window ....
lpf = create1TransSinc(windowLength, transFreq, sampFreq, LOW_PASS);
lpf_hamming = createWindow(lpf, NULL, windowLength, HAMMING);



// generating sine and cosine wave ....
float a = 1, wl;
float w = 2 * M_PI * f;

for(int n = 0; n < b; n++)
    {
      wl = w * n;
      cosine[n] = a * cosf(wl/sampFreq);
      sine[n] = a * sinf(wl/sampFreq);
    }
```

APPENDIX – V

(Code for GPU kernels performing mixer operation and creating complex)

```
// GPU kernel function for multiplication of input signal with cosine and
sine function ....

__global__ void multi_sine_cosine(float *dev_sine, float *dev_cosine,
float *dev_op_sine, float *dev_op_cosine,
float *dev_input_host)
{
      intidx = threadIdx.x + blockIdx.x * blockDim.x;

float temp1, temp2;

      temp1 =  dev_input_host[idx] * dev_cosine[idx];
      temp2 = dev_input_host[idx] * dev_sine[idx];

dev_op_cosine[idx] = temp1;
dev_op_sine[idx] = temp2;
}



// GPU kernel function for assigning LPF values as real(sine
multiplication) and complex(cosine multiplication) ....

__global__ void comp(cufftComplex *dev_comp, float *dev_op_sine_conv,
float *dev_op_cosine_conv, int c)
{
int i;


intidx = threadIdx.x + blockIdx.x * blockDim.x;
int stride = blockDim.x * gridDim.x;

for(i = idx; i < c; i+=stride)
        {
dev_comp[i].x = dev_op_cosine_conv[i];
dev_comp[i].y = -1*dev_op_sine_conv[i];
        }
}
```

APPENDIX –VI

(GPU kernel for Convolution without using GPU shared memory)

```
// GPU kernel for convoluting sine and cosine multiplication data with
filter coefficients with hamming window ....

__global__ void conv(float *dev_op_sine, float *dev_op_cosine, float
*dev_op_sine_conv, float *dev_op_cosine_conv, float *dev_lpf_hamming, int
b, intwindowLength)
{
      inti,k,l;

      intidx = threadIdx.x + blockIdx.x * blockDim.x;
int stride = blockDim.x * gridDim.x;
float temp1, temp2;

for(i = idx; i < b; i+=stride)
      {
            temp1 = 0;
            temp2 = 0;

            for(k = 0; k <windowLength; k++)
            {
                  l = windowLength - k;

                  temp1 +=  dev_op_sine[i+l] * dev_lpf_hamming[k];
                        temp2 +=  dev_op_cosine[i+l] * dev_lpf_hamming[k];

            }

dev_op_sine_conv[i] = temp1;
dev_op_cosine_conv[i] = temp2;

      }
}
```

APPENDIX –VII

(GPU kernel for Convolution using GPU shared memory)

```
// GPU kernel for convoluting sine and cosine multiplication data with
filter coefficients with hamming window ....

#define BLOCK_SIZE 512
#define WINDOW 500

__global__ void conv(float *dev_op_sine, float *dev_op_cosine, float
*dev_op_sine_conv, float *dev_op_cosine_conv, float *dev_lpf_hamming, int
b, intwindowLength)
{
      inti,l;

      __shared__ float dats[BLOCK_SIZE];
      __shared__ float datc[BLOCK_SIZE];
__shared__ float coeff[WINDOW];

intthreadid = threadIdx.x;
      intidx = threadIdx.x + blockIdx.x * 256;

float temp1, temp2;

      dats[threadid] = dev_op_sine[idx];
      datc[threadid] = dev_op_cosine[idx];


if(threadid<windowLength)
{
coeff[threadid] = dev_lpf_hamming[threadid];
      }
         __syncthreads();


if(threadid< 256)
      {
        temp1 = 0;
        temp2 = 0;
for( i = 0; i <windowLength ; ++i)
           {
             l = windowLength - i;

             temp1 += dats[threadid+l] * coeff[i];

             temp2 += datc[threadid+l] * coeff[i];
            }

dev_op_sine_conv[idx] = temp1;

dev_op_cosine_conv[idx] = temp2;
      }
}
```

APPENDIX – VIII

(Code showing usage of all kernel functions and calculating time)

```
// defining grid and block dimensions

dim3 block(512);
dim3 grid(((NX/512)+1)*(BATCH));

// doing multiplication with sine and cosine wave ....

cudaEvent_t start, stop; // defining the event variables

HANDLE_ERROR(cudaEventCreate(&start)); // creating events
HANDLE_ERROR(cudaEventCreate(&stop));

HANDLE_ERROR(cudaEventRecord( start, 0 )); // starting event

multi_sine_cosine<<<grid,block>>>(dev_sine, dev_cosine, dev_op_sine,
dev_op_cosine, dev_input_host);

HANDLE_ERROR(cudaEventRecord( stop, 0 )); // stopping the event
HANDLE_ERROR(cudaEventSynchronize( stop)); // synchronizing the timings

floatelapsedtime; // defining elapsed time
HANDLE_ERROR(cudaEventElapsedTime( &elapsedtime, start, stop )); //
elapsed time

x += elapsedtime;

HANDLE_ERROR(cudaEventDestroy( start )); // destroying events start and
stop
HANDLE_ERROR(cudaEventDestroy( stop ));



// doing LPF by using FIR filter using hamming window ....

HANDLE_ERROR(cudaMemcpy(dev_lpf_hamming, lpf_hamming, windowLength *
sizeof(float), cudaMemcpyHostToDevice));

cudaEvent_tstart_conv, stop_conv; // defining the event variables

HANDLE_ERROR(cudaEventCreate(&start_conv)); // creating events
HANDLE_ERROR(cudaEventCreate(&stop_conv));

HANDLE_ERROR(cudaEventRecord( start_conv, 0 )); // starting event

conv<<<grid,block>>>(dev_op_sine, dev_op_cosine, dev_op_sine_conv,
dev_op_cosine_conv, dev_lpf_hamming, b, windowLength);
```

```
HANDLE_ERROR(cudaEventRecord( stop_conv, 0 )); // stopping the event
HANDLE_ERROR(cudaEventSynchronize( stop_conv)); // synchronizing the
timings

floatelapsedtime_conv; // defining elapsed time
HANDLE_ERROR(cudaEventElapsedTime( &elapsedtime_conv, start_conv,
stop_conv )); // elapsed time

y += elapsedtime_conv;

HANDLE_ERROR(cudaEventDestroy( start_conv )); // destroying events start
and stop
HANDLE_ERROR(cudaEventDestroy( stop_conv ));



// make complex values ....

cudaEvent_tstart_comp, stop_comp; // defining the event variables

HANDLE_ERROR(cudaEventCreate(&start_comp)); // creating events
HANDLE_ERROR(cudaEventCreate(&stop_comp));

HANDLE_ERROR(cudaEventRecord( start_comp, 0 )); // starting event

comp<<<grid,block>>>(dev_comp, dev_dec_sine_conv, dev_dec_cosine_conv,
(c/DF));

HANDLE_ERROR(cudaEventRecord( stop_comp, 0 )); // stopping the event
HANDLE_ERROR(cudaEventSynchronize( stop_comp)); // synchronizing the
timings

floatelapsedtime_comp; // defining elapsed time
HANDLE_ERROR(cudaEventElapsedTime( &elapsedtime_comp, start_comp,
stop_comp )); // elapsed time

z += elapsedtime_comp;

HANDLE_ERROR(cudaEventDestroy( start_comp )); // destroying events start
and stop
HANDLE_ERROR(cudaEventDestroy( stop_comp ));
```

APPENDIX – IX

(Code for performing FFT in GPU using CUFFT library)

```
// plan and memory for input FFT....
cufftHandleplanip;
cufftComplex *ip_fft;
cudaMalloc((void**)&ip_fft, sizeof(cufftComplex)*(NX/2+1)*BATCH);

    //Create a 1D FFT plan
cufftPlan1d(&planip, NX, CUFFT_R2C, BATCH);

    //allocating memory on host to store the copy of fft values calculated
in device
    Complex *ip_fft_result = (Complex
*)malloc((BATCH)*(NX/2+1)*sizeof(Complex));


    // plan and memory for output FFT ....
cufftHandle plan;
cufftComplex *data;
cudaMalloc((void**)&data, sizeof(cufftComplex)*(NX+1)*(BATCH/DF));

    //Create a 1D FFT plan
cufftPlan1d(&plan, NX, CUFFT_C2C, (BATCH/DF));

    //allocating memory on host to store the copy of fft values calculated
in device
    Complex *result = (Complex *)malloc((BATCH/DF)*(NX)*sizeof(Complex));



// calculating R2C FFT in GPU using CUDA-C for plotting input FFT
cufftExecR2C(planip, (cufftReal*)dev_input_host, ip_fft);

// calculating C2C FFT in GPU using CUDA-C for plotting output FFT
cufftExecC2C(plan, (cufftComplex*)dev_comp, data,CUFFT_FORWARD);




//Destroy the CUFFT plan
cufftDestroy(plan);
cudaFree(data);
cufftDestroy(planip);
cudaFree(ip_fft);
```