IMPLEMENTATION OF MAD-BASED RFI FILTERING ALGORITHM ON CPU AND GPU

Student Project
By
Shubham Balte

Electronics And Communication Engineering,
Sardar Vallabhbhai National Institute of Technology,
Surat

Under the Guidance of
Mr. Kaushal Buch

GIANT METREWAVE RADIO TELESCOPE

NATIONAL CENTER FOR RADIO ASTROPHYSICS

TATA INSTITUTE OF FUNDAMENTAL RESEARCH

P.B. 6, Narayangaon P.O., Tal-Junnar,Dist-Pune 410504
Maharashtra

May 2022 - July 2022

ABSTRACT

This project looks into a software implementation of the MAD-based RFI filtering algorithm on the uGMRT system. The filtering is currently being done on FPGAs, where the customizability is low. Shifting the filtration on software with the GWB compute nodes will help free up space on the FPGA for more exploration and provide an easier path for alteration and experimentation with the RFI filtering algorithms.

The project explores the implementation being computed on the CPU and the GPU. It looks at the pros and cons in terms of performance, customizability, resource usage, and the integration potential in the current system.

Multiple implementations are done on both platforms while looking at different assumptions that can be considered to improve performance and their impact. The ways to implement the software system were also highlighted and tested for potential compatibility problems.

Acknowledgments

I would like to express my gratitude to Mr. Kaushal Buch for giving me this opportunity to explore my interests in the practical world. His helpfulness and passion to make me interested not just in my own work but in the surrounding work being done to make the observations at GMRT possible have been an outstanding experience. I would also like to thank the group head, Mr. Ajith Kumar, for his motivation and suggestions during the work.

I want to thank Mr. Harshavardhan Reddy for his help with setting up the GPB test machine and helping to test the code on the snode machine, and everyone who helped me understand and explained to me the various components and their inner workings that make up the GMRT antenna as a whole.

I would also like to thank Prof. Yashwant Gupta, Centre Director, and Prof. Jayaram Chengalur, Director, TIFR, for allowing me to work on the Upgraded GMRT project at GMRT.

I would also like to thank my parents and fellow STP students for supporting me during this project.

Shubham Balte

Contents

Abstract **1**

Acknowledgements **2**

List of Figures and Tables **4**

Chapter 01: Introduction

1.1 Overview of uGMRT **6**

1.2 RFI issues at GMRT **7**

1.3 GWB correlator and current RFI mitigation method: **8**

Chapter 02: Histogram-based algorithm

2.1 Need for the histogram method **10**

2.2 Algorithm for histogram method **10**

2.3 Comparison of different median algorithms over multiple datasets **11**

2.4 Computational estimates for different histogram methods **12**

Chapter 03: Implementation on CPU

3.1 Algorithm: **14**

3.2 Functional Testing **16**

3.3 Benchmarking of MAD filter **17**

3.4 Visualization of the CPU MAD filter **19**

3.5 Different compiler versions **20**

3.6 Benchmarking of MOM code on different compilers **21**

3.7 Timing analysis of different functions of MOM code **22**

Chapter 04: Implementation on GPU

4.1 Basic architectural differences between GPU and CPU **23**

4.2 Algorithm **24**

4.3 Benchmarking of MOM filter **25**

Chapter 05: Integrating with GWB

5.1 Structure of current GWB code **26**

5.2 Benchmarks of integration code **27**

Conclusions **29**

Future Scope **30**

References **31**

Appendix **32**

List of Figures and Tables

Figure 1.1.1 GMRT Antenna being taken down for maintenance
Figure 1.2.1 - An illustration of the RFI received from various sources
Figure 1.3.1 - Block Diagram for GWB correlator
Figure 2.2.1 - Visualization of histogram
Figure 2.3.1 - Different median algorithms
Figure 3.1.1 - CPU MOM Implementation algorithm
Figure 3.2.1 - Functional testing block diagram
Figure 3.3.1 - MAD filter time per window for different assumptions
Figure 3.3.2 - Median calculation time per window for different assumptions
Figure 3.4.1 - 3-sigma MAD Filtering in replacement mode
Figure 3.4.2 - 3-sigma MAD Filtering in threshold mode
Figure 3.6.1 - Worst case performance of MOM filter on GPU
Figure 4.1.1 - Comparison of typical CPU and GPU architecture
Figure 4.2.1 - Detailed illustration of histogram kernel function
Figure 4.2.2 - Algorithm of MOM filter on GPU
Figure 4.3.1 - MOM filtering on GPU at different threshold levels
Figure 4.3.2 - Comparison of different GPU MOM algorithms
Figure 5.1.1 - Structure of the FPGA + Compute Node
Figure 5.2.1 - Utilization of all CPU cores during online snode testing
Figure A.1 - Function to read header files

Table 2.3.1 - Comparison of different median algorithms
Table 2.4.1 - Computational estimates for different histogram methods
Table 3.7.1 - Function wise timing distribution
Table A.1 - Details of benchmarked datasets

Chapter 01: Introduction

1.1 Overview of uGMRT

Giant Metrewave Radio Telescope is an observatory which explores the metre wavelength range of the radio spectrum. It is set up by the National Centre for Radio Astrophysics (NCRA). GMRT consists of 30 fully steerable gigantic parabolic dishes of 45m in diameter which are spread over distances of up to 25 km. The site was selected because it fulfills the following important criteria such as
1) Low man-made radio noise.
2) Low wind speed.
3) It has a geographical latitude which is sufficiently north of the geomagnetic equator in order to have a reasonably quiet ionosphere and yet be able to observe a good part of the southern sky as well.

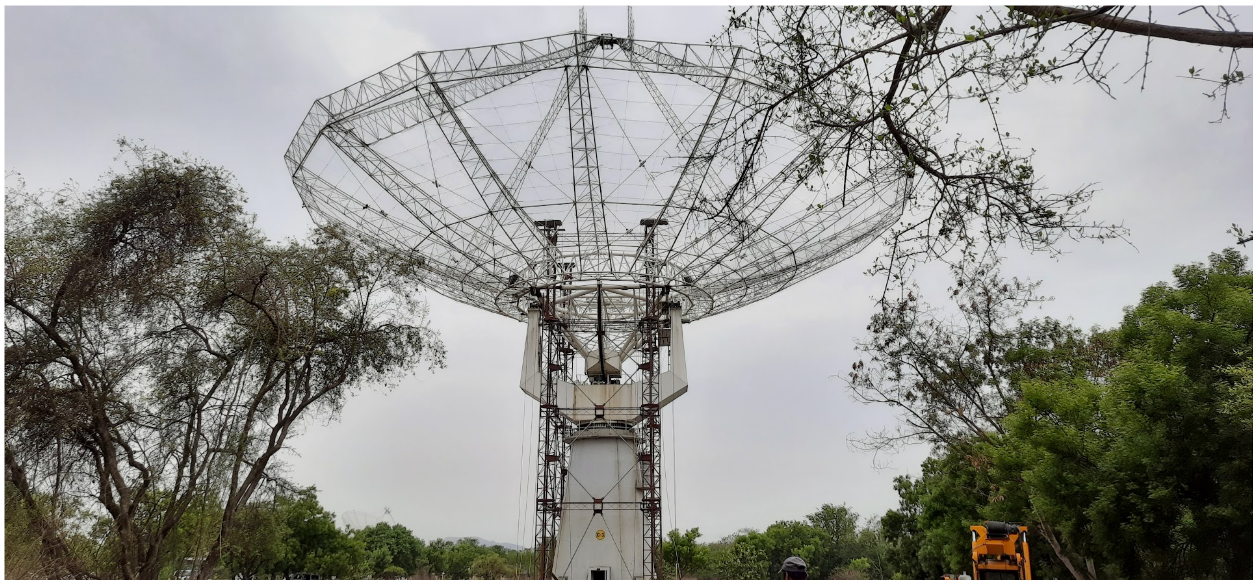

Figure 1.1.1 GMRT Antenna (C03) being taken down for maintenance

In the electromagnetic spectrum, the metre wavelength range of spectrum has been particularly chosen for study with GMRT because man-made radio interference is quite less in this part of the spectrum in India and there are many outstanding astrophysical phenomena which are best studied at metre wavelengths.

1.2 RFI issues at GMRT

Radio Frequency Interference (RFI) is the 'unwanted signal' produced due to Natural and Man-made activities. Television transmissions, RADAR, Satellite communication, Wi-Fi networking power-line radiation, spark ignition noise and radiation due to electronic devices are considered as man-made radio interference. Radio astronomical signals are very weak (typically -110 dBm at the input of the radio telescope receiver). Thus radio telescopes are very sensitive to capturing such weak signals here and are more prone to interference. RFI can damage the quality of the data severely and excision of it can cause loss of data also.

There are two methods for handling the RFI situation
1. Proactive methods - Prevention of RFI by establishing 'Radio Quiet Zones'
2. Reactive methods – Signal processing for RFI mitigation after reception of the astronomical data at different stages of the receiver chain.

Digital signal processing can help improve the data signal quality effectively by removing RFI without much loss of quality. Figure 1.2.1 illustrates the RFI and noise signals in a time series signal received at GMRT.

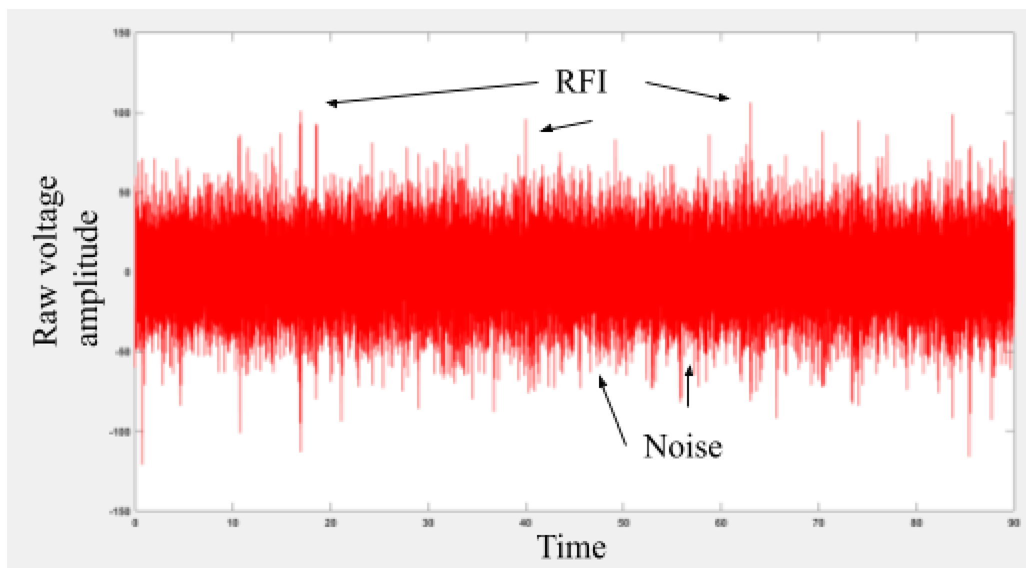

Figure 1.2.1 - An illustration of the RFI received from various sources

1.3 GWB correlator and current RFI mitigation method:

The current digital backend system of the Upgraded GMRT (uGMRT) is known as the GMRT Wideband Backend (GWB). When the time series raw voltage data is provided at the ADC of the system, an FPGA node is configured to perform a 16x16 MOM filtration on 4 antennas at once. Then this digitized and filtered data is sent to the CPU + GPU compute node for further processing and data storage.

MAD is defined as

$$\mathrm{MAD} = \mathrm{median}_j[|x - \mathrm{median}(x_i)|]$$

and for a normal distribution, the standard deviation is related to this as

$$\sigma = 1.4826 \times \mathrm{MAD}$$

This sigma value is computed for a window of size 'W' and is used to eventually compute a threshold value where:

$$\text{upper threshold} = \mathrm{median}_j + N*\sigma$$
$$\text{lower threshold} = \mathrm{median}_j - N*\sigma$$

Here, N is the threshold level, usually set at N = 3. These values are used to flag and filter every element in the window, and can do 3 things with the RFI values:

1. Do nothing and just flag the value
2. Replace RFI with a constant
3. Replace with the threshold
4. Replace with digital noise

The GWB system consists of 16 FPGA + Compute nodes, each receiving 2 polarizations of 2 antennas through 2 ADC units attached to them. This filtering method, while fast, consumes an entire FPGA unit and is difficult to reconfigure. Hence the project tries to look into a software implementation to try and perform the RFI filtering on the Compute nodes with CPU and/or GPU power to make room for more complex filters on the FPGAs and provide more flexibility for the MOM-based software implementation. Figure 1.3.1 shows of the block diagram of the GWB backend system.

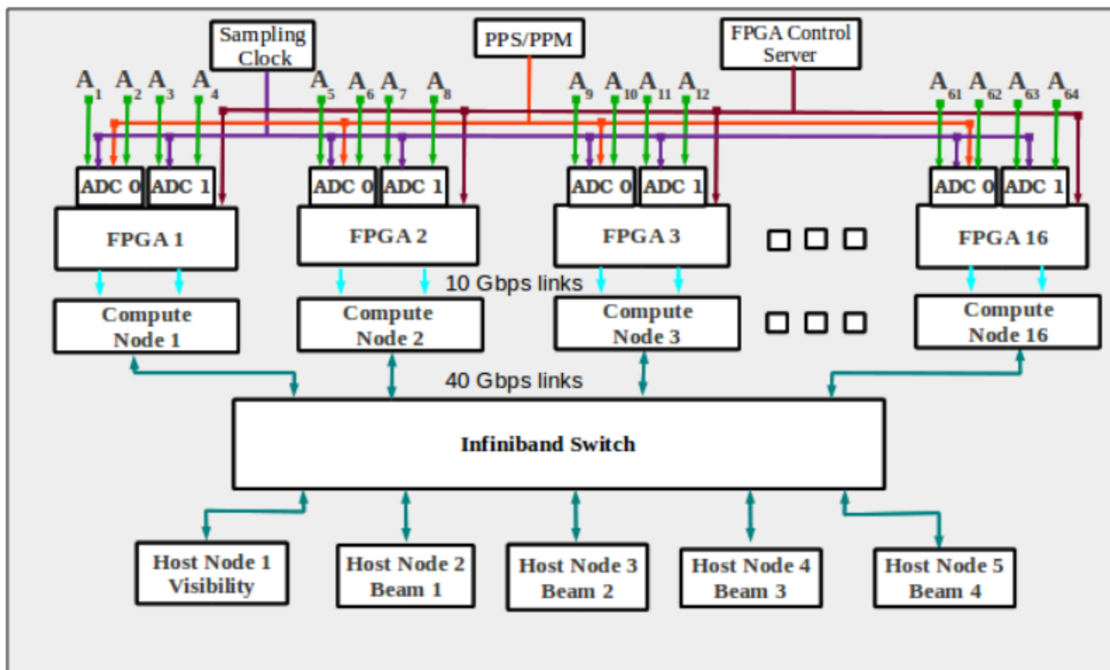

Figure 1.3.1 - Block Diagram for GWB correlator [1]

Chapter 02: Histogram-based Median Computation

2.1 Need for the histogram method

The general method to compute the median is to sort the window of size and find the center element, but the sorting is generally computationally expensive and doesn't provide significant benefit over the cost.

Different sorting algorithms have varying time complexities, but as we only need the N/2th largest element in our array for our calculations, we can just avoid sorting entirely. The histogram method for median calculation is one such algorithm that can be used to find the median of a dataset without sorting the array without extensive computational cost and hence saving us time.

2.2 Algorithm for histogram method

Here, we first construct the histogram of the window of size say, n. To find the median, we find the cumulative sum until the sum is equal to or greater than (n+1)/2 for odd n and n/2 for even n. The bin number when this condition is satisfied is the median. An example of a histogram is shown in Figure 2.2.1

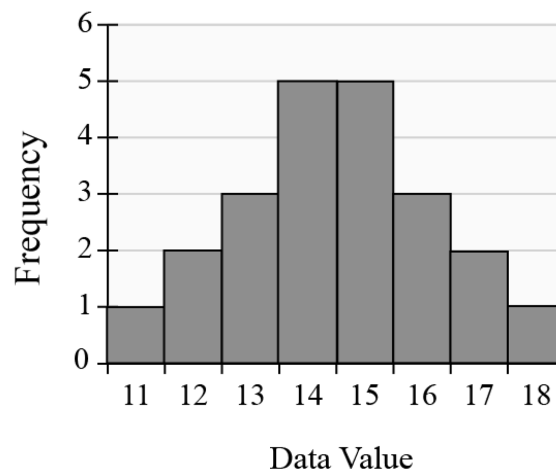

Figure 2.2.1 - Visualization of histogram

2.3 Comparison of different median algorithms over multiple datasets

- Table 2.3.1 shows the timings taken to perform MAD-based filtering with different algorithms used to compute the median, with Figure 2.3.1 helping with its visualization in a line chart.
- The Real Time column pertains to how much time every data point in the window arrives in, which is 2.5 microseconds at 400 MHz sampling rate for the first 1024 window size. Other columns represent the time taken in microseconds for each window to be completely filtered.
- Hence for a software implementation, the code needs to be faster than real time, ie. below the real time line in the chart shown in Figure 2.3.1
- As seen from the data, we can see that the histogram method has the best bet at coming close to real-time for the implementation of MAD-based filtering.

Window size	Real Time	Histogram	Quicksort	Heapsort	Bubble Sort
1024	2.5	65	220	1014	4399
2048	5	118	452	2040	17943
4096	10	235	955	4430	75089
8192	20	460	1971	9474	317732
16384	40	911	4184	19952	2133515

Table 2.3.1 - Comparison of different median algorithms

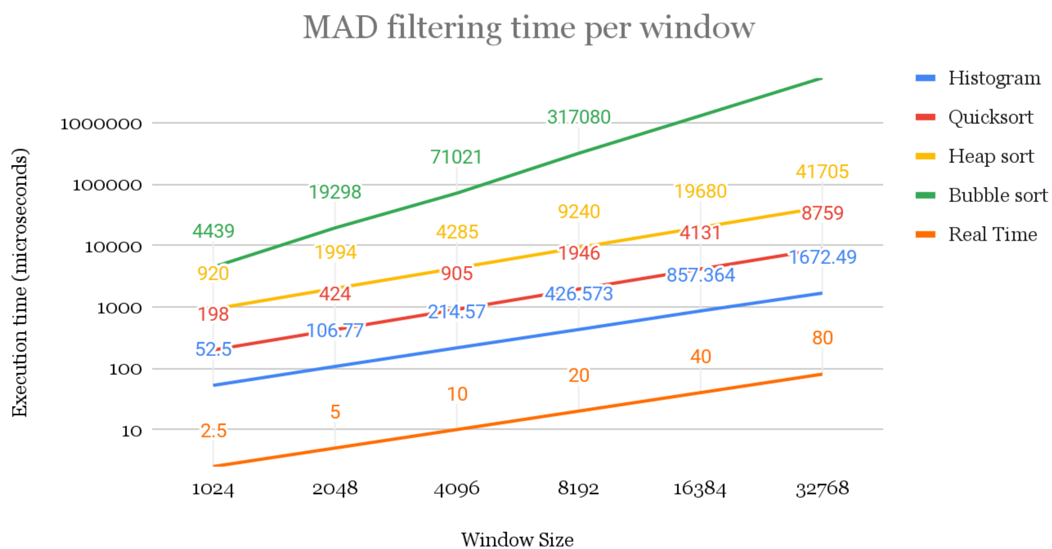

Figure 2.3.1 - Different median algorithms

2.4 Computational estimates for different histogram methods

As the histogram method is the fastest median algorithm but still not meeting real-time requirements, there are multiple assumptions that can be made to get us closer to real-time. Some of these have been used in the FPGA design as well and are tested in the system over a long period of time.

Table 2.2 shows the computational estimates for these different assumptions, and also adds a column to do MOM-based filtering (Median of MAD) to see how much different it is going to be from MAD filtering in terms of computation.

Step	MAD	First median = 0	MAD estimations of every 4th sample	MOM
1. Copy data into array	W (copy operation) W (iterator increment)	W (copy operation) W (iterator increment)	W (copy operation) W (iterator increment)	W (copy operation) W (iterator increment)
2. Calculating first median	256 (making the histogram array) W (offset) W (histogram increment) W (iterator increment) 256 (calculating sum) 256 (iterator increment)	N/A	256 (making the histogram array) W/4 (offset) W/4 (histogram increment) W/4 (iterator increment) 256 (calculating sum) 256 (iterator increment)	256 (making the histogram array) W (offset) W (histogram increment) W (iterator increment) 256 (calculating sum) 256 (iterator increment)
3. Making array of absolute deviation	W (deviation) W (absolute) W(iterator increment)	W(absolute) **no iterator as it was done during step 1	W (deviation) W (absolute) W(iterator increment)	W (deviation) W (absolute) W(iterator increment)

4. Calculating MAD	256 (making the histogram array) W (offset) W (histogram increment) W (iterator increment) 256 (calculating sum) 256 (iterator increment)	256 (making the histogram array) W (offset) W (histogram increment) W (iterator increment) 256 (calculating sum) 256 (iterator increment)	256 (making the histogram array) W/4 (offset) W/4 (histogram increment) W/4 (iterator increment) 256 (calculating sum) 256 (iterator increment)	256 (making the histogram array) W (offset) W (histogram increment) W (iterator increment) 256 (calculating sum) 256 (iterator increment)
5. Calculating MOM (occurs 1/M times)	N/A	N/A	N/A	4M (making the array of medians) M (iterator increment) 3M + 768 (Median calculation) 8 (Updating thresholds)
6. Updating thresholds	1 (sigma) 3 (upper th) 3 (lower th)	1 (sigma) 3 (upper th) 3 (lower th)	1 (sigma) 3 (upper th) 3 (lower th)	**occurs during step 5
7. Filtering data	W (output vector assign) W (flag vector assign) W (iterator increment) F (flag counter increment)	W (output vector assign) W (flag vector assign) W (iterator increment) F (flag counter increment)	W (output vector assign) W (flag vector assign) W (iterator increment) F (flag counter increment)	W (output vector assign) W (flag vector assign) W (iterator increment) F (flag counter increment)
Total Operations per second	14W + F + 1543	9W + F + 1031	11W + F + 1543	14W + F + 1544 + 776/M

Table 2.4.1 - Computational estimates for different histogram methods

Chapter 03: Implementation on CPU

3.1 Algorithm:

The MOM filtering algorithm for the code flow is shown in Figure 3.1.1. The code takes the following parameters as input:

1. Input data - A .txt file containing signed 8 bit signed integers
2. Window size (W)
3. MOM Size (M)
4. Threshold - The N value to filter (N-sigma filtering)
5. RPL Option (Digital noise, Constant, Threshold, Bypass)

According to whether the filewriting option is commented out or not in th code, the code after running will provide the following output:

1. Average time taken for each window
2. Total data points filtered
3. Total RFI points flagged
4. Percentage filtering
5. Ouput .txt file of filtered data (optional)
6. Output .txt file of flagging data (optional)

All benchmarking is done on the snode machine with the following specifications:

```
model name  : Intel(R) Xeon(R) CPU E5-2667 v3 @ 3.20GHz
stepping    : 2
microcode   : 0x3d
cpu MHz     : 3195.976
cache size  : 20480 KB
cpu cores   : 8
cpu threads : 16
```

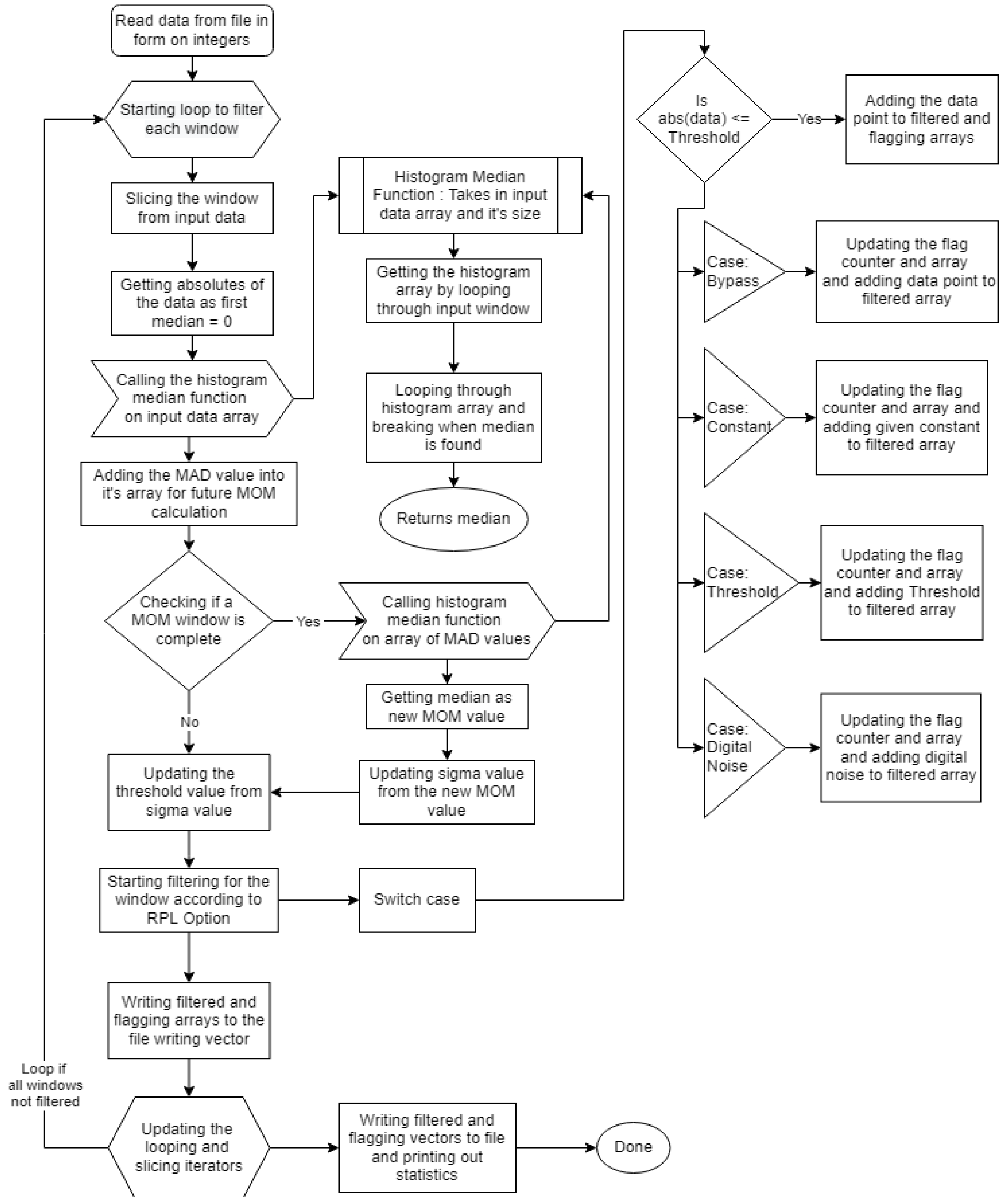

Figure 3.1.1 - CPU MOM Implementation algorithm

3.2 Functional Testing

Every version of code is tested with the methods provided in Figure 3.2 to make sure that the program is accurate as per expectations before any benchmarking is done.

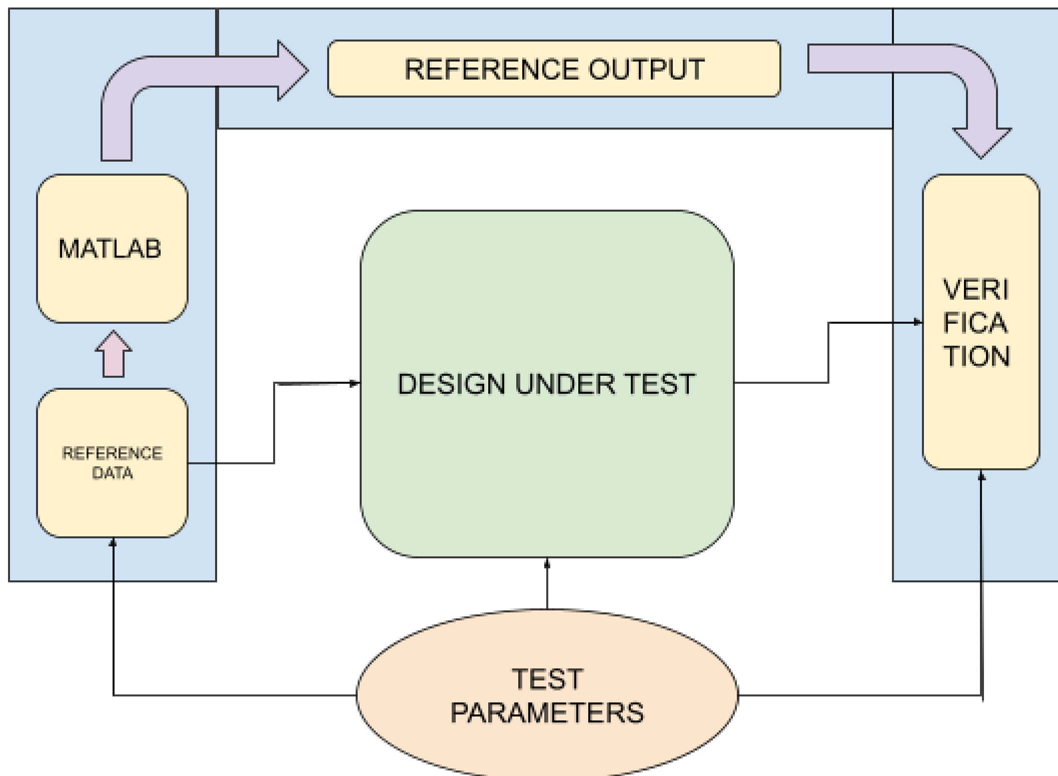

Figure 3.2.1 - Functional testing block diagram

Design Under Test -

- The DUT is the C++ algorithm that is being profiled. These are the multiple algorithms for computing the MAD filter. The different algorithms tested under this are the multiple ways to calculate the median for filtering, ie. median through quicksort, heap sort, bubble sort, and the histogram method.

Test Parameters -

- The test parameters are the window size, threshold value, replacement options, and the reference data to be filtered. These options are passed as command line arguments to the DUT which then proceeds to read the data and filter it according to the given parameters.

MATLAB -

- The MAD filtering algorithm is also implemented in MATLAB software and used to generate the output files to be used as a golden reference to verify the functionality of the design, and find out the rate of errors that occur by the assumptions and limitations of the histogram algorithm.

Reference Data -

- The reference data is data from multiple sources that need to be filtered. We are currently working with raw input voltage data that is in the form of a time series of signed 8-bit integers. The current sources of data are :
    - Simulation data
    - Antenna data (B2, B3, B4, B5 bands from the antennas C11 and C12 taken simultaneously)
    - Digital generated noise
    - 4-bit data

Verification -

- Verification is the C++ diff algorithm called 'filereader' which takes two .txt files and compares the data between them, listing out the points at which the files are different, the number and percentage of differences.
- This is used to compare the reference output from MATLAB against the output generated by the DUT to analyze both outputs.

3.3 Benchmarking of MAD filter

- The different assumptions of the histogram method were profiled on multiple datasets, and more probes were added for getting separate timings for median calculations and the filtering part.
- All the assumptions were compared against a MATLAB reference to verify that the error rate they introduce due to the differences is within acceptable levels.
- The various set of assumptions we have for improving the code is:
    - Keeping the 1st median as 0
    - Sampling every 4th data point for median
    - Both assumptions together

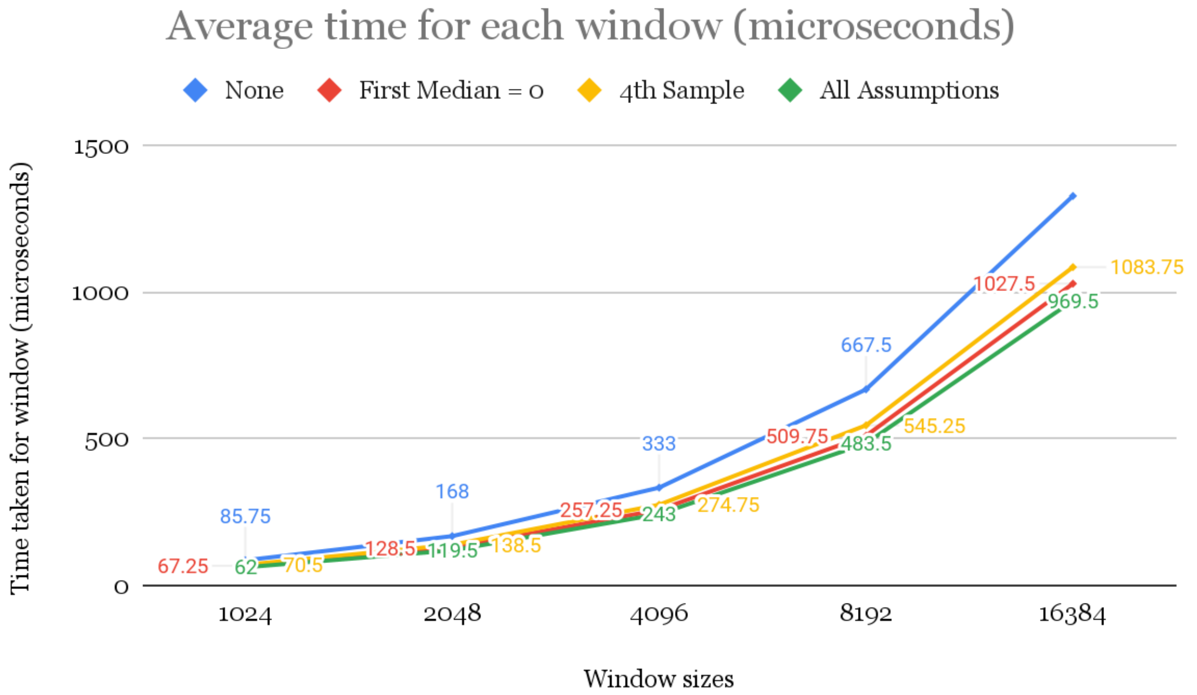

Figure 3.3.1 - MAD filter time per window for different assumptions

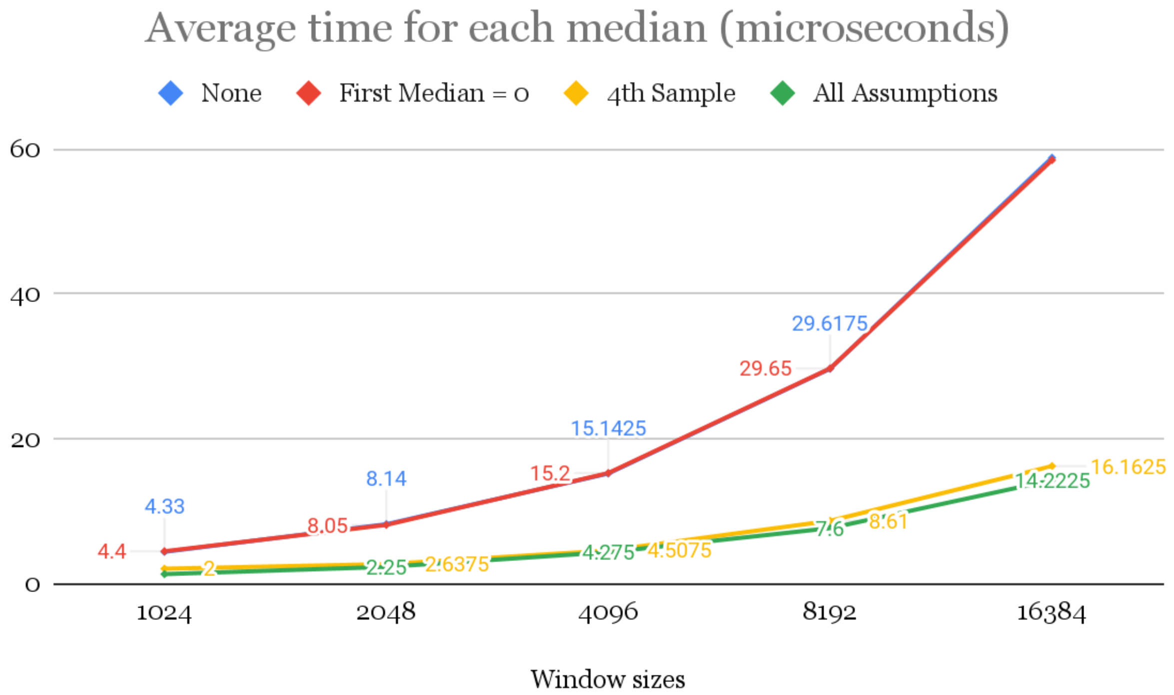

Figure 3.3.2 - Median calculation time per window for different assumptions

3.4 Visualization of the CPU MAD filter

- Data - B3_C11_1, Window size = 16384, N = 3
- The data is taken by the C11 antenna from the Band 3 spectrum (250-300MHz) and filtered at 16384 window size and 3 sigma filtering
- Histogram method is used as it is considered for further optimizations.

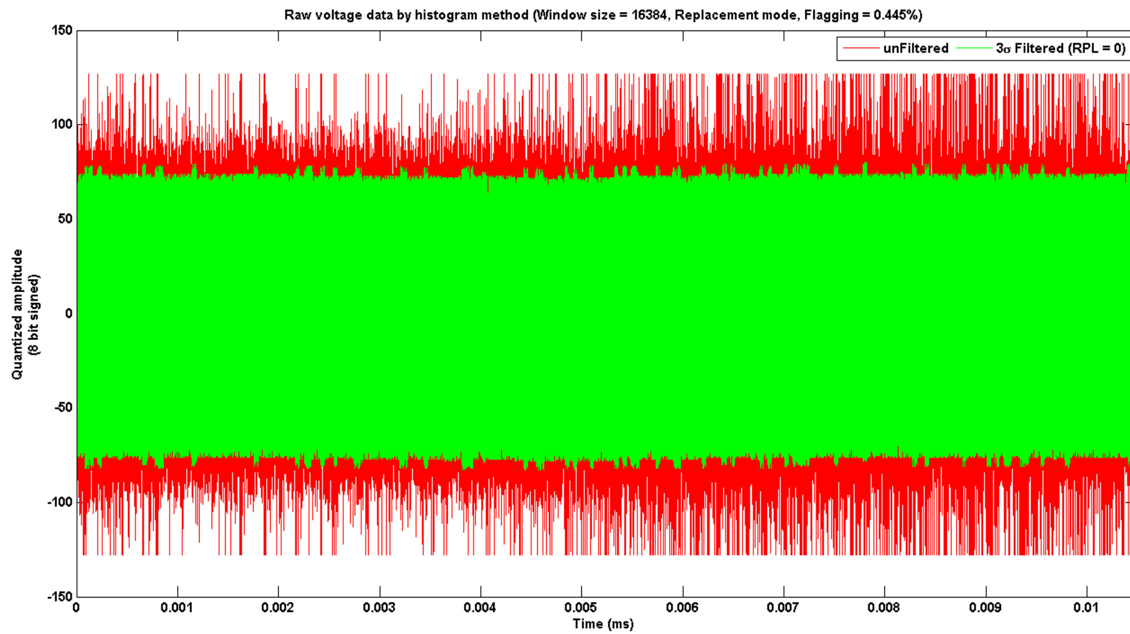

Figure 3.4.1 - 3-sigma MAD Filtering in replacement mode

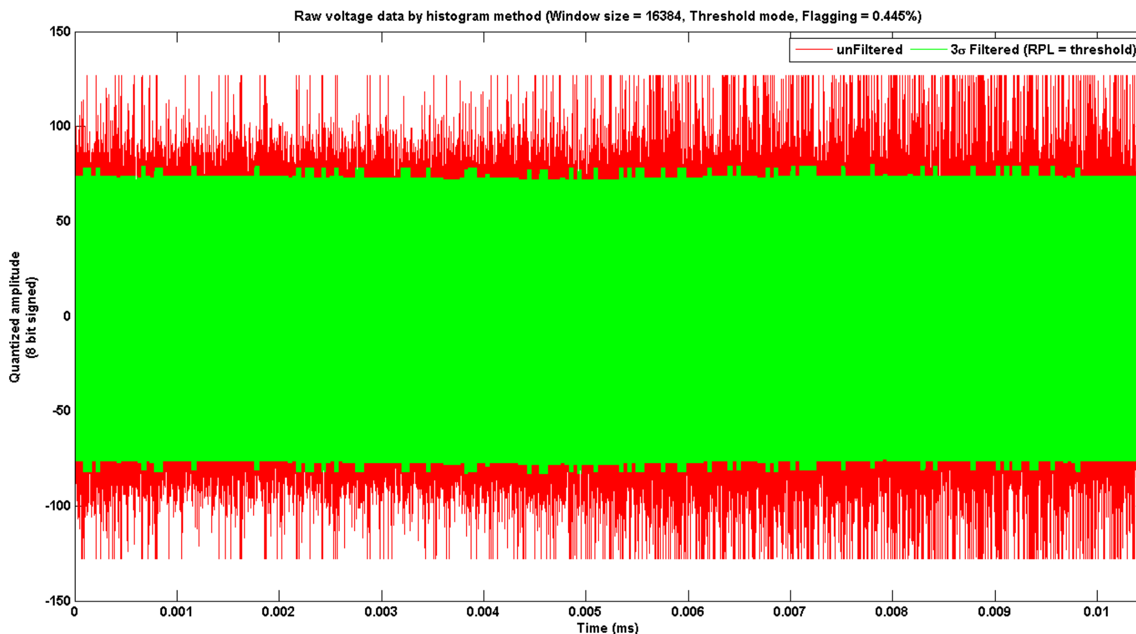

Figure 3.4.2 - 3-sigma MAD Filtering in threshold mode

3.5 Different compiler versions:

After multiple revisions and optimization improvements to the algorithms, compiler-based commands, a performance difference between two different compiler versions. Based on this assumption, some versions of the filtering algorithm were tested with different compiler versions on the GPB machine - identical to the snode machine (Intel Xeon 2667 v3 with CentOS 7). Right now we haven't added the header file reading functionality because the exact format isn't known, but the basic function is made and is ready to be put whenever the format is finalized.

The default gcc compiler version coming with the machines is GCC 4.8.5:

```
[rfiuser@gpbcorr6 ~]$ gcc --version
gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-44)
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

The different gcc version installed was GCC 11.2.1 from devtoolset-11:

```
[rfiuser@gpbcorr6 ~]$ scl enable devtoolset-11 "gcc --version"
gcc (GCC) 11.2.1 20220127 (Red Hat 11.2.1-9)
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

The new GCC version was installed separately and is invoked within a different command to ensure that it does not interfere with the default GCC version or commands.

Installation command for the different compiler -

```
$ sudo yum install devtoolset-11
```

Command to call the different compiler -

```
$ scl enable devtoolset-11 "any usual gcc command"
```

As we can see, the different gcc compiler is called as part of a different tool, and hence can be called separately but identically enough.

3.6 Benchmarking of MOM code on different compilers:

This technique is most preferable for longer bursts of RFI. In this technique the Median of MAD values is carried out and that is called Median of MAD - MoM.

MoM = Median(MAD1 ,MAD2 ,MAD3 ,.....MADn)
-where n = window size

For MoM computation, three Median computations are required. For real-time performance, it is difficult to buffer these data values. So, the calculated current MoM value is applied to the next cycle. Also in order to optimize median computation, third median computation is multiplexed with second median computation. The MAD value is stored in Block memory for W window cycle. At the last MoM window cycle, the main data path for MAD computation block is switched to provide initially stored MAD value and median of these values has been computed and this median value is called as Median of MAD - MoM.

Hence, we add another parameter called a MOM window size (M) which represents the number of windows (of size W) used to calculate an MOM value from. That is, an MxW MOM would be computing the MAD value of M windows each of size W, and then taking the median of those M values is used to compute the sigma value, and hence the threshold, which in turn is used to filter out the next window.

The 3 versions used for performance testing are (with compilation instructions):

1. Basic (Using first median = 0)
2. 4th sample (Uses only every 4th sample for median calculation)
3. openMP (Uses 4 cores to filter the data faster)

The results are the deviation from real-time in seconds (Compute time - 1) done on 16x16 MOM with Digital Noise as RPL Option to show the worst case performance with both compiler versions. Real-time is the horizontal axis.

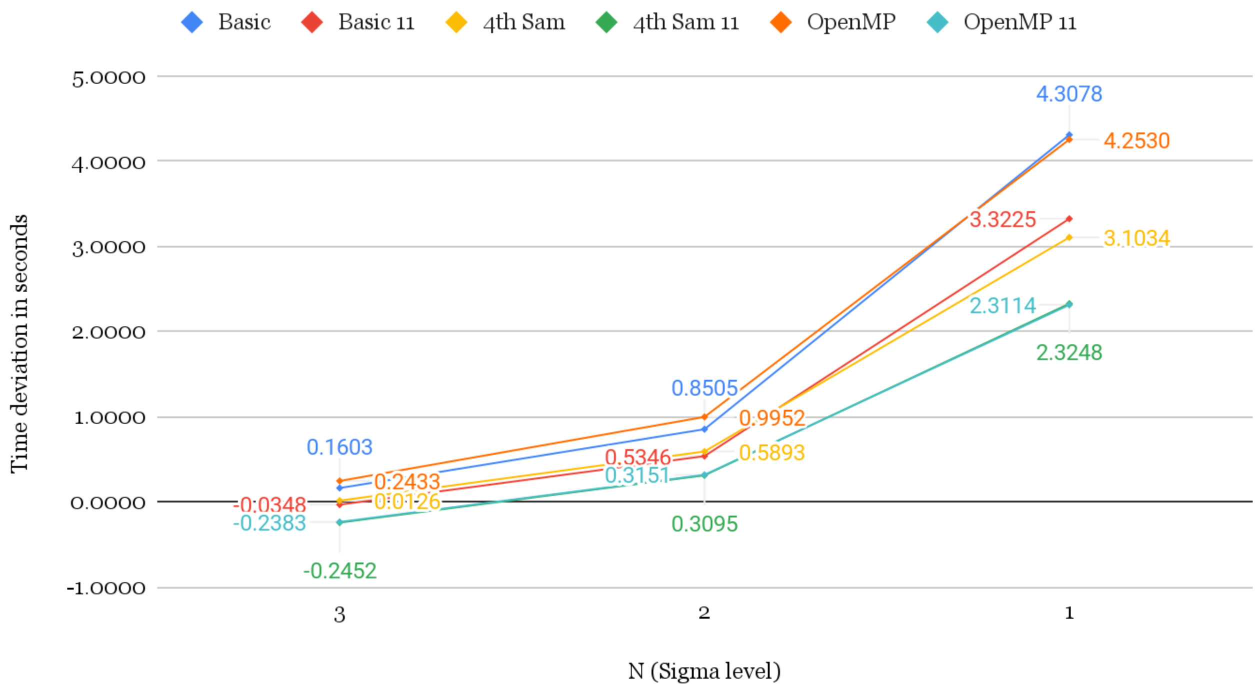

Figure 3.6.1 - Worst case performance of MOM filter on GPU

As we can see, the MOM filtering with optimized code barely passes the real-time barrier at the 3-sigma filtering threshold for one antenna per core, with the 4th sampling observations. This comparison is with data where the RFI is barely 1-1.5%. If the RFI increases further, let's say 2-5% which usually happens on Band 2 observations, the filter cannot keep up (this is seen in the 2 sigma threshold level which shows about 5-6%% RFI at this threshold). This presents a highly dicey situation and it is apparent that this sort of filter cannot be kept in a release.

3.7 Timing analysis of different functions of MOM code:

The best code sample we have written as of yet is the MOM code using every 4th sample for histogram calculation, compiled by GCC 11. This code was probed with more timing calculations to get an idea of how much time every section of the code takes. The table below depicts the acquired results for different window sizes. The RPL option is taken as replacement with digital noise to show worst case performance.

Timing is represented in microseconds							
RPL	N	Window Size	MOM Size	Each MAD Calculation	Threshold + overhead	Filtering	Avg Time/Window
3	3	8192	8192	1.08	1.64	12.1	14.82
3	3	16384	8192	2.17	3.19	24.8	30.16
3	3	16384	16384	2.17	3.36	23.38	28.91
3	2	8192	8192	1.08	1.62	24.55	27.25
3	2	16384	8192	2.19	3.15	46.66	52
3	2	16384	16384	2.43	3.48	45.06	50.97
3	1	8192	8192	1.17	2.11	75.19	78.47
3	1	16384	8192	2.13	3.34	129.32	134.79
3	1	16384	16384	2.19	3.78	115.71	121.68

Table 3.7.1 - Function wise timing distribution

A counter-intuitive result, as the expectation was that the MAD, ie. median calculations would take longer than the filtering + RPL time, but the results were re-ran and the code was checked multiple times. The best explanation for this that can be thought of is the branch prediction performance of the CPU. In both cases, we look at every data point in the window and do a task with it, hence both functions are O(N) in time complexity as seen from window sizes. But there are multiple branches taken which aren't consistent. Usually the CPU pipeline is loaded with a branch predicted result to improve performance, which turns out great in histogram as it's a simple loop with less branches; but having an if-else ladder inside a loop likely causes the branch prediction to be thrown off and results in slower performance due increased pipeline flushes. However, this is just speculation, and actual reason might be different.

Chapter 04: Implementation on GPU

4.1 Basic architectural differences between GPU and CPU

The GPU is highly parallel, with thousands of CUDA cores in a single chip. While the per-core performance in terms of clock speed is only about a fourth compared to a CPU core, due to the immense number of cores available, the GPU can compute repetitive tasks with very high throughput.

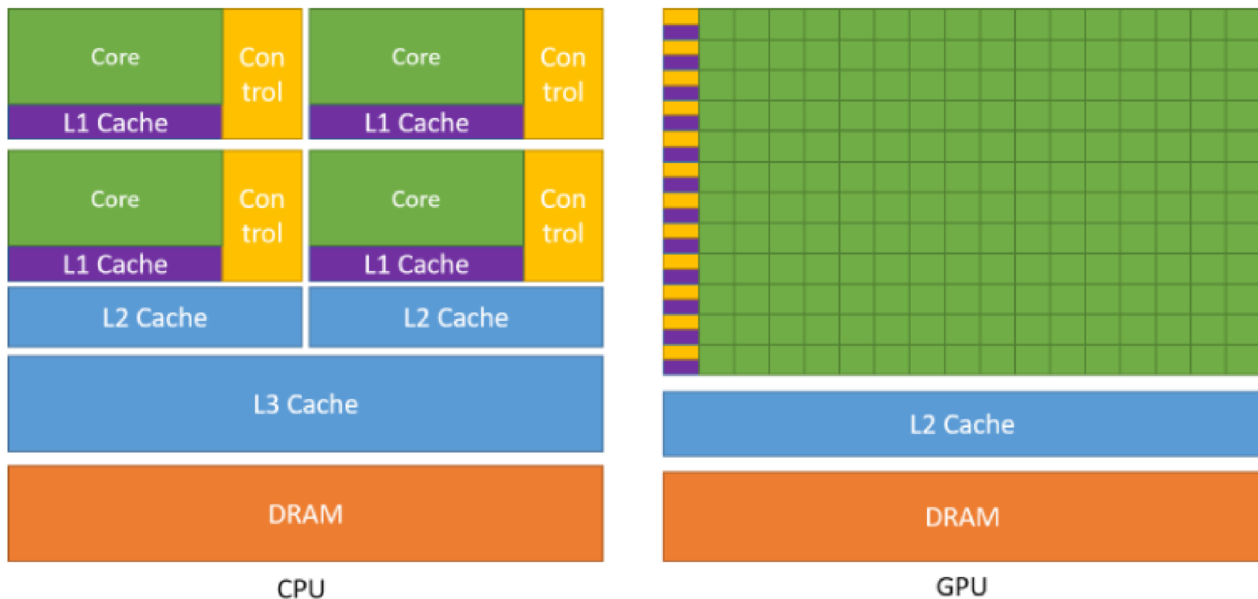

Figure 4.1.1 - Comparison of typical CPU and GPU architecture [9]

The programming model for the GPU is centered around blocks and threads, where the developer initiates functions called CUDA kernels to be run on the device (GPU) with a certain number of blocks having a certain number of threads per block. Every thread in every block executes the algorithm inside a kernel function. This doesn't depend on the structure of the GPU cores, for the most part, to keep the program portable across different models of GPUs and to make it easy to write a CUDA program. The nvcc compiler with the Cuda toolkit takes care of the hardware organization of the blocks and threads.

4.2 Algorithm

The GPU algorithm works mostly on a fork-join architecture : The CPU acts as the host which calls kernel functions that act on the GPU, with pointers to the data that exists on the GPU itself given to the kernel function. For example, the histogram generation function takes in pointers to the input data which is organized in time slices of M windows of W size each. It also takes the pointer to the histogram output array which is organized as an array with sequential histograms corresponding to every window. There are M blocks initialized with T threads each. The code in the kernel function is executed by every thread, hence each thread needs to be pointed correctly to the section of the input and output data arrays. The thread compute the histogram for the elements for each section and store it into the histogram array. Other kernel functions operate similarly. An illustration for the detailed histogram kernel is shown in Figure 4.2.1 and an overall summary of the complete algorithm is given in Figure 4.2.2.

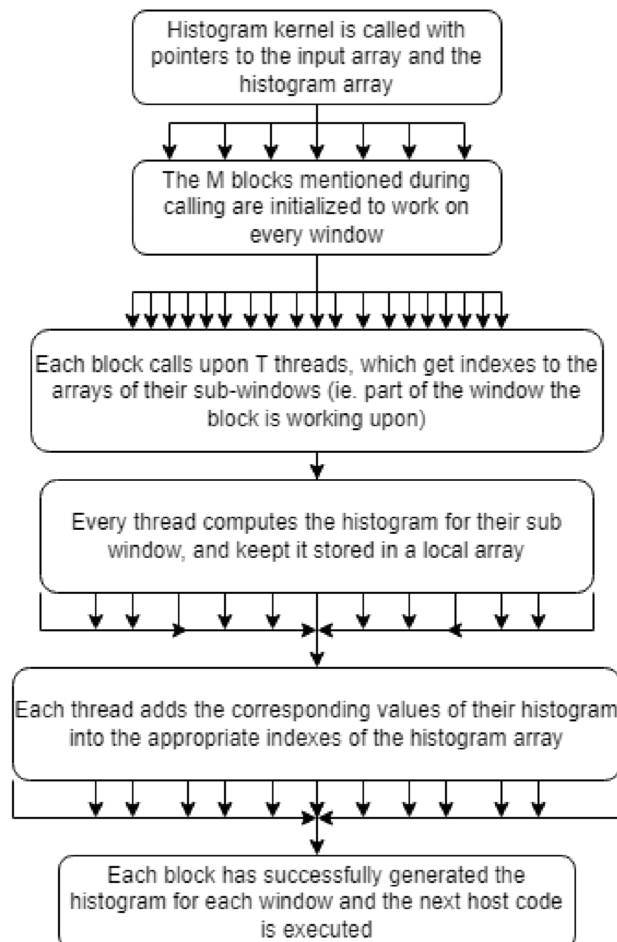

Figure 4.2.1 - Detailed illustration of histogram kernel function

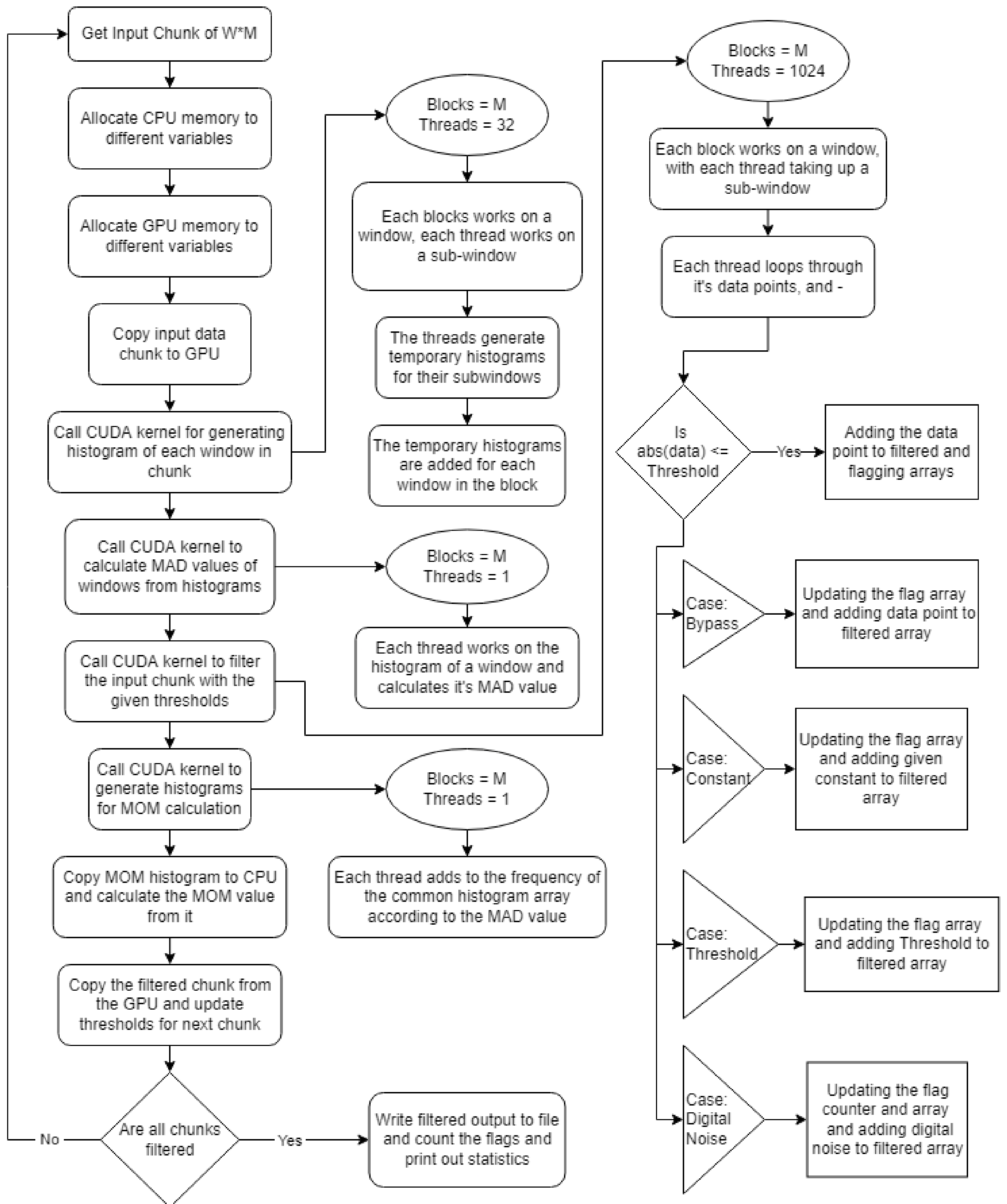

Figure 4.2.2 - Algorithm of MOM filter on GPU

4.3 Benchmarking of MOM filter:

The next two plots (Figure 4.3.1 and 4.3.2) Provide the illustrations of the MOM filtering algorithm as benchmarked on the snode machine. The snode machine is equipped with the Nvidia Tesla K40c GPUs, which were the devices the cuda kernels in the code were executed on. As we can see from 4.3.1, the time taken to compute a MOM window is independent of the RPL option chosen. The compute time is also in a linear relation to the window size, ie. the algorithm is in O(n). Figure 4.3.2 shows that if we adopt the 4th sample histogram assumption used in the current FPGA system, we can almost get twice as fast as real time for a 16x16 window (real time for a 16x16 window is 671 milliseconds.

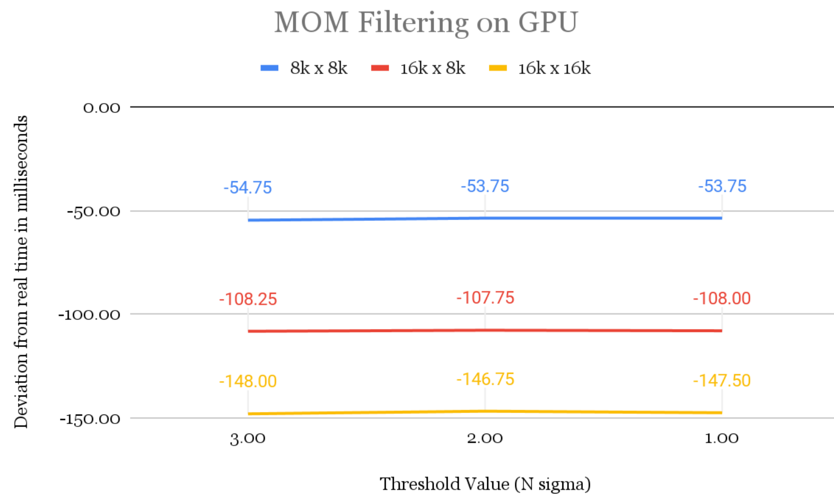

Figure 4.3.1 - MOM filtering on GPU at different threshold levels

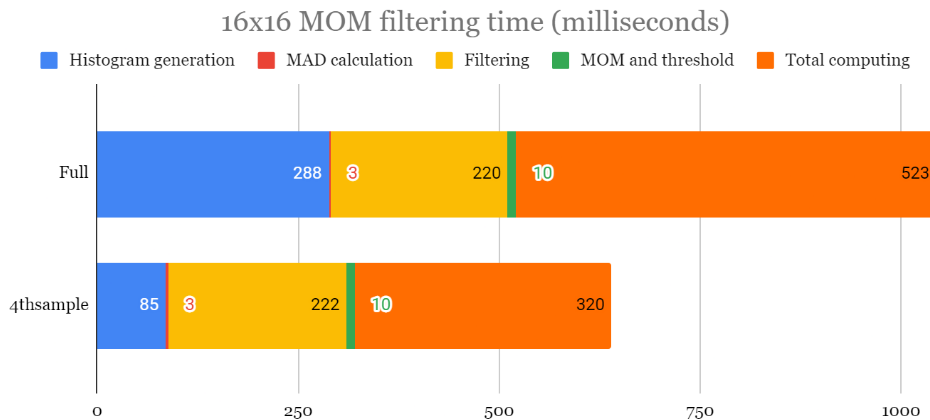

Figure 4.3.2 - Comparison of different GPU MOM algorithms

Chapter 05: Integrating with GWB

5.1 Structure of current GWB code:

The current way the filtering works is with an FPGA before the data is sent to the correlator. At the correlator, the CPU is the best place the current code could be integrated with the least modifications to the system.

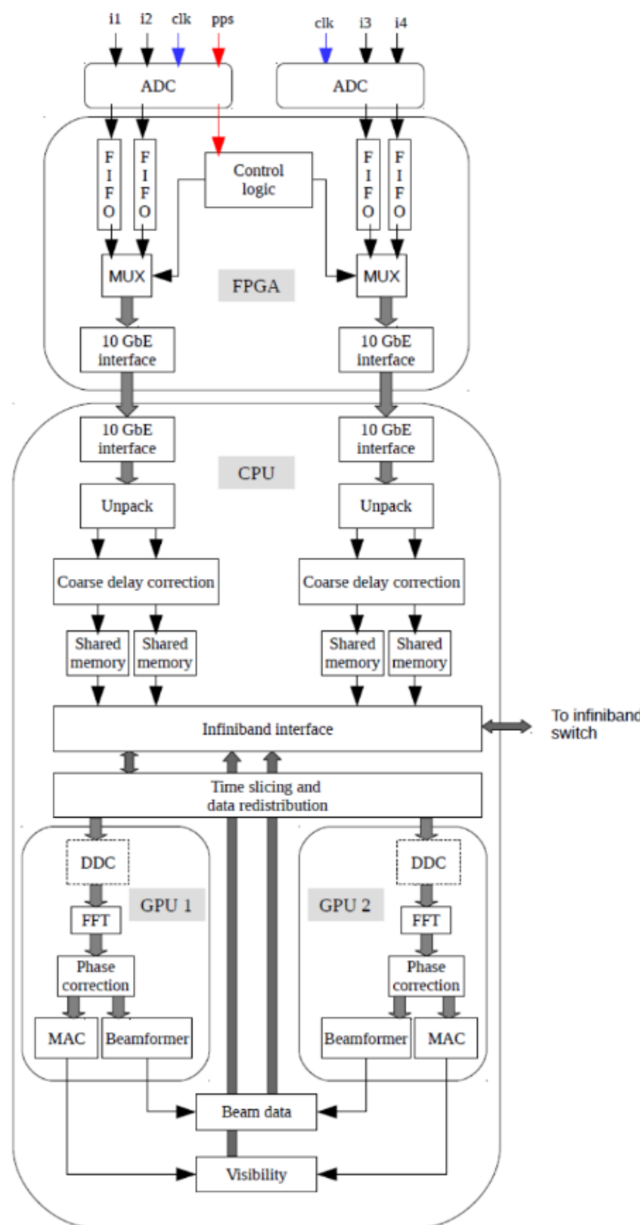

Figure 5.1.1 - Structure of the FPGA + Compute Node [2]

5.2 Benchmarks of integration code:

The CPU code works linearly on sequential raw voltage data. This code currently works for a single data stream (ie. one antenna) but initializing instances of this algorithm on different cores is equivalent to working on multiple antennas at the same time in terms of benchmarking. Hence the code was initialized in that manner on different cores using the '*taskset*' command during execution while the snode correlator was running.

As seen from the terminal instances and the core utilization in Fig 7.2.1, the algorithm runs on one core per stream and is not bottlenecked by the snode correlator running in parallel. This means that if the CPU code is optimized further and starts working in real-time, then integration with the current GWB system will be a trivial task.

Figure 5.2.1 - Utilization of all CPU cores during online snode testing

The GPU code is a different situation. The algorithm is implemented assuming that the raw voltage data will be on the GPU as a single stream, but the GWB structure shows that the current GPU memory buffer contains smaller streams from multiple

antennas at a time and not a single stream. Hence the algorithm will need to be rewritten to accommodate testing and integration of the GPU code into the GWB machine.

Conclusions

- From this attempt at the software implementation of the RFI filtering algorithm, we can say that a software implementation is possible and more reconfigurable than the FPGA-based implementation.
- The CPU can filter a single stream of raw voltage data in near real-time under ideal conditions, while the GPU can filter a raw voltage chunk under any conditions consistently at about twice the real-time.
- The CPU has the limitations of having to rely on the assumptions currently existing in the FPGA system, and it cannot provide filtration on higher RFI levels in real-time.
- The GPU version has the potential of beating real time requirements, and not being dependant on the level of RFI or the replacement option asked for by the user.
- The GPU version has a structural mismatch with the currently implemented code and the way the data on the GWB is structured and fed in; while also having the issue of having to compute power available to spare for the implementation and no pipelining possibilities.

Future Scope

- While there are hurdles of available GPU compute power or upgradation of the CPU compiler and software versions, this project serves as a baseline and gives a path to a completely integrated implementation in the future. T
- he CPU code was tested with the current GWB system running in parallel fashion on the snode machine, and it performed without trouble. Hence by improvements to the code or hardware upgrades, it provides the easiest integration path by taking the continuous data stream coming from each antenna and filtering it.
- The GPU code has a preliminary implementation of the appropriate GWB structure already written in this project which can be verified with a standard reference and implemented without issues as a standalone filter.
- The code can be optimized further by initializing arrays in multiple dimensions for the GPU to work on together. For the GPU integration, even a minor hardware upgrade could be considered as it would greatly provide more compute power to help run the filter alongside other applications.
- Hence through some more optimization and upgrades, a complete software-based RFI filtration algorithm at GMRT is achievable.

References

[1] S. H. Reddy *et al.*, “A Wideband Digital Back-End for the Upgraded GMRT,” *Journal of Astronomical Instrumentation*, vol. 06, no. 01, Mar. 2017, doi: 10.1142/s2251171716410117.

[2] K. D. Buch, K. Naik, S. Nalawade, S. Bhatporia, Y. Gupta, and B. Ajithkumar, “Real-Time Implementation of MAD-Based RFI Excision on FPGA,” *Journal of Astronomical Instrumentation*, vol. 08, no. 01, Mar. 2019, doi: 10.1142/s2251171719400063.

[3] R. Joshi, “rohinijoshi06/mad-filter-gpu,” *GitHub*, Apr. 18, 2020. https://github.com/rohinijoshi06/mad-filter-gpu

[4] GeeksForGeeks, “Inter Process Communication (IPC),” *GeeksforGeeks*, Jan. 24, 2017. https://www.geeksforgeeks.org/inter-process-communication-ipc/

[5]“std::normal_distribution - cppreference.com,” *en.cppreference.com*. https://en.cppreference.com/w/cpp/numeric/random/normal_distribution

[6] “c++ - Performance of built-in types : char vs short vs int vs. float vs. double,” *Stack Overflow*. https://stackoverflow.com/questions/5069489/performance-of-built-in-types-char-vs-short-vs-int-vs-float-vs-double

[7] “linux - How to change the default GCC compiler in Ubuntu?,” *Stack Overflow*. https://stackoverflow.com/questions/7832892/how-to-change-the-default-gcc-compiler-in-ubuntu/7834049#7834049

[8] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, “GPU Computing Graphics Processing UnitsVpowerful, programmable, and highly parallelVare increasingly targeting general-purpose computing applications,” doi: 10.1109/JPROC.2008.917757.

[9] “CUDA C++ Programming Guide,” *docs.nvidia.com*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Appendix

A.1 Reading header files:

The settings for the filter, like window size, threshold level, replacement option, etc are currently being given by a command line argument while initializing program execution. The readhdr.cpp file contains a function to address this and implement the reading and parsing of a header file provided by the observer to get these settings.

Figure A.1 shows the format of the header file alongside the cpp code to read it. The final version of the software implementation would have this function formatted according to the finalized header file format, the current implementation is just for demonstration.

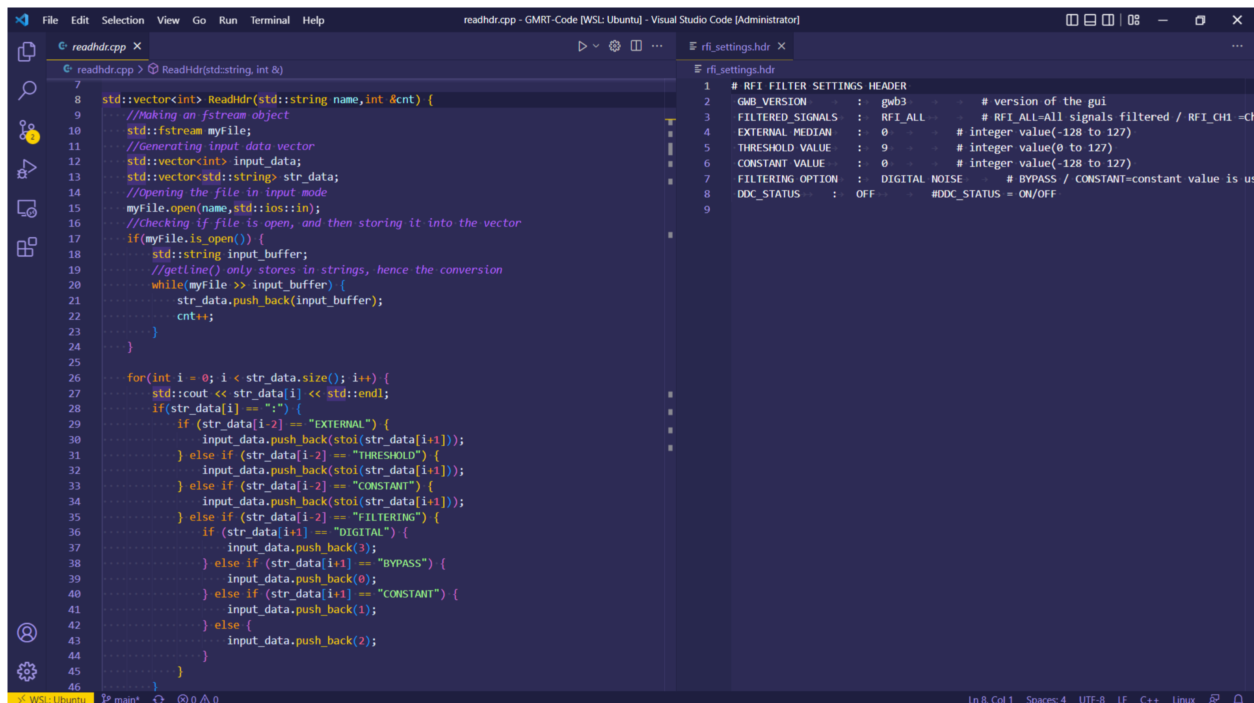

Figure A.1 - Function to read header files

A.2 Datasets used for testing:

These are the details of all the datasets that were used for testing the MAD and MOM filtering code:

Category	Name	Data length	Data length (condensed)
Small Sample data	data16300.txt	16300	16 KB
	c11A.txt	327680	320 KB
Simulator data	1us.txt	1632256	1.5 MB
Band 2 Antenna Data	B2_C11_1.txt	4194304	4 MB
	B2_C12_1.txt	4194304	5 MB
Band 3 Antenna Data	B3_C11_1.txt	4194304	6 MB
	B3_C12_1.txt	4194304	7 MB
Band 4 Antenna Data	B4_C11_1.txt	4194304	8 MB
	B4_C12_1.txt	4194304	9 MB
Band 5 Antenna Data	B5_C11_1.txt	4194304	10 MB
	B5_C12_1.txt	4194304	11 MB
4 Bit Data	4bit_C01_short .txt	327680	320 KB
	4bit_C01.txt	16384000	16 MB
	4bit_C11.txt	16384002	16 MB
Long Data for MOM	1g.txt	449998848	430 MB
	2g.txt	999997440	953 MB
Simulated Noise data	noise_ascii1.txt	268435456	256 MB

Table A.1 - Details of benchmarked datasets