Performance Optimization of CPU-based Real-time RFI filtering System for GWB

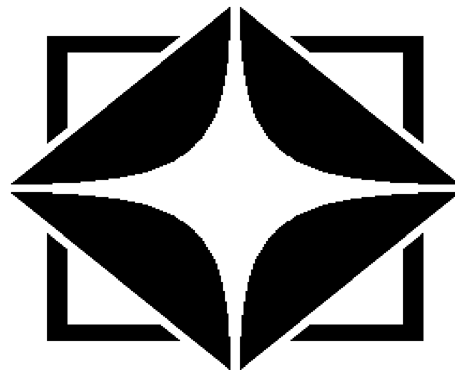
Student Project
by

Satyarth Gupta

Computer Science and Engineering
Indian Institute of Technology, Guwahati

Under the guidance of

Mr. Kaushal D. Buch



NCRA • TIFR

GIANT METREWAVE RADIO TELESCOPE
NATIONAL CENTRE FOR RADIO ASTROPHYSICS
TATA INSTITUTE OF FUNDAMENTAL RESEARCH
Narayangaon, Tal-Junnar, Dist-Pune
May 2023- July 2023

Abstract

The report presents a comprehensive analysis of the implementation and optimization of real-time radio frequency interference (RFI) mitigation algorithms in the context of the Giant Metrewave Radio Telescope (GMRT) and the GMRT Wideband Backend (GWB) processing system. The goal is to improve the real-time performance and efficiency of RFI mitigation techniques by leveraging SSE instructions and other optimizations. The report outlines the benchmarking results, functional testing, and integration with the GWB processing system. Additionally, it explores the power detection technique and its optimization using SSE instructions. The report concludes with an overview of the achieved improvements and outlines future directions for further enhancements.

Acknowledgements

I would like to express my heartfelt appreciation and gratitude to my mentor, Mr. Kaushal D. Buch, Engineer F, Digital Backend Group, GMRT, for their invaluable guidance, encouragement, and insightful suggestions throughout this project. Their unwavering support and teachings were instrumental in the completion of this report. I am deeply grateful for their constant assistance and direction.

I would also like to thank Dr. Jayanta Roy , Sanjay Kudale, and Harshavardhan Reddy for discussion regarding this project. The discussions with them were helpful in understanding the CPU programming and the GMRT Wideband Backend (GWB).

I extend my thanks to Mr. B. Ajith Kumar, Group Coordinator, GMRT Backend Group, for his cooperation, support, and permission to work within the Digital Back-end Group.

I would like to express my sincere appreciation to the members of the Digital Backend group and the entire staff of GMRT.

List of contents

1.1 Brief on GMRT Array and uGMRT 7
1.2 Digital Backend 7
1.3 A brief on Radio Frequency Interference 8
    1.3.1 Narrowband RFI 8
    1.3.2 Broadband RFI 8
1.4 A brief introduction to GMRT Wideband Backend 10
1.5 Current RFI mitigation method 11
2.1 RFI filter C++ code based on median calculation optimisation 12
2.2 Benchmarking results of MoM_4th sample 14
2.3 MoM 4th sample with pre-generated noise 15
2.4 Benchmarking results from pre-generated noise approach 16
2.5 Major bottlenecks and possible optimisations 17
3.1 SIMD,SSE and AVX instructions - an overview 18
    3.1.1 SSE supported by snode 18
    3.1.2 AVX over SSE 19
3.2 Converting C++ code to use AVX instructions for filtering 19
3.3 Flow of the overall code with flowchart 22
3.4 Benchmarking on sample datasets 25
3.5 Error in previous pre generating noise function 28
3.6 Functional results 30
3.7 Running on four cores via taskset 31
4.1 GWB processing requirement 32
4.2 Converting C++ code to C code 33
4.3 Reading .hdr header file to pass parameters 34
    4.3.1 About.hdr file 34
    4.3.2 Reading the parameters from .hdr file and the code 34
4.4 Double buffering of input and output buffers 36
4.5 Implementing OpenMP for parallel filtering 37
4.6 Functional test results 39
6.1 Working of the code 40
6.2 Earlier implementation - with pre generated noise 41
6.3 Functional testing - against voltage based MoM filter 41
6.4 Benchmarking results 43
rfiuser@snode specifications 46
List of generic parameters 46
Codes (C++ and C) 46
    MoM_4thsample.cpp 46
AVX instructions and their usage **55**

List of figures

Figure 1.1.1 - GMRT Antenna (courtesy GMRT archives) 7
Figure 1.3.1 - Narrowband RFI at GMRT Courtesy: Mayuresh Surnis 8
Figure 1.3.2 - Broadband RFI in band 2 at GMRT 9
Figure 1.3.3 - RFI Detection setup 9
Figure 1.3.4 - Spectrum analyser 9
Figure 1.3.5 - Distribution of radio signals (courtesy - Bonnie Baker) 10
Figure 1.4.1 - Block diagram for GWB correlator 10
Figure 2.1.1 - noise replacement mode 13
Figure 2.1.2 - Flowchart of MoM_4thsample.cpp 13
Figure 2.2.1 - Benchmarking plots for 1g.txt on MoM_4thsample.cpp 14
Figure 2.3.1 - pre generation of noise - module 15
Figure 2.3.2 - noise populated in the array 15
Figure 2.3.3 - Pre generated noise used in noise replacement 15
Figure 2.4.1 - Comparison of deviation of from real time pre generated noise against generating at every MoM window 16
Figure 2.4.2 - Benchmarking results and comparison with 2g.txt 17
Figure 3.1.1 - SSE flags in cpu 18
Figure 3.2.1 - Changes made to achieve same functionality as previous C++ code 19
Figure 3.2.2 - Converted constant replacement mode 21
Figure 3.2.3 - Converted bypass mode 21
Figure 3.2.4 - Converted threshold replacement mode 22
Figure 3.2.5 - Noise replacement mode, pregenerated noise 22
Figure 3.4.1 - Benchmarking plots of 2g.txt. All other curves overlap with green one 25
Figure 3.4.2 - Band 2 data deviation plot 26
Figure 3.4.3 - Band 3 data deviation plot 26
Figure 3.4.4 - Band 4 data deviation plot 27
Figure 3.4.5 - Band 5 data deviation plot 27
Figure 3.5.1 - Previous noiseMaker with stddev declared to 1 and type cast to int 28
Figure 3.5.2 - Histogram of noise generated by earlier NoiseMaker function 28
Figure 3.5.3 - Modified noise function with sigma as an argument 29
Figure 3.5.4 - Histogram of noise generated by modified NoiseMaker 29
Figure 3.5.5 - kurtosis of noise2(unmodified) 30
Figure 3.5.6 - kurtosis of noise1(modified) 30
Figure 3.6.1 - Functional verification flowchart 30
Figure 3.6.2 - Overlay plots for filtered over unfiltered signal, N =3, zero replacement 30
Figure 3.6.3 - Overlay plots for filtered over unfiltered signal, N =2, threshold replacement 31
Figure 3.7.1 - CPU and memory utilisation during running with taskset 31
Figure 4.1.1 - Flowchart of execution sequence in the code for GWB integration 33
Figure 4.3.1 - a typical .hdr file 34

Figure 4.3.2 - structure of RfiFilterSettings 34
Figure 4.4.1 - structure of a double buffer 37
Figure 4.4.2 - double buffer allocation 37
Figure 4.5.1 - OpenMP implementation 38
Figure 4.6.1 - Functional results with constant replacement, N = 3 39
Figure 4.6.2 - Functional results with threshold replacement, N = 2 40
Figure 4.6.3 - Flowchart of functional verification procedure [4] 40
Figure 5.1.1 - basic working of power detection 41
Figure 5.2.1 - Noise replacement in power detection technique 42
Figure 5.3.1 - Flagging percentage for various threshold factor of power detection filter compared against voltage based filter 43
Figure 5.4.1 - Comparison of deviation with N for subwindow of 16,32,64,128,256. Window size is 1024. 44
SELF DECLARED SSE/AVX FUNCTIONS 59

List of tables

Table 2.4.1 - Quantitative data for snode, mode - noise replacement 16
Table 3.2.1 - Description of each line and function inside loop 21
Table 3.4.1 - Time taken for 1 MAD window for each replacement option 25

Chapter 1 : Introduction

1.1 Brief on GMRT Array and uGMRT

The GMRT, or the Great Metrewave Radio Telescope, is an impressive radio telescope located near Pune, India. It's managed by the National Centre for Radio Astrophysics (NCRA) at the Tata Institute of Fundamental Research (TIFR). The GMRT is renowned worldwide as one of the biggest and most sensitive radio telescopes that operates at metre wavelengths. The GMRT comprises a group of 30 antennas (figure 1.1.1), each with a diameter of 45 metres. These antennas function at various frequencies ranging from 150 MHz to 1450 MHz. To achieve a lightweight and wind-resistant design, the GMRT employs an innovative antenna construction method called SMART. This approach involves using wire mesh panels and rope trusses, making the entire array cost-effective to build [1].

Figure 1.1.1 GMRT Antenna (courtesy GMRT archives)

1.2 Digital Backend

The backend systems are housed at the CEB, with separate systems for processing the legacy and upgraded GMRT signals. The first stage of these systems is the analog processing chain

that provides the final baseband signals, which are then digitised and processed by the digital backend systems – the GMRT Software Backend (GSB) for the legacy GMRT, and the GMRT Wideband Backend (GWB) for the upgraded GMRT. The GWB processes a maximum of 400 MHz band for each of two polarisations for all the GMRT antennas.

1.3 A brief on Radio Frequency Interference

RFI, or Radio Frequency Interference, refers to unwanted electromagnetic signals or noise that can disrupt or degrade the quality of radio frequency signals used in communication systems, including wireless devices, radios, and radar systems [2]. RFI can originate from various sources, such as power lines, electrical equipment, electronic devices, and even natural phenomena like lightning. The presence of RFI can lead to signal distortion, reduced range, and increased error rates in communication systems.

1.3.1 Narrowband RFI

Narrowband RFI refers to radio frequency interference that occurs within a limited portion of the spectrum. These signals typically exhibit consistency over an extended period and generally do not cause permanent harm to the system. The occurrence of narrowband RFI is often attributed to frequency overlap from sources such as mobile towers and other communication devices. Figure 1.3.1 illustrates narrowband RFI observed at GMRT, where the distinct peaks in the image represent narrowband RFI within the 325 MHz observing band of GMRT.

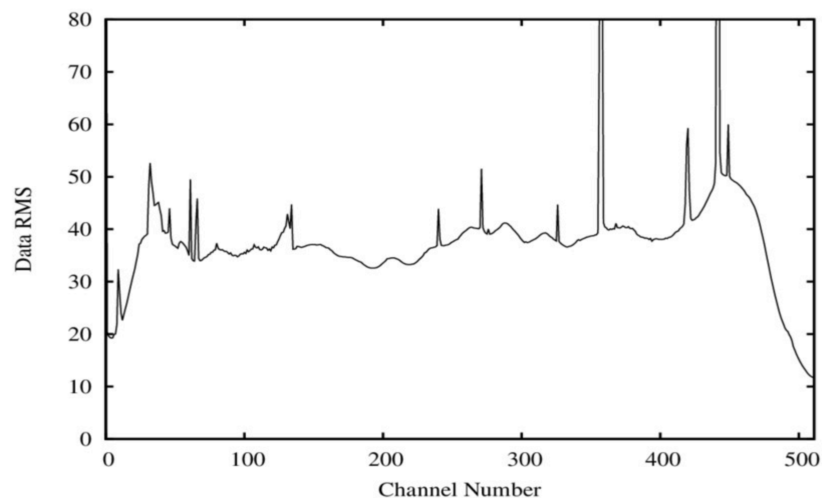


Figure 1.3.1 - Narrowband RFI at GMRT Courtesy: Mayuresh Surnis

1.3.2 Broadband RFI

Broadband RFI, on the other hand, manifests as impulsive signals with high energy content capable of overpowering the underlying astronomical signals. This type of interference

occurs over short durations and has the potential to cause permanent damage to electronic receiver systems due to its intense nature. Figure 1.3.2 showcases broadband RFI observed at GMRT, predominantly caused by high voltage power lines, which is why it is commonly referred to as powerline RFI.

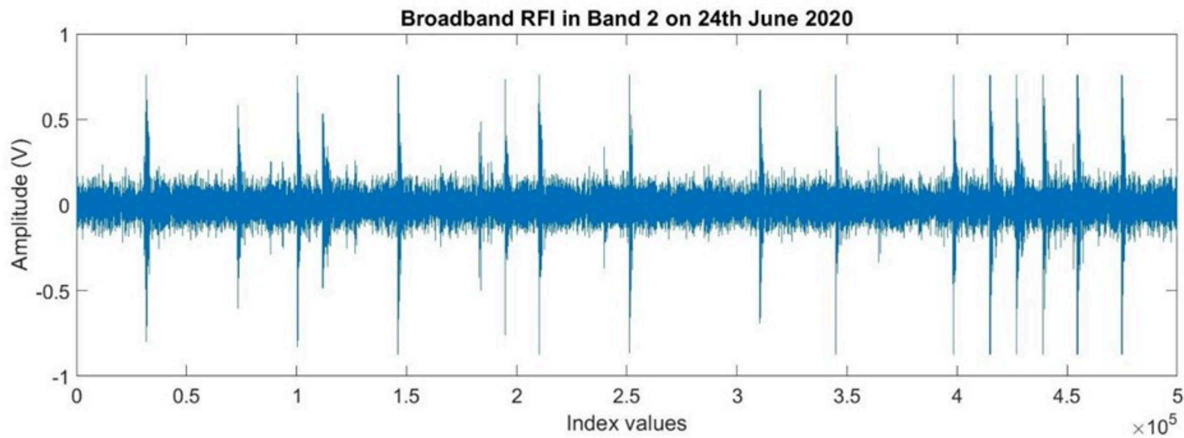


Figure 1.3.2 - Broadband RFI in band 2 at GMRT

Figure 1.3.3 shows the log-periodic antenna setup to measure RFI from the powerline at GMRT Main electric Supply, and figure 1.3.4 shows detected RFI on the spectrum analyser.

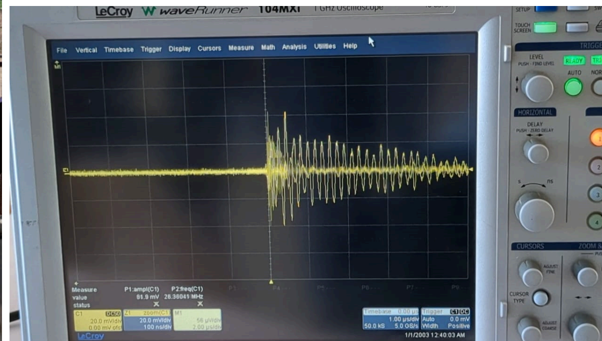


Figure 1.3.3 RFI Detection setup

Figure 1.3.4 Spectrum analyser

The radio signals received from outer space are often assumed to follow a normal distribution, as depicted in the figure 1.3.5. However, these signals are susceptible to interference from RFI, which is primarily a local phenomenon. The amplitude of RFI is typically stronger than that of the original signal, resulting in the useful signal being concentrated primarily within the range of three standard deviations, at the centre of the distribution curve. So the signals out of this range which have high amplitude are mitigated using different algorithms.

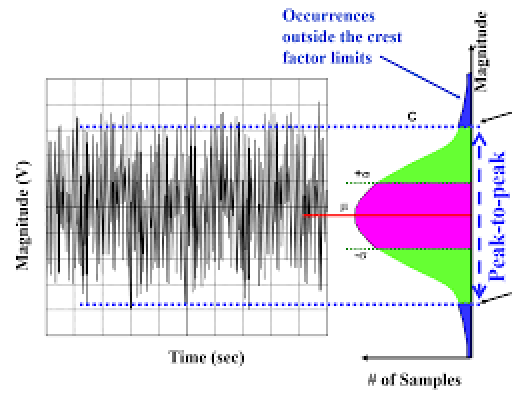


Figure 1.3.5 - Distribution of radio signals (courtesy - Bonnie Baker)

1.4 A brief introduction to GMRT Wideband Backend

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 is currently implemented on FPGA {provide a citation to my paper}. However, due to certain other backend feature requirements on FPGA, the filter needs to be moved to CPU. Hence the project tries to look into a software implementation to try and perform the RFI filtering on the Compute nodes with CPU to make room for more complex filters on the FPGAs and provide more flexibility for the MoM-based software implementation. Figure 1.4.1 shows the block diagram of the GWB backend system [3].

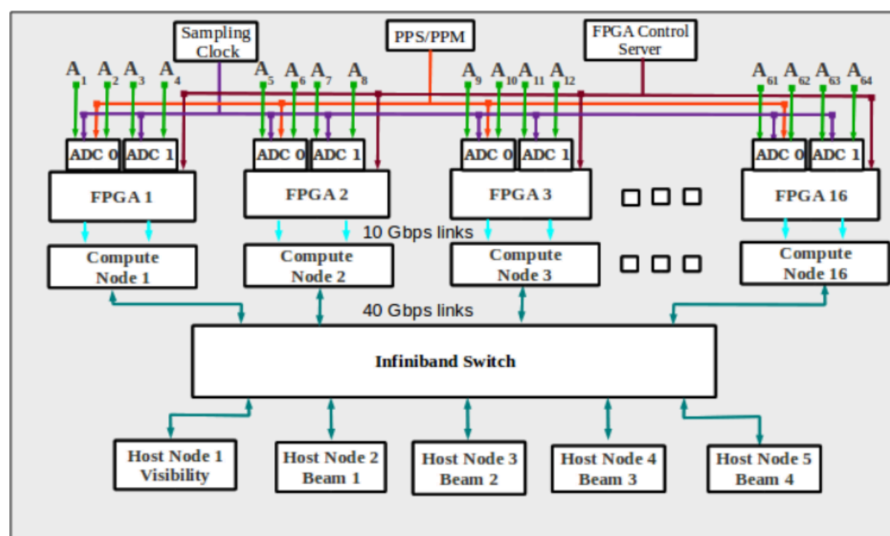


Figure 1.4.1 - Block diagram for GWB correlator

1.5 Current RFI mitigation method

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(median of MADs) filtration on 4 antennas. Then this digitised and filtered data is sent to the CPU + GPU compute node for further processing and data storage.
Earlier, the MoM_4thsample.cpp code(refer appendix) computed MoM via Histogram Method over every 4th sample, to optimise performance.

MAD(median absolute deviation) is defined as [4]

$$MAD = median_j[|x - median(x_i)|]$$

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

$$\sigma = 1.4826 \times MAD$$

However, for longer bursts of RFI, having a median of several MAD values called the Median-of-MAD(MoM) is preferred. In that case,

$$\sigma = 1.4826 \times MoM$$

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

$$\text{upper threshold} = median_j + N*\sigma$$
$$\text{lower threshold} = median_j - N*\sigma$$

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

Chapter 2 : Overview and benchmarking of the previous versions on CPU

In this chapter, we will provide an overview of the MoM_4thsample.cpp code and discuss the benchmarking process based on the average deviation per second of data. We will explore the functionality of the code and present the benchmarking results.

2.1 RFI filter C++ code based on median calculation optimisation

The initial code, named MoM_4thsample.cpp, of the RFI filter, was based on calculating median and MAD from histogram sort. To optimise histogram sorting, median was calculated using every 4th sample after sorting. Also in the noise replacement, the noise was generated at every sample which contributed to increase in calculation time, affecting real time performance. Overall, the code was far from real time performance.

For benchmarking, the data used was a long data set (1g.txt) containing around 449 million samples, each of them being of 1 byte (from -128 to 127). It was collected at a bandwidth of 200 MHz and it is of band 3.

For 200 MHz bandwidth, the Nyquist sampling criterion sets the sampling frequency to be

$$\text{Sampling frequency, } f_s = 2 \times 200 \text{ MHz} = 400 \text{ MHz}$$

Samples come at a time interval of

$$1/f_s = 1/400 \text{ MHz} = 2.5 \text{ ns}$$

For a 16K (16,384) window of samples, we get a time of

$$16{,}384 \times 2.5 \text{ ns} = 40.96 \text{ µs} \approx 40 \text{ µs}$$

We are required to be able to process 1 signal window of 16 K (filter it), within 40 microseconds.

Filter time for cpp code was calculated using `high_resolution_clock::now()` from chrono library

The code of MoM_4thsample.cpp is present in the appendix.
Figure 2.1.1 shows a snippet of Noise replacement mode, figure 2.1.2 shows flow of execution of MoM_4thsample.cpp

```
case 3:
    //Generating a gaussian noise distribution with 0 mean and 1 std deviation
    default_random_engine generator(time(0));
    normal_distribution<float> dist(0,1);

    for(int k = 0; k < WINDOW_SIZE; k++) {
        int ansk = ans[k];
        if(std::abs(ansk) <= upper_th) {
            //Adding the filtered value to the output array
            filtered_output[k] = ansk;
            flag_output[k] = 0;
        } else {
            //Increasing flag counter
            flag_count++;
            //Adding the filtered value to the output array
            int noise = dist(generator)*sigma;
            filtered_output[k] = std::abs(noise) > 127 ? (-1^k)*127 : noise;
            flag_output[k] = 1;
        }
    }
    break;
```

Figure 2.1.1 noise replacement mode

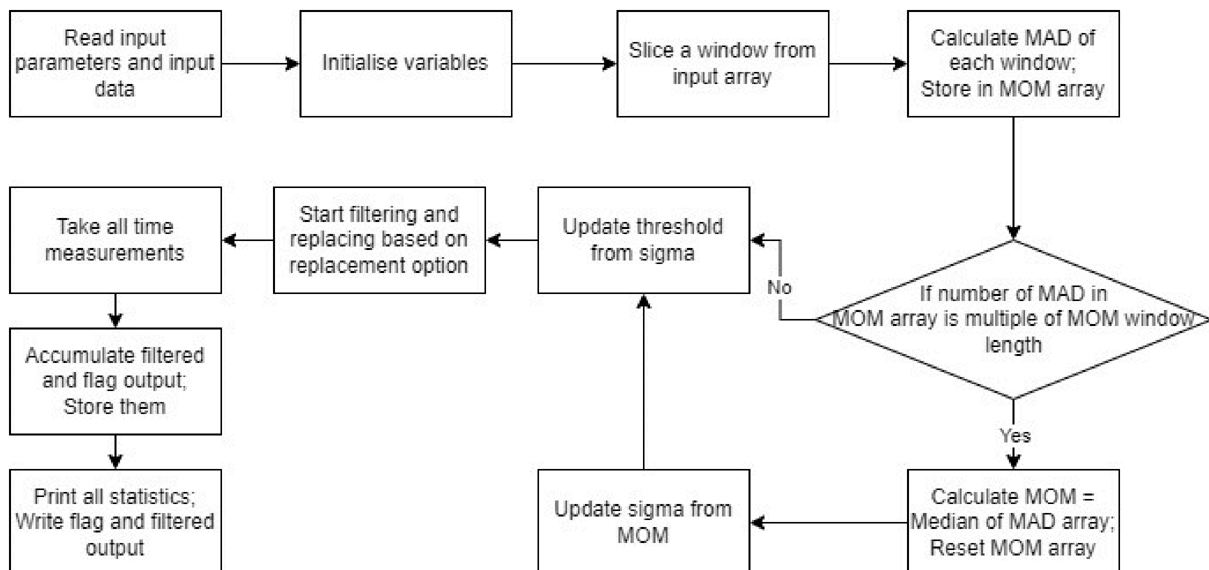


Figure 2.1.2 Flowchart of MoM_4thsample.cpp

2.2 Benchmarking results of MoM_4th sample

We ran the code on rfiuser@snode (specifications in appendix) machine and tracked the time taken for it to run on different modes with different values of threshold factor n. We then calculated deviation of actual time taken to filter vs the time it takes 1 MAD window to arrive (around 40 microseconds).

```
Deviation = (actual filtering time/real sampling time) - 1
```

The deviation should be less than 0 for the filtration to occur in less than 1 MAD window sampling time.
Plots of benchmarking are in figure 2.2.1

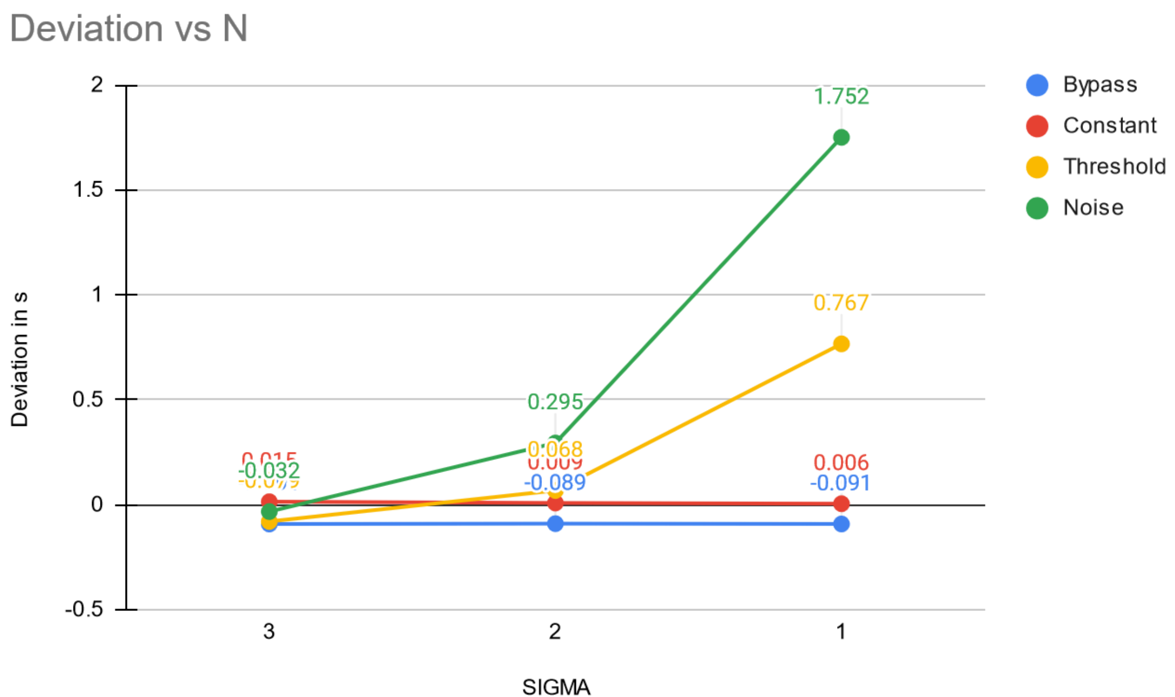


Figure 2.2.1 Benchmarking plots for 1g.txt on MoM_4thsample.cpp

We can see that threshold and noise replacement are falling way before real time requirements especially for lower threshold factors. Also for bypass and constant replacement the deviation is still not enough below real time requirements.

For noise replacement, we can see that the noise samples are generated at every MAD window, which could suggest the extreme deviation from real time.

2.3 MoM 4th sample with pre-generated noise

To reduce the time from generation of noise samples at each MoM window, a different approach of pre-generating the noise in an array(figure 2.3.1) and storing it(figure 2.3.2), was followed. Figure 3.2.3 shows the noise replacement mode utilising pre generated noise.

```
//********************** Noise Generation Module ****************************************//
int* NoiseMaker(int arr[], int len) {

    // Define random generator with Gaussian distribution
    const double mean = 0.0;
    const double stddev = 1.0;
    std::default_random_engine generator(time(0));
    std::normal_distribution<float> dist(mean, stddev);

    // Add Gaussian noise
    for (int i = 0; i < len; i++) {
        arr[i] = dist(generator);
    }
    //std::cout << oper) << std::endl;

    return 0;
}
```

Figure 2.3.1 - pre generation of noise - module

```
//Pregeneration of noise for the given window size
int noise_window[WINDOW_SIZE];
NoiseMaker(noise_window,WINDOW_SIZE);
```

Figure 2.3.2 - noise populated in the array

```
case 3:

    for(int k = 0; k < WINDOW_SIZE; k++) {
        int ansk = ans[k];
        if(abs(ansk) <= upper_th) {
            //Adding the filtered value to the output array
            filtered_output[k] = ansk;
            flag_output[k] = 0;
        } else {
            //Increasing flag counter
            flag_count++;
            //Adding noise from noise array
            filtered_output[k] = abs(noise_window[k]*sigma) > 127 ? (-1^k)*127 : noise_window[k]*sigma;
            flag_output[k] = 1;
        }
    }
    break;
```

Figure 2.3.3 - Pre generated noise used in noise replacement

This approach reduces the time overhead from generation of noise at every MoM window. At different instances of running this program noise generated is different, hence each antenna

would have its own pre generated noise window randomly generated and different from others.

2.4 Benchmarking results from pre-generated noise approach

Benchmarking results from this approach for 1g.txt(band 3 data) are given in figure 2.4.1 and table 2.4.1

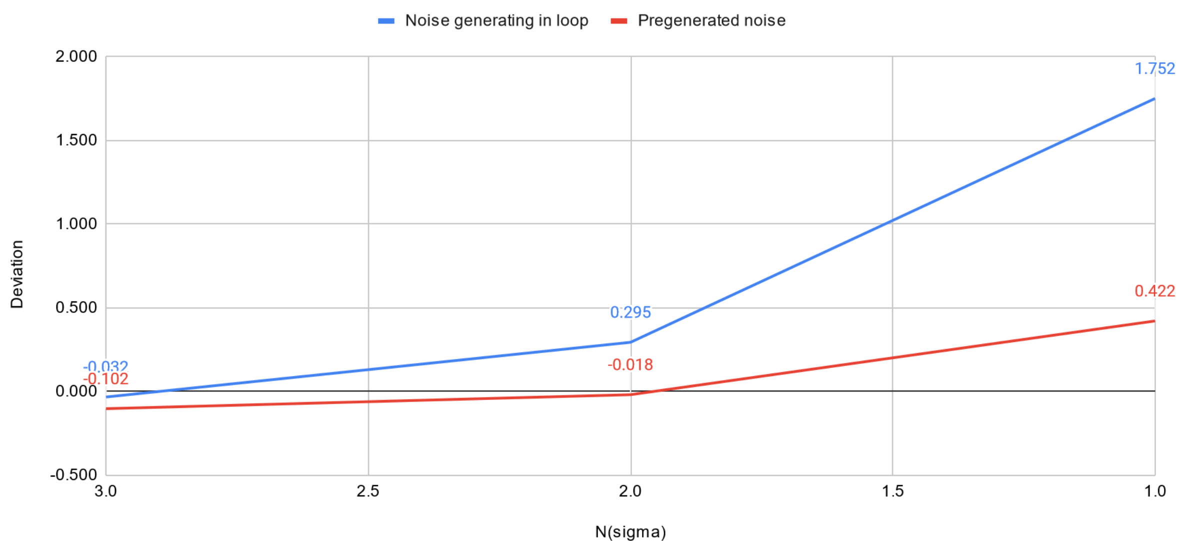


Figure 2.4.1 - Comparison of deviation of from real time pre generated noise against generating at every MoM window

N	Avg time per MAD window (microseconds)(noise replacement) for non-pre generated noise	Avg time per MAD window (microseconds)(noise replacement) for pre generated noise	Percentage improvement
3	39.647916	36.795704	7.2
2	53.03834	40.215656	24.2
1	112.702931	58.264191	48

Table 2.4.1 - Quantitative data for snode, mode - noise replacement

We can see that there is significant improvement made by this approach at lower threshold factors as shown in table 2.4.1. This approach of generating noise has been later used with AVX (Advanced Vector Extensions) instructions.

Plot for 2g.txt is shown in figure 2.4.2.

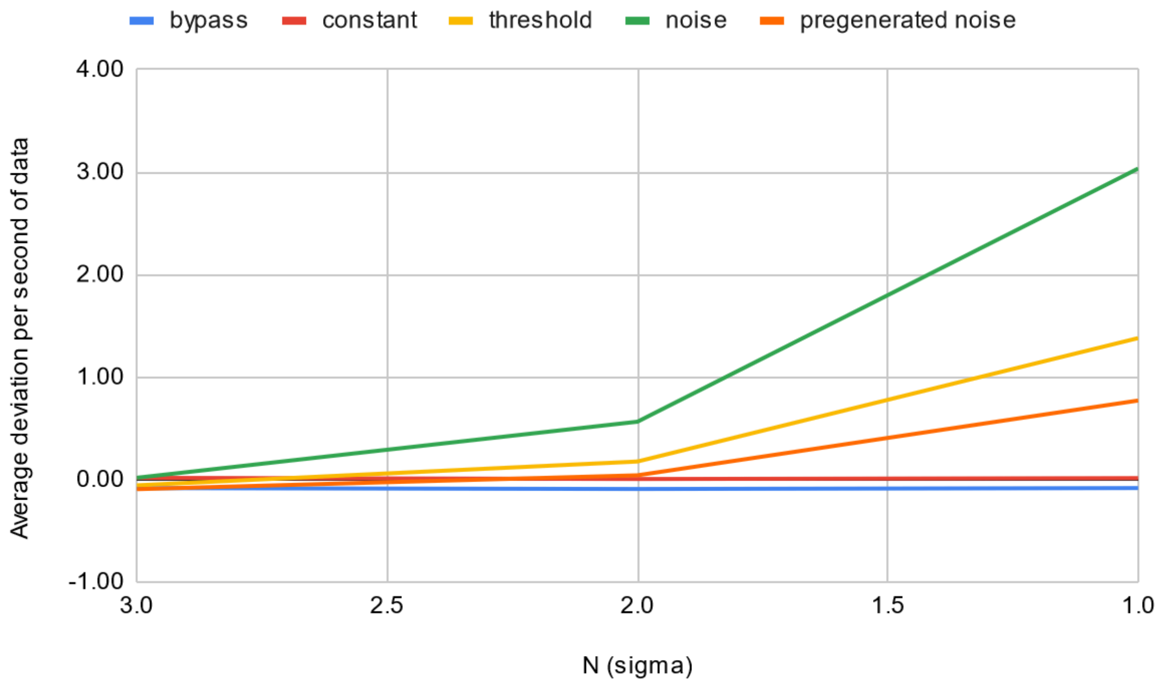


Figure 2.4.2 - Benchmarking results and comparison with 2g.txt

We can see that noise replacement with pre generated noise works better than threshold replacement.

2.5 Major bottlenecks and possible optimisations

In both of the previous codes, there is a common pattern where each element encountered in a window undergoes branch prediction based on whether it surpasses a threshold or not. The execution time of the if-else blocks varies in different replacement modes, depending on whether the element exceeds the threshold or not. Consequently, there is a reliance on the threshold factor N, introducing performance dependencies.

Branch prediction can lead to pipeline stalls, while inefficient memory access patterns and loop structures can result in significant overhead. To mitigate these issues, several optimizations can be implemented. These include code restructuring to minimise branch mispredictions, leveraging parallel computation techniques, optimising memory access by aligning data and enhancing data locality, and applying loop unrolling techniques to optimise loops. Compiler optimizations, such as utilising aggressive optimization flags, can further improve performance. By addressing these bottlenecks and employing suitable optimizations, it is possible to reduce overhead and enhance the code's execution speed.

Chapter 3 : SSE and AVX instructions based optimisations

In this chapter, we explore the use of SIMD (Single Instruction, Multiple Data), SSE (Streaming SIMD Extensions), and AVX (Advanced Vector Extensions) instructions for optimising the real-time radio frequency interference (RFI) mitigation algorithms in the context of the Giant Metrewave Radio Telescope (GMRT) and the GMRT Wideband Backend (GWB) processing system. SIMD instructions enable parallel processing by applying a single instruction to multiple data elements simultaneously, thereby improving performance by performing operations in parallel.

3.1 SIMD,SSE and AVX instructions - an overview

SIMD (Single Instruction, Multiple Data), SSE (Streaming SIMD Extensions), and AVX (Advanced Vector Extensions) are instruction sets that enable parallel processing on modern processors. SIMD is a programming paradigm that allows a single instruction to be applied to multiple data elements simultaneously. SSE introduced 128-bit registers for simultaneous operations on multiple data elements. AVX extended SSE by introducing wider 256-bit registers and additional instructions, and AVX-512 further extended it to 512 bits. These instructions enhance performance in various applications by executing operations on multiple data elements in parallel [5].

3.1.1 SSE supported by snode

In figure 3.1.1 and 3.1.2 we get output of $ cat /proc/cpuinfo | grep sse and $ cat /proc/cpuinfo | grep avx respectively

```
[rfiuser@snode ~]$ cat /proc/cpuinfo | grep sse
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush d
ts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs
bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx
 smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_
timer aes xsave avx f16c rdrand lahf_lm abm epb ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority e
pt vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid cqm xsaveopt cqm_llc cqm_occup_llc dth
erm ida arat pln pts spec_ctrl intel_stibp flush_l1d
```

Figure 3.1.1 - SSE flags in cpu

```
[rfiuser@snode ~]$ cat /proc/cpuinfo | grep avx
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush d
ts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs
bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx
 smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_
timer aes xsave avx f16c rdrand lahf_lm abm epb ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority e
pt vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid cqm xsaveopt cqm_llc cqm_occup_llc dth
erm ida arat pln pts spec_ctrl intel_stibp flush_l1d
```

Figure 3.1.2 - AVX flags in cpu

3.1.2 AVX over SSE

For our implementation, we opted for AVX instructions due to their ability to work with 256 bits [5] or 32 bytes at a time. By leveraging AVX, we can efficiently handle 32 8-bit numbers and execute operations more effectively, resulting in improved performance and faster execution times.

3.2 Converting C++ code to use AVX instructions for filtering

In this conversion, the original code snippet has been modified to utilise AVX instructions for improved performance (shown in figure 3.2.1). The purpose of the code remains the same, which is to filter and process 1 MAD window of data based on the given threshold value. The array a is a vector slice of length 1 MAD window. Major changes made are highlighted in figure 3.2.2.

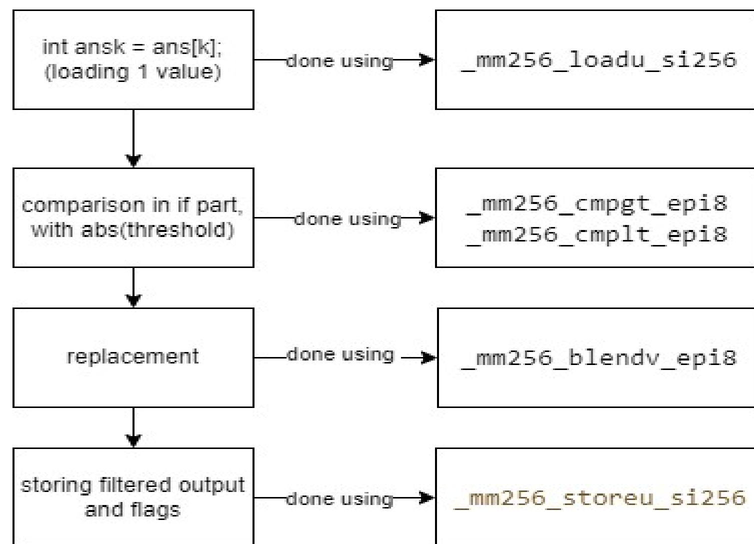


Figure 3.2.1 - Changes made to achieve same functionality as previous C++ code

```
case 1: //CONSTANT REPLACEMENT
{
    int vector_iterator=0;  //iterator is incremented by 32 in each iteration
    __m256i threshold=_mm256_set1_epi8(THRES);  //constant vector of threshold values
    __m256i th_neg=_mm256_set1_epi8((int8_t)(upper_th*(-1)));
    __m256i constant = _mm256_set1_epi8(RPL_CONST); //constant vector of replacement constant values
    for(int8_t *p=a;p<a+WINDOW_SIZE;p=p+32)
    {
        // cout<<_mm256_extract_epi8(constant,0);
        __m256i v=_mm256_loadu_si256((__m256i*) (p));  //load 32 numbers in v[iterator:iterator+31]
        __m256i cmp_gt=_mm256_cmpgt_epi8(v,threshold);//mask, of comparing values of v with threshold
        __m256i cmp_lt=_mm256_cmpgt_epi8(th_neg,v);//mask, of comparing values of v with threshold
        __m256i cmp=_mm256_or_si256(cmp_gt,cmp_lt);
        __m256i result = _mm256_blendv_epi8(v, constant, cmp);
        // result = _mm256_blendv_epi8(result, constant, cmp_lt);
        cmp=_mm256_mullo_epi8(cmp,cmp);
        _mm256_storeu_si256((__m256i*) (flag_output+vector_iterator), cmp);//storing flags
        _mm256_storeu_si256((__m256i*) (filtered_output+vector_iterator), result);//storing filtered output
        vector_iterator+=32;//incrementing counter by 32
    }
}
```

Figure 3.2.2 - Converted constant replacement mode

The optimised code uses 256-bit wide AVX registers (__m256i) to process 32 elements (8-bit numbers) in parallel [6]. Here's an explanation of the AVX implementation:

Initialization:

1. vector_iterator is an iterator variable used to track the current position within the arrays a, flag_output, and filtered_output.
2. threshold is a vector that holds the threshold value (THRES), broadcasting it to all elements of the vector.
3. th_neg is a vector that holds the negation of the threshold value (upper_th * -1), used for the lower threshold comparison.
4. constant is a vector that holds the replacement constant value (RPL_CONST), broadcasting it to all elements of the vector.

Looping: The loop iterates over the array a in steps of 32 (the width of the AVX registers). Each statement is described in table 3.2.1

Code	Description
`__m256i v = _mm256_loadu_si256((__m256i*) (p))`	Loads 32 elements from the array starting at the current position
`__m256i cmp_gt = _mm256_cmpgt_epi8(v, threshold)`	Compares each element of v with the threshold value
`__m256i cmp_lt = _mm256_cmpgt_epi8(th_neg, v)`	Compares each element of v with the negated threshold value

`__m256i cmp = _mm256_or_si256(cmp_gt, cmp_lt)`	Combines the masks from the greater-than and less-than comparisons
`__m256i result = _mm256_blendv_epi8(v, constant, cmp)`	Replaces the elements in v with the corresponding elements
`_mm256_storeu_si256((__m256i*)(flag_output+vector_iterator), cmp)`	Stores the cmp mask values in the flag_output array
`_mm256_storeu_si256((__m256i*)(filtered_output+vector_iterator), result)`	Stores the filtered output values in the filtered_output array
`vector_iterator += 32`	Advances the vector_iterator by 32 to process the next set of elements

Table 3.2.1 - Description of each line and function inside loop

Similar changes have been made for other replacement modes and are shown below(figure 3.2.3, 3.2.4, 3.2.5).

```
case 0: //BYPASS
{
    int vector_iterator=0;  //iterator is incremented by 32 in each iteration
    __m256i threshold=_mm256_set1_epi8(THRES);  //constant vector of threshold values
    __m256i th_neg=_mm256_set1_epi8((int8_t)(upper_th*(-1)));
    for(int8_t *p=a;p<a+WINDOW_SIZE;p=p+32)
    {
        __m256i v=_mm256_loadu_si256((__m256i*) (p));  //load 32 numbers in v[iterator:iterator+31]
        __m256i cmp_gt=_mm256_cmpgt_epi8(v,threshold);//mask, of comparing values of v with threshold
        __m256i cmp_lt=_mm256_cmpgt_epi8(th_neg,v);//mask, of comparing values of v with threshold
        __m256i cmp=_mm256_or_si256(cmp_gt,cmp_lt);

        cmp=_mm256_mullo_epi8(cmp,cmp);
        _mm256_storeu_si256((__m256i*) (flag_output+vector_iterator), cmp);//storing flags
        _mm256_storeu_si256((__m256i*) (filtered_output+vector_iterator), v);//storing filtered output
        vector_iterator+=32;//incrementing counter by 32
    }
}
```

Figure 3.2.3 - Converted bypass mode

```
case 2:     // THRESHOLD REPLACEMENT
    {
    int vector_iterator=0;  //iterator is incremented by 32 in each iteration
    __m256i threshold=_mm256_set1_epi8(THRES); //constant vector of threshold values
    __m256i th_neg=_mm256_set1_epi8((int8_t)(upper_th*(-1))); //constant vector of negative threshold values
    for(int8_t *p=a;p<a+WINDOW_SIZE;p=p+32)
    {
        __m256i v=_mm256_loadu_si256((__m256i*) (p));  //load 32 numbers in v[iterator:iterator+31]

        __m256i cmp_gt=_mm256_cmpgt_epi8(v,threshold);//mask, of comparing values of v with threshold
        __m256i cmp_lt=_mm256_cmpgt_epi8(th_neg,v);//mask, of comparing values of v with threshold
        __m256i cmp=_mm256_or_si256(cmp_gt,cmp_lt);


        __m256i result = _mm256_blendv_epi8(v, threshold, cmp_gt);
        result=_mm256_blendv_epi8(result,th_neg,cmp_lt);
        cmp=_mm256_mullo_epi8(cmp,cmp);
        _mm256_storeu_si256((__m256i*) (flag_output+vector_iterator), cmp);//storing flags
        _mm256_storeu_si256((__m256i*) (filtered_output+vector_iterator), result);//storing filtered output
        vector_iterator+=32;//incrementing counter by 32
    }
    }
```

Figure 3.2.4 - Converted threshold replacement mode

```
case 3: // NOISE REPLACEMENT
{
    int vector_iterator=0;  //iterator is incremented by 32 in each iteration
    __m256i threshold=_mm256_set1_epi8(THRES);  //constant vector of threshold values
    __m256i th_neg=_mm256_set1_epi8((int8_t)(upper_th*(-1))); //constant vector of negative threshold values
    for(int8_t *p=a;p<a+WINDOW_SIZE;p=p+32)
    {
        __m256i noise_v=_mm256_loadu_si256((__m256i*)(integer_noise+vector_iterator));   //loading pregenerated noise
        __m256i v=_mm256_loadu_si256((__m256i*) (p));   //load 32 numbers in v[iterator:iterator+31]
        __m256i cmp_gt=_mm256_cmpgt_epi8(v,threshold);//mask, of comparing values of v with threshold
        __m256i cmp_lt=_mm256_cmpgt_epi8(th_neg,v);//mask, of comparing values of v with threshold
        __m256i cmp=_mm256_or_si256(cmp_gt,cmp_lt);
        __m256i result = _mm256_blendv_epi8(v, noise_v, cmp);

        cmp=_mm256_mullo_epi8(cmp,cmp);
        _mm256_storeu_si256((__m256i*) (flag_output+vector_iterator), cmp);//storing flags
        _mm256_storeu_si256((__m256i*) (filtered_output+vector_iterator), result);//storing filtered output
        vector_iterator+=32;//incrementing counter by 32
    }

}break;
```

Figure 3.2.5 - Noise replacement mode, pregenerated noise

3.3 Flow of the overall code with flowchart

The program begins by parsing command-line arguments to obtain input parameters, including the input filename, window size, MoM window size, threshold factor, replacement option, and replacement constant. It then reads the input file into a vector and determines the number of slices based on the window size and data count.

To record the time with arguments, a file is opened. Average time, average filter time, and average median time variables are initialised to calculate the average times per window. A flag count variable is also initialised to keep track of the number of flags.

Arrays for filtered output, flag output, and noise generation are created. The program then iterates through each window, performing the following steps:

a. Slicing the input vector into a smaller window.
b. Calculating the median of the window using the HistoMedian function.
c. Updating the threshold based on the calculated median.
d. Applying threshold-based filtering using SIMD instructions and the chosen replacement option.
e. Updating the flag count based on the flagged elements.
f. Measuring the time taken for filtering and median calculation.
g. Storing the filtered output and flags in vectors.

After processing all windows, the program calculates and prints the average times per window for total time, filtering time, and median time. The filtered output and flags vectors are written to output files. Finally, the file recording time with arguments is closed.

The flowchart for C++ code is given in figure 3.3.1

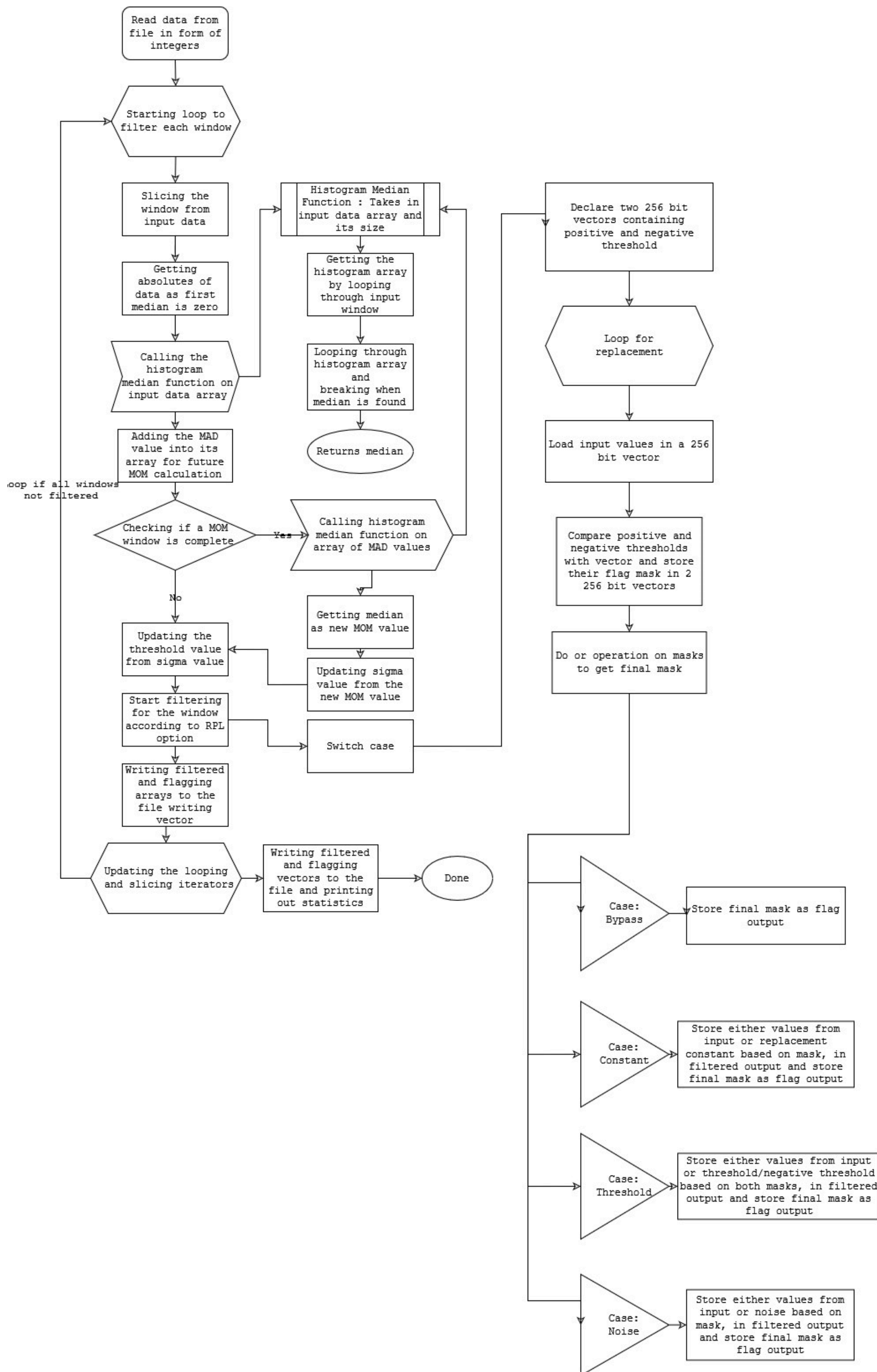


Figure 3.3.1 - Flowchart of AVX instruction based filter

3.4 Benchmarking on sample datasets

Here is a benchmarking result on 2g.txt in figure 3.4.1

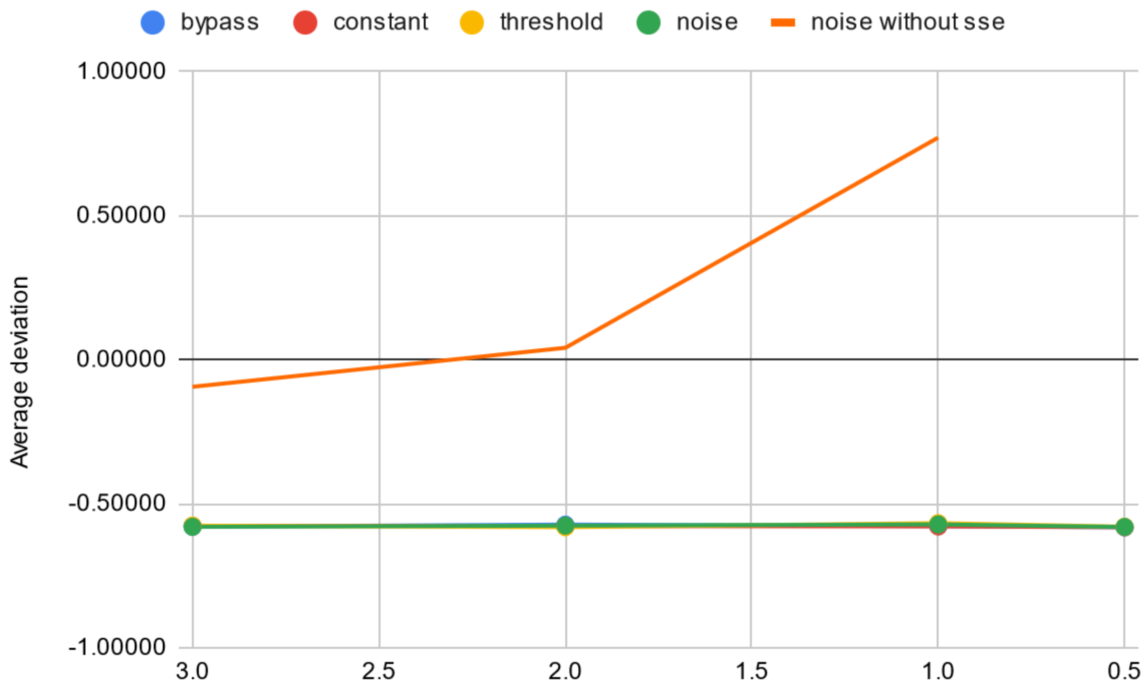


Figure 3.4.1 Benchmarking plots of 2g.txt. All other curves overlap with green one

N	Avg time taken for 1 MAD window (microsecond)(noise with AVX)	Avg time taken for 1 MAD window (microsecond)(noise without AVX)	Speedup
3	17.199279	37.111084	2.16
2	17.366871	42.654362	2.46
1	17.536086	72.543278	4.14
0.5	17.174638	NA	NA

Table 3.4.1 - Time taken for 1 MAD window for each replacement option

We can see that the improvement is drastic, with the filtering time being around 14-15 microsecond for 1 MAD window. Also we see that the dependence on threshold factor and replacement option has disappeared.

Benchmarking results on other datasets are shown in figure 3.4.2, 3.4.3, 3.4.4, 3.4.5. The bandwidth is 200MHz, that gives us a sampling rate of 400 MHz

Band 2 : 130 - 260 MHz

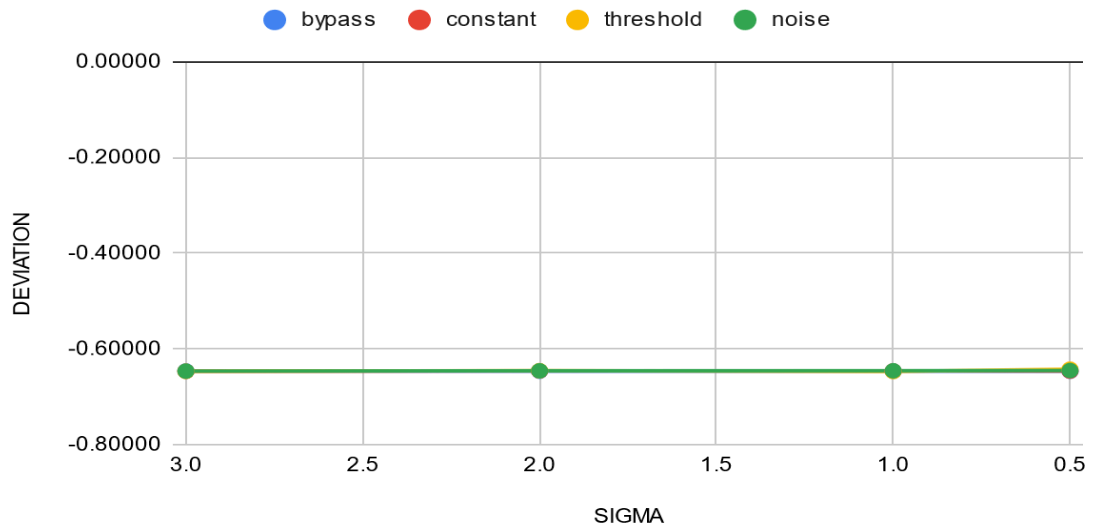


Figure 3.4.2 - Band 2 data deviation plot

Band 3 : 250 - 500 MHz

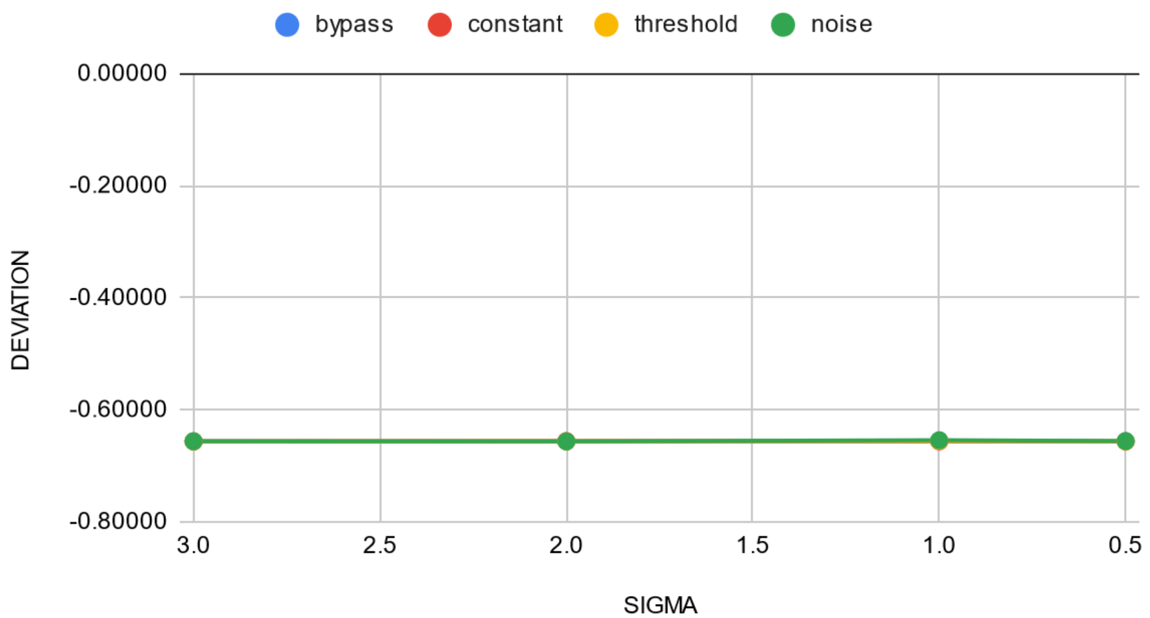


Figure 3.4.3 - Band 3 data deviation plot

Band 4 : 550 - 850 MHz

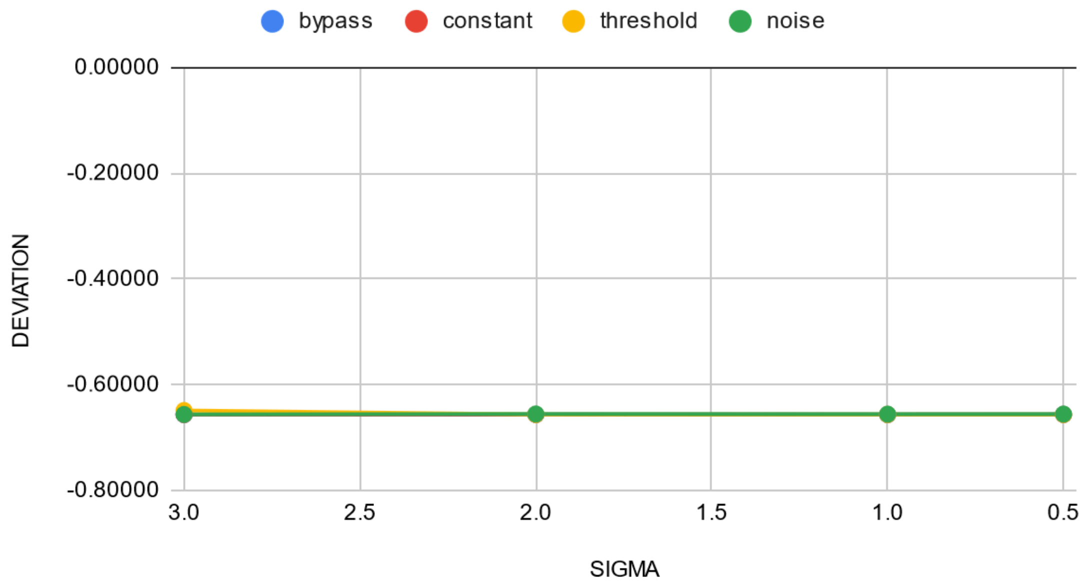


Figure 3.4.4 - Band 4 data deviation plot

Band 5: 1050 - 1450 MHz

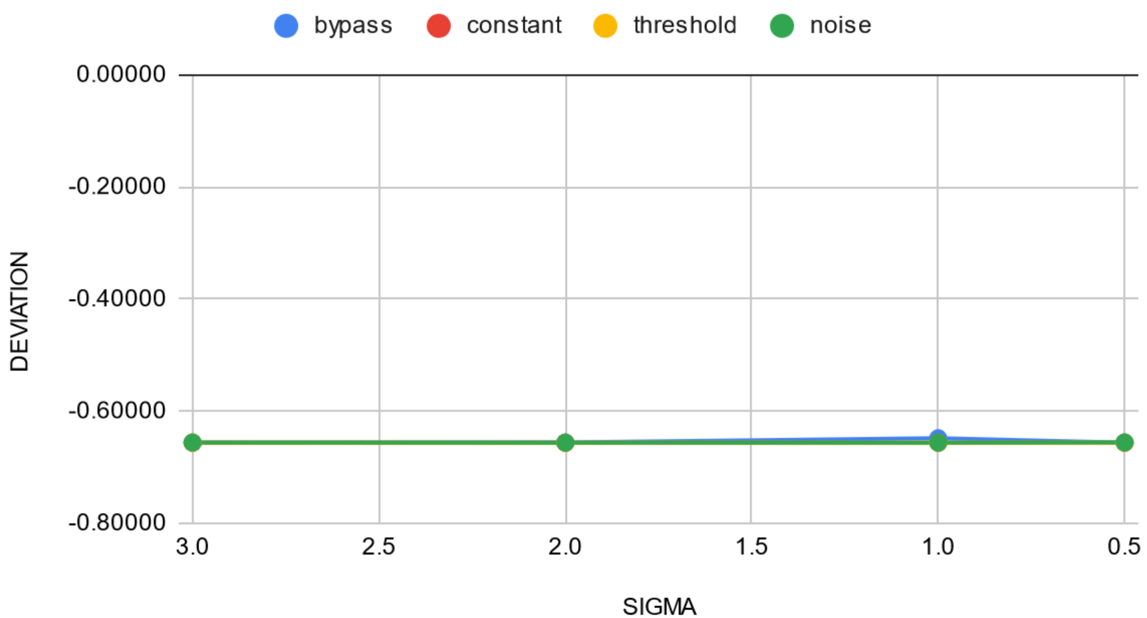


Figure 3.4.5 - Band 5 data deviation plot

The plots exhibit consistent behaviour across all frequency bands, indicating reliability in the filtering process. Additionally, the filtering operation is performed at a significantly faster

rate than real-time, showcasing its efficiency. Moreover, there is a notable independence from both the filtering option and the threshold factor, suggesting that the filtering results remain consistent regardless of the chosen parameters.

3.5 Error in previous pre generating noise function

In the noise generating function of MoM_4th sample(MoM filtering function without AVX) with pre-generated noise(figure 3.5.1), there was an error in noise generating functions, which first created noise of standard deviation 1, truncated it to integer, then scale by calculated standard deviation. This led the noise to take some discrete integer values, of non gaussian nature (figure 3.5.2).

```
int* NoiseMaker(int arr[], int len) {

    // Define random generator with Gaussian distribution
    const double mean = 0.0;
    const double stddev = 1.0;
    std::default_random_engine generator(time(0));
    std::normal_distribution<float> dist(mean, stddev);

    // Add Gaussian noise
    for (int i = 0; i < len; i++) {
        arr[i] = dist(generator);
    }
    //std::cout << oper) << std::endl;

    return 0;
}
```

Figure 3.5.1 - Previous noiseMaker with stddev declared to 1 and type cast to int

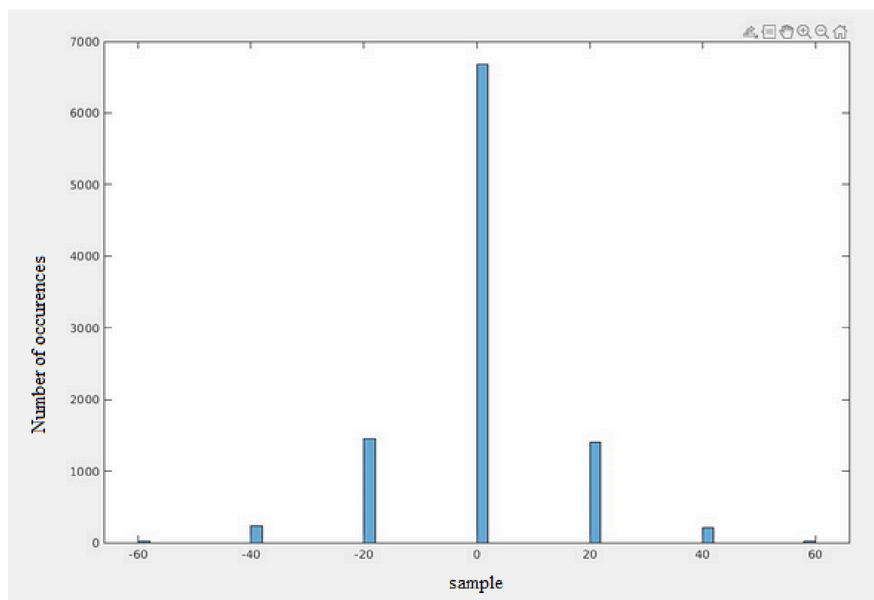


Figure 3.5.2 - Histogram of noise generated by earlier NoiseMaker function

To solve this, the noise generating function was modified such that it took the value of sigma as argument and generated the noise for each MoM window as shown in figure 3.5.3.

```
void NoiseMaker(int8_t arr[], int len, int sigma) {

    // Define random generator with Gaussian distribution
    const double mean = 0.0;
    const double stddev = sigma;
    std::default_random_engine generator(time(0));
    std::normal_distribution<float> dist(mean, stddev);

    // Add Gaussian noise
    for (int i = 0; i < len; i++) {
        arr[i] = dist(generator);
    }
    //std::cout << oper) << std::endl;

    //return 0;
}
```

Figure 3.5.3 - Modified noise function with sigma as an argument

The noise generated (figure 3.5.4) by this noise function was much more similar to Gaussian noise as verified by the kurtosis(figure 3.5.5 and figure 3.5.6).

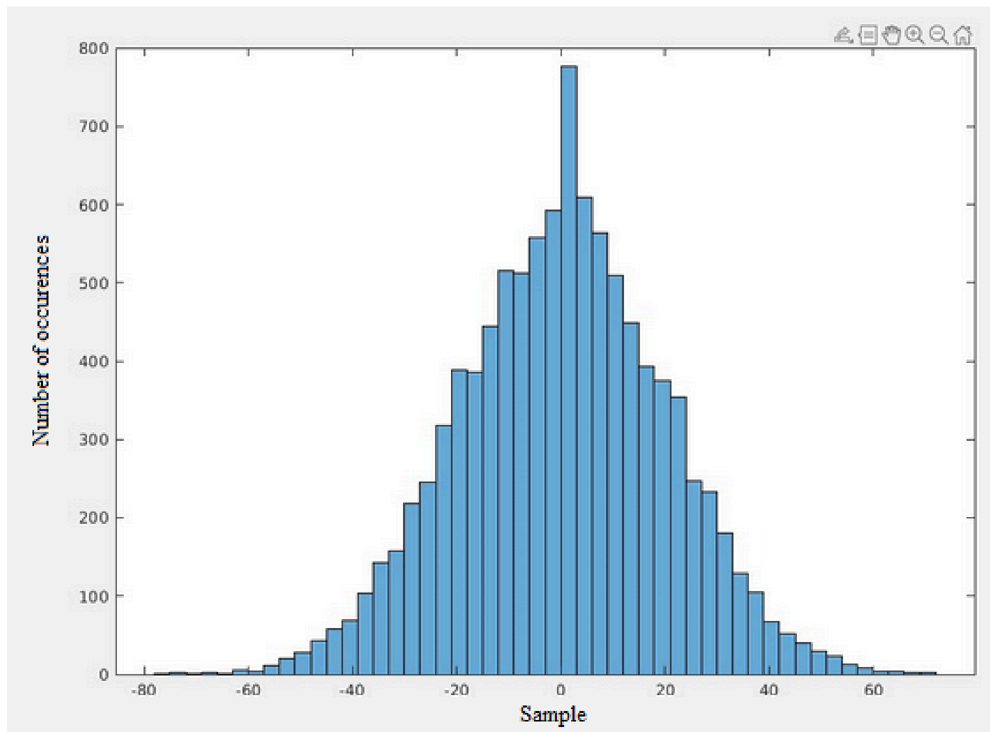


Figure 3.5.4 - Histogram of noise generated by modified NoiseMaker

```
>> kurtosis(noise2)

ans =

    5.1232
```

Figure 3.5.5 kurtosis of noise2(unmodified)

```
>> kurtosis(noise1)

ans =

    3.0468
```

Figure 3.5.6 kurtosis of noise1(modified)

3.6 Functional results

- The output of AVX code was tested against golden reference and for windows with the same threshold, the filtered data was perfectly aligning(figure 3.6.1).

Here are some plots of filtered data against raw data in figure 3.6.2 and 3.6.3

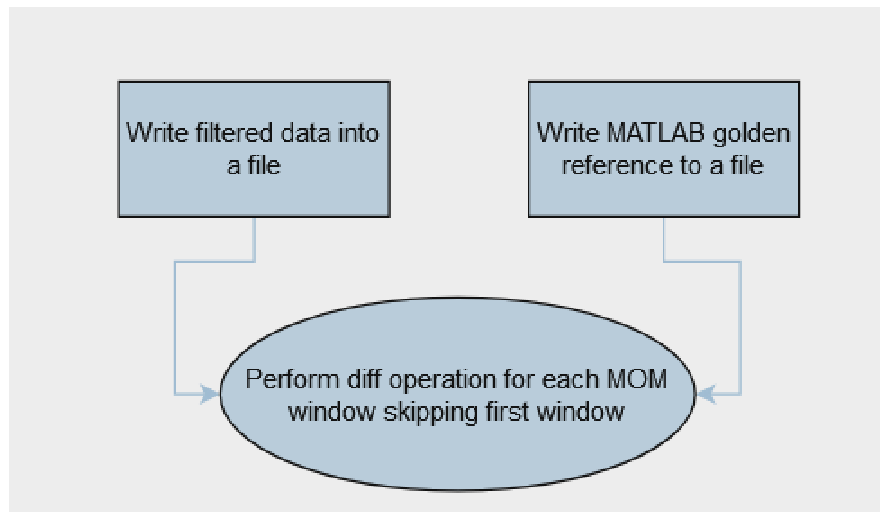


Figure 3.6.1 - Functional verification flowchart

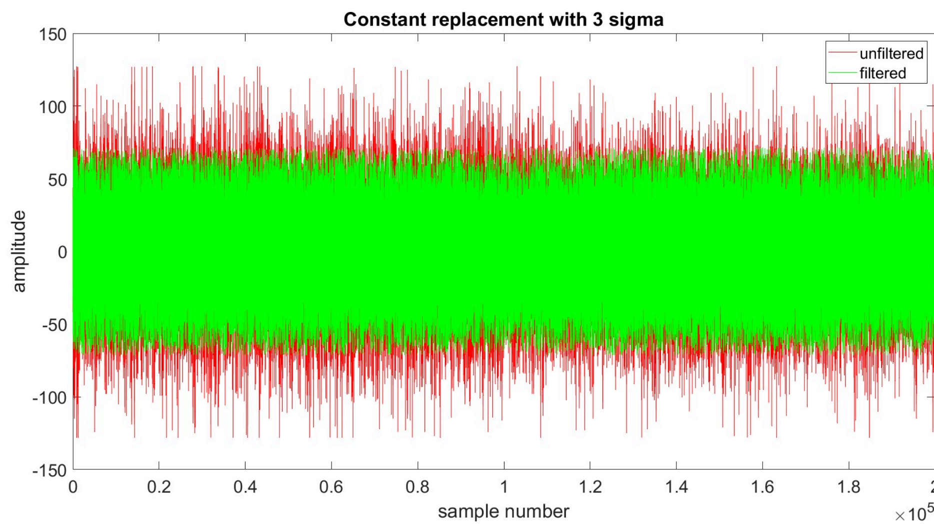


Figure 3.6.2 - Overlay plots for filtered over unfiltered signal, N =3, zero replacement

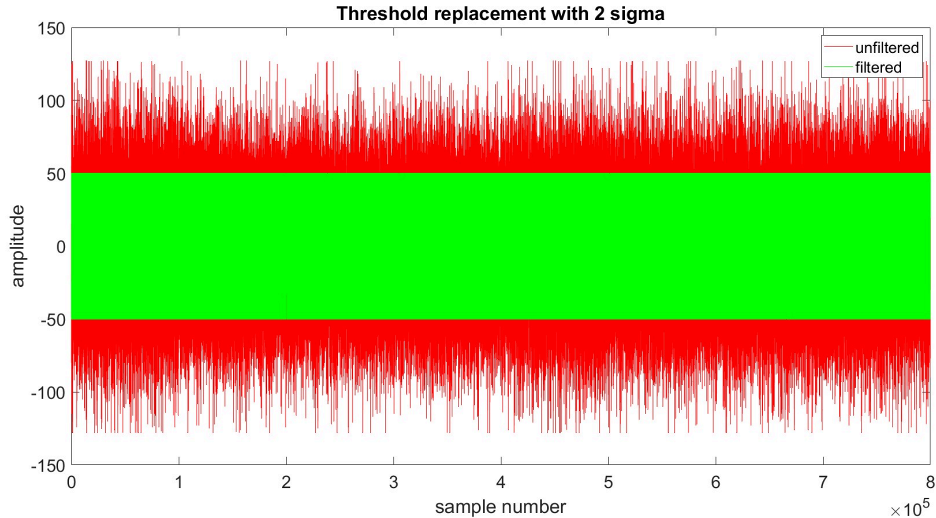


Figure 3.6.3 - Overlay plots for filtered over unfiltered signal, N =2, threshold replacement

3.7 Running on four cores via taskset

```
top - 23:08:41 up 38 days,  7:47,  3 users,  load average: 0.62, 0.32, 0.51
Tasks: 305 total,   5 running, 300 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu8  : 97.7 us,  2.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu9  : 97.0 us,  3.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu10 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu11 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu12 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu13 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu14 : 97.3 us,  2.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu15 : 97.0 us,  3.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 65556324 total, 38559036 free,  2292176 used, 24705112 buff/cache
KiB Swap: 13421772+total, 13421772+free,        0 used. 48017448 avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
129819 rfiuser   20   0  536980 292756   1056 R 100.0  0.4   0:16.43 MOM_separate+
129821 rfiuser   20   0  536980 292596   1056 R 100.0  0.4   0:16.43 MOM_separate+
129818 rfiuser   20   0  536980 289296   1056 R  99.7  0.4   0:16.43 MOM_separate+
```

Figure 3.7.1 - CPU and memory utilisation during running with taskset

We employed the "taskset" instruction, which enabled us to run the program concurrently on four different cores. Specifically, we pinned the program to cores 8, 9, 14, and 15, utilising the available resources optimally. By leveraging the "taskset" instruction, we were able to harness the parallel processing capabilities of these cores(figure 3.7.1).

Chapter 4 : Preparing the code for integration with GWB

4.1 GWB processing requirement

During the integration stage of the AVX-optimised C++ code for the RFI filter, we encountered the following challenges:

1. Compatibility with GWB Library: The GWB library or code was originally written in C and compiled using the gcc compiler, while our AVX-optimised code was written in C++ and compiled using the g++ compiler. We had to ensure compatibility between the two by making any necessary adjustments or modifications to the code or build process.
2. Double Buffer System: To optimise the performance of the filter, we implemented a double buffer system. This system allowed for simultaneous reading of the input signal from one buffer while writing the filtered signal to another buffer. Suppose we have a counter or clock or some loop iterator variable, when read occurs in (count%2)th buffer, filter and write occurs in (count+1 %2)th buffer. This approach minimised the data dependency and improved overall efficiency.
3. OpenMP Implementation: To simulate filtering on multiple antennas simultaneously, we utilised OpenMP, a parallel programming framework. By incorporating OpenMP directives into the code, we were able to parallelize the filter execution and distribute the workload across multiple threads. This resulted in improved performance by leveraging the available processing resources effectively.

Additionally, as part of the integration stage, we designed the RFI filter to be implemented as a function. This function takes input arguments, such as the input signal and filter parameters, and returns the filtered signal as output. By encapsulating the filter logic within a function, it promotes code modularity and reusability, allowing the filter to be easily integrated into larger software systems or used in different contexts with varying input data and parameters. This approach enhances the flexibility and maintainability of the RFI filter implementation.

Overall, these challenges were successfully addressed during the integration stage, ensuring compatibility with the GWB library, implementing a double buffer system, and incorporating OpenMP for parallel execution on multiple instances of the filter. Figure 4.1.1 shows the flow of program execution.

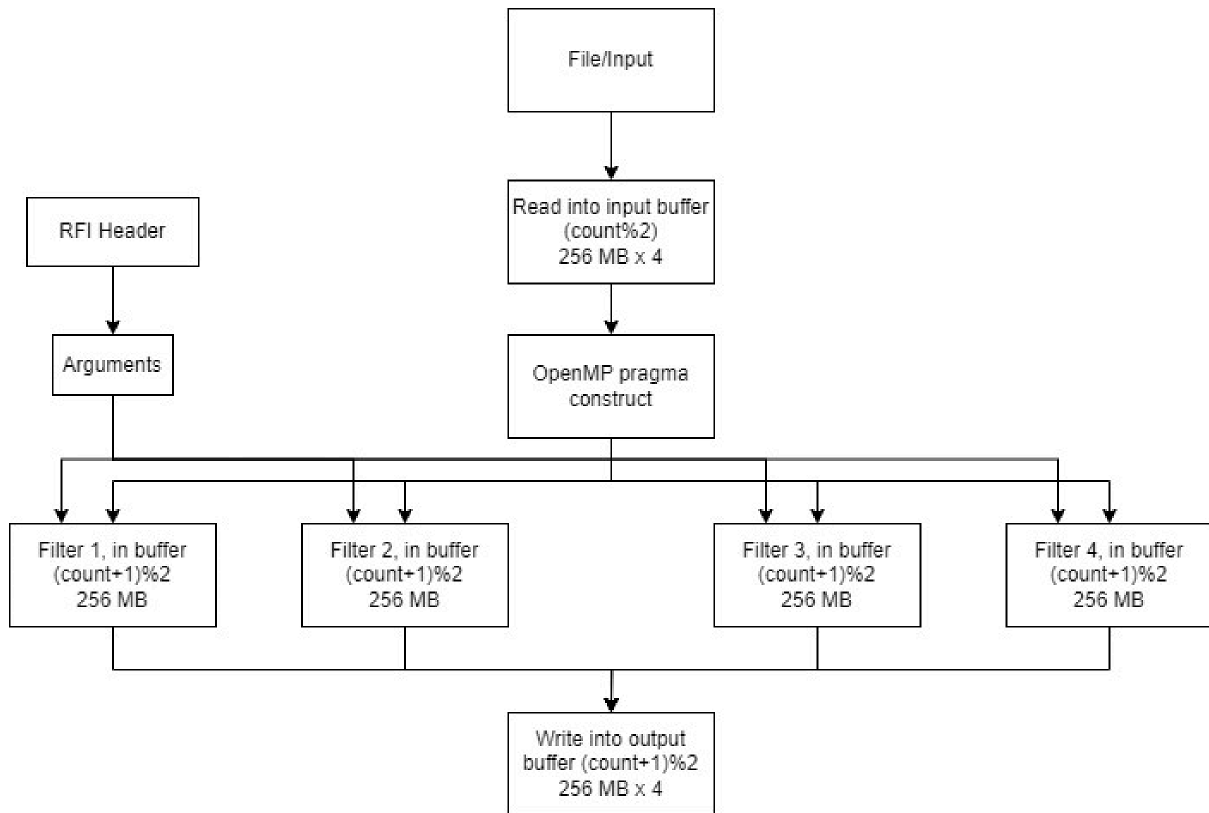


Figure 4.1.1 - Flowchart of execution sequence in the code for GWB integration

4.2 Converting C++ code to C code

The C++ code was converted to C code to ensure compatibility with a C environment and remove dependencies on C++ features.

- In order to convert the code, the C++ functions were replaced with their C equivalents and modified to work with arrays instead of vectors.
- Several helper functions, such as push_back, ReadFile, WriteFile, HistoMedian, and VectorSlice, were added and implemented to provide the required functionality.
- Dynamic memory allocation was used to create and manage arrays instead of vectors.
- Time benchmarking was performed using functions like timeval or clock() from the time.h header.

The resulting C code retains the core functionality of the original C++ code and can be compiled and executed in a C programming environment.

4.3 Reading .hdr header file to pass parameters

When working with a filtering process, the .hdr file contains the essential parameters required for the filtration. These parameters include information about input parameters for filtration like window size, threshold factor etc.

4.3.1 About.hdr file

The .hdr file serves as a container for the necessary parameters used in the filtering process. Figure 4.3.1 presents an example of the content typically found within a .hdr file.

```
# RFI FILTER SETTINGS HEADER
 GWB_VERSION          :       gwb3                  # version of the gui
 FILTERED_SIGNALS     :       RFI_ALL               # RFI_ALL=All signals filtered / RFI_CH1 =Channel 1 only / RFI_CH2=Channel 2 only
 EXTERNAL MEDIAN      :       0                     # integer value(-128 to 127)
 THRESHOLD VALUE      :       2                     # integer value(0 to 127)
 CONSTANT VALUE       :       15                    # integer value(-128 to 127)
 FILTERING OPTION     :       CONSTANT                      # BYPASS / CONSTANT=constant value is used / THRESHOLD / DIGITAL_NOISE
 DDC_STATUS           :       OFF                   #DDC_STATUS = ON/OFF
```

Figure 4.3.1 - a typical .hdr file

4.3.2 Reading the parameters from .hdr file and the code

```
typedef struct {        //Structure which contains information from header file
    char gwbVersion[500];
    char filteredSignals[500];
    int externalMedian;
    float thresholdValue;
    int constantValue;
    int filteringOption;
    char ddcStatus[500];
    int mom_win;
    int mad_win;
} RfiFilterSettings;
```

Figure 4.3.2 - structure of RfiFilterSettings

The code snippet of reader function is as follows:

```
RfiFilterSettings readRfiFilterSettings(const char* filename) {
    RfiFilterSettings settings;
    FILE* file = fopen(filename, "r");
    if (file == NULL) {
        printf("Failed to open the file '%s'.\n", filename);
        exit(1);
    }
    char line[MAX_LINE_LENGTH];
    while (fgets(line, sizeof(line), file)) {
```

```
        char* key = strtok(line, ":");
        char* value = strtok(NULL, "#");
        key = truncateWhitespace(key);
        value = truncateWhitespace(value);
        if (key != NULL && value != NULL) {
            if (strcmp(key, "GWB_VERSION") == 0)
                strcpy(settings.gwbVersion, value);
            else if (strcmp(key, "FILTERED_SIGNALS") == 0)
                strcpy(settings.filteredSignals, value);
            else if (strcmp(key, "EXTERNALMEDIAN") == 0)
                settings.externalMedian = atoi(value);
            else if (strcmp(key, "THRESHOLDVALUE") == 0)
                settings.thresholdValue = atof(value);
            else if (strcmp(key, "CONSTANTVALUE") == 0)
                settings.constantValue = atoi(value);
            else if (strcmp(key, "FILTERINGOPTION") == 0) {
                if (strcmp(value, "BYPASS") == 0)
                    settings.filteringOption = 0;
                else if (strcmp(value, "THRESHOLD") == 0)
                    settings.filteringOption = 2;
                else if (strcmp(value, "DIGITAL_NOISE") == 0)
                    settings.filteringOption = 3;
                else if (strcmp(value, "CONSTANT") == 0)
                    settings.filteringOption = 1;
            }
            else if (strcmp(key, "DDC_STATUS") == 0)
                strcpy(settings.ddcStatus, value);
            else if (strcmp(key, "MoM_WINDOW_SIZE") == 0)
                settings.mom_win = atoi(value);
            else if (strcmp(key, "MAD_WINDOW_SIZE") == 0)
                settings.mad_win = atoi(value);
        }
    }
    fclose(file);
    return settings;
}
```

Explanation to the code :

The readRfiFilterSettings function is designed to read a .hdr file containing filtering parameters and extract the information into a RfiFilterSettings structure shown in figure 4.3.2. Here's an explanation of its functioning:

1. The function takes the filename of the .hdr file as input and returns a RfiFilterSettings structure containing the extracted parameters.
2. It opens the .hdr file using the provided filename and checks if the file was successfully opened. If not, it displays an error message and exits.
3. The function reads the .hdr file line by line using fgets and stores each line in the line variable.
4. For each line, it tokenizes the line based on the ":" delimiter using strtok.
5. The key and value tokens are extracted using strtok, and any leading or trailing whitespace is removed using the truncateWhitespace function.
6. The extracted key-value pair is then processed based on the key.
7. If the key matches a specific parameter (e.g., "GWB_VERSION"), the corresponding value is assigned to the appropriate field in the RfiFilterSettings structure.
8. The process continues for each key-value pair in the .hdr file.
9. After processing all the lines, the file is closed, and the populated RfiFilterSettings structure is returned.
10. The caller can now access the extracted filtering parameters from the returned structure.

The truncateWhitespace function is a helper function used by readRfiFilterSettings to remove any whitespace characters from a given string. It iterates through the characters in the input string, copying only the non-whitespace characters to the result string. The resulting string is then null-terminated and returned.

By using the readRfiFilterSettings function, the program can easily read and extract the filtering parameters from a .hdr file, enabling the customization and configuration of the filtering process based on the specified parameters.

4.4 Double buffering of input and output buffers

In Figure 4.4.1, you can see the design of something called a double buffer. It's represented as a big array with a size of 2 times (4 times 256 times 2^{20}). This means it has four different sections, and each section has 256 rows and 2^{20} columns. Each of these sections is used for running a separate filter, and all four filters work at the same time. This arrangement helps make the processing faster and more efficient. With this setup, the filters can simultaneously analyse and process signals within the buffer. Reading and filtering occurs in separate buffers. Suppose we have a counter or clock or some loop iterator variable, when read occurs in (count%2)th buffer, filter and write occurs in (count+1 %2)th buffer.

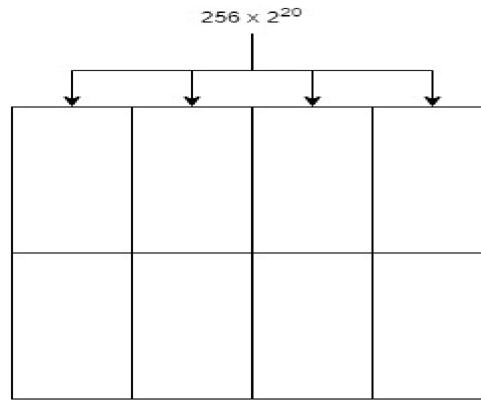


Figure 4.4.1 structure of a double buffer

The double buffering in C code is used for input and output of the signals. Figure 4.4.2 shows the snippet for allocation of memory for input and output which is double buffered. Numbers is input buffer and filtered is output buffer.

```
const int chunk_size=268435456;

int8_t** numbers;
numbers=(int8_t**)malloc(2*sizeof(int8_t*));
numbers[0]=(int8_t*)malloc(4*chunk_size*sizeof(int8_t));
numbers[1]=(int8_t*)malloc(4*chunk_size*sizeof(int8_t));

int8_t** filtered;
filtered=(int8_t**)malloc(2*sizeof(int8_t*));
filtered[0]=(int8_t*)malloc(4*chunk_size*sizeof(int8_t));
filtered[1]=(int8_t*)malloc(4*chunk_size*sizeof(int8_t));
```

Figure 4.4.2 - double buffer allocation

4.5 Implementing OpenMP for parallel filtering

Figure 4.5.1 shows the OpenMP implementation of the RFI filter. Before implementing, inside the '.bashrc' file, the GOMP_CPU_AFFINITY was added/set to 8, 9, 14, 15 to give preference to these cores as per availability. The OMP_PROC_BIND was set to true, then this '.bashrc' was sourced.

```
export GOMP_CPU_AFFINITY="8 9 14 15"
export OMP_PROC_BIND=true
```

```
do {
    ReadBinFile(binaryFile1, numbers[(count % 2)], data_len1, chunk_size);
    ReadBinFile(binaryFile2, (numbers[(count % 2)] + chunk_size), data_len2, chunk_size);
    ReadBinFile(binaryFile3, (numbers[(count % 2)] + (chunk_size * 2)), data_len3, chunk_size);
    ReadBinFile(binaryFile4, (numbers[(count % 2)] + (chunk_size * 3)), data_len4, chunk_size);
    omp_set_num_threads(4);
    #pragma omp parallel for private(cr) schedule(static)
    for (int cr = 0; cr < 4; cr++) {
        if (cr == 0) {
            filter(numbers[(count + 1) % 2],
            filtered[(count + 1) % 2],
            chunk_size,WINDOW_SIZE, MOM_WIN, N, RPL_OPTION, RPL_CONST, &threshold1);
        }
        if (cr == 1) {
            filter((numbers[(count + 1) % 2] + chunk_size),
            (filtered[(count + 1) % 2] + chunk_size),
            chunk_size, WINDOW_SIZE, MOM_WIN, N, RPL_OPTION, RPL_CONST, &threshold2);
        }
        if (cr == 2) {
            filter((numbers[(count + 1) % 2] + (2 * chunk_size)),
            (filtered[(count + 1) % 2] + (2 * chunk_size)),
            chunk_size, WINDOW_SIZE, MOM_WIN, N, RPL_OPTION, RPL_CONST, &threshold3);
        }
        if (cr == 3) {
            filter((numbers[(count + 1) % 2] + (3 * chunk_size)),
            (filtered[(count + 1) % 2] + (3 * chunk_size)),
            chunk_size, WINDOW_SIZE, MOM_WIN, N, RPL_OPTION, RPL_CONST, &threshold4);
        }
    }
    WriteFile(op_flname1, filtered[(count + 1) % 2], chunk_size);
    count++;
} while (count < chunk_n);
```

Figure 4.5.1 - OpenMP implementation

The explanation to the above implementation is given as follows.

1. #pragma omp parallel for private(cr) schedule(static): This pragma directive indicates that the following for-loop should be executed in parallel using multiple threads. The loop iterator variable cr is declared private to each thread, meaning that each thread will have its own copy of this variable. The schedule(static) clause specifies that the loop iterations should be evenly distributed among the threads.
2. The subsequent code within the for-loop corresponds to different statements executed by each thread:
   a. Statement 1: The thread with cr value 0 reads a binary file (binaryFile1) into the numbers array and performs filtering operations on the data, storing the filtered results in the filtered array.
   b. Statement 2: The thread with cr value 1 reads a different section of the binary file (binaryFile2) into the numbers array and performs filtering operations on this section, storing the filtered results in the corresponding section of the filtered array.
   c. Statement 3: The thread with cr value 2 reads another section of the binary file (binaryFile3) into the numbers array and performs filtering operations on this section, storing the filtered results in the corresponding section of the filtered array.
   d. Statement 4: The thread with cr value 3 reads the remaining section of the binary file (binaryFile4) into the numbers array and performs filtering operations on this section, storing the filtered results in the corresponding section of the filtered array.

3. The code then proceeds to write the filtered data from the filtered array into separate output files (op_flname1, op_flname2, op_flname3, op_flname4) using the WriteFile function.
4. The loop continues until the condition count<chunk_n is met, where chunk_n represents the number of iterations predicted in the data(this is later replaced by a subroutine which exits the program at the end of input). This ensures that all data chunks are processed.

4.6 Functional test results

The output of openMP code was tested with MATLAB golden reference and it was plotted against it for 3 sigma zero replacement(Figure 4.6.1) and 2 sigma threshold replacement(Figure 4.6.2).

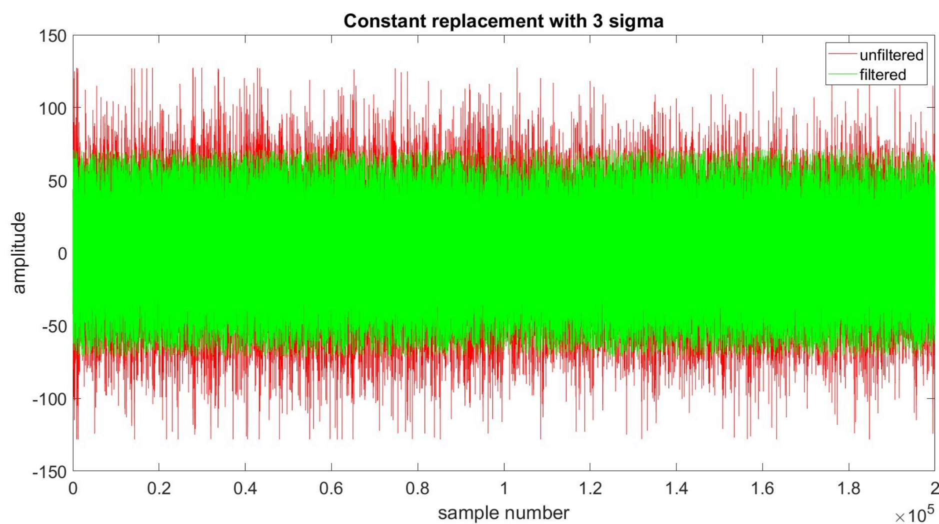


Figure 4.6.1 - Functional results with constant replacement, N = 3

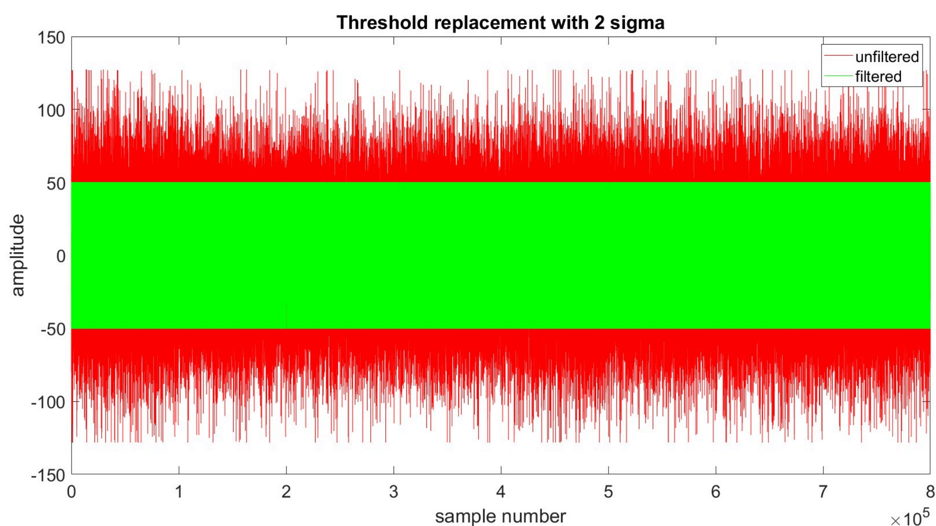


Figure 4.6.2 - Functional results with threshold replacement, N = 2

The flags are also being compared with golden reference apart from the filtered data.

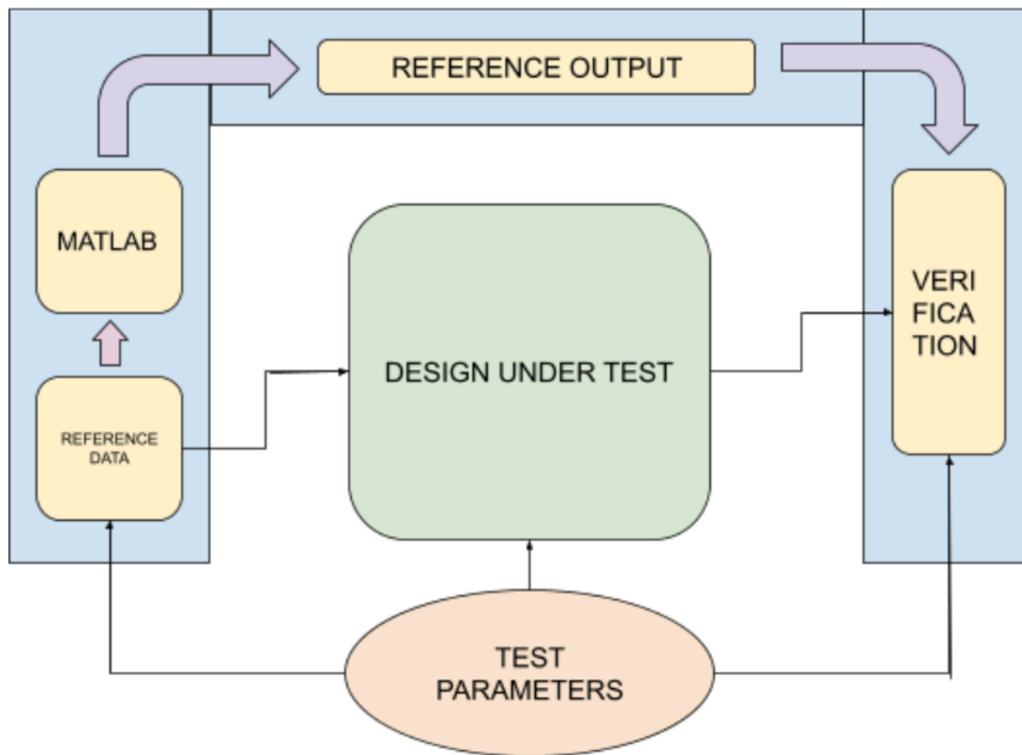


Figure 4.6.3 - Flowchart of functional verification procedure [4]

Verification is done using 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.This is used to compare the reference output and flags from MATLAB against the output and flags generated by the design under test to analyse both outputs (figure 4.6.3).

Chapter 5 : Exploring Power Detection Technique

The power detection technique works on improved SNR as it squares and averages the time-domain signal and decision is made on the averaged power signal. The number of samples to be averaged are optimally arrived at considering the typical duration of RFI burst from the power-line sparking instances.

5.1 Working of the code

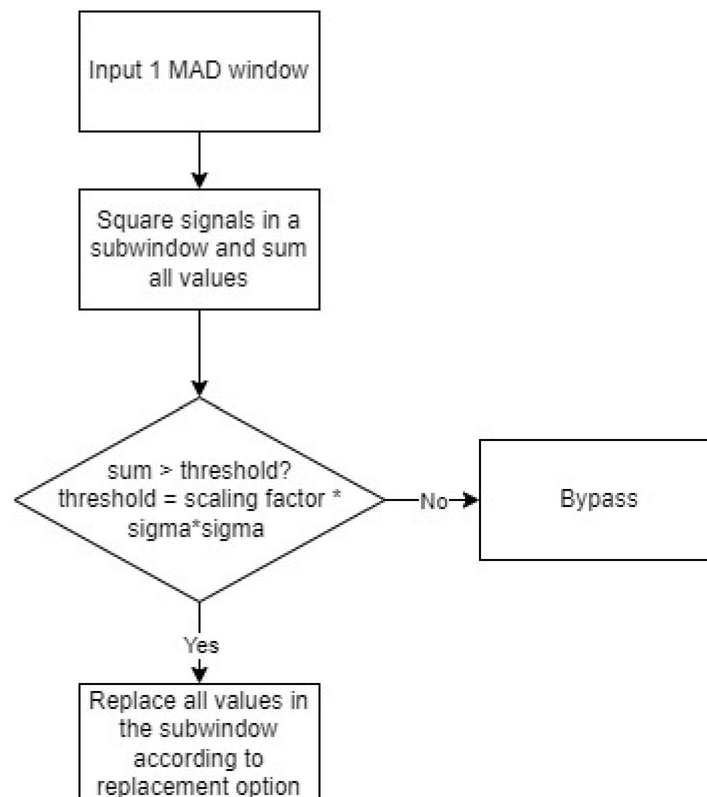


Figure 5.1.1 basic working of power detection

Figure 5.1.1 shows the basic flow of execution of the code. The parameters included file address, MAD computation window length, MoM computation window length, subwindow size, scaling factor (determines aggressiveness of filter) , replacement option and replacement constant for constant replacement.

5.2 Earlier implementation - with pre generated noise

Figure 5.2.1 shows the implementation, with noise replacement. The code, based on the replacement option, performs filtering on sub-windows within the current window. If the accumulated power in a sub-window exceeds the upper threshold, the code replaces the sub-window values with the replacement constant and sets the flag output accordingly. The flag count is updated accordingly.

```
case 1:
    for(int j = 0; j < n_sub; j++) {
        //Going through every element in the sub window
        int st_index = j*sub_window_size;           //Points to the starting value
        int data_accum = 0;
        //Applying power detection algorithm [getting sum(x^2)]
        for(int k = st_index; k < st_index + sub_window_size; k++) {
            data_accum += arr[k]*arr[k];
        }
        //Filtering out the window and adding flags if it's out of threshold
        if(data_accum > upper_th) {
            for(int k = st_index; k < st_index + sub_window_size; k++) {
                //int noise = dist(generator)*sigma;
                filtered_output[k] = abs(noise_window[k]*sigma) > 127 ? ((-1)^k)*127 : noise_window[k]*sigma;;
                flag_output[k] = 1;
            }
            //Updating the entire sub window's flag counter at once
            flag_count += sub_window_size;
        } else {
            for(int k = st_index; k < st_index + sub_window_size; k++) {
                filtered_output[k] = ans[k];
                flag_output[k] = 0;
            }
        }
    }
    break;
```

Figure 5.2.1 - Noise replacement in power detection technique

5.3 Functional testing - against voltage based MoM filter

The filtered data was compared with the data filtered by voltage based MoM filter for different subwindows and a window size of 1024 (figure 5.3.1). Also, it was benchmarked with a window size of 16384 and different subwindows for noise replacement.

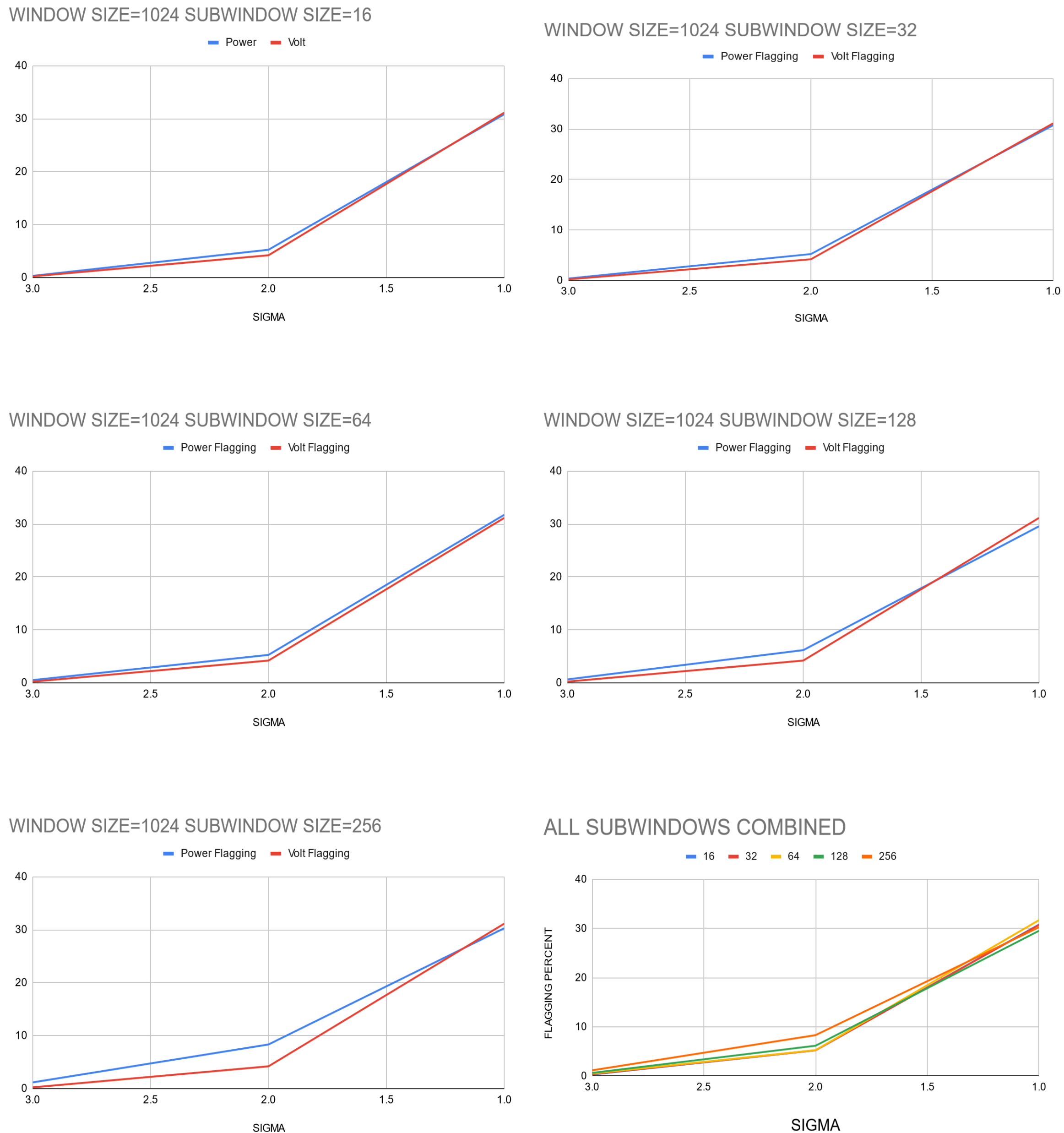


Figure 5.3.1 - Flagging percentage for various threshold factor of power detection filter compared against voltage based filter

The detector operates on chi-squared distribution (sum of square of Gaussian distributed samples). The squaring and accumulation operation on raw voltage samples results in a central chi-square distribution whose mean and variance would depend on that of the voltage domain data and the number of samples accumulated. For a large number of samples (n), the distribution converges towards the Gaussian distribution.

The relationship between the mean (μ_p) and standard deviation (σ_p) in the chi-square domain as computed from the respective mean (μ) and standard deviation (σ) in the time-domain is given by

$$\mu_p = n\sigma^2$$

$$\sigma_p = \sqrt{(2n)}.\sigma^2$$

Thus the detection threshold (γ) in terms of μp and σp can be given as

$$\gamma = \lambda\sigma^2(n + \sqrt{(2n)})$$

where λ is the aggressiveness parameter. This form can also be arrived at by referring to the chi-square statistics table for a given n and λ. Thus if the square and accumulated sample is greater than γ, it is detected as RFI and the entire block of n samples is flagged as RFI.

$$\sum_{k=1}^{n} |X_k|^2 > \gamma$$

The value of n depends on the number of samples that a typical RFI burst corrupts (depending on the duration of burst and the sampling rate of the system).

5.4 Benchmarking results

The code was benchmarked using GMRT band-3 data(1g.txt). The window size used was 16384 with the option of noise replacement. Figure 5.4.1 demonstrates the plots of deviation from real time in 1 second versus sigma.

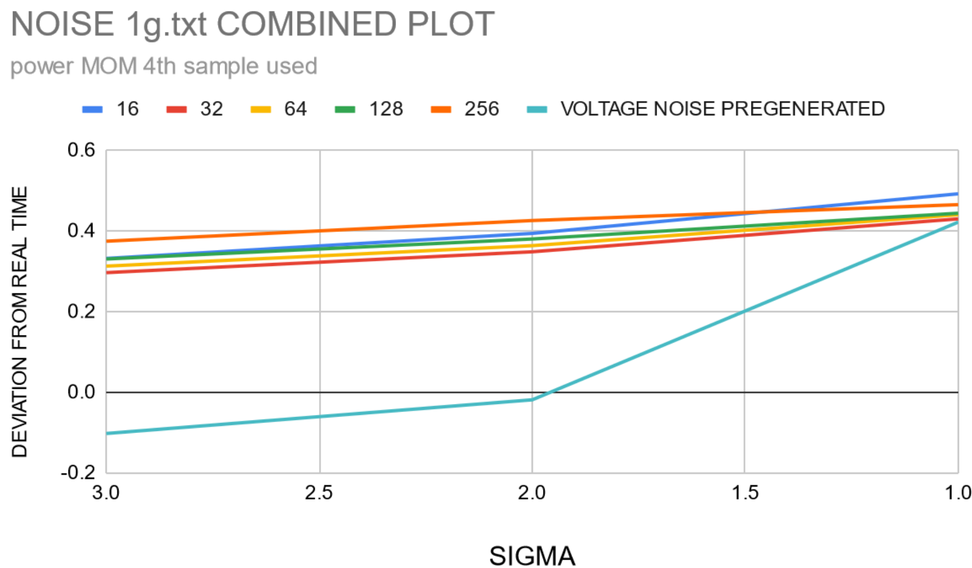


Figure 5.4.1 - Comparison of deviation with N for subwindow of 16,32,64,128,256. Window size is 1024.

As we can see this code runs much slower than the pre-generated noise version, for all threshold factors. This can be accounted for squaring the signal multiple times and adding all those values, in a subwindow; along with replacing all values in the sub window in case the accumulated sum turns out to be greater than threshold.

Chapter 6 : Power Detection Technique using SSE and AVX instructions

---

The preceding approach to noise filtration, which utilises power for its implementation, can be enhanced in CPU performance by incorporating AVX instructions. This optimization closely resembles the methodology employed in the earlier chapters for voltage-based noise filters.

6.1 Converting the code

```
long data_accum = 0;    // LONG DATA ACCUMULATOR
for(int16_t *k=b;k<b+sub_window_size;k=k+16)    //PASSING THROUGH EACH 16 NUMBERS PARTITION
{
    __m256i v=_mm256_loadu_si256((__m256i*) (k));
    __m256i v_sq = _mm256_mullo_epi16(v, v);   //SET OF 7 INSTRUCTIONS PERFORMING HORIZONTAL ADDITION OF
NUMBERS IN THE VECTOR v_sq
    __m256i v_sq1 = _mm256_cvtepi16_epi32(_mm256_extracti128_si256(v_sq, 0));
    __m256i v_sq2 = _mm256_cvtepi16_epi32(_mm256_extracti128_si256(v_sq, 1));
    __m256i sum1 = _mm256_add_epi32(v_sq1,v_sq2);
    sum1 = _mm256_hadd_epi32(sum1, zero0);
    sum1 = _mm256_hadd_epi32(zero0, sum1);
    sum1 = _mm256_hadd_epi32(sum1, zero0);
    __m128i w = _mm_add_epi32(_mm256_extracti128_si256(sum1, 0), _mm256_extracti128_si256(sum1, 1));
    //ACCUMULATING SUM
    data_accum += _mm_extract_epi32(w,1);
}
```

Figure 6.1.1 Snippet of code highlighting the data accumulation part in a subwindow

In contrast to the previous iteration of power-based noise detection, the data accumulation segment of the code has evolved into a more intricate set of instructions. In each cycle of the for loop illustrated in Figure 6.1.1, the code now operates on 16 numbers simultaneously. Each cycle of the for loop in figure 6.1.1 works on 16 numbers at a time.

Unlike our earlier approach in the voltage-based noise filter, we refrained from processing 32 numbers. Instead, we opted for a different strategy, aiming to accumulate 16-bit squares of numbers within a 256-bit AVX vector. Additionally, there has been a shift in using the "long" data type for the variable "data_accum."

```
if (data_accum > upper_th)  // IF ACCUMULATED DATA IS GREATER THAN THRESHOLD
{
    for (int k = 0; k < sub_window_size; k += 16)   //REPLACE WITH CONSTANT
    {
        __m256i noise_v=_mm256_loadu_si256((__m256i*)(integer_noise + k + st_index));
        _mm256_storeu_si256((__m256i *)(filtered_output_oneWindow + st_index + k), noise_v);
        _mm256_storeu_si256((__m256i *)(flag_output_oneWindow + st_index + k), one1);
    }
    // Updating the entire sub window's flag counter at once
    flag_count += sub_window_size;
}
else
{
    for (int k = 0; k < sub_window_size; k += 16)
    {
        __m256i v = _mm256_loadu_si256((__m256i *)(&b[k]));
        _mm256_storeu_si256((__m256i *)(filtered_output_oneWindow + st_index + k), v);
        _mm256_storeu_si256((__m256i *)(flag_output_oneWindow + st_index + k), zero0);
    }
}
```

Figure 6.1.2 - Noise filter with constant replacement snippet

The filter and replacement part is comparatively simpler, as we do not need to compare each data point and we can replace entire subwindows at once.

6.2 Functional test results

Extensive testing was conducted on the code using established datasets, and a comprehensive side-by-side comparison was performed with the earlier C++ code. The results from both implementations were found to be precisely identical, affirming the consistency of the newer version with its predecessor in C++.

Conclusions and future scope

- This project provided a comprehensive examination of the implementation and real-time performance enhancement of real-time radio frequency interference (RFI) mitigation algorithms within the Giant Metrewave Radio Telescope (GMRT) and the GMRT Wideband Backend (GWB) processing system.
- The project helped in enhancing the real-time efficiency and performance of RFI mitigation techniques by leveraging SSE instructions and other optimizations. The report presents the results of benchmarking, functional testing, and the integration of these techniques with the GWB processing system.
- However, the integration process has posed challenges, particularly regarding the inability to fully utilise cores with OpenMP. Nonetheless, there is ample room for improvement in this aspect.
- We observed that upgrading the compiler from version 4.8.5 to gcc 11 yielded significantly better performance, with the filtering time for a single MAD window dropping to less than 5 microseconds, compared to 17 microseconds in gcc 4.8.5. Upgrading the compiler can bring benefits to filtering and other operations.
- The next steps would be to successfully integrate the design with the GWB and carry out detailed testing. Also, the mechanism of treating the flags can be looked at.

References

[1] Low-Frequency Radio Astronomy
http://www.ncra.tifr.res.in/ncra/gmrt/gmrt-users/low-frequency-radio-astronomy

[2] Kaushal D. Buch, Kishor Naik, Swapnil Nalawade, Shruti Bhatporia, Yashwant Gupta and B. Ajithkumar, "Real-Time Implementation of MAD-Based RFI Excision on FPGA", 2019 Radio Frequency Interference (RFI), 2016, pp. 11-15, doi: 10.1109/RFINT.2016.7833523.

[3] Suda Harshavardhan Reddy, Sanjay Kudale1 ,Upendra Gokhale, Irappa Halagalli, Nilesh Raskar ,Kishalay De, Shelton Gnanaraj, Ajith Kumar B and Yashwant Gupta, "A Wideband Digital Back-End for the Upgraded GMRT", Journal of Astronomical Instrumentation, Vol. 6, No. 1 (2017) 1641011

[4] Shubham Balte, "Implementation Of Mad-Based Rfi Filtering Algorithm On CPU And GPU", NCRA STP Report, July 2022

[5] Intel® Instruction Set Extensions Technology
https://www.intel.in/content/www/in/en/support/articles/000005779/processors.html

[6] Intel Intrinsics Guide
https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=SSE_ALL

[7] Mastering OpenMP Performance
https://www.openmp.org/wp-content/uploads/openmp-webinar-vanderPas-20210318.pdf

[8] A "Hands-on" Introduction to OpenMP
https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf

Appendix

Final versions

File Names	Description
c_version_rfi.c	Runs on single core, without any double buffering
double_buff_rfi.c	Single core, uses double buffer
5_release…	Has.hdr reader implemented, double buffer, with openMP , to run on 4 cores simultaneously

rfiuser@snode specifications

```
processor       : 15
vendor_id       : GenuineIntel
cpu family      : 6
model           : 63
model name      : Intel(R) Xeon(R) CPU E5-2667 v3 @ 3.20GHz
stepping        : 2
microcode       : 0x3d
cpu MHz         : 3196.450
cache size      : 20480 KB
physical id     : 1
siblings        : 8
core id         : 7
cpu cores       : 8
apicid          : 30
initial apicid  : 30
fpu             : yes
fpu_exception   : yes
cpuid level     : 15
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse
sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc ap
erfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4
_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm epb ssbd ibrs ibpb stibp tpr_shadow vnmi
 flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid cqm xsaveopt cqm_llc cqm_occup_llc dtherm id
a arat pln pts spec_ctrl intel_stibp flush_l1d
bogomips        : 6397.30
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

List of generic parameters

Parameter	**Data type**	**Range or constraints**	**Usual values**
MAD window size	int	> 32	16384, less common - 1024, 4096
MoM window size	int	>1	16384
Threshold factor	float	> 0	3, 2, 1, 0.5
Replacement option	int	{0, 1, 2, 3}	1, 3
Replacement constant	Int - 8 bit	-128 to 127	0
Subwindow size for power detection	int	Should divide MAD window and > 1	16, 32, 64, 128, 256

Codes (C++ and C)

MoM_4thsample.cpp

```
/*
This code does MOM filtering while assuming the first median to be zero
to save computation
*/

/*
This version will be assuming first median to be 0 always.
It also includes the configuration options being given from a header
file instead of the command line.
It will be optimized with OpenMP and -O3 level ove optimizations from
the compiler directives.

export GOMP_CPU_AFFINITY="2 3 11 12 4 5 6 7 13 5"
*/

#include<iostream>
#include<bits/stdc++.h>
#include<chrono>
#include<fstream>
#include<vector>
#include<iterator>
#include<random>
```

```
using namespace std;
using namespace std::chrono;

//*********************** File reader and writer functions
*****************************************//

vector<int> ReadFile(string &name,int &cnt) {
    //Making an fstream object
    fstream myFile;
    //Generating input data vector
    vector<int> input_data;
    //Opening the file in input mode
    myFile.open(name,ios::in);
    //Checking if file is open, and then storing it into the vector
    if(myFile.is_open()) {
        int input_buffer;
        //getline() only stores in strings, hence the conversion
        while(myFile >> input_buffer) {
            input_data.push_back(input_buffer);
            cnt++;
        }
    }
    myFile.close();
    return input_data;
}

void WriteFile(vector<int> &write_data,string name,int &data_length) {
    //Making an fstream object
    fstream myFile;
    //Opening the file in write mode
    myFile.open(name,ios::out);
    //Checking if file is open, then writing to it
    if(myFile.is_open()) {
        for(int i = 0; i < data_length;i++) {
            myFile << write_data.at(i) << endl;
        }
    }
    myFile.close();
}

//********************** Slicing the vector into windows
*****************************************//
```

```
vector<int> VectorSlice(vector<int> &v,int X, int Y) {
    auto first = v.cbegin() + X;
    auto last = v.cbegin() + Y + 1;

    std::vector<int> vec(first, last);
    return vec;
}

// ******************** Histogram method for median calculation
*******************************//

//Function to calculate the median through histogram - inputs are the
array and it's length
int HistoMedian(int input_arr[], int len) {

    //Making an array to store frequencies, using indexes as data
points (0-128 as it's absolutes of signed 8 bit data)
    int histo_array[129] = {};

    //Sifting through the input array and incrementing histogram array
counters
    for(int i = 0; i < len; i = i + 4) {
        //Increasing the frequency counter of the histogram array acc
to input
        histo_array[input_arr[i]]++;
    }

    //Calculating the median from the histogram array
    int sum = 0;
    for (int i = 0; i < 129; i++) {
        sum = sum + histo_array[i];
        //Return median if the cumulative sum >= len/2 and break the
loop
        if(sum > len/8) {
            return i;
        }
    }
    return 0;
}

//*********************** MAD Calculation
**********************************//
int main(int argc, char *argv[]) {
```

```
    //************************* Getting input parameters from command
line ********************************//

    string INPUT_FILENAME = argv[1]; //Name of the file with data
    const int WINDOW_SIZE = stoi(argv[2]);   //Window Size for all
methods
    const int MOM_WIN = stoi(argv[3]);          //Number of windows for
MOM calculation
    const int N = stoi(argv[4]);                //Threshold factor  (th
= sigma*N)
    const int RPL_OPTION = stoi(argv[5]);       //% 0 = bypass; 1 =
constant; 2 = threshold; 3 = digital noise
    const int RPL_CONST = stoi(argv[6]);        //Constant to be
replaced by in rpl = 1

    //Getting the whole data vector
    vector<int> numbers;
    int data_count = 0;
    numbers = ReadFile(INPUT_FILENAME,data_count);

    //Getting a variable for avg time
    long double avg_time = 0, avg_filter =0, avg_mediantime = 0;

    //Getting a count for number of flags
    long int flag_count = 0;

    //Calculating number of slices
    int n_slices = data_count/WINDOW_SIZE;

    //Making a filtered output and flagging array
    int filtered_output[WINDOW_SIZE];
    int flag_output[WINDOW_SIZE];

    //Making a vector for writing to file
    vector<int> filtered;

    //Making the MAD buffer array to calculate MOM value
    int mad_buffer[n_slices];

    //Initiating a counter for slicing
    int start = 0;
```

```
    //Declaring threshold variables for the windows without MOM filter
(first MOM_WIN windows)
    float sigma;
    float upper_th,lower_th;
    upper_th = 127;

    //Main loop for iterating through each window
    for(int i = 0;i < n_slices ; i++) {
        //Making a vector for the window
        vector<int> ans;
        //cout << "start value is - " << start << "----------------||"
<< endl;
        //cout << "End value is - " << start + WINDOW_SIZE - 1 <<
"---------"<< i + 1 << "------||" << endl;
        ans = VectorSlice(numbers,start,start + WINDOW_SIZE - 1);

        //Starting the stopwatch for timer of current window
        auto start_time = high_resolution_clock::now();

        //Making the array to pass for median (directly taking
absolutes as first median is 0)
        int arr[WINDOW_SIZE];
        for(int j = 0; j < WINDOW_SIZE; j++) {
            arr[j] = abs(ans[j]);
        }

        //Starting stopwatch for median calculation
        //auto start_med = std::chrono::high_resolution_clock::now();
        //Adding the MAD value into buffer
        mad_buffer[i] = HistoMedian(arr,WINDOW_SIZE);
        //Stopping timer and calculating median time required
        //auto stop_med= std::chrono::high_resolution_clock::now();

        //Updating the sigma value for threshold calculations, if we
reach the next MOM window
        if((i+1)%MOM_WIN == 0) {

            //Making an array for MOM calculation from mad_buffer (in
short, slicing)
            int mom_buffer[MOM_WIN];
            for(int m = 0; m < MOM_WIN; m++) {
                mom_buffer[m] = mad_buffer[i + 1 - MOM_WIN + m];
            }
```

```
            //Getting MOM value
            //Calculating the median of the data
            int mom_value = HistoMedian(mom_buffer,MOM_WIN);
            //Calculating the thresholds
            sigma = 1.4826*mom_value;
        }

        //Updating the upper and lower thresholds according to the last
MOM sigma
        //Done after the first MOM window has passed, to apply these to
second one.
        if(i + 1 >= MOM_WIN) {
            upper_th = + N*sigma;
        }

        //Starting filtering timer
        //auto start_filter =
std::chrono::high_resolution_clock::now();

        //************************ Filtering with switch case
**************************************//
        switch(RPL_OPTION) {
            case 0:
                for(int k = 0; k < WINDOW_SIZE; k++) {
                    int ansk = ans[k];
                    if(abs(ansk) <= upper_th) {
                        //Adding the filtered value to the output array
                        filtered_output[k] = ansk;
                        flag_output[k] = 0;
                    } else {
                        //Increasing flag counter
                        flag_count++;
                        //Adding the filtered value to the output array
                        filtered_output[k] = ansk;
                        flag_output[k] = 1;
                    }
                }
                break;
            case 1:
                for(int k = 0; k < WINDOW_SIZE; k++) {
                    int ansk = ans[k];
                    if(abs(ansk) <= upper_th) {
```

```
                    //Adding the filtered value to the output array
                    filtered_output[k] = ansk;
                    flag_output[k] = 0;
                } else {
                    //Increasing flag counter
                    flag_count++;
                    //Adding the filtered value to the output array
                    filtered_output[k] = RPL_CONST;
                    flag_output[k] = 1;
                }
            }
            break;
        case 2:
            for(int k = 0; k < WINDOW_SIZE; k++) {
                int ansk = ans[k];
                if(abs(ansk) <= upper_th) {
                    //Adding the filtered value to the output array
                    filtered_output[k] = ans[k];
                    flag_output[k] = 0;
                } else {
                    //Increasing flag counter
                    flag_count++;
                    //Adding the filtered value to the output array
                    filtered_output[k] = (ansk/abs(ansk))*upper_th;
                    flag_output[k] = 1;
                }
            }
            break;
        case 3:
            //Generating a gaussian noise distribution with 0 mean
and 1 std deviation
            default_random_engine generator(time(0));
            normal_distribution<float> dist(0,1);

            for(int k = 0; k < WINDOW_SIZE; k++) {
                int ansk = ans[k];
                if(std::abs(ansk) <= upper_th) {
                    //Adding the filtered value to the output array
                    filtered_output[k] = ansk;
                    flag_output[k] = 0;
                } else {
                    //Increasing flag counter
                    flag_count++;
```

```
                        //Adding the filtered value to the output array
                        int noise = dist(generator)*sigma;
                        filtered_output[k] = std::abs(noise) > 127 ?
(-1^k)*127 : noise;
                        flag_output[k] = 1;
                    }
                }
                break;
        }

        //Increasing the slicing position
        start = start + WINDOW_SIZE;

        //Stopping timer for filtering
        //auto stop_filter = std::chrono::high_resolution_clock::now();

        //Stopping the timer for total time
        auto stop_time = high_resolution_clock::now();

        //Getting average times for the window
        auto duration = duration_cast<microseconds>(stop_time -
start_time);
        avg_time = ((avg_time*i) + duration.count())/(i+1);
        //auto duration_med = duration_cast<microseconds>(stop_med -
start_med);
        //avg_mediantime = ((avg_mediantime*i) +
duration_med.count())/(i+1);
        //auto duration_filter =
duration_cast<microseconds>(stop_filter - start_filter);
        //avg_filter = ((avg_filter*i) +
duration_filter.count())/(i+1);


        for(int f = 0; f < WINDOW_SIZE; f++) {
            filtered.push_back(filtered_output[f]);
        }

    }

    //Printing the statistics
    cout << "Avg total time per window required (microsec) = " <<
avg_time << endl;
```

```
    cout << "Avg filtering time per window required (microsec) = " <<
avg_filter << endl;
    cout << "Avg median time per window required (microsec) = " <<
avg_mediantime << endl;
    //float out_size = filtered_output.size();
    //cout << "|||_____ Final output size - " << out_size << endl;
    cout << "|||_____ Number of flags - " << flag_count << endl;
    //float percentage = (flag_count/out_size)*100;
    //cout << "|||_____ Percentage flagging - " << percentage << endl;


    //Writing the output vector to the file
    int out_sz = filtered.size();
    cout << "|||_____ Final output size - " << out_sz << endl;
    WriteFile(filtered,"output.txt",out_sz);
    cout << "Done writing filtered output file" << endl;

/*
    //Writing the flag vector to the file and getting a counter for
number of flags
    out_sz = flag_output.size();
    cout << "|||_____ Number of flags - " << flag_count << endl;
    WriteFile(flag_output,"output_flags.txt",out_sz);
    cout << "Done writing filtered output file" << endl;
*/

    cout << endl;
    return 0;
}
```

pwd_final_mod.cpp

```
/*
This code does MOM filtering while assuming the first median to be zero
to save computation
*/

/*
This version will be assuming first median to be 0 always.
It also includes the configuration options being given from a header
file instead of the command line.
It will be optimized with OpenMP and -O3 level ove optimizations from
the compiler directives.
```

```
*/

#include<iostream>
#include<bits/stdc++.h>
#include<chrono>
#include<fstream>
#include<vector>
#include<iterator>
#include<random>

using namespace std;
using namespace std::chrono;

//************************ File reader and writer functions
*****************************************//

vector<int> ReadFile(string &name,int &cnt) {
   //Making an fstream object
   fstream myFile;
   //Generating input data vector
   vector<int> input_data;
   //Opening the file in input mode
   myFile.open(name,ios::in);
   //Checking if file is open, and then storing it into the vector
   if(myFile.is_open()) {
       int input_buffer;
       //getline() only stores in strings, hence the conversion
       while(myFile >> input_buffer) {
           input_data.push_back(input_buffer);
           cnt++;
       }
   }
   myFile.close();
   return input_data;
}

void WriteFile(vector<int> &write_data,string name,int &data_length) {
   //Making an fstream object
   fstream myFile;
   //Opening the file in write mode
   myFile.open(name,ios::out);
   //Checking if file is open, then writing to it
   if(myFile.is_open()) {
```

```
        for(int i = 0; i < data_length;i++) {
            myFile << write_data.at(i) << endl;
        }
    }
    myFile.close();
}

//********************** Slicing the vector into windows
*****************************************//
vector<int> VectorSlice(vector<int> &v,int X, int Y) {
    auto first = v.cbegin() + X;
    auto last = v.cbegin() + Y + 1;
    std::vector<int> vec(first, last);
    return vec;
}

//********************** Noise Generation Module
*****************************************//
int* NoiseMaker(int arr[], int len) {

    // Define random generator with Gaussian distribution
    const double mean = 0.0;
    const double stddev = 1.0;
    std::default_random_engine generator(time(0));
    std::normal_distribution<float> dist(mean, stddev);

    // Add Gaussian noise
    for (int i = 0; i < len; i++) {
        arr[i] = dist(generator);
    }
    //std::cout << oper << std::endl;

    return 0;
}


// ******************** Histogram method for median calculation
*******************************//

//Function to calculate the median through histogram - inputs are the
array and it's length
int HistoMedian(int input_arr[], int len) {
```

```
    //Making an array to store frequencies, using indexes as data points
(0-128 as it's absolutes of signed 8 bit data)
    int histo_array[129] = {};

    //Sifting through the input array and incrementing histogram array
counters
    for(int i = 0; i < len; i+=4) {
        //Increasing the frequency counter of the histogram array acc to
input
        histo_array[input_arr[i]]++;
    }

    //Calculating the median from the histogram array
    int sum = 0;
    for (int i = 0; i < 129; i++) {
        sum = sum + histo_array[i];
        //Return median if the cumulative sum >= len/2 and break the
loop
        if(sum > len/8) {
            return i;
        }
    }
    return 0;
}

//*********************** MAD Calculation
**********************************//
int main(int argc, char *argv[]) {

    //************************* Getting input parameters from command
line ********************************//

    string INPUT_FILENAME = argv[1];            //Name of the file with
data
    const int WINDOW_SIZE = stoi(argv[2]);      //Window Size for all
methods
    const int MOM_WIN = stoi(argv[3]);          //Number of windows for
MOM calculation
    const int SUB_WIN = stoi(argv[4]);          //Sub window size to
take for power detection
    const int SCL_FCTR = stoi(argv[5]);         //Threshold factor  (th
= sigma*sigma*SCL_FCTR)
```

```
    const int RPL_OPTION = stoi(argv[6]);       // Replacement option (0
= rpl with const, 1 = rpl with noise)
    const int RPL_CONST = stoi(argv[7]);        // Replacement constant
for rpl = 0

    //Getting the whole data vector
    vector<int> numbers;
    int data_count = 0;
    numbers = ReadFile(INPUT_FILENAME,data_count);

    //Getting a variable for avg time
    long double avg_time = 0, avg_filter =0, avg_mediantime = 0;

    //Getting a count for number of flags
    long int flag_count = 0;

    //Calculating number of slices
    int n_slices = data_count/WINDOW_SIZE;

    //Making a filtered output and flagging array
    int filtered_output[WINDOW_SIZE];
    int flag_output[WINDOW_SIZE];

    //Making a vector for writing to file
    vector<int> filtered;
    vector<int> flags;

    //Making the MAD buffer array to calculate MOM value
    int mad_buffer[n_slices];

    //Initiating a counter for slicing
    int start = 0;

    //Pregeneration of noise for the given window size
    int noise_window[WINDOW_SIZE];
    NoiseMaker(noise_window,WINDOW_SIZE);

    //Declaring threshold variables for the windows without MOM filter
(first MOM_WIN windows)
    float sigma;
    float upper_th;
    upper_th = 127;
    FILE* fptr2=fopen("dacpp.txt","w");
```

```
    //Main loop for iterating through each window
    for(int i = 0;i < n_slices ; i++) {
        //Making a vector for the window
        vector<int> ans;
        //cout << "start value is - " << start << "---------------||" <<
endl;
        //cout << "End value is - " << start + WINDOW_SIZE - 1 <<
"---------"<< i + 1 << "------||" << endl;
        ans = VectorSlice(numbers,start,start + WINDOW_SIZE - 1);

        //Starting the stopwatch for timer of current window
        auto start_time = high_resolution_clock::now();

        //Making the array to pass for median (directly taking absolutes
as first median is 0)
        int arr[WINDOW_SIZE];
        for(int j = 0; j < WINDOW_SIZE; j++) {
            arr[j] = abs(ans[j]);
        }

        //Starting stopwatch for median calculation
        auto start_med = std::chrono::high_resolution_clock::now();
        //Adding the MAD value into buffer
        mad_buffer[i] = HistoMedian(arr,WINDOW_SIZE);
        //Stopping timer and calculating median time required
        auto stop_med= std::chrono::high_resolution_clock::now();

        //Updating the sigma value for threshold calculations, if we
reach the next MOM window
        if((i+1)%MOM_WIN == 0) {

            //Making an array for MOM calculation from mad_buffer (in
short, slicing)
            int mom_buffer[MOM_WIN];
            for(int m = 0; m < MOM_WIN; m++) {
                mom_buffer[m] = mad_buffer[i + 1 - MOM_WIN + m];
            }

            //Getting MOM value
            //Calculating the median of the data
            int mom_value = HistoMedian(mom_buffer,MOM_WIN);
            cout << mom_value << endl;
            //Calculating the thresholds
```

```
            sigma = 1.4826*mom_value;
        }

        //Updating the upper and lower thresholds according to the last
MOM sigma
        //Done after the first MOM window has passed, to apply these to
second one.
        if(i + 1 >= MOM_WIN) {
            upper_th = SCL_FCTR*sigma*sigma;
        }
        if(((i + 1) % MOM_WIN) ==0) {
            printf("%f--> Threshold\n",upper_th);
        }
        //Starting filtering timer
        auto start_filter = std::chrono::high_resolution_clock::now();

        int n_sub = WINDOW_SIZE/SUB_WIN;
        int sub_window_size = SUB_WIN;

        //************************ Filtering with switch case
***************************************//
        switch(RPL_OPTION) {
            case 0:
                for(int j = 0; j < n_sub; j++) {
                    //Going through every element in the sub window
                    int st_index = j*sub_window_size;           //Points
to the starting value
                    //int16_t* b=&
                    int data_accum = 0;
                    //Applying power detection algorithm [getting
sum(x^2)]
                    for(int k = st_index; k < st_index +
sub_window_size; k++) {
                        data_accum += arr[k]*arr[k];
                    }

                    fprintf(fptr2,"%d\n",data_accum);
                    //Filtering out the window and adding flags if it's
out of threshold
                    if(data_accum > upper_th) {
                        for(int k = st_index; k < st_index +
sub_window_size; k++) {
                            filtered_output[k] = RPL_CONST;
```

```
                            flag_output[k] = 1;
                        }
                        //Updating the entire sub window's flag counter
at once
                        flag_count += sub_window_size;
                    } else {
                        for(int k = st_index; k < st_index +
sub_window_size; k++) {
                            filtered_output[k] = ans[k];
                            flag_output[k] = 0;
                        }
                    }
                }
                break;
            case 1:
                //Generating a gaussian noise distribution with 0 mean
and 1 std deviation
                default_random_engine generator(time(0));
                normal_distribution<float> dist(0,1);

                for(int j = 0; j < n_sub; j++) {
                    //Going through every element in the sub window
                    int st_index = j*sub_window_size;           //Points
to the starting value
                    int data_accum = 0;
                    //Applying power detection algorithm [getting
sum(x^2)]
                    for(int k = st_index; k < st_index +
sub_window_size; k++) {
                        data_accum += arr[k]*arr[k];
                    }
                    //Filtering out the window and adding flags if it's
out of threshold
                    if(data_accum > upper_th) {
                        for(int k = st_index; k < st_index +
sub_window_size; k++) {
                            //int noise = dist(generator)*sigma;
                            filtered_output[k] =
abs(noise_window[k]*sigma) > 127 ? (-1^k)*127 : noise_window[k]*sigma;;
                            flag_output[k] = 1;
                        }
                        //Updating the entire sub window's flag counter
at once
```

```
                        flag_count += sub_window_size;
                    } else {
                        for(int k = st_index; k < st_index +
sub_window_size; k++) {
                            filtered_output[k] = ans[k];
                            flag_output[k] = 0;
                        }
                    }
                }
                break;
        }

        //Increasing the slicing position
        start = start + WINDOW_SIZE;

        //Stopping timer for filtering
        auto stop_filter = std::chrono::high_resolution_clock::now();

        //Stopping the timer for total time
        auto stop_time = high_resolution_clock::now();

        //Getting average times for the window
        auto duration = duration_cast<microseconds>(stop_time -
start_time);
        avg_time = ((avg_time*i) + duration.count())/(i+1);
        auto duration_med = duration_cast<microseconds>(stop_med -
start_med);
        avg_mediantime = ((avg_mediantime*i) +
duration_med.count())/(i+1);
        auto duration_filter = duration_cast<microseconds>(stop_filter -
start_filter);
        avg_filter = ((avg_filter*i) + duration_filter.count())/(i+1);


        for(int f = 0; f < WINDOW_SIZE; f++) {
            filtered.push_back(filtered_output[f]);
            flags.push_back(flag_output[f]);
        }

    }

    //Printing the statistics
```

```
   std::cout << "Avg total time per window required (microsec) = " <<
avg_time << endl;
   std::cout << "Avg filtering time per window required (microsec) = "
<< avg_filter << endl;
   std::cout << "Avg median time per window required (microsec) = " <<
avg_mediantime << endl;
   float out_size = filtered.size();
   std::cout << "|||_____ Final output size - " << out_size << endl;
   std::cout << "|||_____ Number of flags - " << flag_count << endl;
   float percentage = (flag_count/out_size)*100;
   std::cout << "|||_____ Percentage flagging - " << percentage <<
endl;


   //Writing the output vector to the file
   int out_sz = filtered.size();
   std::cout << "|||_____ Final output size - " << out_sz << endl;
   WriteFile(filtered,"output_cpp.out",out_sz);
   std::cout << "Done writing filtered output file" << endl;

   //Writing the flag vector to the file and getting a counter for
number of flags
   out_sz = flags.size();
   cout << "|||_____ Number of flags - " << flag_count << endl;
   WriteFile(flags,"flags_cpp.out",out_sz);
   cout << "Done writing filtered output file" << endl;


   std::cout << endl;
   return 0;
}
```

SSE and AVX instructions, with their usage

__m256i _mm256_set1_epi8 (char a)

...

Synopsis

```
__m256i _mm256_set1_epi8 (char a)
#include <immintrin.h>
Instruction: Sequence
CPUID Flags: AVX
```

Description

Broadcast 8-bit integer `a` to all elements of `dst`. This intrinsic may generate the `vpbroadcastb`.

Operation

```
FOR j := 0 to 31
        i := j*8
        dst[i+7:i] := a[7:0]
ENDFOR
dst[MAX:256] := 0
```

__m256i _mm256_loadu_si256 (__m256i const * mem_addr)

vmovdqu

Synopsis

```
__m256i _mm256_loadu_si256 (__m256i const * mem_addr)
#include <immintrin.h>
Instruction: vmovdqu ymm, m256
CPUID Flags: AVX
```

Description

Load 256-bits of integer data from memory into `dst`. `mem_addr` does not need to be aligned on any particular boundary.

Operation

```
dst[255:0] := MEM[mem_addr+255:mem_addr]
dst[MAX:256] := 0
```

__m256i _mm256_cmpgt_epi8 (__m256i a, __m256i b)

vpcmpgtb

Synopsis

```
__m256i _mm256_cmpgt_epi8 (__m256i a, __m256i b)
#include <immintrin.h>
Instruction: vpcmpgtb ymm, ymm, ymm
CPUID Flags: AVX2
```

Description

Compare packed signed 8-bit integers in a and b for greater-than, and store the results in dst.

Operation

```
FOR j := 0 to 31
	i := j*8
	dst[i+7:i] := ( a[i+7:i] > b[i+7:i] ) ? 0xFF : 0
ENDFOR
dst[MAX:256] := 0
```

__m128i _mm_cmplt_epi8 (__m128i a, __m128i b)

pcmpgtb

Synopsis

```
__m128i _mm_cmplt_epi8 (__m128i a, __m128i b)
#include <emmintrin.h>
Instruction: pcmpgtb xmm, xmm
CPUID Flags: SSE2
```

Description

Compare packed signed 8-bit integers in a and b for less-than, and store the results in dst. Note: This intrinsic emits the pcmpgtb instruction with the order of the operands switched.

Operation

```
FOR j := 0 to 15
	i := j*8
	dst[i+7:i] := ( a[i+7:i] < b[i+7:i] ) ? 0xFF : 0
ENDFOR
```

__m256i _mm256_or_si256 (__m256i a, __m256i b)

vpor

Synopsis

```
__m256i _mm256_or_si256 (__m256i a, __m256i b)
#include <immintrin.h>
Instruction: vpor ymm, ymm, ymm
CPUID Flags: AVX2
```

Description

Compute the bitwise OR of 256 bits (representing integer data) in a and b, and store the result in dst.

Operation

```
dst[255:0] := (a[255:0] OR b[255:0])
dst[MAX:256] := 0
```

__m256i _mm256_blendv_epi8 (__m256i a, __m256i b, __m256i mask) vpblendvb

Synopsis

```
__m256i _mm256_blendv_epi8 (__m256i a, __m256i b, __m256i mask)
#include <immintrin.h>
Instruction: vpblendvb ymm, ymm, ymm, ymm
CPUID Flags: AVX2
```

Description

Blend packed 8-bit integers from `a` and `b` using `mask`, and store the results in `dst`.

Operation

```
FOR j := 0 to 31
	i := j*8
	IF mask[i+7]
		dst[i+7:i] := b[i+7:i]
	ELSE
		dst[i+7:i] := a[i+7:i]
	FI
ENDFOR
dst[MAX:256] := 0
```

void _mm256_storeu_si256 (__m256i * mem_addr, __m256i a) vmovdqu

Synopsis

```
void _mm256_storeu_si256 (__m256i * mem_addr, __m256i a)
#include <immintrin.h>
Instruction: vmovdqu m256, ymm
CPUID Flags: AVX
```

Description

Store 256-bits of integer data from `a` into memory. `mem_addr` does not need to be aligned on any particular boundary.

Operation

```
MEM[mem_addr+255:mem_addr] := a[255:0]
```

AVX and SSE instructions used in Power Detection Code

__m256i _mm256_cvtepi16_epi32 (__m128i a)

Synopsis

```
__m256i _mm256_cvtepi16_epi32 (__m128i a)
#include <immintrin.h>
Instruction: vpmovsxwd ymm, xmm
CPUID Flags: AVX2
```

Description

Sign extend packed 16-bit integers in a to packed 32-bit integers, and store the results in dst.

Operation

```
FOR j:= 0 to 7
        i := 32*j
        k := 16*j
        dst[i+31:i] := SignExtend32(a[k+15:k])
ENDFOR
dst[MAX:256] := 0
```

__m256i _mm256_mullo_epi16 (__m256i a, __m256i b)

Synopsis

```
__m256i _mm256_mullo_epi16 (__m256i a, __m256i b)
#include <immintrin.h>
Instruction: vpmullw ymm, ymm, ymm
CPUID Flags: AVX2
```

Description

Multiply the packed signed 16-bit integers in a and b, producing intermediate 32-bit integers, and store the low 16 bits of the intermediate integers in dst.

Operation

```
FOR j := 0 to 15
        i := j*16
        tmp[31:0] := SignExtend32(a[i+15:i]) * SignExtend32(b[i+15:i])
        dst[i+15:i] := tmp[15:0]
ENDFOR
dst[MAX:256] := 0
```


__m256i _mm256_cvtepi16_epi32 (__m128i a)

Synopsis

```
__m256i _mm256_cvtepi16_epi32 (__m128i a)
#include <immintrin.h>
Instruction: vpmovsxwd ymm, xmm
CPUID Flags: AVX2
```

Description

Sign extend packed 16-bit integers in a to packed 32-bit integers, and store the results in dst.

Operation

```
FOR j:= 0 to 7
        i := 32*j
        k := 16*j
        dst[i+31:i] := SignExtend32(a[k+15:k])
ENDFOR
dst[MAX:256] := 0
```

__m256i _mm256_add_epi32 (__m256i a, __m256i b)

Synopsis

```
__m256i _mm256_add_epi32 (__m256i a, __m256i b)
#include <immintrin.h>
Instruction: vpaddd ymm, ymm, ymm
CPUID Flags: AVX2
```

Description

Add packed 32-bit integers in a and b, and store the results in dst.

Operation

```
FOR j := 0 to 7
	i := j*32
	dst[i+31:i] := a[i+31:i] + b[i+31:i]
ENDFOR
dst[MAX:256] := 0
```

__m256i _mm256_hadd_epi32 (__m256i a, __m256i b)

Synopsis

```
__m256i _mm256_hadd_epi32 (__m256i a, __m256i b)
#include <immintrin.h>
Instruction: vphaddd ymm, ymm, ymm
CPUID Flags: AVX2
```

Description

Horizontally add adjacent pairs of 32-bit integers in a and b, and pack the signed 32-bit results in dst.

Operation

```
dst[31:0] := a[63:32] + a[31:0]
dst[63:32] := a[127:96] + a[95:64]
dst[95:64] := b[63:32] + b[31:0]
dst[127:96] := b[127:96] + b[95:64]
dst[159:128] := a[191:160] + a[159:128]
dst[191:160] := a[255:224] + a[223:192]
dst[223:192] := b[191:160] + b[159:128]
dst[255:224] := b[255:224] + b[223:192]
dst[MAX:256] := 0
```

__m128i _mm256_extracti128_si256 (__m256i a, const int imm8)

Synopsis

```
__m128i _mm256_extracti128_si256 (__m256i a, const int imm8)
#include <immintrin.h>
Instruction: vextracti128 xmm, ymm, imm8
CPUID Flags: AVX2
```

Description

Extract 128 bits (composed of integer data) from `a`, selected with `imm8`, and store the result in `dst`.

Operation

```
CASE imm8[0] OF
0: dst[127:0] := a[127:0]
1: dst[127:0] := a[255:128]
ESAC
dst[MAX:128] := 0
```

__m128i _mm_add_epi32 (__m128i a, __m128i b)

Synopsis

```
__m128i _mm_add_epi32 (__m128i a, __m128i b)
#include <emmintrin.h>
Instruction: paddd xmm, xmm
CPUID Flags: SSE2
```

Description

Add packed 32-bit integers in `a` and `b`, and store the results in `dst`.

Operation

```
FOR j := 0 to 3
	i := j*32
	dst[i+31:i] := a[i+31:i] + b[i+31:i]
ENDFOR
```

```
int _mm_extract_epi32 (__m128i a, const int imm8)
```

Synopsis

```
int _mm_extract_epi32 (__m128i a, const int imm8)
#include <smmintrin.h>
Instruction: pextrd r32, xmm, imm8
CPUID Flags: SSE4.1
```

Description

Extract a 32-bit integer from `a`, selected with `imm8`, and store the result in `dst`.

Operation

```
dst[31:0] := (a[127:0] >> (imm8[1:0] * 32))[31:0]
```

SELF DECLARED SSE/AVX FUNCTIONS

```
typedef signed char __v32qs __attribute__ ((__vector_size__ (32)));
extern __inline __m256i
__attribute__ ((__gnu_inline__, __always_inline__, __artificial__))
_mm256_mullo_epi8 (__m256i __A, __m256i __B)
{
  return (__m256i) ((__v32qs)__A * (__v32qs)__B);
}
```

To multiply and return lower order bits of 8-bit signed integers