Optimising Real-time Performance of a CPU-based RFI Filtering Algorithm

Student Project
by
Saptarshi Nag

Computer Science and Engineering (Artificial Intelligence)
Defence Institute of Advanced Technology, Girinagar, Pune

Under the guidance of

Kaushal D. Buch

NCRA • TIFR

GIANT METREWAVE RADIO TELESCOPE
NATIONAL CENTRE FOR RADIO ASTROPHYSICS
TATA INSTITUTE OF FUNDAMENTAL RESEARCH
Narayangaon, Tal- Junnar, Dist- Pune
Dec 2023 - Jan 2024

Abstract

The report presents a comprehensive analysis of implementing different optimisation methods for real-time radio frequency interference (RFI) mitigation in the context of the Giant Metrewave Radio Telescope (GMRT) and the GMRT Wideband Backend (GWB) processing system. The goal is to improve RFI mitigation techniques' real-time performance and efficiency by leveraging the Intel Streaming SIMD Extension (SSE) instructions, AVX instructions and other optimisations like in-place memory swapping for filtering, using separate memory for parallel processing and optimising the OpenMP loop. The report outlines the benchmarking results and functional testing carried out as part of this project. Additionally, it explores the power detection technique and its optimisation using SSE instructions, AVX instructions and several other optimisation methods like in-place swapping and look-up tables. The report concludes with an overview of the improvements and outlines future directions for further enhancements.

Acknowledgements

I sincerely appreciate and thank my mentor, Kaushal D. Buch, Engineer F, Digital Backend Group, GMRT, for his invaluable guidance, encouragement, and insightful suggestions throughout this project. His steadfast support and teachings played a pivotal role in completing this report. I am profoundly thankful for his continuous assistance and guidance.

I would like to express my sincere appreciation to the entire staff of GMRT.

List of Contents

List of Figures

List of Tables

Chapter 1: Introduction

1.1 Brief on GMRT Array and uGMRT

The GMRT, or the Giant Metrewave Radio Telescope, is an impressive radio telescope near Pune, India. It is managed by the National Centre for Radio Astrophysics (NCRA) of the Tata Institute of Fundamental Research (TIFR). The GMRT is renowned worldwide as one of the largest and most sensitive radio telescopes 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. The GMRT employs an innovative antenna construction method called SMART to achieve a lightweight and wind-resistant design. This approach involves wire mesh panels and rope trusses, making the entire array cost-effective.

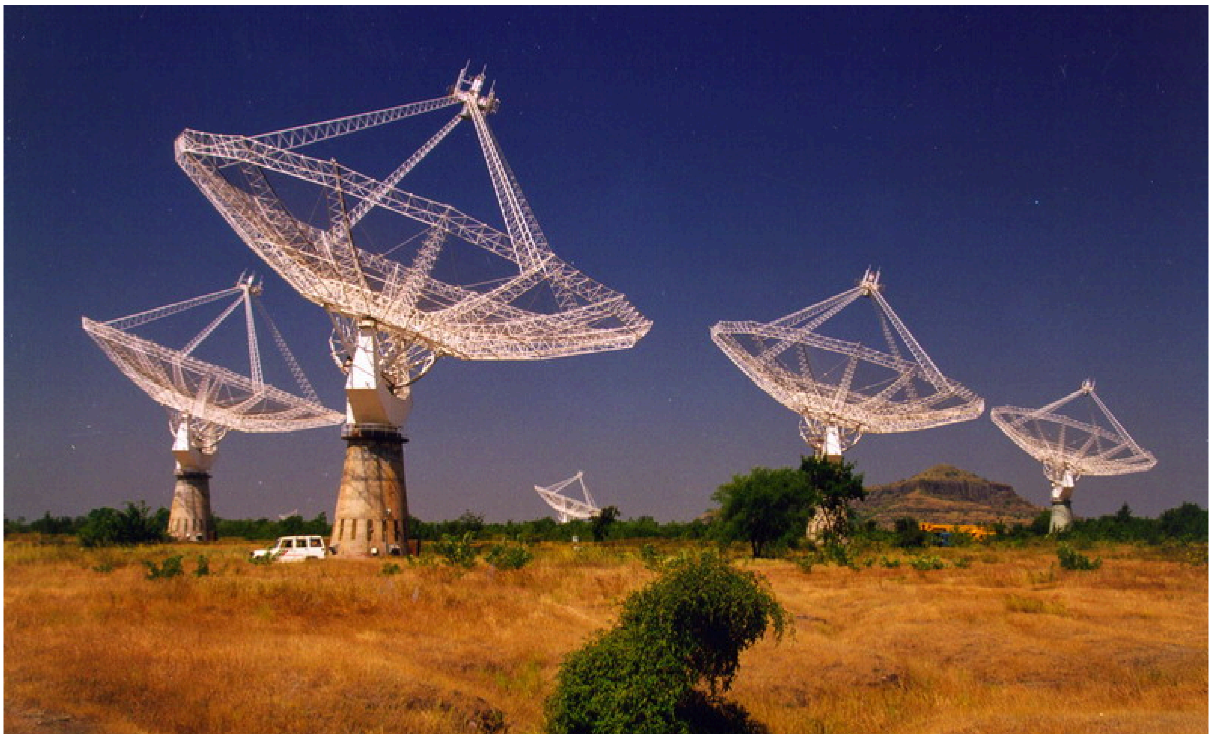


Figure 1.1.1 - A few GMRT antennas (Source: GMRT official website) [1]

1.2 A brief on Radio Frequency Interference (RFI)

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. 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 A brief introduction to GMRT Wideband Backend

The GMRT Wideband Backend (GWB) comprises 16 FPGA + Compute nodes, each equipped with 2 ADC units receiving 2 polarisations from 2 antennas. Initially implemented on FPGA, the filtering method is being reconsidered for migration to CPU due to additional backend feature requirements on FPGA. Consequently, the project aims to develop and optimise a software-based approach for RFI filtering on the Compute nodes using CPU resources. This transition from FPGA to CPU is required to free up resource availability on FPGA which is a requirement for upcoming parallel backend and Spotlight cluster. Refer to Figure 1.3.1 for the block diagram of the GWB backend system.

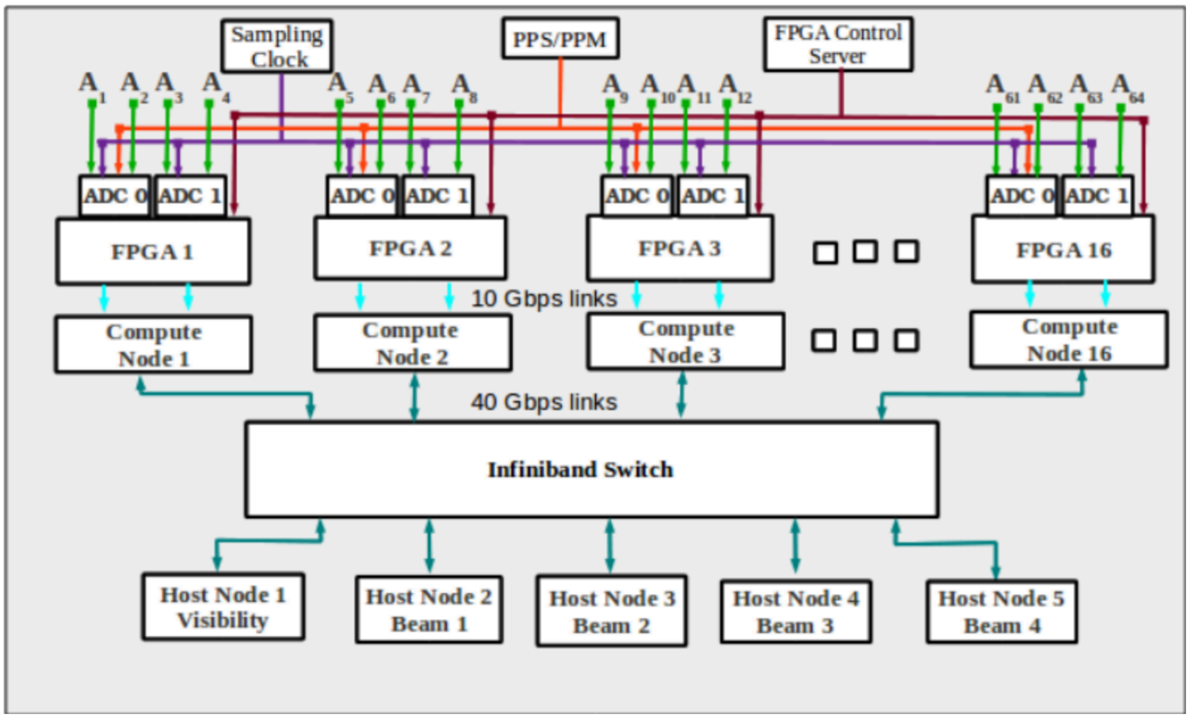


Figure 1.3.1 - Block diagram for GWB correlator (Source: report of Shubham Balte) [2]

1.4 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 four antennas. Then the digitised and filtered data is sent to the CPU + GPU compute node for further processing and data storage. [3]

MAD(median absolute deviation) is defined as
MAD = $\mathrm{median}_j[|x - \mathrm{median}(x_i)|]$

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

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

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

In that case, $\sigma = 1.4826 \times \mathrm{MoM}$.

The sigma value is computed for a window of size ‘W’ and is used to eventually
compute a threshold value where:

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

Here, N is the threshold level, usually set at N = 3. These values are used to flag.

and filter every element in the window, and can do four things with the RFI values:

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

1.5 Previous Work

The goal was to create a CPU-based real-time RFI filtering algorithm. The 1st approach was to create a C program for RFI filtering. However, it was not fast and optimised enough to filter in real-time. Also, it was not integrated with GWB. In the 2nd iteration, the program was optimised with SSE instructions and AVX instructions. Also, some other approaches like Double buffer and openmp loop were utilised after that.

1.5.1 Double buffer

A robust RFI filtering algorithm was developed, leveraging a double buffer setup to execute distinct filters concurrently, enhancing processing speed and efficiency. It has four different sections, each with 256 rows and 2^{20} columns. Each of these sections is used for running a separate filter, and all four filters work simultaneously. This configuration allows simultaneous reading and filtering of signal to occur in separate buffers, contributing to optimised performance and throughput.

1.5.2 OpenMP loop

OpenMP (Open Multi-Processing) is an API (Application Programming Interface) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran. One of its primary features is the ability to parallelise loops efficiently. [4]

By adding OpenMP directives to a loop, the compiler can automatically distribute iterations of the loop across multiple threads, thereby enabling concurrent execution on multi-core processors. It can lead to significant performance improvements for computationally intensive tasks.

OpenMP loop was used to simplify the process of parallelising loops, allowing developers to harness the computational power of modern multi-core architectures more efficiently.

1.5.3 SIMD, SSE and AVX instructions

SIMD (Single Instruction, Multiple Data) is a computer architecture paradigm that enables parallel data processing. It allows a single instruction to operate on multiple data elements simultaneously [5,6].

SSE (Streaming SIMD Extensions) extends the x86 instruction set architecture that introduces SIMD instructions. SSE enhances performance by enabling parallel operations on multiple data elements within a single CPU cycle [5,6].

AVX (Advanced Vector Extensions) is an expansion of SSE that further enhances SIMD capabilities in x86 processors. AVX introduces wider vector registers and additional instructions, enabling even greater parallelism and performance improvements in tasks that can be parallelised [6,7].

The previous algorithm utilised AVX over SSE AVX instructions. By leveraging AVX, we can efficiently handle 32 8-bit numbers and execute operations more effectively, resulting in improved performance and faster execution times.

1.5.4 Flowchart for previous algorithms

The old 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 flag vectors are written to output files. Finally, the file recording time with arguments is closed. [3]

The flowchart for C++ code is given in figure 1.5.4.1

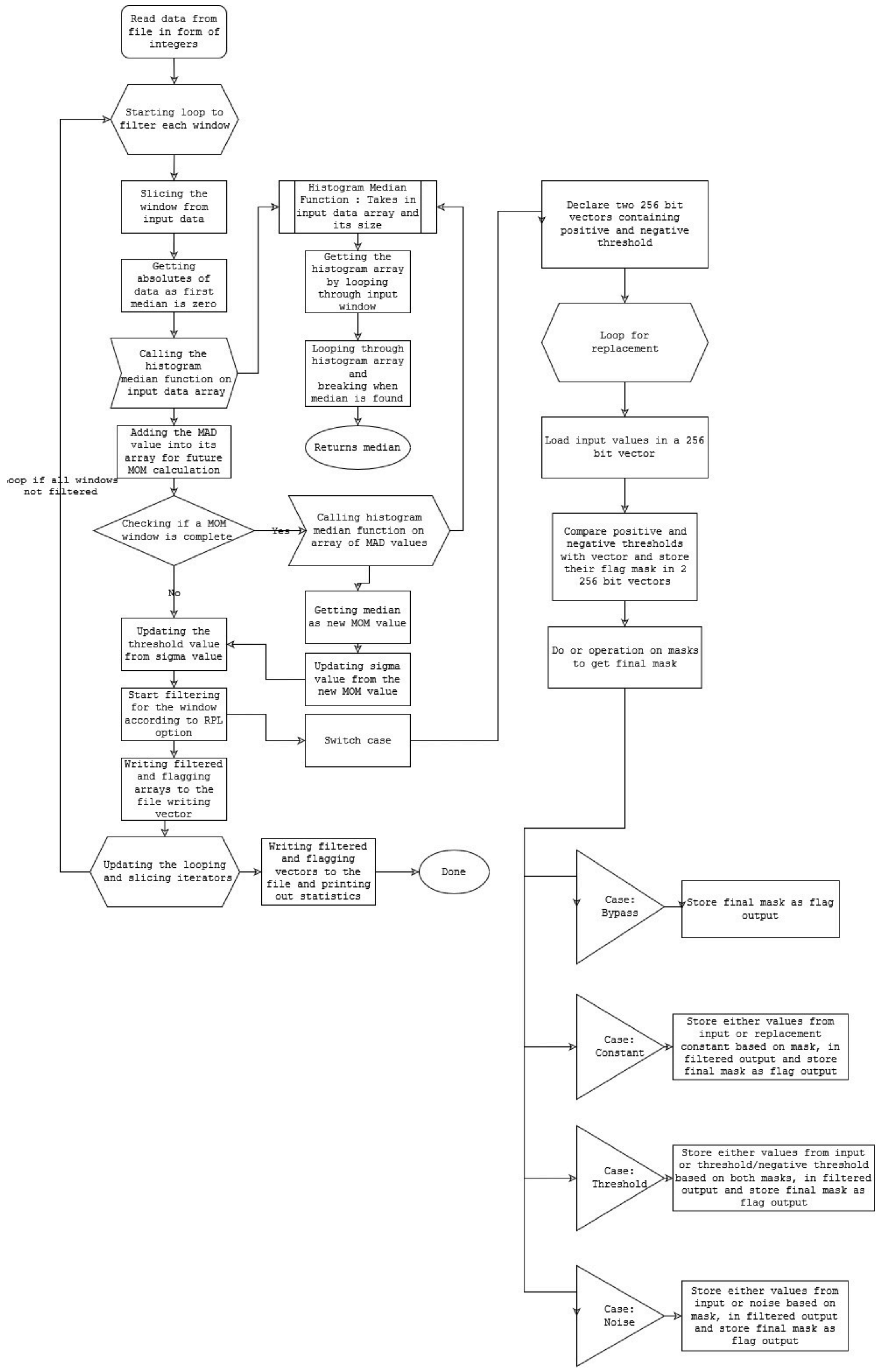

Read data from file in form of integers
Starting loop to filter each window
Slicing the window from input data
Getting absolutes of data as first median is zero
Calling the histogram median function on input data array
Adding the MAD value into its array for future MOM calculation
Loop if all windows not filtered
Checking if a MOM window is complete
Yes
No
Updating the threshold value from sigma value
Start filtering for the window according to RPL option
Writing filtered and flagging arrays to the file writing vector
Updating the looping and slicing iterators
Writing filtered and flagging vectors to the file and printing out statistics
Done
Histogram Median Function : Takes in input data array and its size
Getting the histogram array by looping through input window
Looping through histogram array and breaking when median is found
Returns median
Calling histogram median function on array of MAD values
Getting median as new MOM value
Updating sigma value from the new MOM value
Switch case
Declare two 256 bit vectors containing positive and negative threshold
Loop for replacement
Load input values in a 256 bit vector
Compare positive and negative thresholds with vector and store their flag mask in 2 256 bit vectors
Do or operation on masks to get final mask
Case: Bypass
Store final mask as flag output
Case: Constant
Store either values from input or replacement constant based on mask, in filtered output and store final mask as flag output
Case: Threshold
Store either values from input or threshold/negative threshold based on both masks, in filtered output and store final mask as flag output
Case: Noise
Store either values from input or noise based on mask, in filtered output and store final mask as flag output

Figure 1.5.4.1 - Flowchart of AVX instruction-based filter (Source: Project Report of Satyarth Gupta) [3]

"

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

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. There was a need 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, a double buffer system was implemented. 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 a read occurs in (count%2)th buffer, filter and write occur 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, OpenMP, a parallel programming framework was utilised. By incorporating OpenMP directives into the code, the filter execution was parallelized and it helped to 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, The RFI filter was designed 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. Encapsulating the filter logic within a function, 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 1.5.4.2 shows the flow of program execution.[3]

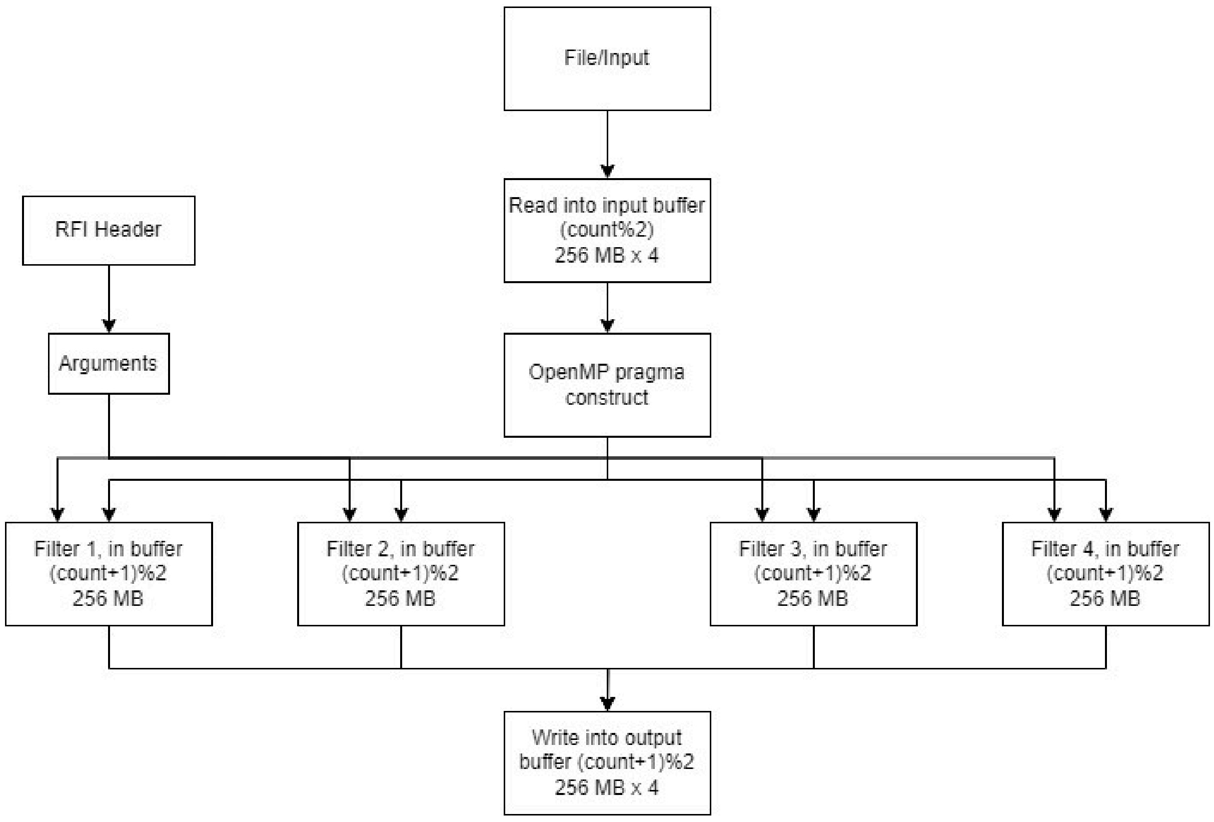

File/Input
Read into input buffer (count%2) 256 MB x 4
RFI Header
Arguments
OpenMP pragma construct
Filter 1, in buffer (count+1)%2 256 MB
Filter 2, in buffer (count+1)%2 256 MB
Filter 3, in buffer (count+1)%2 256 MB
Filter 4, in buffer (count+1)%2 256 MB
Write into output buffer (count+1)%2 256 MB x 4

Figure 1.5.4.2 - Flowchart of execution sequence in the code for GWB integration (Source: Report of Satyarth Gupta) [3]

1.6 Challenges in previous implementation

1.6.1 Machine specification

This is the specification of the snode machine which is obtained using the lscpu command

```
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  16
On-line CPU(s) list:     0-15
Thread(s) per core:      1
Core(s) per socket:      8
Socket(s):               2
NUMA node(s):            2
Vendor ID:               GenuineIntel
CPU family:              6
```

```
Model:                   63
Model name:              Intel(R)  Xeon(R)  CPU  E5-2667  v3  @
3.20GHz
Stepping:                2
CPU MHz:                 3196.600
BogoMIPS:                6393.20
Virtualization:          VT-x
L1d cache:               32K
L1i cache:               32K
L2 cache:                256K
L3 cache:                20480K
NUMA node0 CPU(s):       0,2,4,6,8,10,12,14
NUMA node1 CPU(s):       1,3,5,7,9,11,13,15
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  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 ept vpid fsgsbase tsc_adjust bmi1
avx2 smep bmi2 erms invpcid cqm xsaveopt cqm_llc cqm_occup_llc
dtherm ida arat pln pts spec_ctrl intel_stibp flush_l1d
```

For this program execution, the CPU cores 8,9,14,15 were pinned. This number has been chosen as the rest of the cores are being used for the GWB system.

1.6.2 Program execution time higher than real-time

The main issue of the previously implemented code was it was not working in real-time. It means that input signals will not be filtered. A large amount of data might be lost because of this. Due to this reason, the filtering algorithm could not be integrated into the GWB.

The code worked well in real time when it used a single core for RFI filtering. It took around 25% of real-time, as seen in the graph below (Figure 1.6.2.1)

Calculation of real-time

For single-core, 16k MoM, the real-time calculation is 16k * 2.5 ns = 40.96 µs
For multi-core, 16k MoM, 16k window size, the real-time calculation is 16k * 16k* 2.5ns = 0.671s = 671ms (200 MHz bandwidth mode)

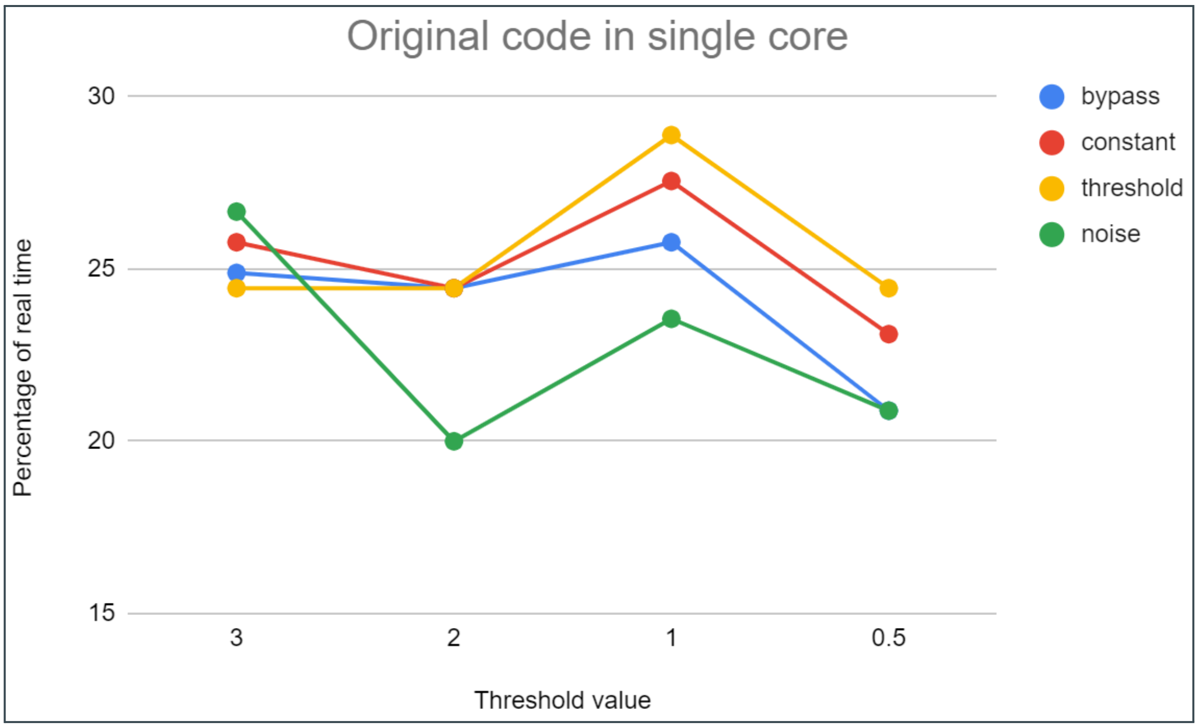


Figure 1.6.2.1 - Filtering time of the previous implementation in single-core

But when it was integrated with GWB, it took more time despite using 4 cores parallelly. As seen in the graph below (Figure 1.6.1.2), the filtering algorithm takes more than 90% of real-time and crosses the threshold multiple times.

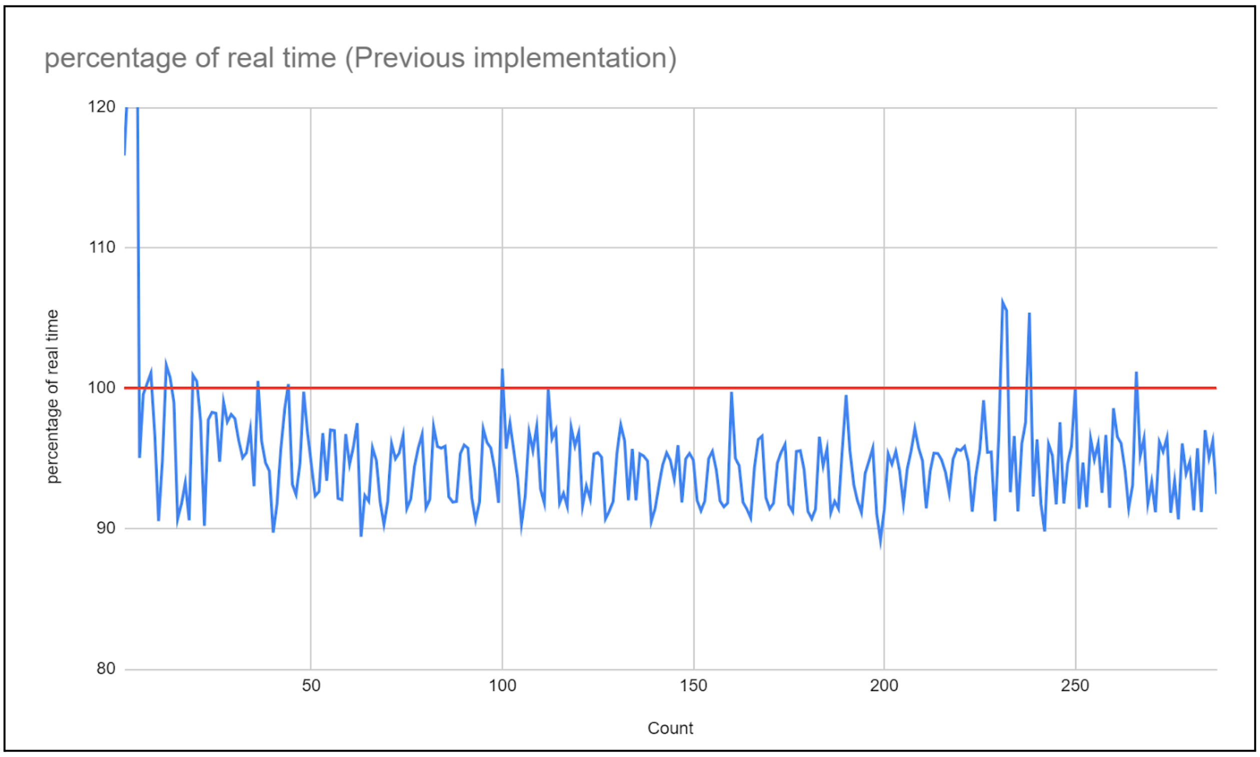


Figure 1.6.2.2 - Filtering time of the previous implementation in multi-core. The X-axis is the number of iterations, each iteration is 0.671s (16k MoM)

1.6.3 Automated verification and benchmarking

In the past implementation, there were no automated verification and benchmarking methods that posed challenges. Even minor modifications needed extensive testing, making the development process complex and longer. Additionally, there was a lack of a comprehensive testing approach to assess multiple programs simultaneously. Furthermore, a streamlined benchmarking method was absent as well. These deficiencies hampered efficiency and accuracy in the development and evaluation phases. As a result, the process encountered delays and inefficiencies, impeding progress and potentially compromising the quality of the outcomes. Implementing robust automated verification and benchmarking procedures could alleviate these issues, enhancing workflow efficiency, reducing testing overhead, and ensuring the reliability and performance of the developed solutions.

Chapter 2: Optimisation for real-time performance

In this chapter, we will provide different optimisation techniques used in the RFI filtering algorithm and discuss the improvement achieved using those methods. We will also explore the effect of compiler versions, operating systems and machines on optimisation. As discussed before in section 1.6.1 about the specification for the GWB system, It has gcc version 4.8.5. So, all optimisation methods and benchmarking in this chapter used gcc version 4.8.5 on snode if not mentioned explicitly.

2.1 Overview of the integrated code and performance benchmarks

The previously implemented code was optimised using a double buffer to read input data and execute distinct filters concurrently. AVX instructions were used for CPU architecture-specific optimisations. An OpenMP loop was used for efficient parallel processing. But still, more than these implementations were needed to achieve real-time performance. It had some room for improvement. As seen in the graph below (Figure 2.1.1), the filtering algorithm takes 95% of real-time on average. On multiple occasions, it crosses the real-time threshold.

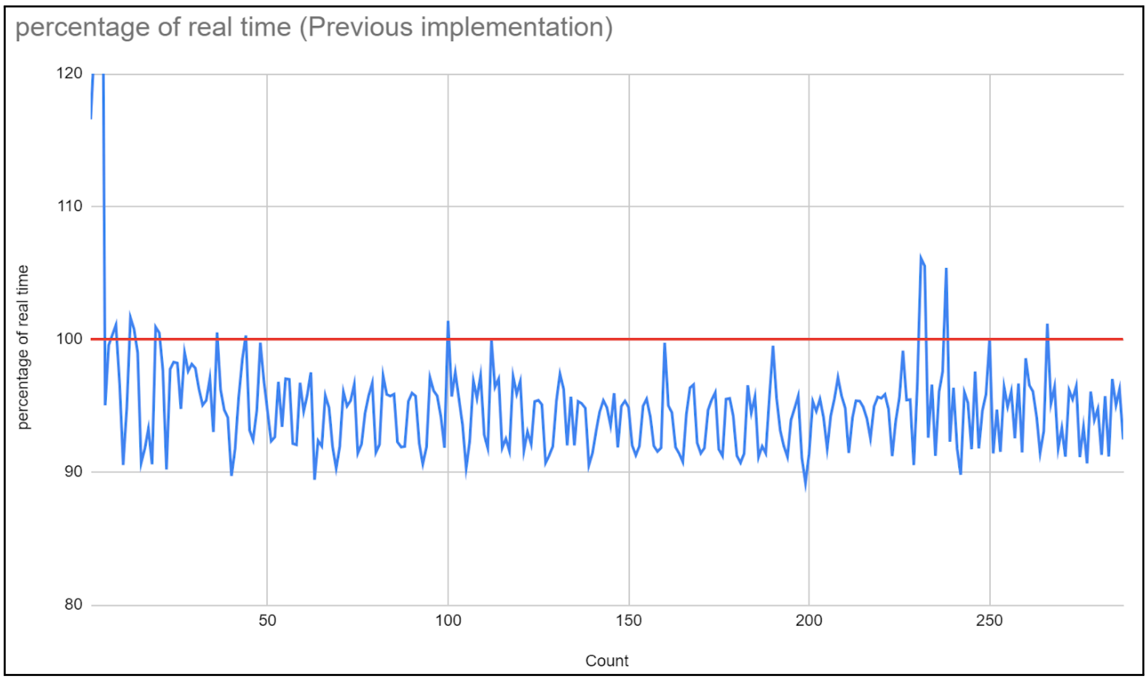


Figure 2.1.1 - Filtering time of unoptimised code in percentage of real-time in multicore

2.2 In-place optimisation

optimising the RFI (Radio Frequency Interference) filtering algorithm by implementing in-place swapping, where the filtered version is written back to the input array, offers several performance benefits:

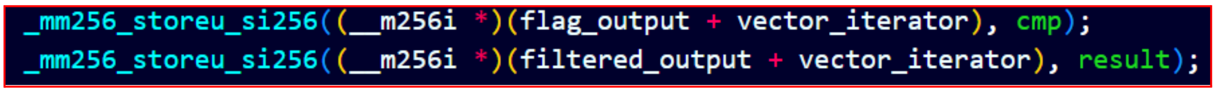
1. Reduced Memory Usage: In-place swapping eliminates the need for allocating additional memory for the filtered array, thereby reducing memory consumption. It is particularly advantageous when dealing with large datasets or limited memory environments.

2. Cache Efficiency: By operating directly on the input array, in-place swapping enhances cache efficiency. It reduces cache misses by ensuring that the processed data remains in the cache, which can lead to significant performance improvements, especially on modern processors with hierarchical caching systems.

3. Lower Overhead: In-place swapping avoids the overhead associated with memory allocation and deallocation and copying data between arrays. It can result in faster execution times and reduced computational overhead.

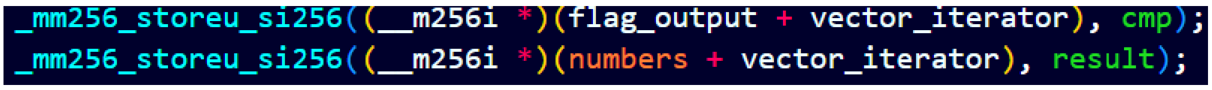
4. Improved Locality of Reference: In-place swapping promotes better locality of reference, as the data being accessed and modified is contiguous in memory. It enhances data access patterns and can lead to more predictable and efficient memory access.

Optimising RFI filtering algorithms with in-place swapping improves performance by leveraging memory efficiency, cache optimisation, and reduced overhead, resulting in faster and more efficient data processing.



```
_mm256_storeu_si256((__m256i *)(flag_output + vector_iterator), cmp);
_mm256_storeu_si256((__m256i *)(filtered_output + vector_iterator), result);
```

Figure 2.2.1 - Old approach to storing filtered data



```
_mm256_storeu_si256((__m256i *)(flag_output + vector_iterator), cmp);
_mm256_storeu_si256((__m256i *)(numbers + vector_iterator), result);
```

Figure 2.2.2 - Inplace swapping to store filtered data

The following graph (Figure 2.2.3) shows the timing seen in the single-core implementation

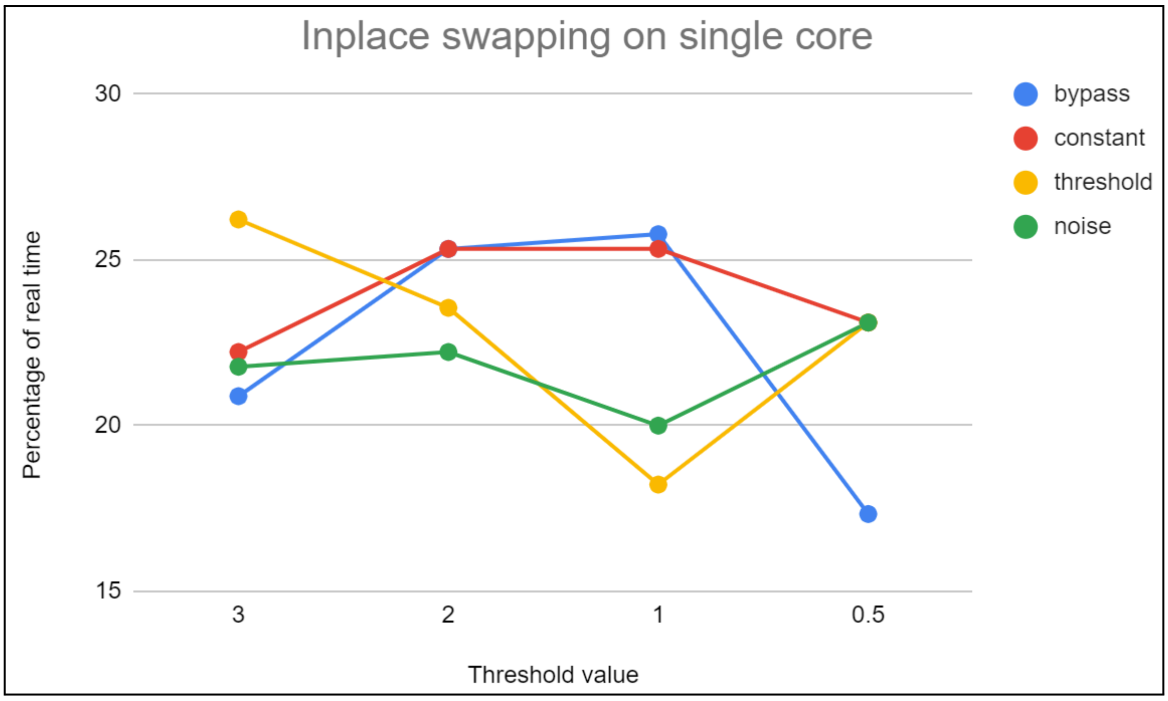


Figure 2.2.3 - Filtering time of the in-place swapping in single-core

Threshold	3	2	1	0.5
Flagging Percentage	0.98	5.20	22.32	39.96

Table 2.2.1 - Flagging percentage

The graph below (Figure 2.2.4) shows the comparison and improvement between the previous implementation and optimized version using in-place swapping when constant replacement is used.

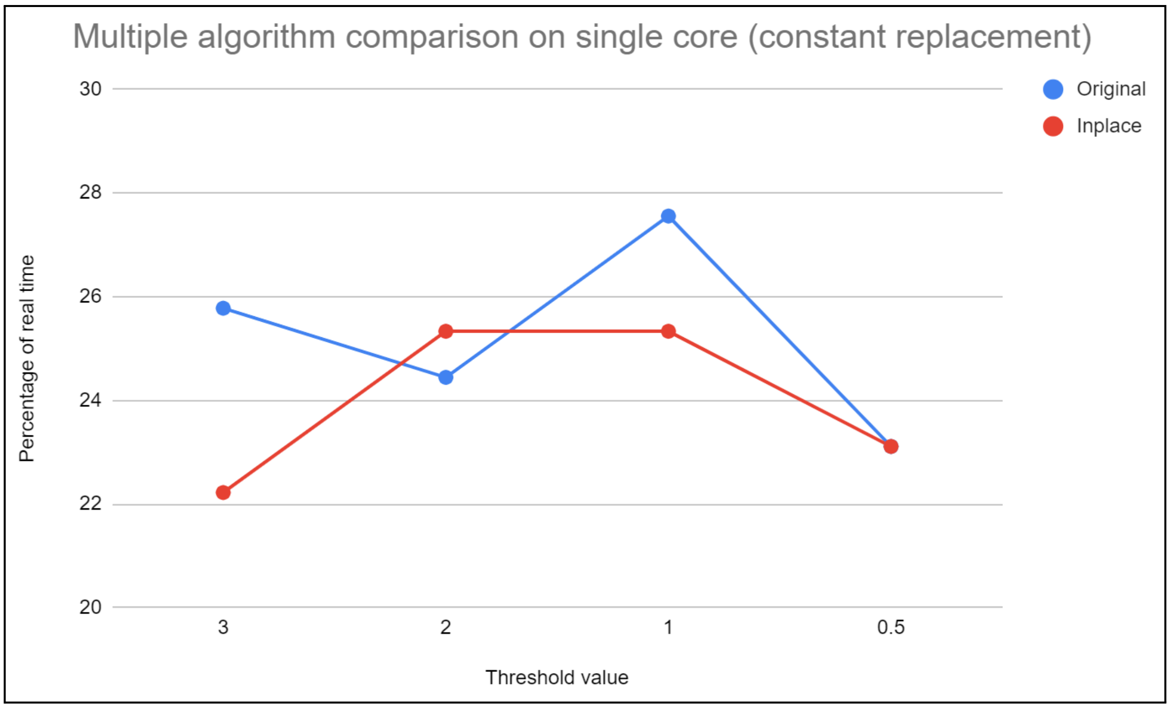


Figure 2.2.4 - Filtering time comparison in single-core

This implementation improves the integrated version. As seen in the graph below (Figure 2.2.5), it takes 93% of real-time. Although not much, it is still an improvement. However, the problem of filtering time going outside the real-time limit still needs to be solved.

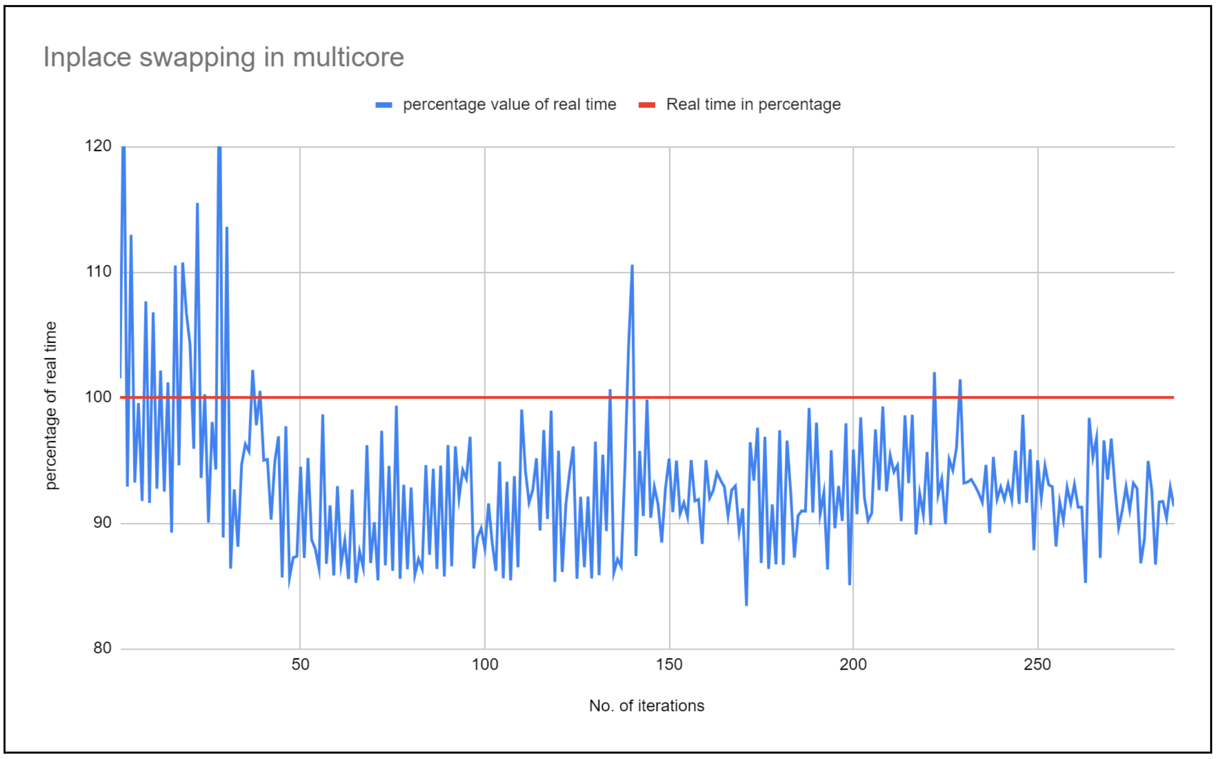


Figure 2.2.5 - Filtering time of the in-place swapping in multi-core

2.3 Four separate memory in double buffer

Implementing four separate memory buffers for each of the sections in the double buffer setup can significantly aid in parallel processing for the following reasons:

1. Reduced Conflicts: With dedicated memory for each buffer, concurrent processing operations encounter fewer clashes for memory access. Conflicts occur when multiple processes attempt to access or modify the same memory location simultaneously, potentially leading to delays. Separate memory spaces mitigate such conflicts, allowing for smoother parallel processing.

2. Enhanced Parallelism: By allocating distinct memory regions for each buffer, parallel processing operations can execute independently without waiting to access shared memory resources. This independence improves parallelism, where multiple filters can simultaneously read, process, and write data without conflict or interference.

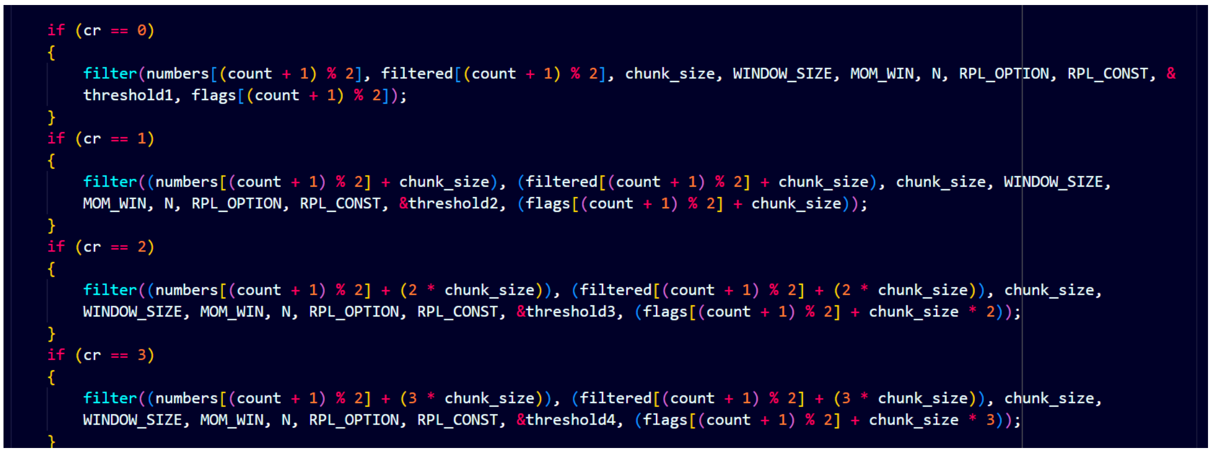
3. Improved Throughput: Dedicated memory for each buffer facilitates efficient data movement and processing, improving throughput. Each filter can operate on its designated data segment without waiting to access shared resources and maximising overall system throughput and performance.

4. Simplified Management: Allocating separate memory for each buffer simplifies memory management and reduces the complexity of data handling. It eliminates the need for intricate synchronisation mechanisms or resource-sharing protocols, streamlining the implementation and maintenance of the parallel processing algorithm.

In summary, employing separate memory buffers for parallel processing in the RFI filtering algorithm enhances performance, throughput, and efficiency by minimising conflicts, maximising parallelism, improving data throughput, and simplifying memory management.

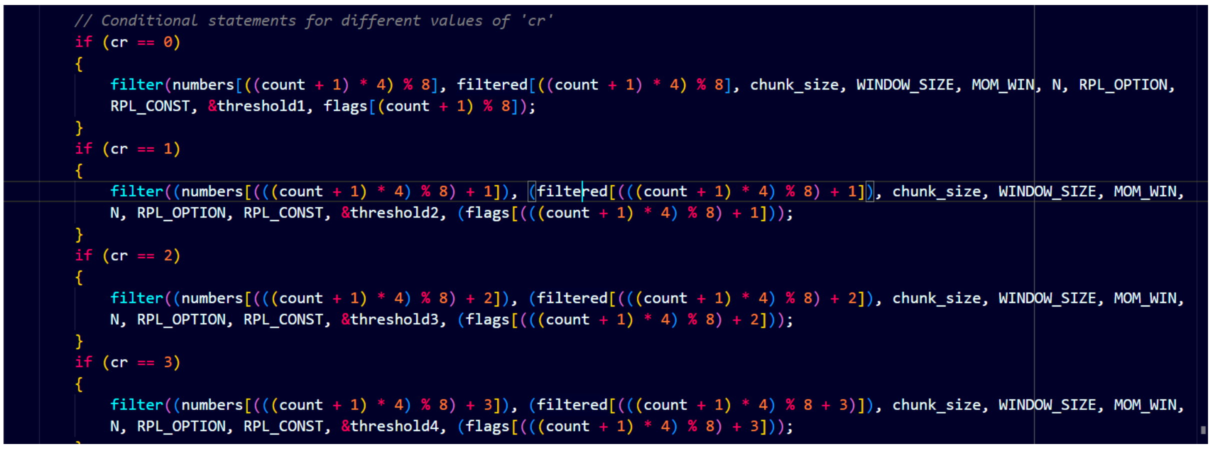
In the previous algorithm, the double buffer was implemented by using two buffers of 1GB each. It was modified and instead of one large 1GB buffer, four smaller 256MB buffers were used. The diagram below (Figure 2.3.3) explains the architecture.

Before the pointer was accessed like this “(count + 1) % 2 + (i*chunk_size)” (i = 0,1,2,3 chunk size = MOM window size X MAD window size)
Now it is accessed like this “(((count + 1) * 4) % 8) + i” (i = 0,1,2,3)



```
if (cr == 0)
{
    filter(numbers[(count + 1) % 2], filtered[(count + 1) % 2], chunk_size, WINDOW_SIZE, MOM_WIN, N, RPL_OPTION, RPL_CONST, &
    threshold1, flags[(count + 1) % 2]);
}
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, (flags[(count + 1) % 2] + chunk_size));
}
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, (flags[(count + 1) % 2] + chunk_size * 2));
}
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, (flags[(count + 1) % 2] + chunk_size * 3));
}
```

Figure 2.3.1 - Pointer offset in original code



```
// Conditional statements for different values of 'cr'
if (cr == 0)
{
    filter(numbers[((count + 1) * 4) % 8], filtered[((count + 1) * 4) % 8], chunk_size, WINDOW_SIZE, MOM_WIN, N, RPL_OPTION,
    RPL_CONST, &threshold1, flags[(count + 1) % 8]);
}
if (cr == 1)
{
    filter((numbers[(((count + 1) * 4) % 8) + 1]), (filtered[(((count + 1) * 4) % 8) + 1]), chunk_size, WINDOW_SIZE, MOM_WIN,
    N, RPL_OPTION, RPL_CONST, &threshold2, (flags[(((count + 1) * 4) % 8) + 1]));
}
if (cr == 2)
{
    filter((numbers[(((count + 1) * 4) % 8) + 2]), (filtered[(((count + 1) * 4) % 8) + 2]), chunk_size, WINDOW_SIZE, MOM_WIN,
    N, RPL_OPTION, RPL_CONST, &threshold3, (flags[(((count + 1) * 4) % 8) + 2]));
}
if (cr == 3)
{
    filter((numbers[(((count + 1) * 4) % 8) + 3]), (filtered[(((count + 1) * 4) % 8 + 3)]), chunk_size, WINDOW_SIZE, MOM_WIN,
    N, RPL_OPTION, RPL_CONST, &threshold4, (flags[(((count + 1) * 4) % 8) + 3]));
```

Figure 2.3.2 - Pointer offset in modified code

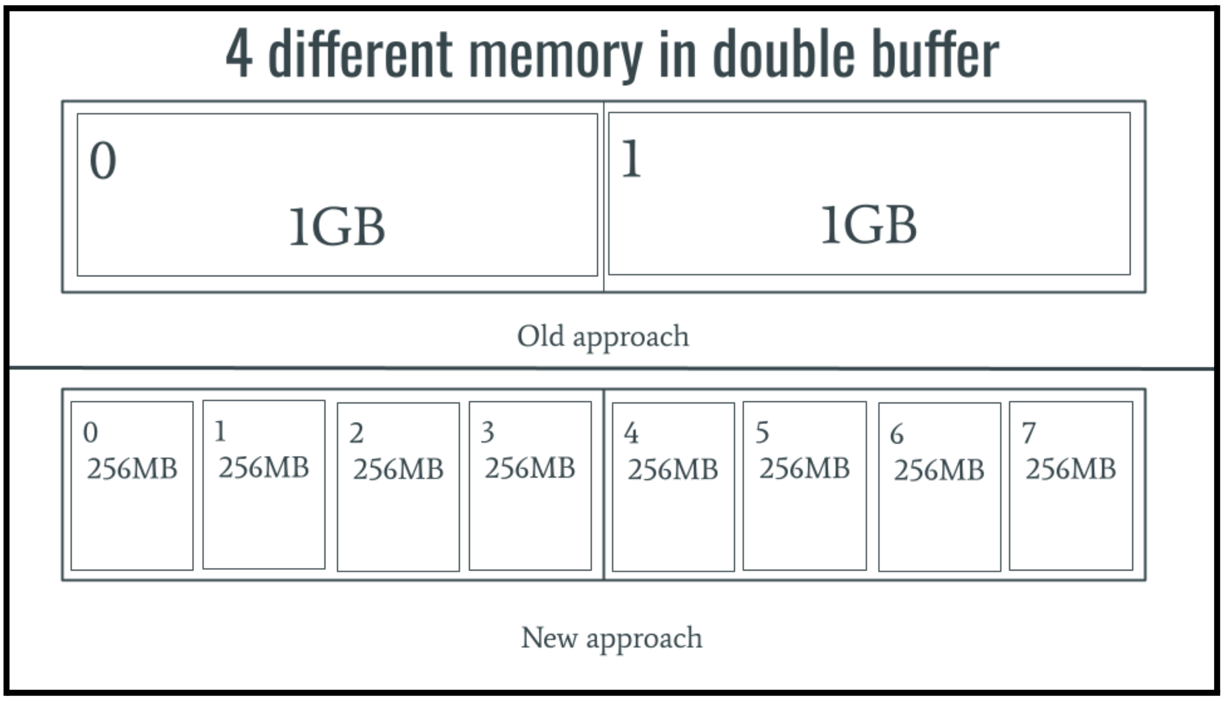


Figure 2.3.3 - Double buffer architecture

As seen in the graph below (Figure 2.3.4) this approach reduced filtering time and in this approach number of values more than real-time was significantly reduced. The average filtering time was 94%.

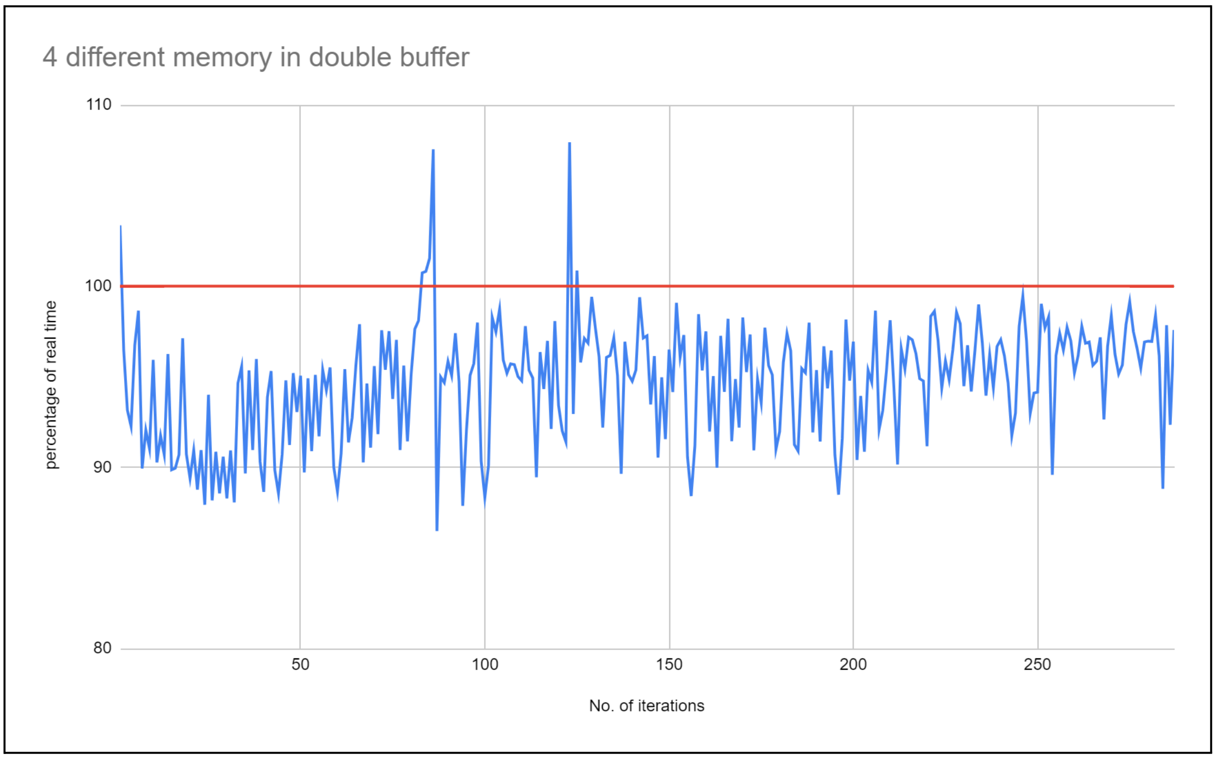


Figure 2.3.4 - Filtering time of 4 separate memory in the buffer in multi-core

2.4 Converting the absolute function to AVX

In the program, there is an absolute calculation that is called every time data is filtered. Previously it was implemented using the in-built abs() function in the `<math.h>` header file. But it is replaced using the avx command. It improved performance significantly.

The performance improvement achieved by replacing the absolute calculation function from the `<math.h>` header with the AVX command stems from several factors: [6]

1. Vectorisation: AVX (Advanced Vector Extensions) instructions enable SIMD (Single Instruction, Multiple Data) operations, allowing the CPU to perform multiple absolute calculations simultaneously on a single instruction. This contrasts with the scalar operation in the standard `<math.h>` function, where each absolute calculation would be executed sequentially.

2. Parallelism: AVX instructions process data in parallel, exploiting the CPU's capabilities to execute multiple operations concurrently. By utilising AVX commands,

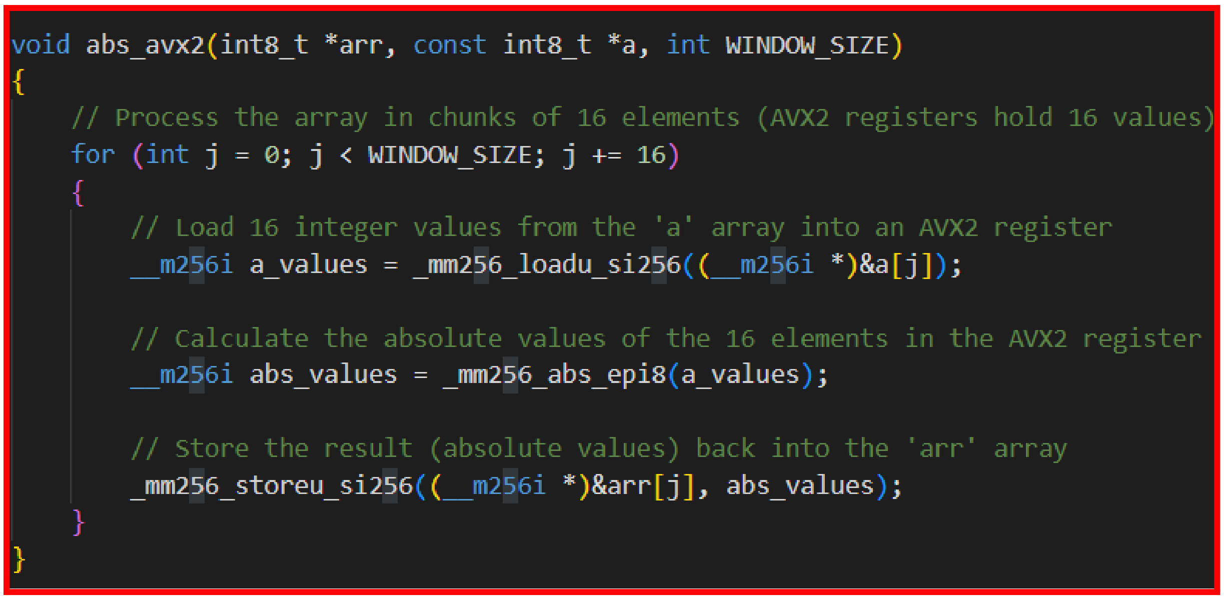
the absolute calculations can be executed efficiently across multiple data elements simultaneously, leveraging the CPU's parallel processing capabilities.

3. Data Movement Efficiency: AVX instructions facilitate efficient data movement between memory and CPU registers. The use of AVX commands such as `_mm256_loadu_si256` and `_mm256_storeu_si256` enables efficient loading and storing of data from memory to CPU registers, minimising data transfer overhead and enhancing overall performance.

4. Optimised Hardware Support: Modern CPUs are optimised to efficiently execute AVX instructions, leveraging specialised hardware units designed to handle SIMD operations effectively. As a result, using AVX commands can leverage the underlying hardware capabilities more efficiently compared to using standard library functions.

5. Reduced Function Call Overhead: By directly implementing the absolute calculation using AVX commands within the loop, the program avoids the overhead associated with function calls to the standard library function from `<math.h>`. This reduction in function call overhead can contribute to improved performance, especially when the function is called repeatedly in performance-critical code sections.

In summary, the use of AVX commands for absolute calculation improves performance by leveraging vectorisation, parallelism, efficient data movement, optimised hardware support, and reduced function call overhead, resulting in faster execution of the absolute calculation algorithm.



```
void abs_avx2(int8_t *arr, const int8_t *a, int WINDOW_SIZE)
{
    // Process the array in chunks of 16 elements (AVX2 registers hold 16 values)
    for (int j = 0; j < WINDOW_SIZE; j += 16)
    {
        // Load 16 integer values from the 'a' array into an AVX2 register
        __m256i a_values = _mm256_loadu_si256((__m256i *)&a[j]);

        // Calculate the absolute values of the 16 elements in the AVX2 register
        __m256i abs_values = _mm256_abs_epi8(a_values);

        // Store the result (absolute values) back into the 'arr' array
        _mm256_storeu_si256((__m256i *)&arr[j], abs_values);
    }
}
```

Figure 2.4.1 - Absolute function written in AVX

This approach significantly improves performance. In the following graph (Figure 2.4.2) the single-core performance is shown.

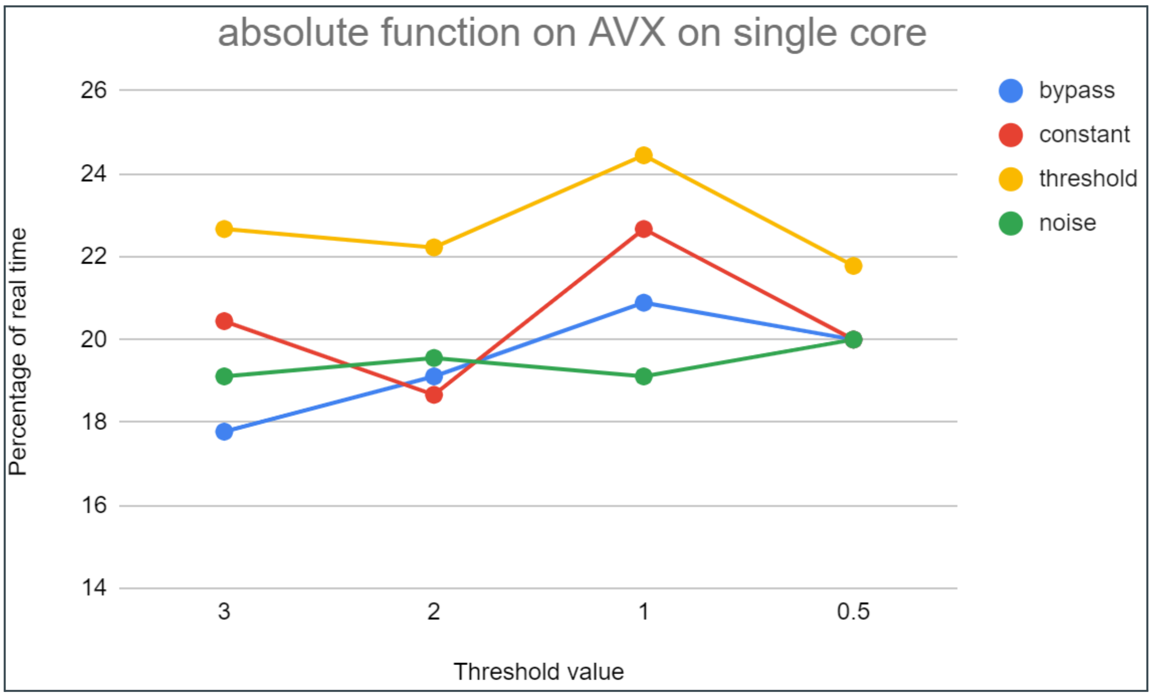


Figure 2.4.2 - Filtering time of the absolute function in AVX in single-core

The graph below (Figure 2.4.3) shows the comparison and improvement between the previous implementation and the optimised version using *the absolute function in AVX* when constant replacement is used.

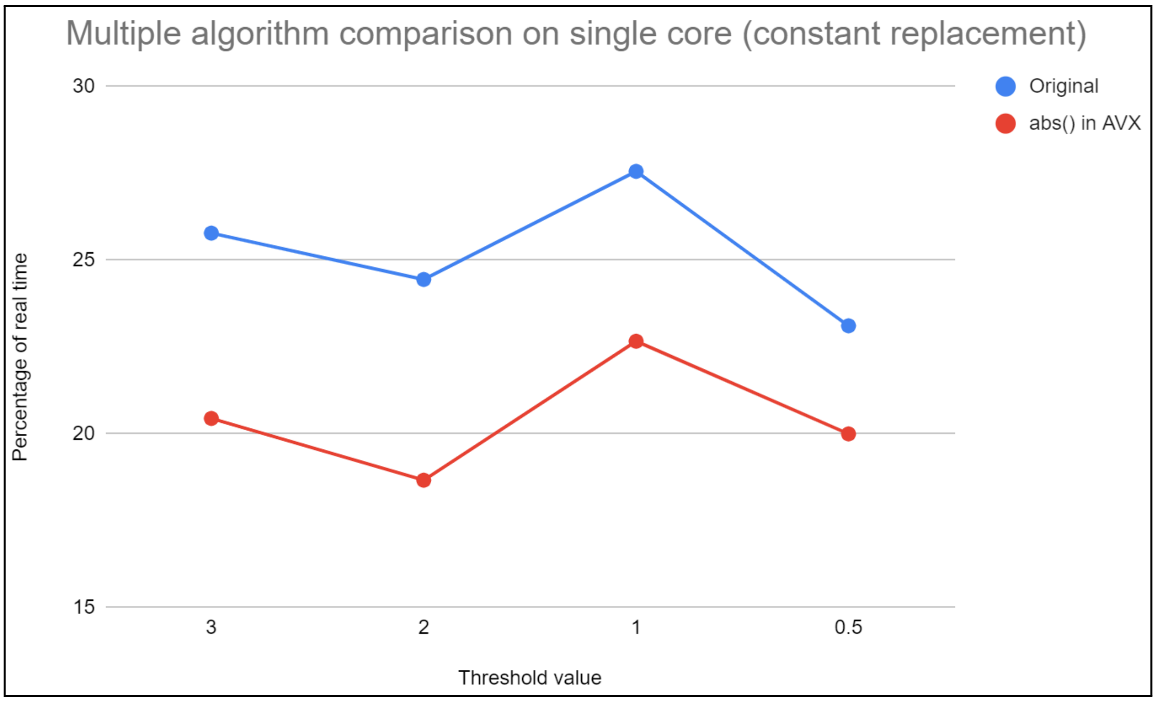


Figure 2.4.3 - Filtering time comparison in single-core

Threshold	3	2	1	0.5
Flagging Percentage	0.98	5.20	22.32	39.96

Table 2.4.1 - Flagging percentage

As this implementation improved the single-core performance, it was implemented together with 4 separate memory for buffers. As seen in the graph below (Figure 2.4.4), it takes 83% of real-time on average. In this iteration, the performance is much better. Also, no values crossed the real-time value.

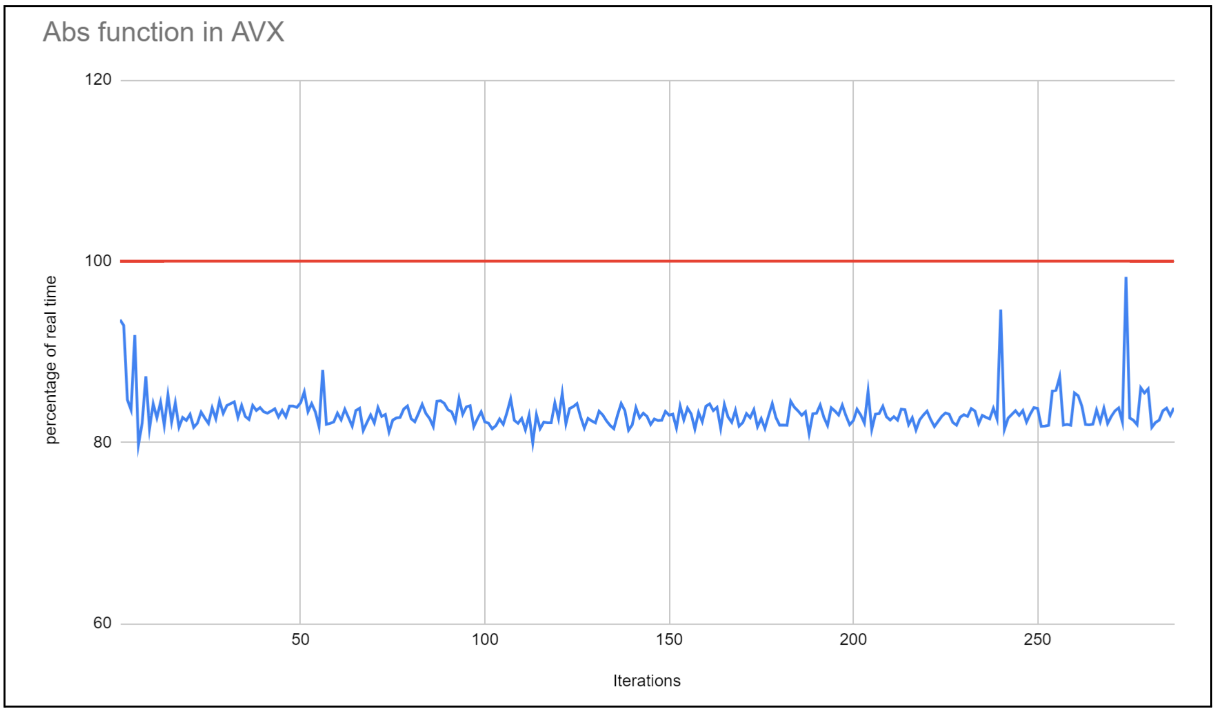


Figure 2.4.4 - Filtering time of the absolute function in AVX in multi-core

2.5 Combined performance: Four separate memory in double buffer + absolute function in AVX

In this approach 4 separate memory and absolute function in avx are used together as these two approaches give the most optimised performance. The outcome is rigorously tested with multiple iterations. Here are the output graphs (Figure 2.5.1.1 to 2.5.1.3).
The average filtering time is written below the graphs.

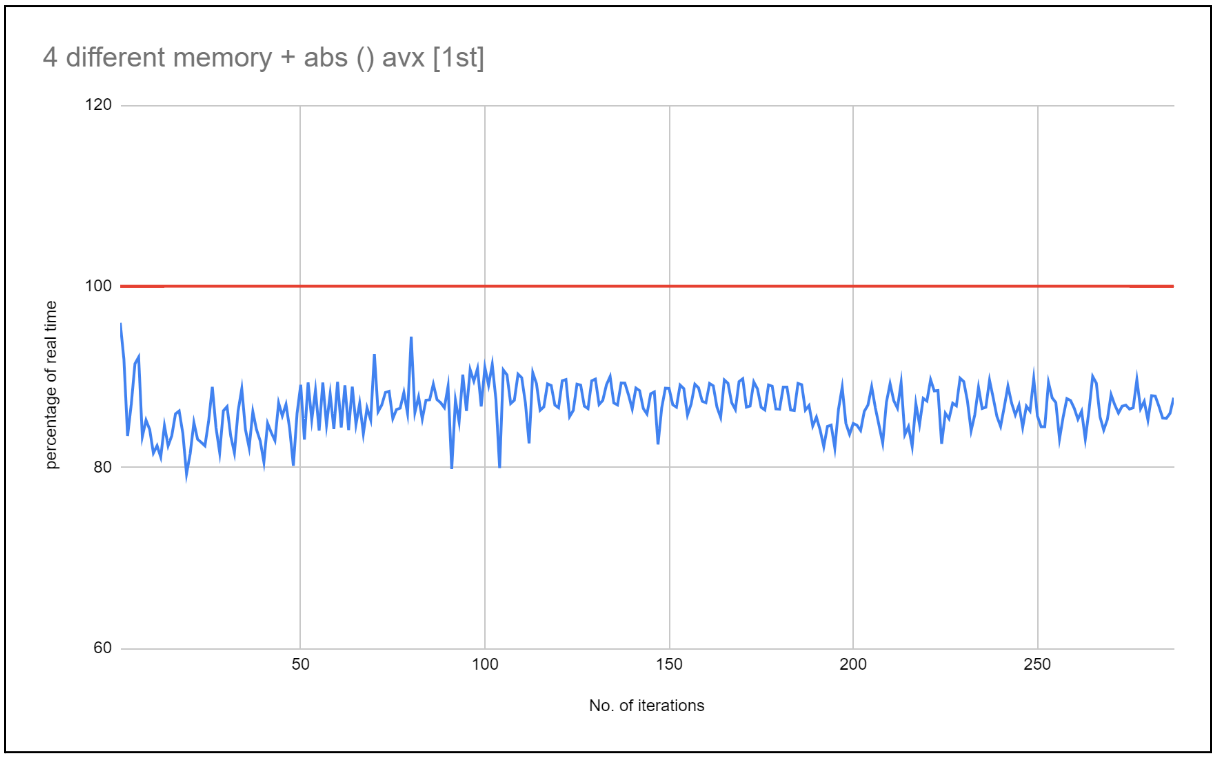


Figure 2.5.1 - Filtering time of 4 different memory + abs function in AVX in multi-core (1)

Average filtering time: 86.6% of real-time
No values outside real-time boundary

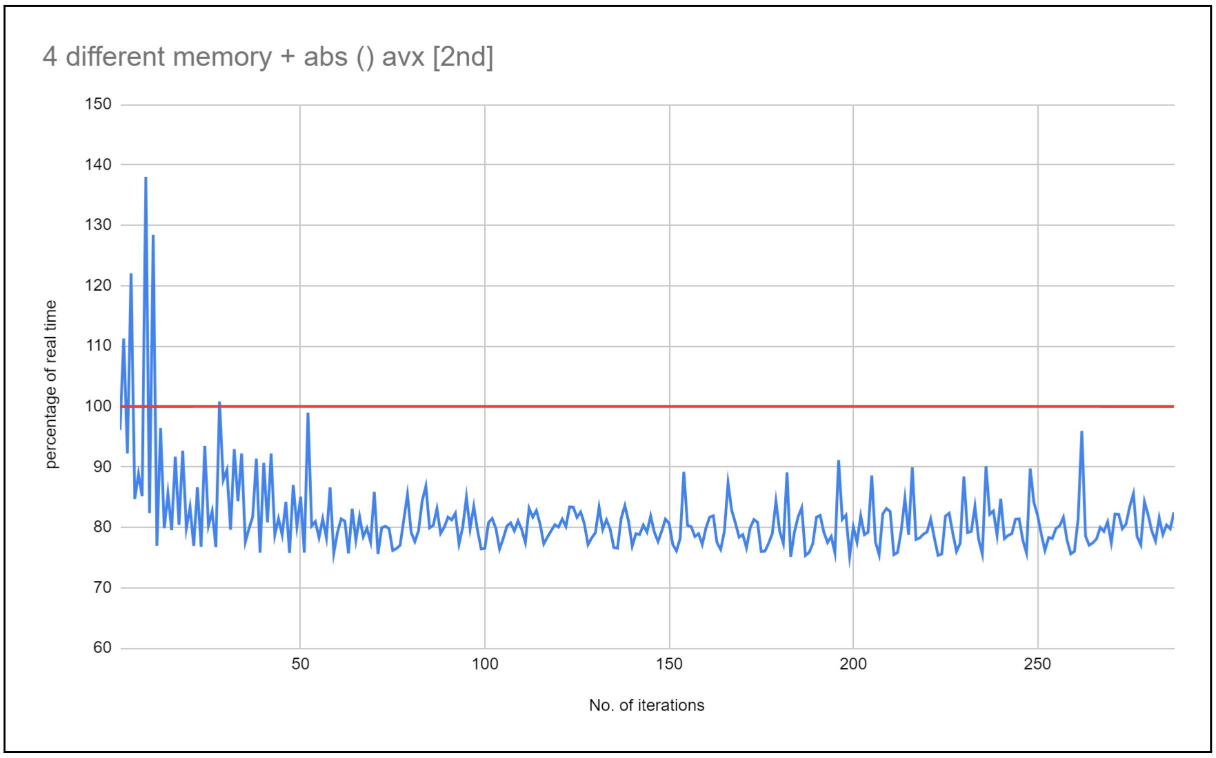


Figure 2.5.2 - Filtering time of 4 different memory + abs function in AVX in multi-core (2)

Average filtering time: 81.5% of real-time
Initially 5 values are outside the real-time boundary and then all values are under real-time

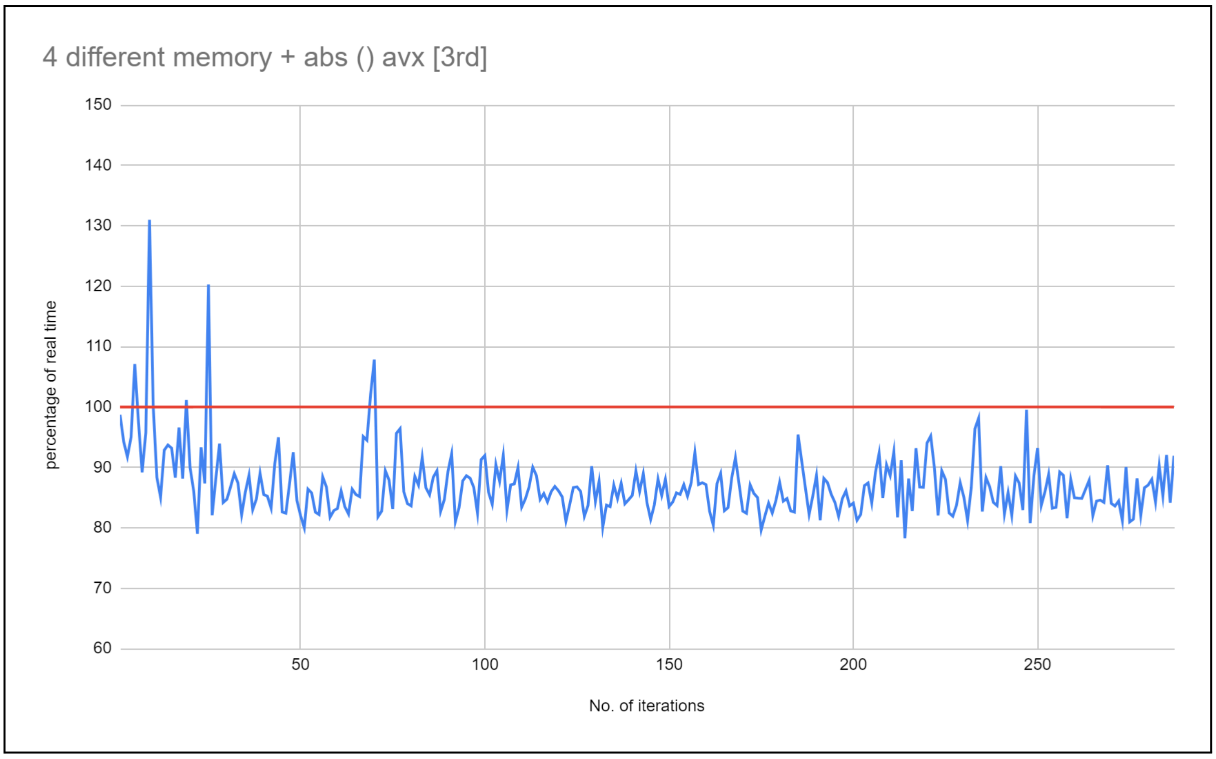


Figure 2.5.3 - Filtering time of 4 different memory + abs function in AVX in multi-core (3)

Average filtering time: 87.1% of real-time

Initially 5 values are outside the real-time boundary and then again 2 values crossed real-time.

This approach gives the most consistent result. Some initial values cross the real-time threshold, but they stay within 85% of the real-time value on average after some iterations

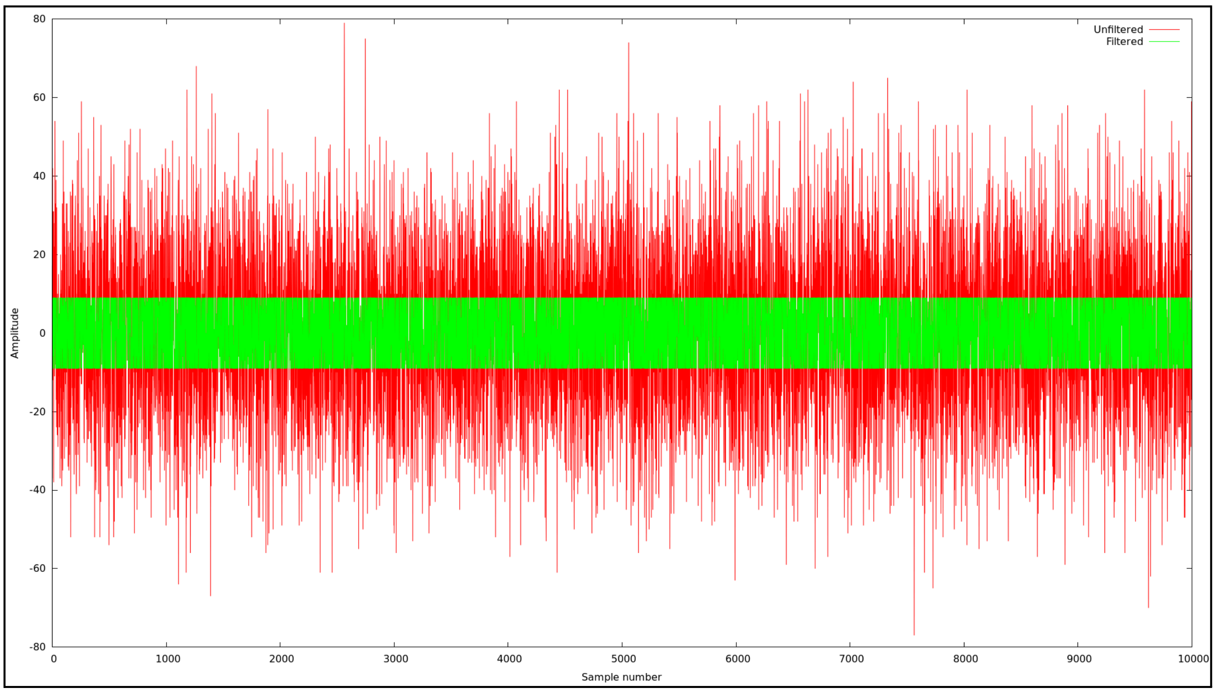


Figure 2.5.4 - Overlay plot of filtered over unfiltered signal, N =0.5, threshold replacement

To further test this method different buffer sizes and window sizes were used. All previous buffers were 256MB in size and window size was 16k x 16k. Now it is benchmarked with 4k x 4k, 6k x 6k, 12k x 12k, 18k x 18k, 24k x 24k, 32k x 32k, 44k x 44k window sizes.

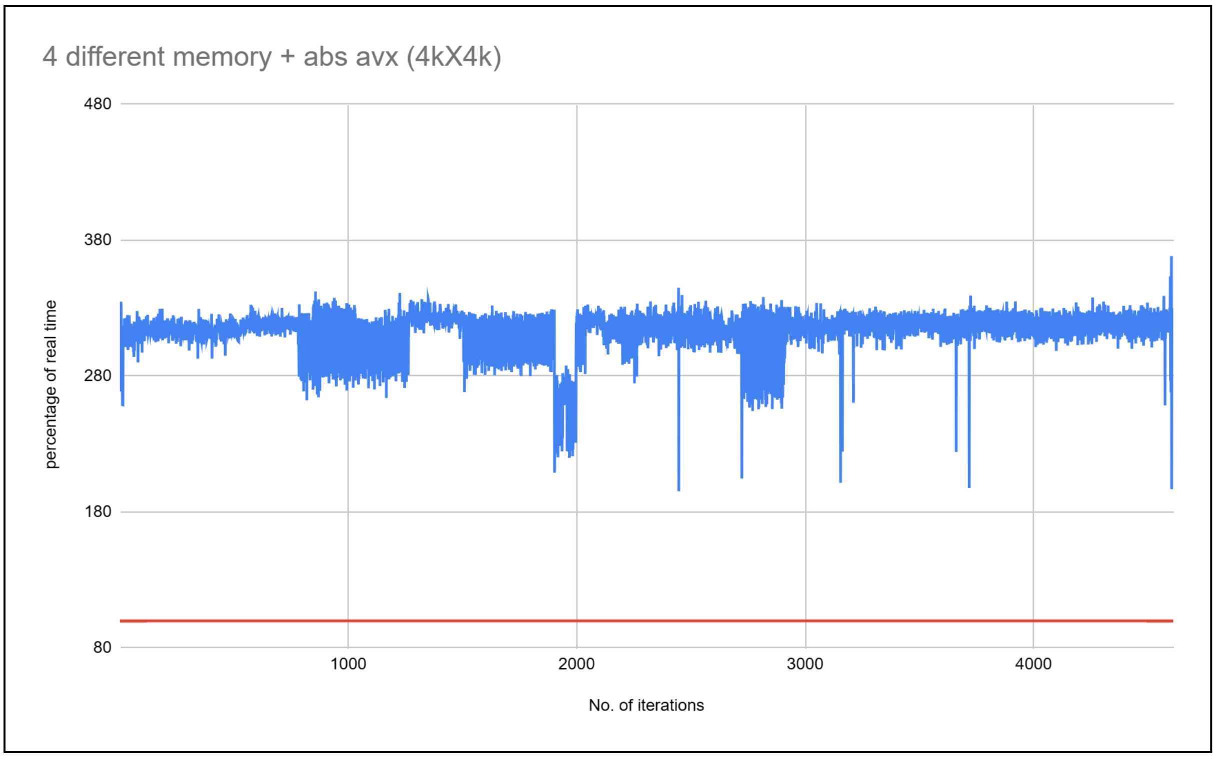


Figure 2.5.5 - Filtering time in 16MB chunk size and 4k x 4k window size

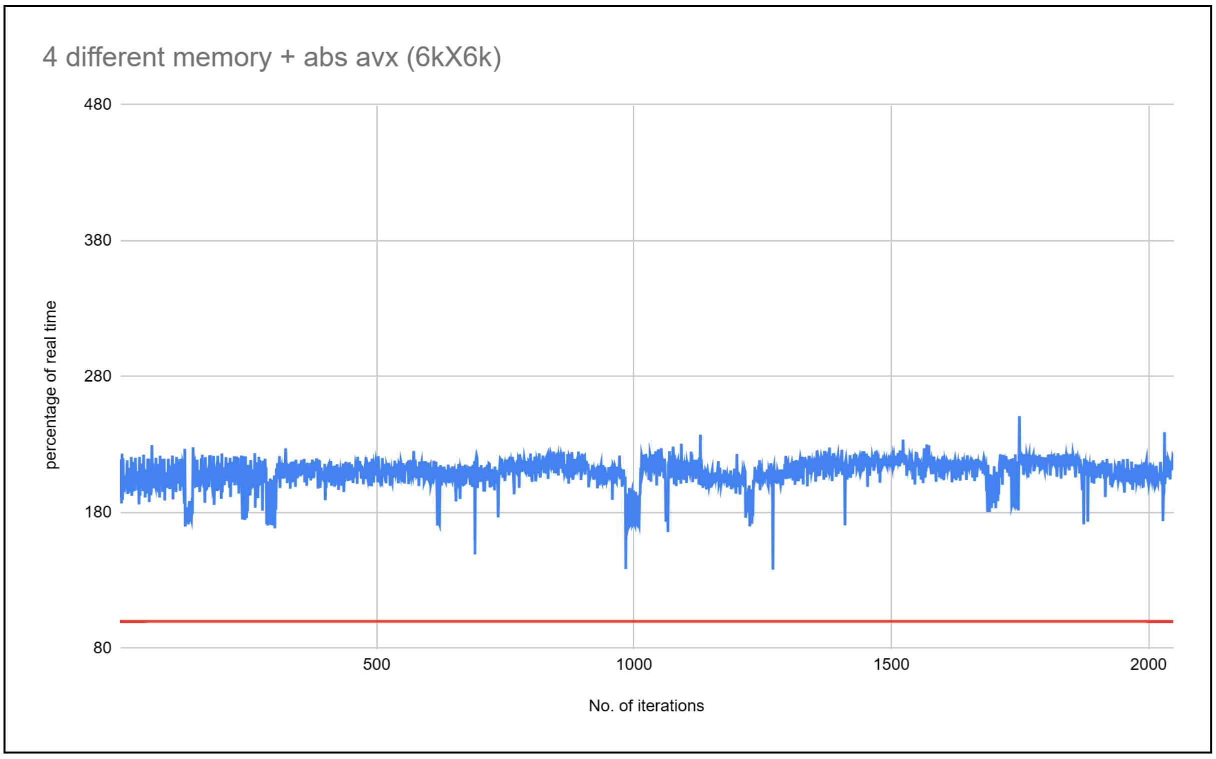


Figure 2.5.6 - Filtering time in 36MB chunk size and 6k x 6k window size

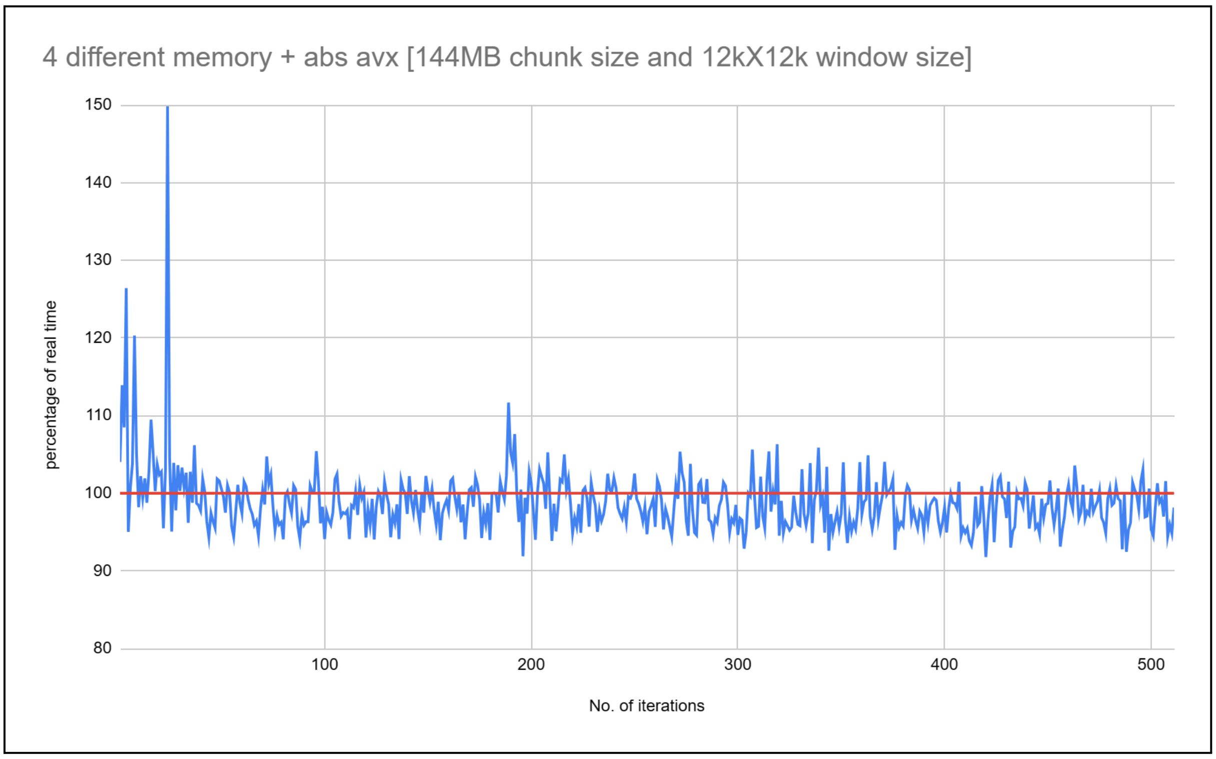


Figure 2.5.7 - Filtering time in 144MB chunk size and 12k x 12k window size

As seen from the graphs, reducing chunk size and window size increases filtering time by a large margin. All iterations went beyond the real-time threshold.

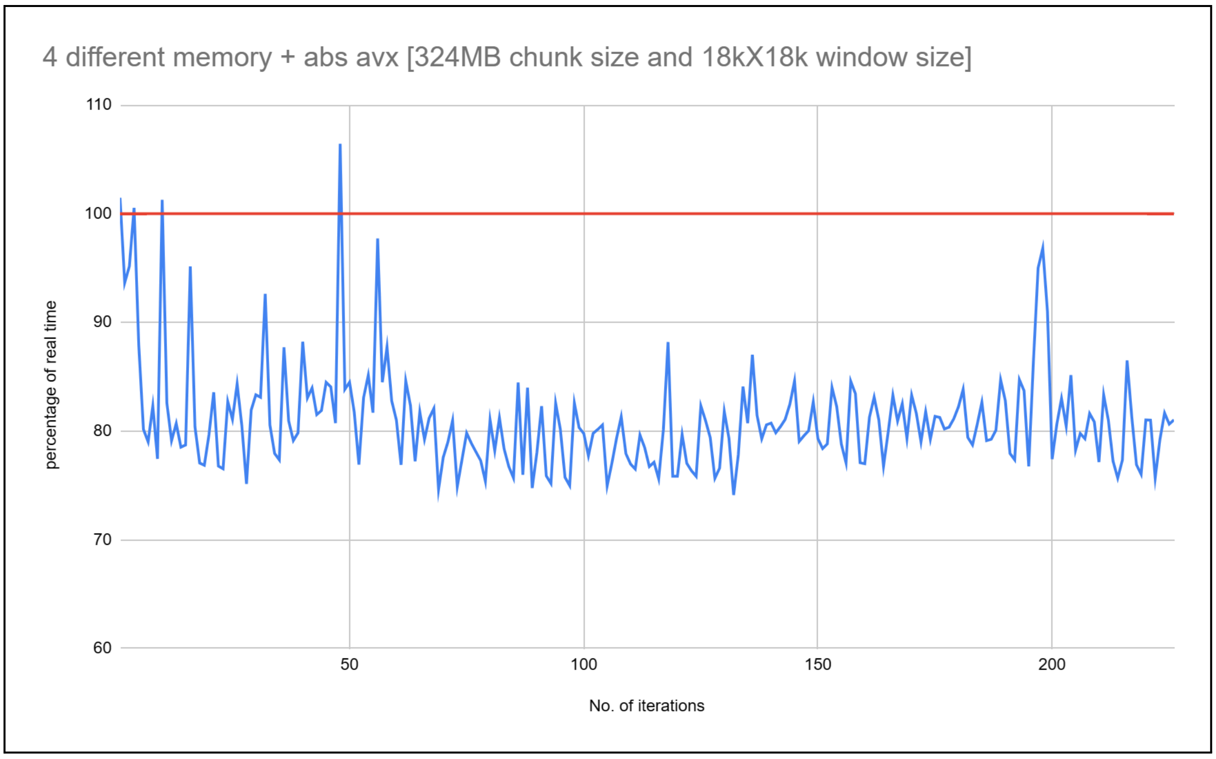


Figure 2.5.8 - Filtering time in 324MB chunk size and 18k x 18k window size

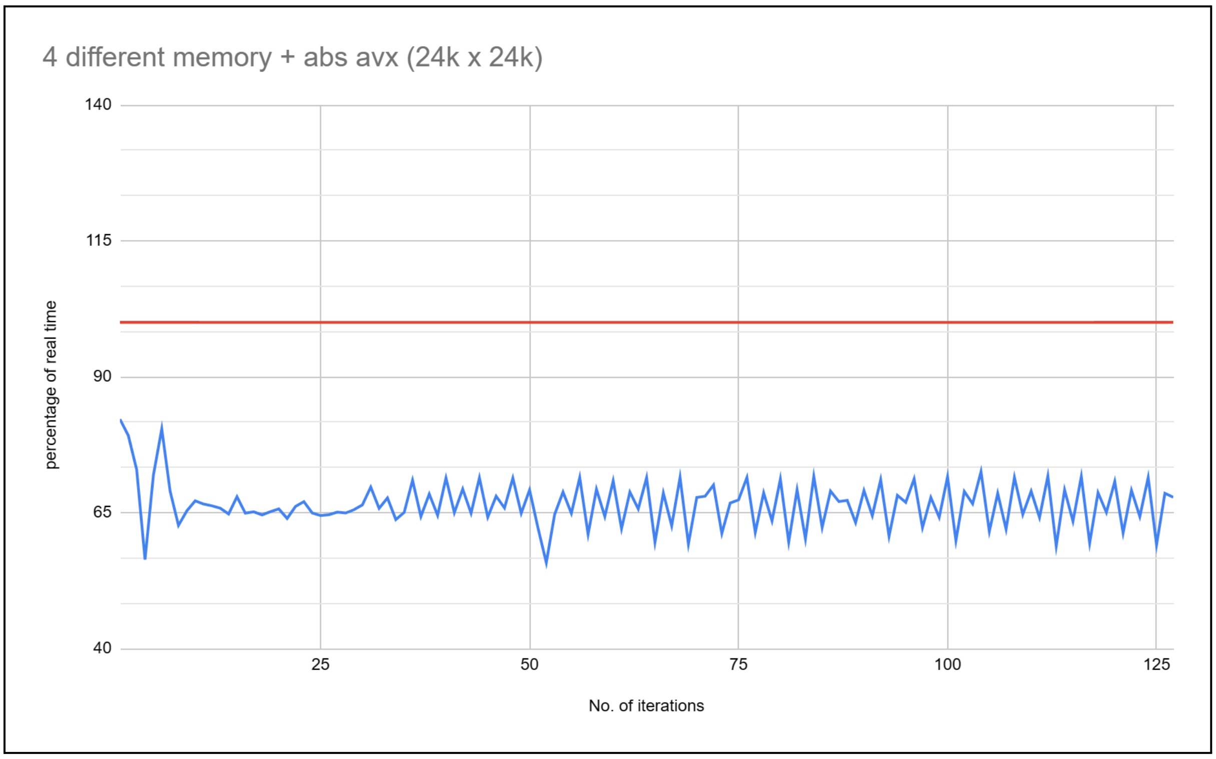


Figure 2.5.9 - Filtering time in 576MB chunk size and 24k x 24k window size

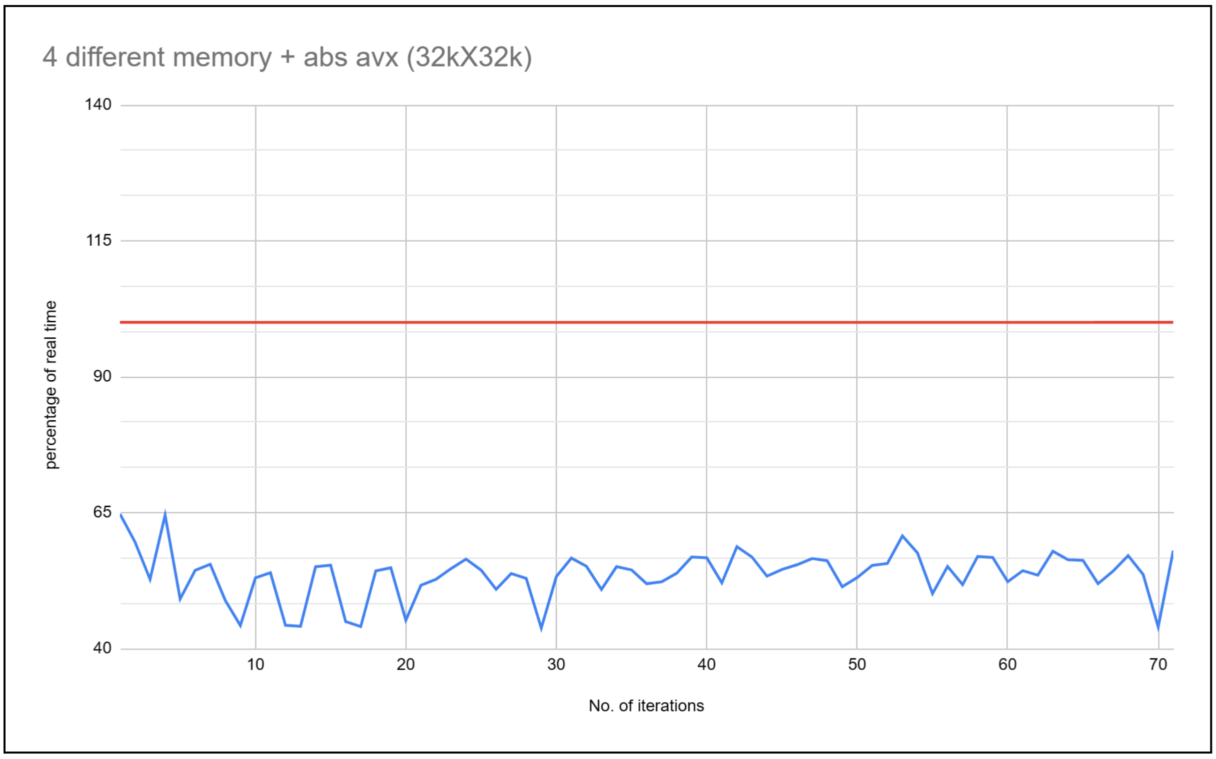


Figure 2.5.10 - Filtering time in 1024MB chunk size and 32k x 32k window size

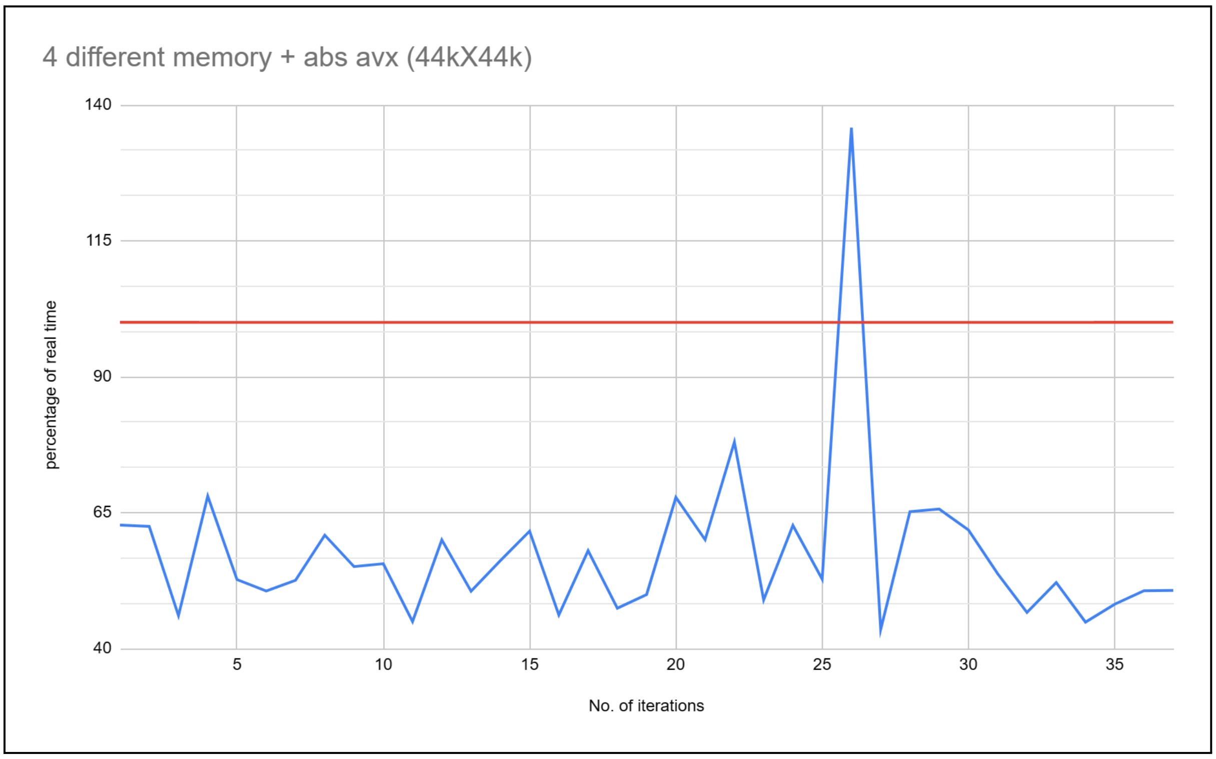


Figure 2.5.11 - Filtering time in 1936MB chunk size and 44k x 44k window size

As seen from the graph, increasing the chunk size decreases filtering time by a large margin and improves performance.

Here is a table that contains the filtering time of various window sizes in terms of the percentage of real-time.

MAD Window Size	MOM window size	Chunk size	Average filtering time (percentage of real-time)
4k	4k	16MB	312.7%
6k	6k	36MB	209.5%
12k	12k	144MB	98.8%
16k	16k	256MB	86%
18k	18k	324MB	81.1%
24k	24k	576MB	66.6%
32k	32k	1024MB	53.6%
44k	44k	1936MB	57.6%

Table 2.5.1 - Filtering time comparison of different window size

From the graphs and table, it is seen that increasing MOM window size and MAD window size decreases filtering time. But the improvement is seen up to a certain size, currently which is 32k x 32k window size. Beyond that, the filtering time starts to increase again.

To further check this, the 32k window size can be checked using gcc version11.x.

2.6 Effect of compiler versions and machines on optimisation

The optimisation of code can be significantly influenced by various factors including compiler versions and underlying hardware characteristics. Different compiler versions may offer varying optimisation levels, ranging from minimal optimisation to aggressive optimisation techniques. Higher optimisation levels can result in more efficient code generation but may also increase compilation time and code size. Here are different optimisation methods on gcc version 11.x on the gpbcorr6 machine.

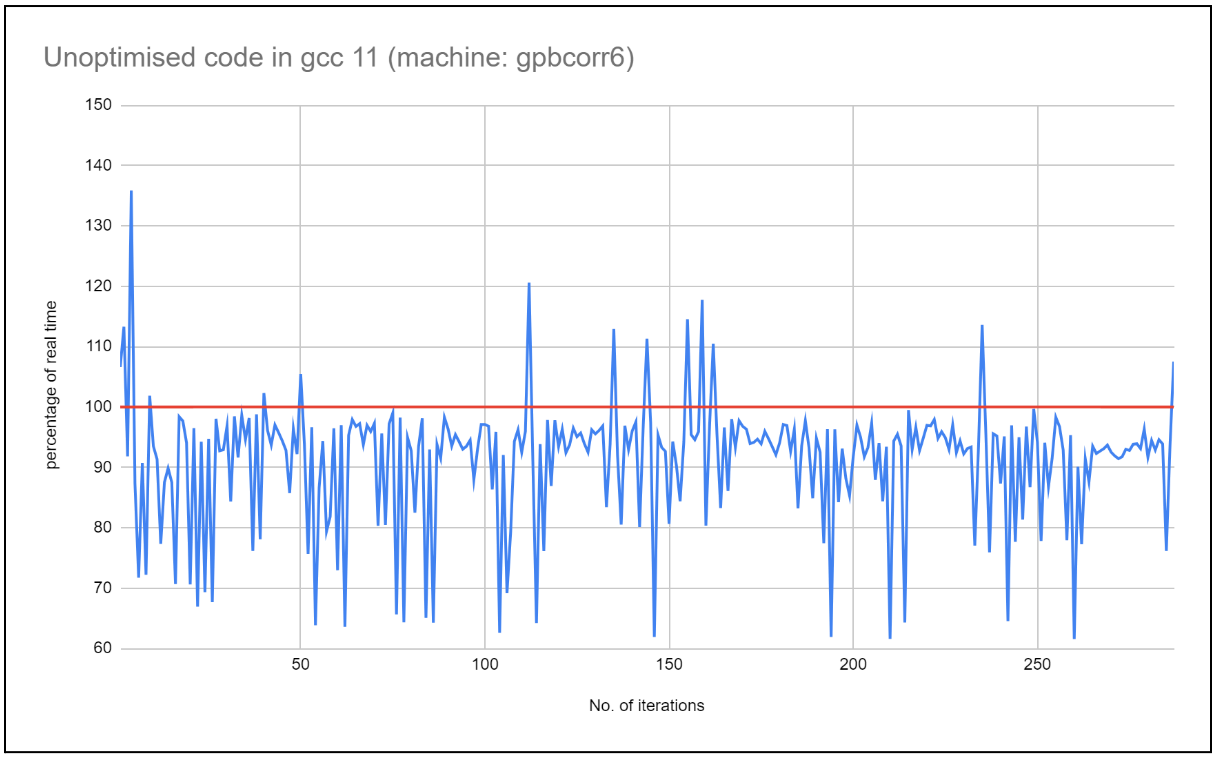


Figure 2.6.1 - Filtering time of unoptimised code in multicore using gcc - 11.x

Average filtering time: 91% real-time, which is an improvement of 4% over the previous implementation using gcc 4.8.5. However, in this case, on multiple iterations, the filtering time crosses the real-time threshold.

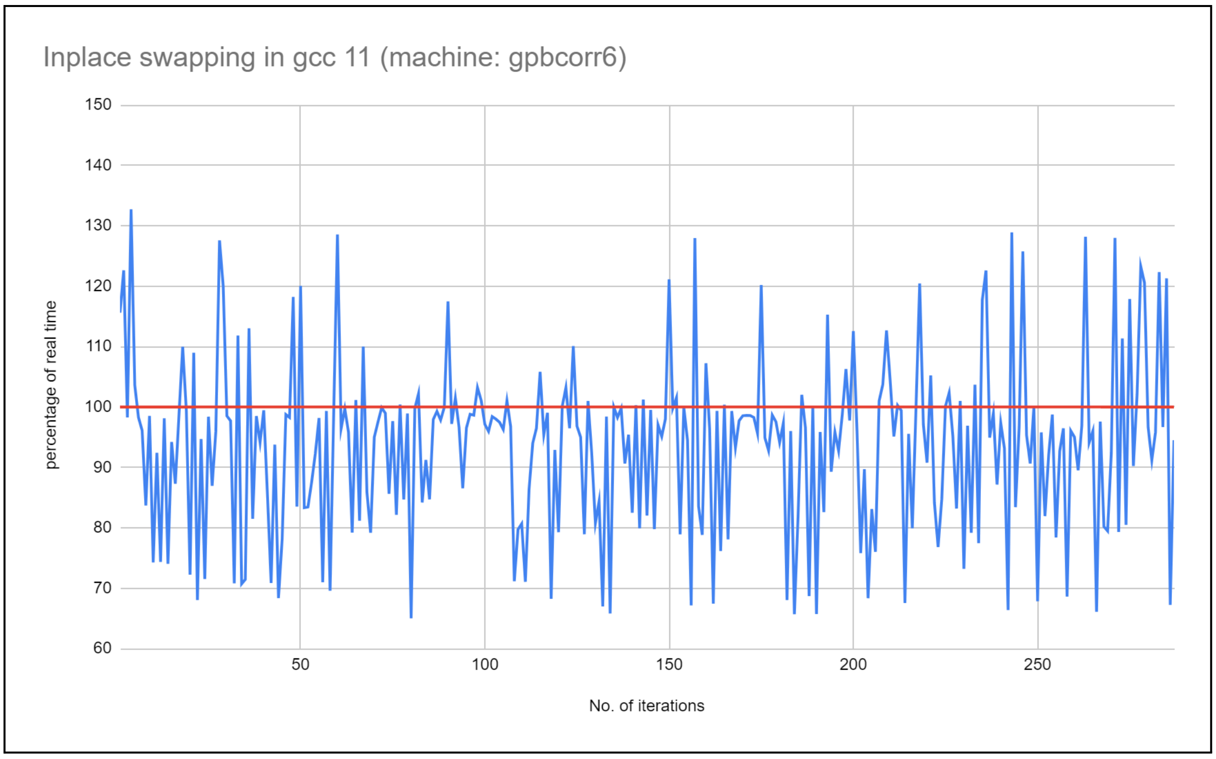


Figure 2.6.2 - Filtering time of inplace-swapping in multicore using gcc - 11.x

Average filtering time: 93% real-time, which is the same as the previous implementation using gcc 4.8.5. But here also on multiple iterations filtering time crosses the real-time threshold.

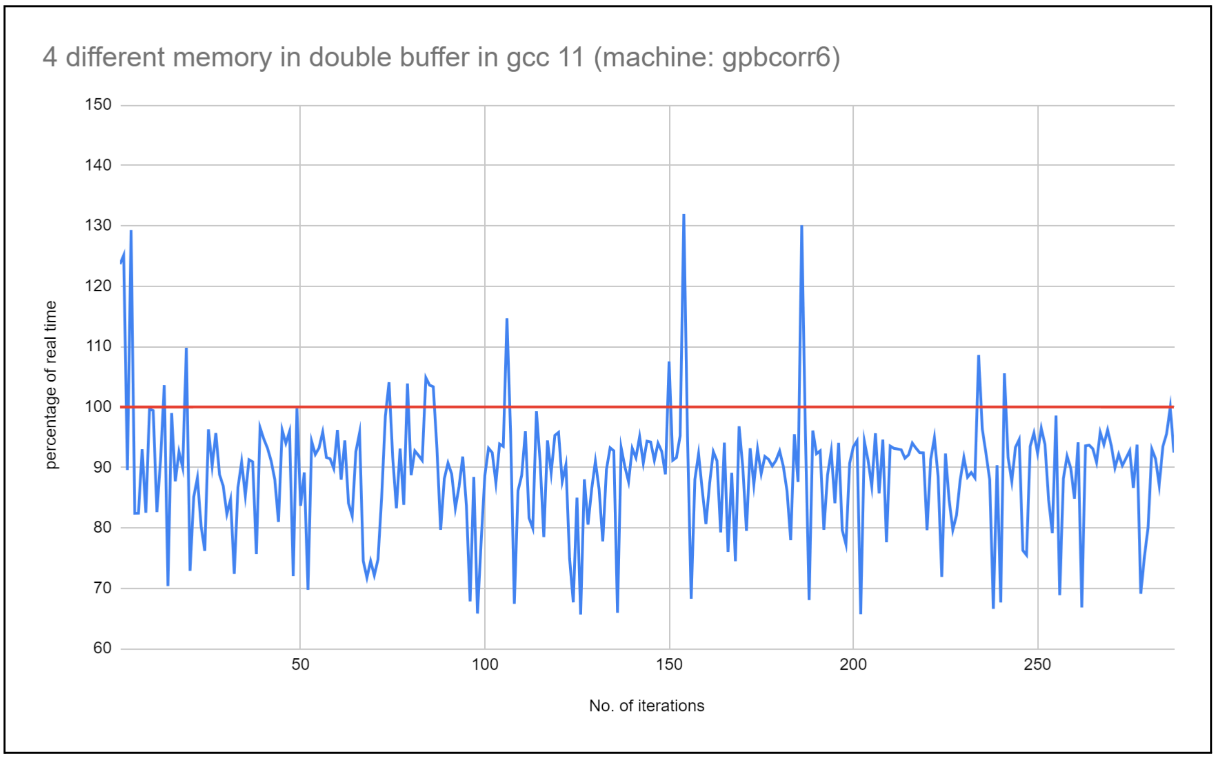


Figure 2.6.3 - Filtering time of 4 different memory in the buffer in multicore using gcc - 11.x

Average filtering time: 89% of real-time, which is an improvement of 5% over the previous implementation using gcc 4.8.5
But here also on multiple iterations filtering time crosses the real-time threshold.

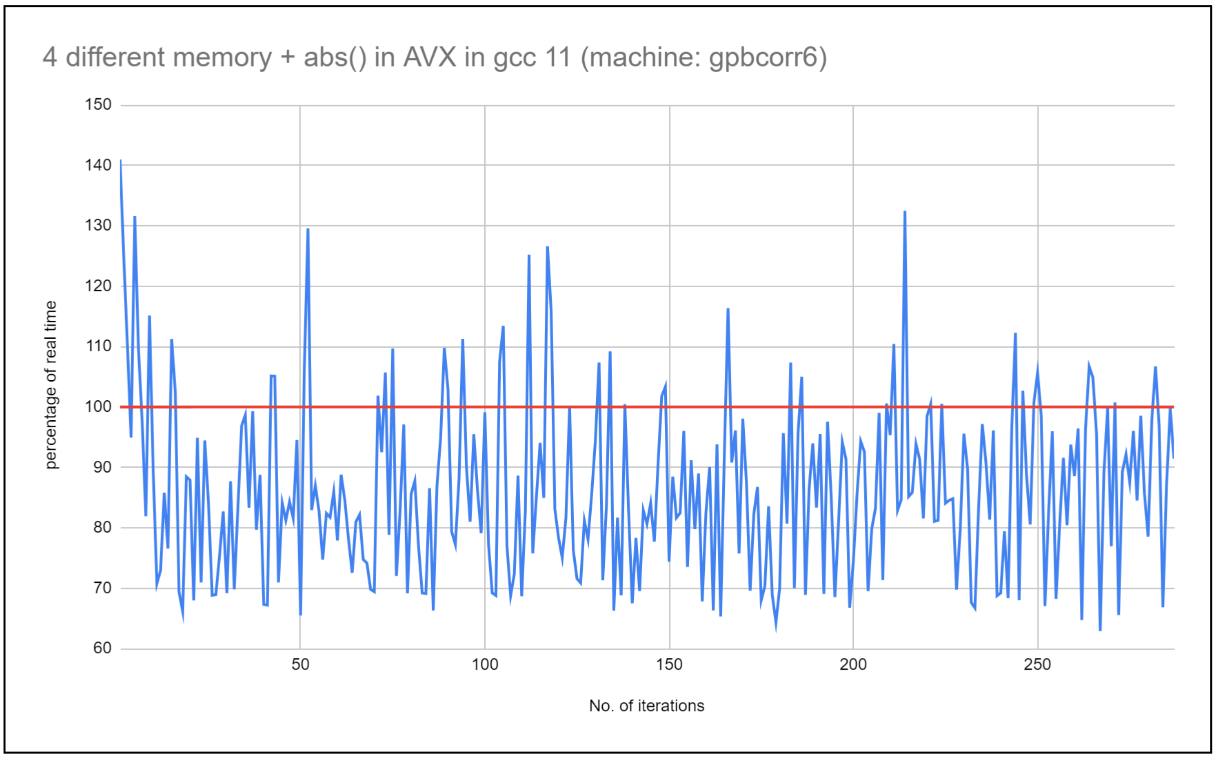


Figure 2.6.4 - Filtering time of 4 different memory in buffer + absolute function in avx in multicore using gcc - 11.x

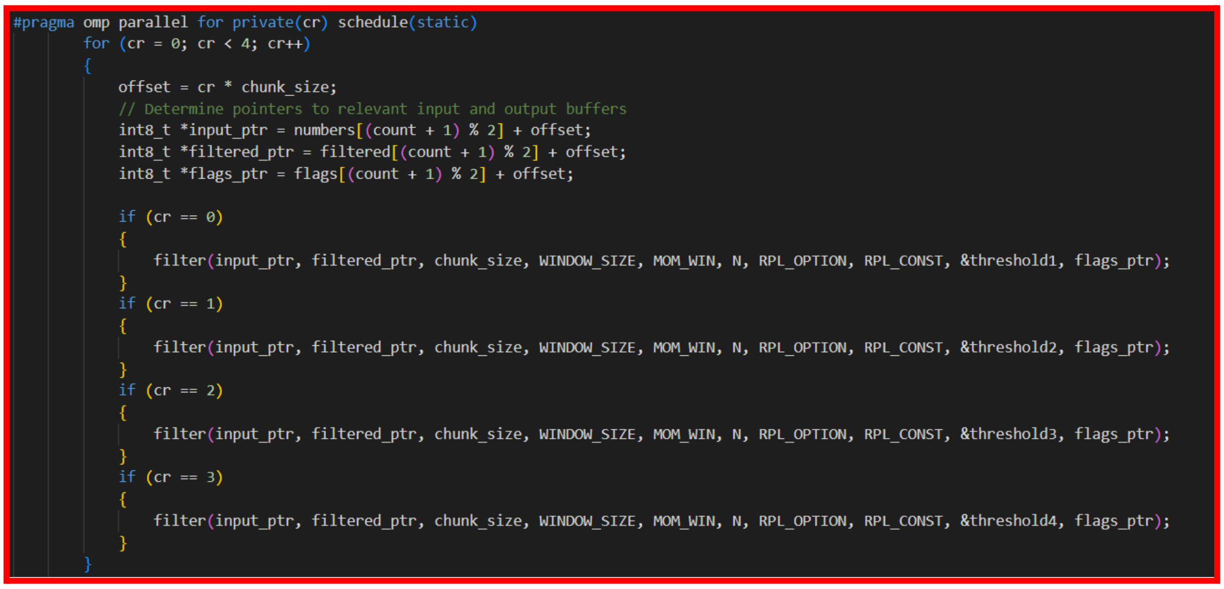
Average filtering time: 86% real-time, which is the same as the previous implementation using gcc 4.8.5
But here also on multiple iterations filtering time crosses the real-time threshold.

So it is seen that in general using gcc compiler version 11 improves performance by reducing filtering time to some degree.

2.7 Other approaches

2.7.1 OpenMP optimisation and its combinations

In the previous approach, the openmp loop had some redundant calculations which have been optimised to reduce time. The modification and benchmarking are shown below.



```
#pragma omp parallel for private(cr) schedule(static)
        for (cr = 0; cr < 4; cr++)
        {
            offset = cr * chunk_size;
            // Determine pointers to relevant input and output buffers
            int8_t *input_ptr = numbers[(count + 1) % 2] + offset;
            int8_t *filtered_ptr = filtered[(count + 1) % 2] + offset;
            int8_t *flags_ptr = flags[(count + 1) % 2] + offset;

            if (cr == 0)
            {
                filter(input_ptr, filtered_ptr, chunk_size, WINDOW_SIZE, MOM_WIN, N, RPL_OPTION, RPL_CONST, &threshold1, flags_ptr);
            }
            if (cr == 1)
            {
                filter(input_ptr, filtered_ptr, chunk_size, WINDOW_SIZE, MOM_WIN, N, RPL_OPTION, RPL_CONST, &threshold2, flags_ptr);
            }
            if (cr == 2)
            {
                filter(input_ptr, filtered_ptr, chunk_size, WINDOW_SIZE, MOM_WIN, N, RPL_OPTION, RPL_CONST, &threshold3, flags_ptr);
            }
            if (cr == 3)
            {
                filter(input_ptr, filtered_ptr, chunk_size, WINDOW_SIZE, MOM_WIN, N, RPL_OPTION, RPL_CONST, &threshold4, flags_ptr);
            }
        }
```

Figure 2.7.1.1 - Optimised openmp loop in multicore

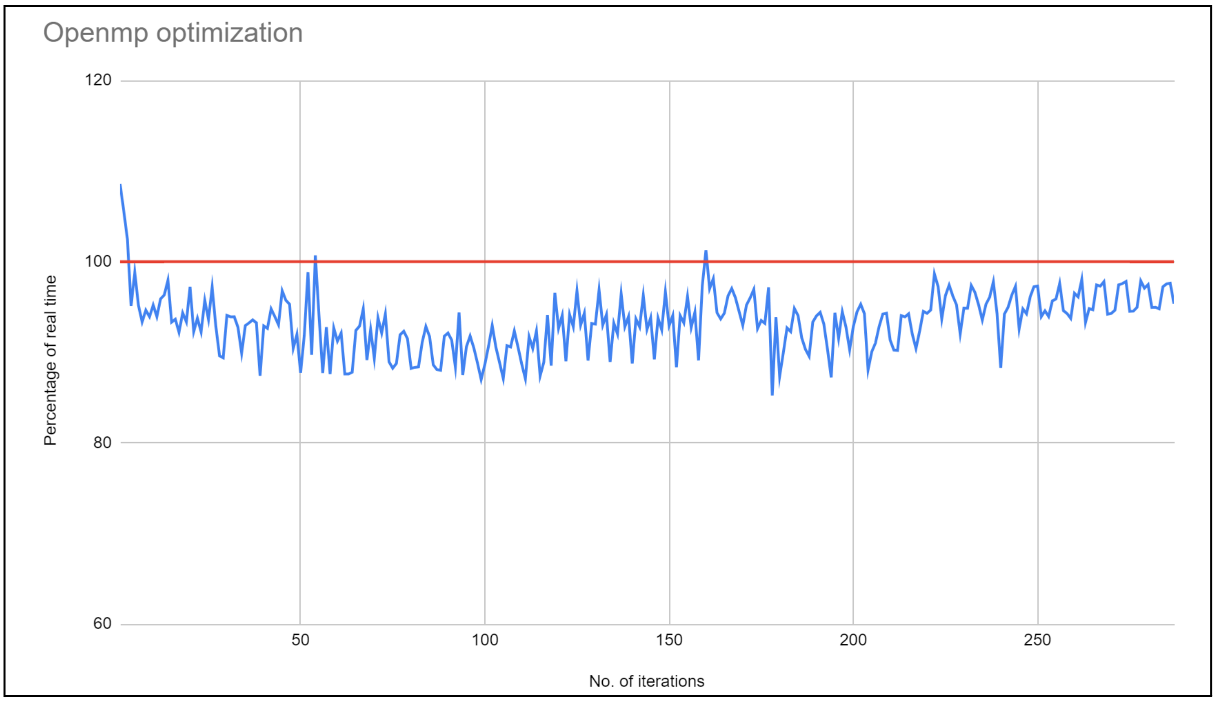


Figure 2.7.1.2 - Filtering time of openmp optimisation in multicore

Average filtering time: 86% of real-time

Here on some iterations filtering time crosses the real-time threshold. otherwise, most values are well under the real-time threshold.

In the previous approach the pointer for input, filtered data and flags were calculated whenever calling the "filter" function. But in the modification, these pointer values are calculated once inside the loop and used in the function call

This optimised openmp loop was used in combination with other approaches as well.

1. OpenMP optimisation + 4 different memory in the buffer

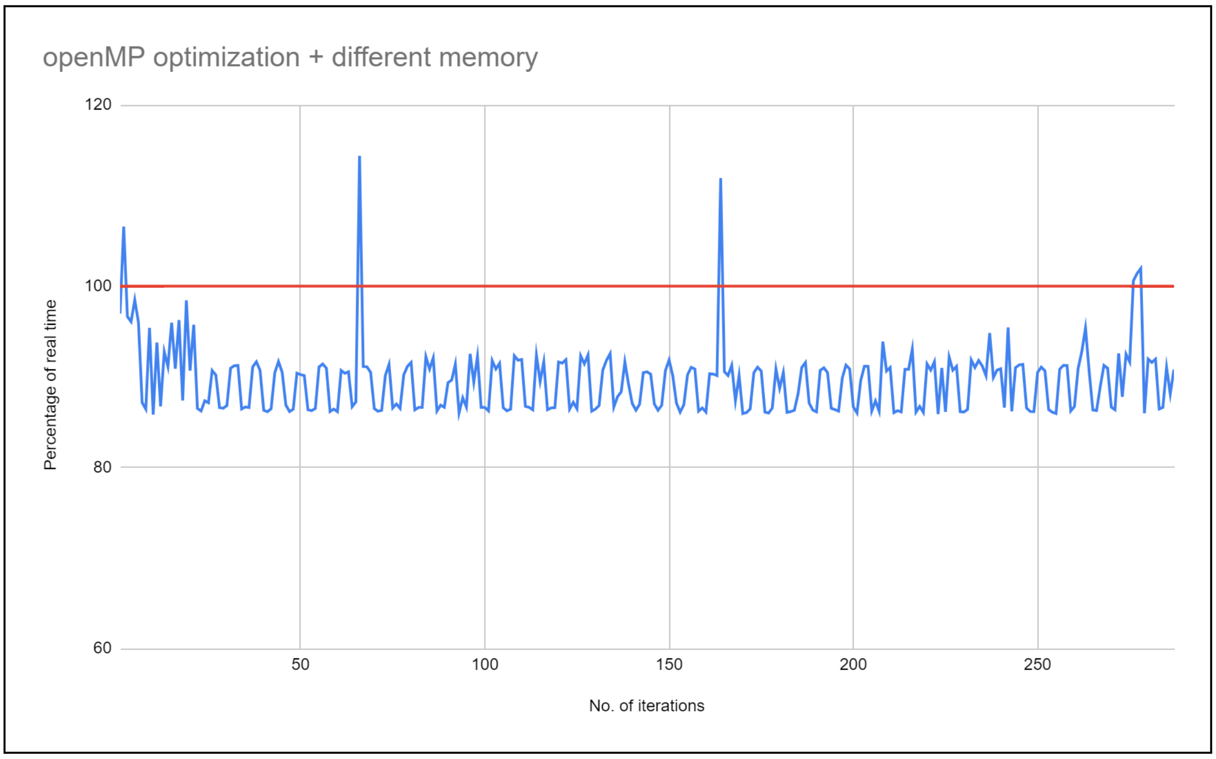


Figure 2.7.1.3 - Filtering time of openmp optimisation + 4 different memory in multicore

Average filtering time: 89.5% of real-time
Here also we can see spikes that crossed real-time percentages during filtering

2. OpenMP optimisation + 4 different memory in buffer + abs function in avx

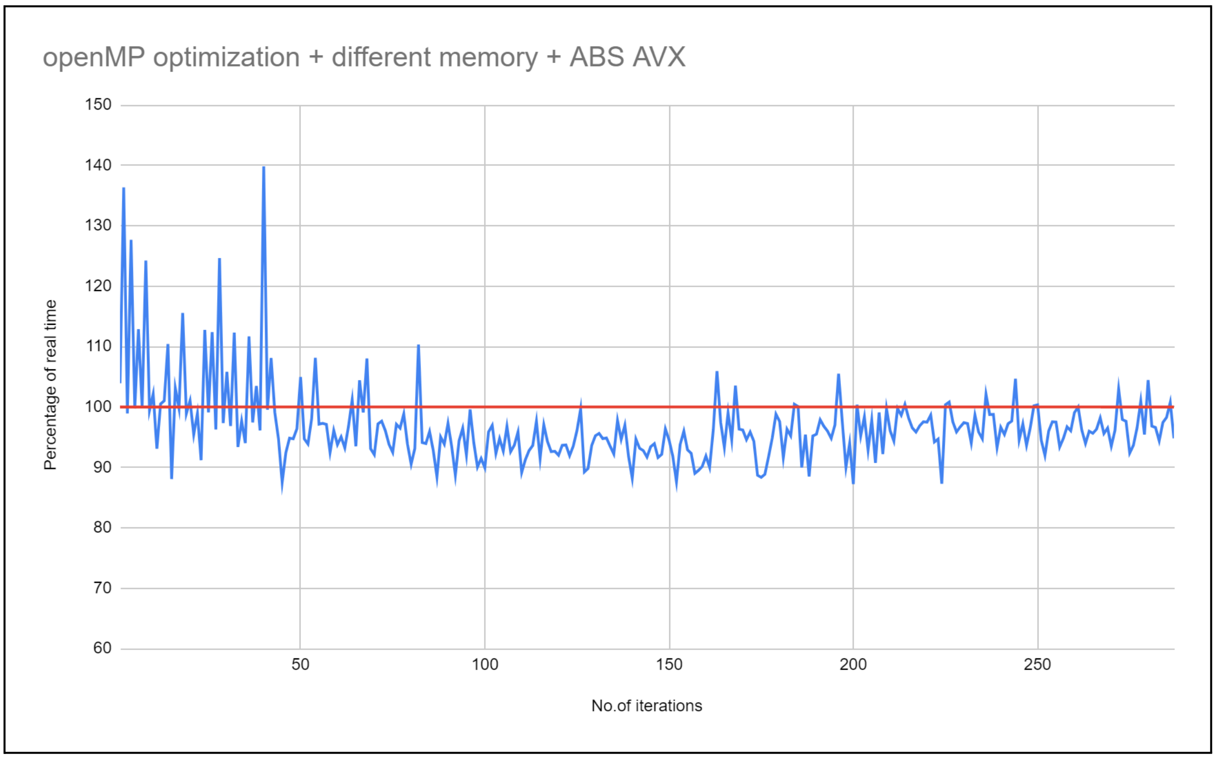


Figure 2.7.1.4 - Filtering time of openmp optimisation + 4 different memory + abs function in avx in multicore

Average filtering time: 96.8% of real-time
This approach is noticed to be very poor in filtering and time taken to filter is very high. Also on multiple occasions, the filtering time is more than the real-time limit

As seen from the graphs, individually the optimised openmp loop reduces filtering time, but if combined with other approaches then the performance becomes worse.

2.7.2 Using pthread for parallelisation

Pthreads (POSIX threads) is a threading library available on Unix-like operating systems, including Linux and macOS. Pthreads provides a standardised API for creating and managing threads in a multithreaded application. [8]

Pthread optimisation refers to the techniques and practices used to improve the performance and efficiency of multithreaded programs developed using the pthreads library. Here are some common pthread optimisation techniques: [9]

1. Thread Pooling: Reusing threads from a pre-allocated pool instead of creating and destroying threads dynamically can reduce the overhead associated with thread creation and termination.

2. Lock-Free Data Structures: Using lock-free data structures and algorithms can eliminate the need for traditional locking mechanisms like mutexes and reduce contention among threads, improving scalability and performance.

3. Thread Affinity: Assigning specific threads to execute on specific CPU cores can reduce cache thrashing and improve cache locality, leading to better performance, especially on multicore systems.

4. Fine-grained locking: Minimising the scope and duration of locks by using fine-grained locking techniques can reduce contention and improve concurrency in multithreaded programs.

Comparison of pthreads with OpenMP

- Benefits of Pthreads over OpenMP:

1. Fine-Grained Control: Pthreads provide fine-grained control over thread creation, management, and synchronisation, allowing developers to optimise performance based on application-specific requirements.

2. Platform Independence: Pthreads are a standardised API available on various Unix-like operating systems, making it easier to write portable multithreaded code that can run across different platforms.

3. Low-Level Optimisation: Pthreads allow developers to implement low-level optimisations tailored to the specific characteristics of the target hardware and workload, providing greater flexibility and control compared to higher-level threading models like OpenMP.

- Drawbacks of Pthreads compared to OpenMP:

1. Complexity: Multithreaded programming with Pthreads can be more complex and error-prone compared to higher-level threading models like OpenMP, as developers are responsible for managing thread creation, synchronisation, and communication explicitly.

2. Verbose Syntax: Pthreads require more lines of code and boilerplate to accomplish common threading tasks compared to OpenMP, which provides a simpler and more concise syntax for parallelising loops and regions of code.

3. Potential for Deadlocks and Race Conditions: Pthreads programming requires careful attention to synchronisation and locking mechanisms to prevent race conditions, deadlocks, and other concurrency-related issues, which can be challenging to debug and diagnose.

Benchmarking of pthread is shown below in Figure 2.7.2.1.

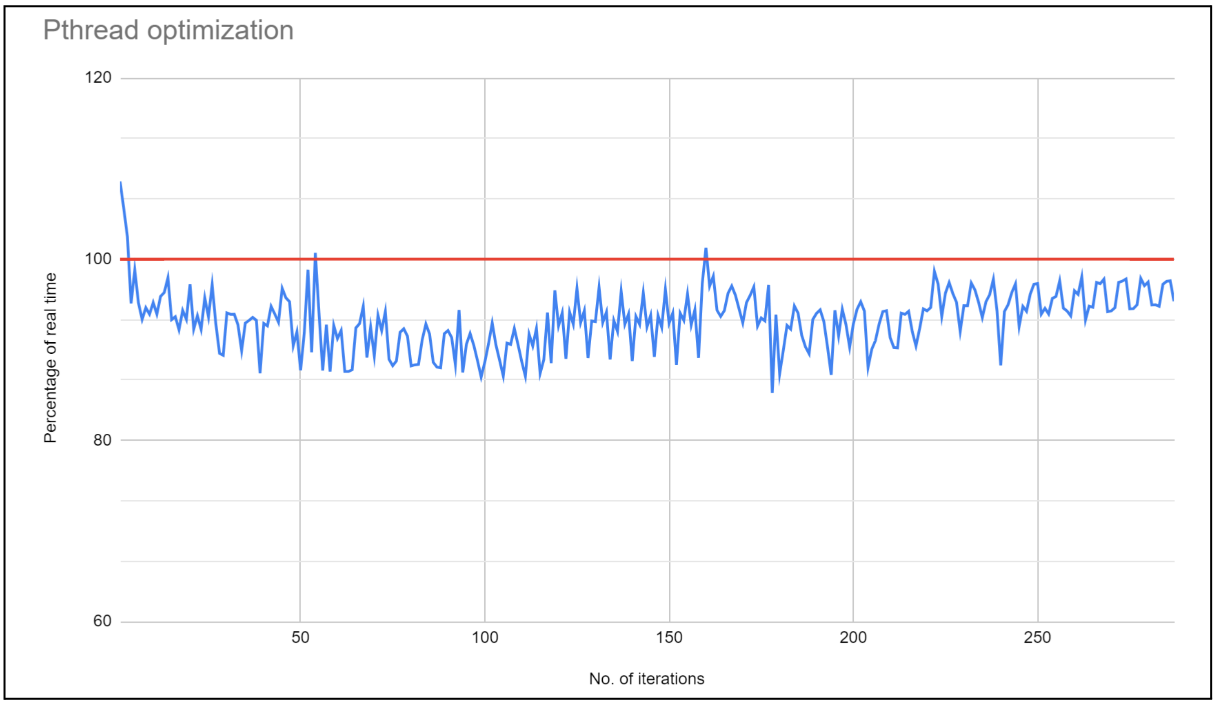


Figure 2.7.2.1 - Filtering time of pthread optimisation in multicore

Average filtering time: 94.2% of real-time

In summary, while pthreads offer greater flexibility and control over thread management and optimisation, they also require more effort and expertise to use effectively compared to higher-level threading models like OpenMP. The choice between pthreads and OpenMP depends on factors such as the level of control and optimisation required, programming complexity, and portability considerations.

Chapter 3: Automated verification of the code

Another essential part of this project was to automate the testing and benchmarking. Simplifying these processes helps to further modify the program. It prevents extensive manual testing. A streamlined benchmarking method is created to improve efficiency.

3.1 Automating the process of verification

Currently, the verification needs manual labour and individually checking each output with the original one. Shell scripts for testing and benchmarking the outputs of single-core and multi-core codes with both small and large datasets are created for this purpose.

Steps for verification:

1. Run the reference code. Here the final code of Satyarth Gupta was used as a golden reference. This program generates .out files for filtered data and flagged data which is used for verification.
2. Compile all C codes that need to be verified
3. Then execute the programs one by one.
4. Each program generates .out files for filtered data and flags. These outputs are compared with the reference outputs with the "diff -s" command.

Details of which shell script to use for each scenario are attached in the appendix.

3.2 Plotting options

A c program file named "2.1.2_rfi_filter_omp_4_memory_abs_avx_plotting.c" is created which is used for plotting filtered data over unfiltered data. It is a slightly modified version of the final code which is used to generate .out files as output. Samples from that output can be taken for plotting the data in Gnuplot. Plotting these data gives output like this (Figure 3.2.1)

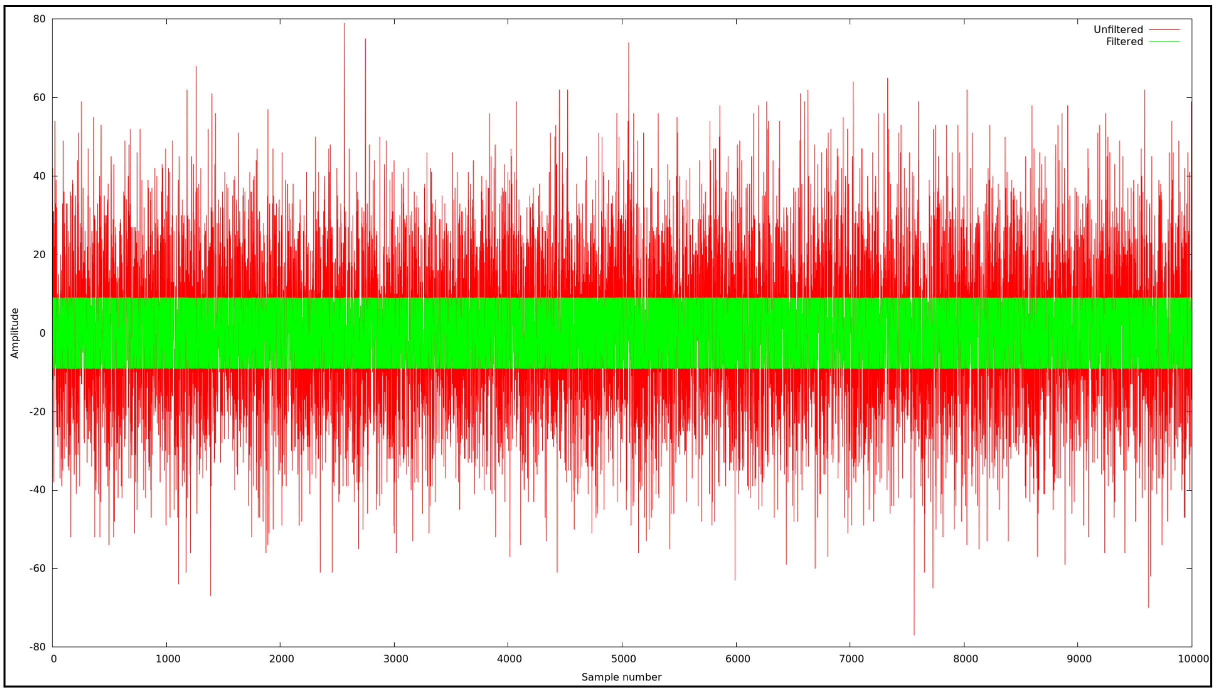


Figure 3.2.1 - Overlay plot of filtered over unfiltered signal, N =0.5, threshold replacement

3.3 Features of testing

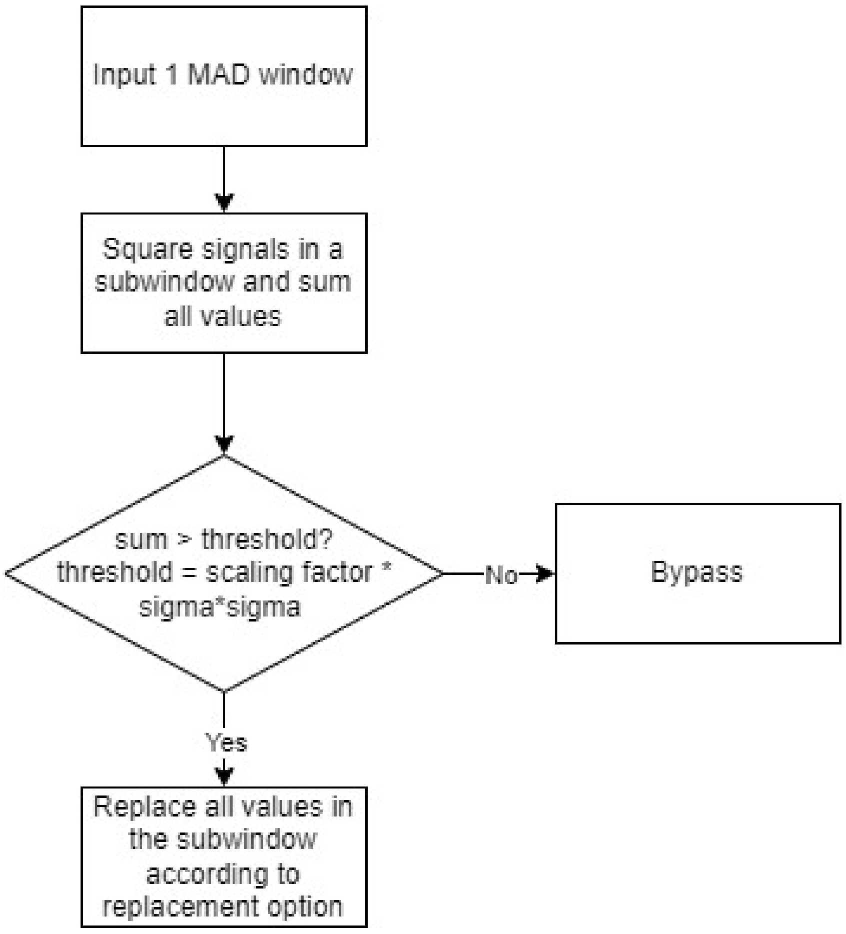
Another C program file named “final_2.1_rfi_filter_omp_4_memory_abs_avx_csv.c” is created which gives flagging percentage along with other essential parameters including window size, MOM window, sigma, replacement option, replacement constant, average filtering time.

Chapter 4: Power Detection Technique

The power detection technique operates by leveraging an improved signal-to-noise ratio (SNR), achieved through the process of squaring and averaging the time-domain signal. This method enables decisions to be made based on the averaged power signal, which helps in better detection of weak RFI signals.
In determining the optimal number of samples for averaging, careful consideration is given to the typical duration of RFI (Radio Frequency Interference) bursts resulting from power-line sparking instances. By selecting an appropriate number of samples, the technique aims to capture the characteristics of these bursts effectively, ensuring accurate detection and analysis of RFI events originating from power-line disturbances.

4.1 Power detection code basics

Figure 4.1.1 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 the aggressiveness of filter), replacement option and replacement constant for constant replacement.



Input 1 MAD window
Square signals in a subwindow and sum all values
sum > threshold? threshold = scaling factor * sigma*sigma
No
Bypass
Yes
Replace all values in the subwindow according to replacement option

Figure 4.1.1 Basic working of power detection (Source: Report of Satyarth Gupta)

4.2 Previous work

Previously the power detection code was implemented with AVX instructions and a C++ program was created as a golden reference. That C program was not integrated with GWB. Also, there was some scope for improvement (Inplace swapping, AVX instruction for absolute function) and some other approaches (look-up table) needed to be tested.

Previous C and C++ programs had two filtering approaches - Threshold filtering and Noise replacement.

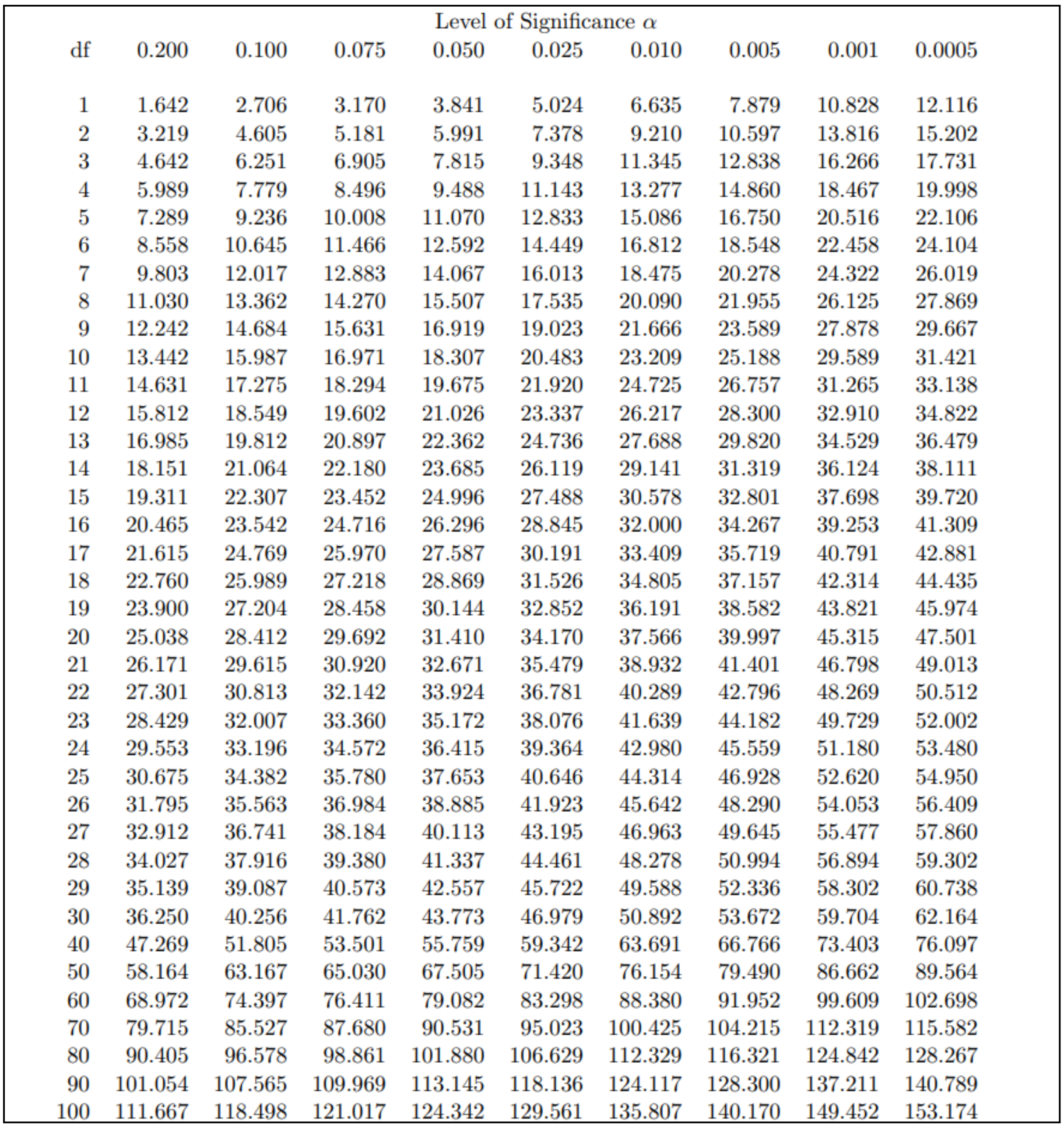
4.3 Finding an appropriate threshold

To find the value of the chi-square (χ^2) statistic given the degrees of freedom and probability (or significance level), we typically refer to a chi-square distribution table (Figure 4.2.1) or use statistical software (see reference for source and Figure 4.2.2). The chi-square distribution is a probability distribution that depends on the degrees of freedom.

A chi-square distribution table provides critical values of the chi-square statistic for various degrees of freedom and probability levels.

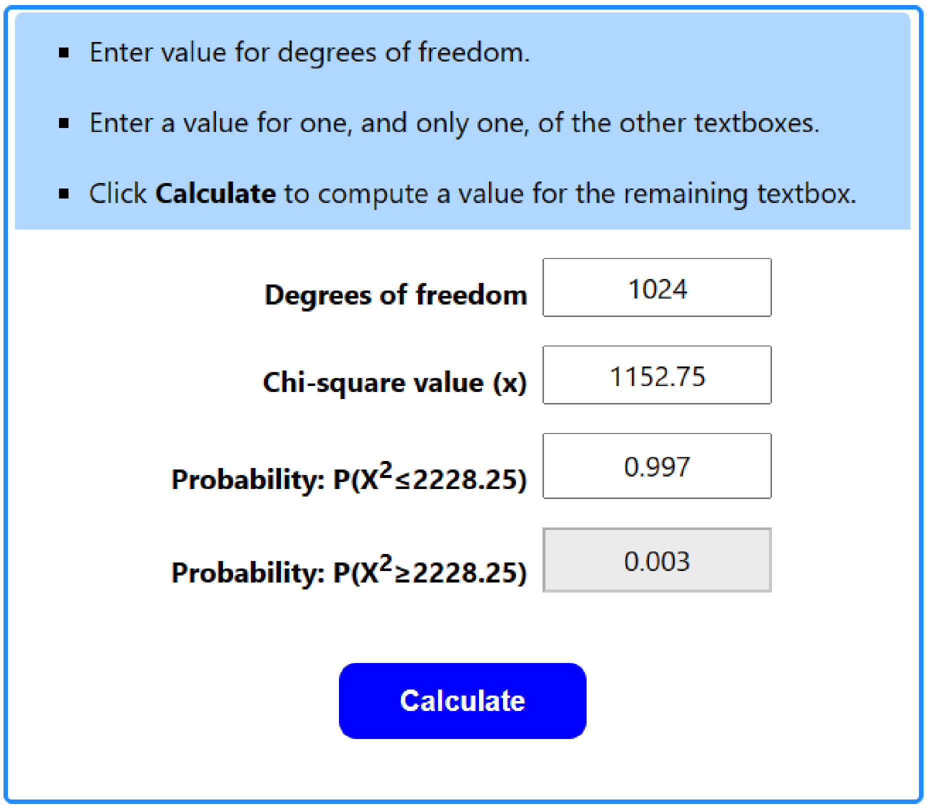
To use the table, locate the row corresponding to the degrees of freedom and find the column corresponding to the probability level (or significance level). The value at the intersection of the row and column is the chi-square statistic.

It's important to note that the chi-square distribution is right-skewed, so the probability value corresponds to the right tail of the distribution. Therefore complement of the probability (1 - probability) needs to be considered when using the table or software function.



	Level of Significance α								
df	0.200	0.100	0.075	0.050	0.025	0.010	0.005	0.001	0.0005
1	1.642	2.706	3.170	3.841	5.024	6.635	7.879	10.828	12.116
2	3.219	4.605	5.181	5.991	7.378	9.210	10.597	13.816	15.202
3	4.642	6.251	6.905	7.815	9.348	11.345	12.838	16.266	17.731
4	5.989	7.779	8.496	9.488	11.143	13.277	14.860	18.467	19.998
5	7.289	9.236	10.008	11.070	12.833	15.086	16.750	20.516	22.106
6	8.558	10.645	11.466	12.592	14.449	16.812	18.548	22.458	24.104
7	9.803	12.017	12.883	14.067	16.013	18.475	20.278	24.322	26.019
8	11.030	13.362	14.270	15.507	17.535	20.090	21.955	26.125	27.869
9	12.242	14.684	15.631	16.919	19.023	21.666	23.589	27.878	29.667
10	13.442	15.987	16.971	18.307	20.483	23.209	25.188	29.589	31.421
11	14.631	17.275	18.294	19.675	21.920	24.725	26.757	31.265	33.138
12	15.812	18.549	19.602	21.026	23.337	26.217	28.300	32.910	34.822
13	16.985	19.812	20.897	22.362	24.736	27.688	29.820	34.529	36.479
14	18.151	21.064	22.180	23.685	26.119	29.141	31.319	36.124	38.111
15	19.311	22.307	23.452	24.996	27.488	30.578	32.801	37.698	39.720
16	20.465	23.542	24.716	26.296	28.845	32.000	34.267	39.253	41.309
17	21.615	24.769	25.970	27.587	30.191	33.409	35.719	40.791	42.881
18	22.760	25.989	27.218	28.869	31.526	34.805	37.157	42.314	44.435
19	23.900	27.204	28.458	30.144	32.852	36.191	38.582	43.821	45.974
20	25.038	28.412	29.692	31.410	34.170	37.566	39.997	45.315	47.501
21	26.171	29.615	30.920	32.671	35.479	38.932	41.401	46.798	49.013
22	27.301	30.813	32.142	33.924	36.781	40.289	42.796	48.269	50.512
23	28.429	32.007	33.360	35.172	38.076	41.639	44.182	49.729	52.002
24	29.553	33.196	34.572	36.415	39.364	42.980	45.559	51.180	53.480
25	30.675	34.382	35.780	37.653	40.646	44.314	46.928	52.620	54.950
26	31.795	35.563	36.984	38.885	41.923	45.642	48.290	54.053	56.409
27	32.912	36.741	38.184	40.113	43.195	46.963	49.645	55.477	57.860
28	34.027	37.916	39.380	41.337	44.461	48.278	50.994	56.894	59.302
29	35.139	39.087	40.573	42.557	45.722	49.588	52.336	58.302	60.738
30	36.250	40.256	41.762	43.773	46.979	50.892	53.672	59.704	62.164
40	47.269	51.805	53.501	55.759	59.342	63.691	66.766	73.403	76.097
50	58.164	63.167	65.030	67.505	71.420	76.154	79.490	86.662	89.564
60	68.972	74.397	76.411	79.082	83.298	88.380	91.952	99.609	102.698
70	79.715	85.527	87.680	90.531	95.023	100.425	104.215	112.319	115.582
80	90.405	96.578	98.861	101.880	106.629	112.329	116.321	124.842	128.267
90	101.054	107.565	109.969	113.145	118.136	124.117	128.300	137.211	140.789
100	111.667	118.498	121.017	124.342	129.561	135.807	140.170	149.452	153.174

Figure 4.3.1 - Chi-square distribution table [10]


- Enter value for degrees of freedom.
- Enter a value for one, and only one, of the other textboxes.
- Click **Calculate** to compute a value for the remaining textbox.

Degrees of freedom	1024
Chi-square value (x)	1152.75
Probability: P(X^2≤2228.25)	0.997
Probability: P(X^2≥2228.25)	0.003

Figure 4.3.2 - Chi-square value calculation using an online tool [11]

Subwindow size	Chi-square value
2048	2228.25
1024	1152.75
512	604.3125
256	322.5625
128	176.34375
64	99.46875

Table 4.3.1 - Table for Chi-square value for each subwindow

4.4 Benchmarking and functional testing

Similar to the voltage detection technique, here also optimisation techniques are used. These are absolute calculations with the avx command, inplace swapping and Lookup table.

This benchmarking is done on a MOM Window size of 16384B
The table for flagging percentage is attached below (Table 4.4.1)

The expected filtering time is same as voltage detection, 40.96 µs for single core, and 671ms for multicore.

Optimisation Technique	Sub window size	Average time (µs)	Flagging percentage
No optimisation	1024	13.122559	57.858467
Abs in AVX	1024	10.681152	57.858467
LU Table	1024	73.242187	57.858467
Inplace	1024	17.39502	57.858467

Table 4.4.1 - Flagging percentage in different optimisations in the same subwindow

Here, is the plot (Figure 4.4.1) comparing the outcomes

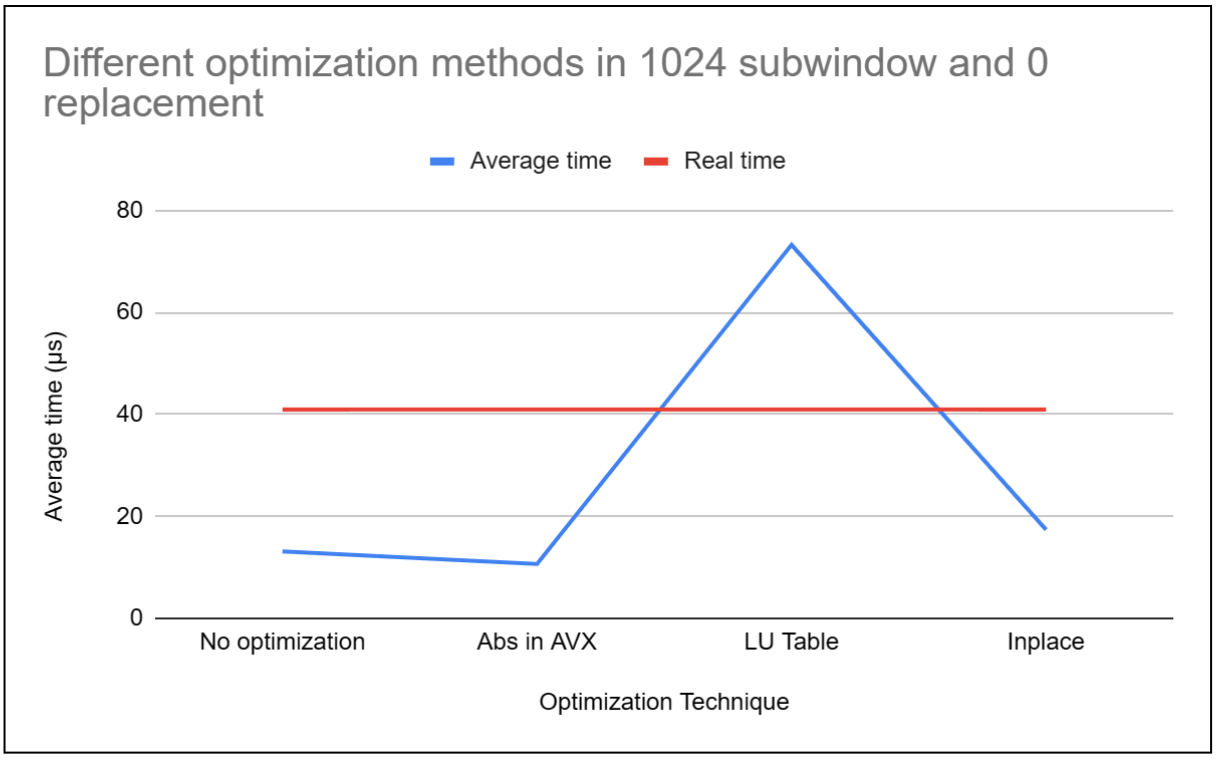


Figure 4.4.1 - Comparing different optimisations in the same subwindow

From the graph (Figure 4.4.1) it is seen that converting the absolute function to AVX gives the best result. Also using the lookup table is not a good approach as the time taken is extremely high and it takes almost twice the real time requirement (40.96 µs).

Considering this outcome, the program where the absolute function is written in avx is rigorously checked in different subwindows to see the effect of subwindow size on filtering time.

Changing the subwindow size changes the flagging percentage. Table 4.4.2 describes that output.

Subwindow size	Average time (μs)	optimisation technique	Flagging percentage
2048	10.986328	abs avx	68.470764
1024	10.681152	abs avx	57.858467
512	10.070801	abs avx	53.361797
256	8.544922	abs avx	51.665974
128	10.070801	abs avx	50.942493
64	9.460449	abs avx	50.629761

Table 4.4.2 - The flagging percentage of abs in avx optimisation in the different subwindow

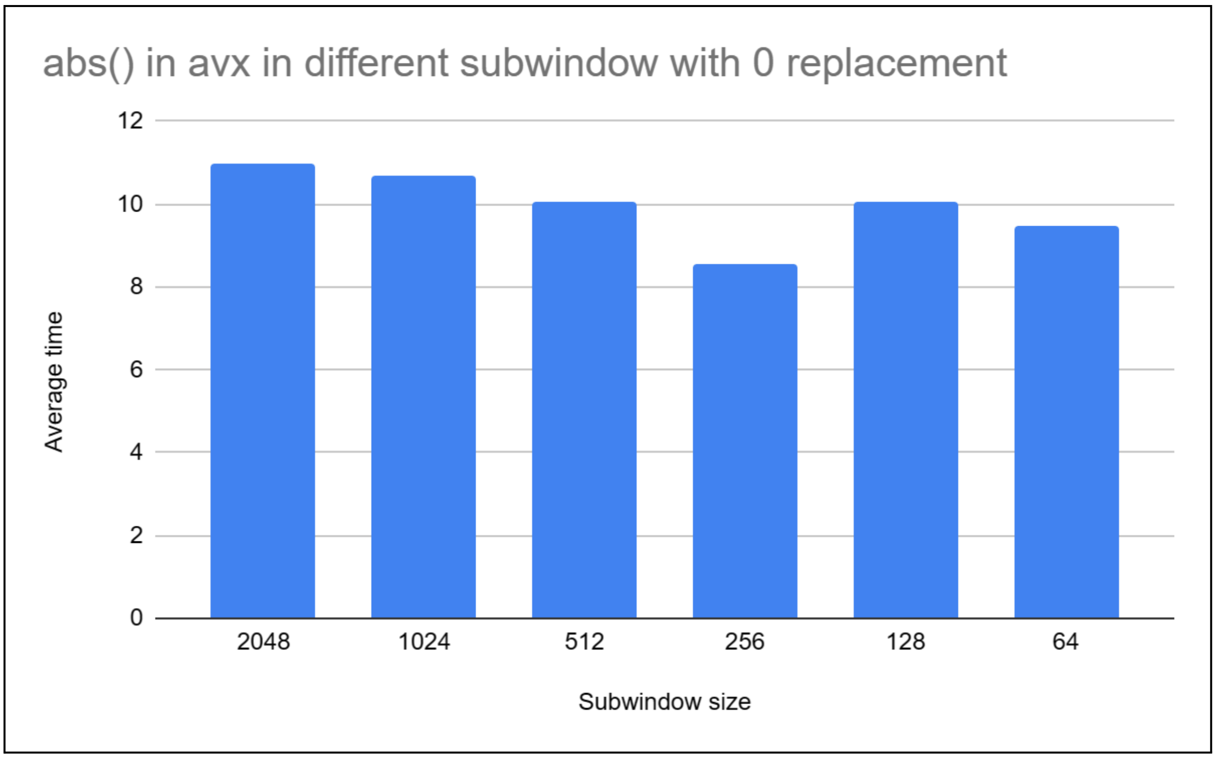


Figure 4.4.2 - Comparing abs in avx optimisation in the different subwindow

4.5 Integration with GWB

Currently, integration with GWB is a work in progress.

Conclusion and future scopes

- This project provided optimisation real-time performance enhancements to the MoM-based real-time radio frequency interference (RFI) mitigation algorithm for the GMRT Wideband Backend (GWB) processing system.
- The project contributed to improving the real-time performance of the MoM-based RFI filtering technique through the use of SSE instructions and the implementation of various optimisations. The report outlines the outcomes of benchmarking, functional testing, and the integration of these techniques into the GWB processing system.
- Currently, the best approach is 4 separate memory in buffer and absolute function (implemented as AVX instructions).
- Code optimisation also depends on a variety of variables, including compiler versions and underlying hardware specs. As we've discussed, it's clear that machine architectures and compiler versions have a significant impact on how well-optimised code performs.
- Increasing the MOM window size and MAD window size improves performance significantly.
- Upgrading to GCC 11 from GCC 4.8.5 improves performance by 4 to 5%
- In the future, performance can be improved by upgrading the hardware and using GPU.
- Power detection implementation can be integrated with GWB.

References

[1] GMRT official website
http://www.gmrt.ncra.tifr.res.in

[2] Shubham Balte, “Implementation Of MAD-Based RFI Filtering Algorithm On CPU And GPU”, NCRA STP Report, July 2022

[3] Satyarth Gupta, “Performance Optimisation of CPU-based Real-time RFI filtering System for GWB”, NCRA STP Report, July 2023

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

[5] CS3330: A quick guide to SSE/SIMD
https://www.cs.virginia.edu/~cr4bd/3330/F2018/simdref.html

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

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

[8] POSIX Threads in OS - GeeksforGeeks
https://www.geeksforgeeks.org/posix-threads-in-os/

[9] POSIX Threads Programming | LLNL HPC Tutorials
https://hpc-tutorials.llnl.gov/posix/

[10] The Chi Square Distribution
https://uregina.ca/~gingrich/appchi.pdf

[11] Chi-Square Calculator
Chi-Square Distribution Probability Calculator (stattrek.com)

Appendix

Modified versions of the voltage detection

Program name	Changes	Outcome
1.0_rfi_filter_omp_original.c	Final code by Satyarth	Original
2.0_rfi_filter_omp_4_memory.c	4 different memory locations in buffer	improvement
final_2.1_rfi_filter_omp_4_memory_abs_avx.c	4 separate memory for buffers + avx instruction for absolute function	most improvement
2.1.1_rfi_filter_omp_4_memory_abs_avx_flag_percentage.c	4 memory + abs() AVX + flag percentage	added some features on the final version
2.1.2_rfi_filter_omp_4_memory_abs_avx_plotting.c	4 memory + abs() AVX + flag percentage + output files for plotting filtered and unfiltered data	added some features on the final version
3.0_rfi_filter_omp_inplace.c	Same array for input and output	improvement
4.0_rfi_filter_omp_omp_optimization.c	Optimizes openmp loop	improvement
5.0_rfi_filter_omp_multi_optimization_1.c	4 seperate memory for buffers + optimised openmp loop	improvement
5.0.1_rfi_filter_omp_multi_optimization_1_ABS_AVX.c	4 seperate memory for buffers + optimised openmp loop + avx instruction for absolute function	much improvement
5.1_rfi_filter_omp_multi_optimization_2.c	Inplace swapping for input and filtered output + 4 separate memory for buffers + optimised openmp loop	improvement

6.0_rfi_filter_omp_pthread.c	Used pthread	similar

Final versions of the power detection

optimised_power_sse.c	C program for power detection using avx. **Includes bypass**, constant replacement, noise replacement
optimised_power_sse_2.c	C program for power detection using avx. **Does not bypass**. Only Constant and noise replacement

List of generic parameters

Parameter	**Data type**	**Range or constraints**	**Usual values**
MAD window size	int	> 32	32768, 16384, less common - 1024, 4096
MoM window size	int	>1	32768, 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, 512, 1024

AVX instructions and their usage

AVX Instruction	**Description**
_mm256_loadu_si256	Loads 256 bits (32 bytes) of integer values from memory into an AVX register.
_mm256_storeu_si256	Stores 256 bits (32 bytes) of integer values from an AVX register into memory.
_mm256_abs_epi8	Calculates the absolute values of 8-bit integers in the AVX register.

The AVX instructions in the abs_avx2 function achieve two main jobs:

- Vectorized Loading and Storing:
  _mm256_loadu_si256((__m256i *)&a[j]): This instruction loads 16 int8_t elements from the address a[j] into an AVX register named a_values. The __m256i type indicates a 256-bit AVX register that can hold 16 signed 8-bit integers. The & before a[j] ensures the address points to the memory location of the first element in the chunk.

  _mm256_storeu_si256((__m256i *)&arr[j], abs_values): This instruction stores the content of the AVX register abs_values back into the memory starting at address arr[j]. Similar to loading, & ensures the address points to the first element where absolute values need to be stored.

- Vectorized Absolute Value Calculation:
  _mm256_abs_epi8(a_values): This instruction performs a single operation on all 16 elements within the AVX register a_values. It calculates the absolute value of each int8_t element and stores the results back in the same register abs_values. This single instruction efficiently replaces 16 separate absolute value calculations typically done in a loop.

In essence, these AVX instructions significantly improve performance by processing 16 elements simultaneously compared to a traditional loop that would process each element individually.

Final C code for RFI filtering optimisation

```
/********************************
Version Information:
    - This version is built on top of rfi_filter_omp_4_memory.c.
      - It uses 4 separate memory buffers and a modified AVX abs()
function.
********************************/

/********************************
Major Features:
    - Assumes the first median to be 0 for computational efficiency.
     - Configuration options are loaded from a header file instead of
the command line.
    - Optimized with OpenMP and -O3 level compiler optimizations.
      - Filtering loop utilizes SSE and AVX instructions for real-time
performance.
    - Implements a double buffer strategy.
      - Inside the buffers, memory locations are not contiguous which
increases read and write time in parallel processing
       - abs() function is written in AVX instruction to increase
efficiency
********************************/

/********************************
Code Sections:
    1. Reading Parameters from .hdr File:
     - Reads and parses configuration parameters from the specified .hdr
file.

    2. Filtering Function:
      - The core filtering function using AVX instructions to achieve
real time performance of the filter.

    3. AVX-based absolute value calculation:
    - AVX instruction is used for efficient filtering
    4. OpenMP Section for 4 Antenna Inputs:
       - Utilizes OpenMP parallelization for processing data from 4
antenna inputs.
********************************/

/********************************
Compilation Instructions:
     - Compile: gcc <.c filename> -o <object filename> -lm -O3 -mavx
-mavx2 -std=c99 -lgomp -fopenmp
Running Instructions:
    - Run: ./<object filename> <.bin file>
********************************/

/********************************
Other versions info:
    - version 1.0: Final code by Satyarth
    - version 2.0: 4 different memory locations in buffer
    - version 2.1: **THIS VERSION** (4 memory + abs() in AVX - Final)
    - version 3.0: Same array for input and output
    - version 4.0: Optimizes openmp loop
     - version 5.0: 4 seperate memory for buffers + optimised openmp
loop
     - version 5.0.1: 4 seperate memory for buffers + optimised openmp
loop + avx instruction for absolute function
     - version 5.1: Inplace swapping for input and filtered output + 4
seperate memory for buffers + optimised openmp loop
    - version 6.0: Used pthread
********************************/

/********************************
Release Information:
- Release 2.1 (4 memory + abs() in AVX - Final)
********************************/

#define _POSIX_C_SOURCE 199309L // Define _POSIX_C_SOURCE to specify
adherence to POSIX.1b-1993 standard

#include <stdio.h>
#include <smmintrin.h>
#include <immintrin.h>
#include <emmintrin.h>
#include <stdlib.h>
```

```
#include <math.h>
#include <time.h>
#include <omp.h>
#include <string.h>
#include <ctype.h>
#include <sys/time.h>

#define CLOCK_PER_MICRO ((double)CLOCKS_PER_SEC) / (1000000.0)
#define MAX_LINE_LENGTH 256

typedef signed char int8_t; // Typedefinition for an 8-bit signed
integer.

/*********************************** SIMD multiplication of 8-bit
integers ***********************************/
/*
   Function: _mm256_mullo_epi8

   Description:
    Multiplies the corresponding packed 8-bit integers of two 256-bit
integer vectors element-wise, producing a vector of 16-bit results.

   Parameters:
   - __A: A 256-bit integer vector.
   - __B: Another 256-bit integer vector.

   Returns:
    - A 256-bit integer vector containing the element-wise product of
corresponding 8-bit integers from __A and __B.

   Usage:
    - This function is an intrinsic provided by certain compilers for
SIMD (Single Instruction, Multiple Data) operations on 256-bit
integers.

   Note:
   - __m256i: This type represents a 256-bit integer vector.
         - __v32qs: This is a type definition using the
__attribute__((__vector_size__(32)) attribute, indicating a vector of
32 bytes (256 bits).
              - __attribute__((__gnu_inline__, __always_inline__,
__artificial__)): These attributes provide hints to the compiler for
inlining and other optimizations.
```

```
   - The actual multiplication operation is performed with (__v32qs)__A
* (__v32qs)__B. The result is cast back to __m256i to match the
expected return type.
*/

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);
}

/***************************************** Structure containing header
file information *******************************/

/*
   Structure: RfiFilterSettings

   Description:
      Represents a structure containing information parsed from an
RFIFilter settings header file.

   Members:
   - gwbVersion: Array to store the version of the GWB (Graphical User
Interface).
      - filteredSignals: Array to store the type of signals to be
filtered.
   - externalMedian: Integer to store the external median value.
   - thresholdValue: Float to store the threshold value.
   - constantValue: Integer to store the constant value.
   - filteringOption: Integer to store the filtering option (0: BYPASS,
1: CONSTANT, 2: THRESHOLD, 3: DIGITAL_NOISE).
      - ddcStatus: Array to store the status of the DDC (Digital
Down-Converter).
   - mom_win: Integer to store the MOM (Method of Moments) window size.
     - mad_win: Integer to store the MAD (Median Absolute Deviation)
window size.

   Note:
   This structure is designed to hold settings information parsed from
an RFIFilter header file.
```

```
    The member types and names correspond to the settings provided in
the header file.
*/

typedef struct
{
    char gwbVersion[500];
    char filteredSignals[500];
    int externalMedian;
    float thresholdValue;
    int constantValue;
    int filteringOption;
    char ddcStatus[500];
    int mom_win;
    int mad_win;
} RfiFilterSettings;

/****************************************    Function    to    truncate
whitespace from a string ******************************/

/*
   Function: truncateWhitespace

   Description:
   Truncates leading and trailing whitespaces from a string.

   Parameters:
   - input: Pointer to the input string.

   Returns:
   - Pointer to the truncated string (allocated in dynamic memory).

   Note:
    This function takes a pointer to a string as input and dynamically
allocates memory for a new string.
     It removes leading and trailing whitespaces from the original
string, creating a truncated version.
    The input string is expected to be null-terminated. The function
allocates memory for the truncated string,
   and the caller is responsible for freeing this memory when it is no
longer needed.
```

```
   The resulting truncated string is returned, and if the input string
is NULL or memory allocation fails,
   the function returns NULL.

   Example:
   If input = "   example   ", the function returns "example".

   Usage:
   char *truncatedString = truncateWhitespace(input);
*/

char *truncateWhitespace(const char *input)
{
    // Return NULL if the input string is NULL
    if (input == NULL)
    {
        return NULL;
    }

    size_t length = strlen(input);

    // Allocate memory for the result string
    char *result = (char *)malloc((length + 1) * sizeof(char));

    if (result == NULL)
    {
        return NULL; // Memory allocation failed
    }

    size_t i, j = 0;

     // Iterate through the input string, omitting leading and trailing
whitespaces
    for (i = 0; i < length; i++)
    {
        if (!isspace(input[i]))
        {
              result[j++] = input[i]; // Copy non-whitespace characters
to the result string
        }
    }

    result[j] = '\0'; // Null-terminate the result string
    // Return the truncated string
    return result;
}

/**********************************     Header     Reader     Function
**********************************/

/*
   Function: readRfiFilterSettings

   Description:
    Reads the header (.hdr) file and populates the RfiFilterSettings
structure with the specified configuration.

   Parameters:
   - filename: Path to the header file.

   Returns:
   - RfiFilterSettings structure containing the parsed configuration.

   Note:
   This function opens the specified header file, reads each line, and
extracts key-value pairs. It then
   populates the fields of the RfiFilterSettings structure based on the
parsed information. The header file
   follows a specific format where keys and values are separated by ':'
and comments start with '#'. The
    function also handles specific key-value mappings and conversions,
such as converting string representations
      of numbers to their respective data types. The resulting
RfiFilterSettings structure holds the configuration
   information for RFI filtering.

   The structure RfiFilterSettings includes the following fields:
   - gwbVersion: Version of the GUI.
   - filteredSignals: Type of signals to be filtered.
   - externalMedian: External median value.
   - thresholdValue: Threshold value for filtering.
   - constantValue: Constant value for replacement.
        - filteringOption: Filtering option (BYPASS, THRESHOLD,
DIGITAL_NOISE, CONSTANT).
   - ddcStatus: DDC status (ON/OFF).
   - mom_win: MOM window size.
   - mad_win: MAD window size.
*/

RfiFilterSettings readRfiFilterSettings(const char *filename)
{
       RfiFilterSettings  settings;              //  Structure  to  store
configuration settings
      FILE  *file  =  fopen(filename,  "r");  //  Open  the  specified header
file

    // Check if the file was successfully opened
    if (file == NULL)
    {
        printf("Failed to open the file '%s'.\n", filename);
        exit(1);
    }

    char line[MAX_LINE_LENGTH];

    // Read each line from the header file and extract key-value pairs
    while (fgets(line, sizeof(line), file))
    {
        char *key = strtok(line, ":");
        char *value = strtok(NULL, "#");
        key = truncateWhitespace(key);
        value = truncateWhitespace(value);

            //  Process  each  key-value  pair  and  populate  the  settings
structure
        if (key != NULL && value != NULL)
        {
            // Handle specific key-value mappings and conversions
            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);
            }
        }
    }
    // Close the header file
    fclose(file);

    // Return the populated configuration settings structure
    return settings;
}

/******************************************** Dot Product Calculation
***********************************************/

/*
   Function: horizontalSum

   Description:
    Computes the horizontal sum of 8-bit unsigned integers in an AVX2
register.

   Parameters:
   - vector: AVX2 register containing 32 unsigned 8-bit integers.

   Returns:
   - The horizontal sum of the 8-bit unsigned integers.

   Note:
    This function uses AVX2 instructions to calculate the horizontal sum
of 8-bit unsigned integers
    stored in the provided AVX2 register. It performs the sum across 32
elements and returns the result
    as a 32-bit integer. The algorithm involves using successive SIMD
instructions to accumulate the sum
   efficiently.
*/

int horizontalSum(__m256i vector)
{
     // Calculate the sum of absolute differences of packed unsigned
8-bit integers
    __m256i sum = _mm256_sad_epu8(vector, _mm256_setzero_si256());

    // Extract two 128-bit lanes from the 256-bit sum
      __m128i sum128 = _mm_add_epi32(_mm256_extractf128_si256(sum, 0),
_mm256_extractf128_si256(sum, 1));
```

```
    // Perform horizontal addition to get a 128-bit sum
      __m128i sum64 = _mm_add_epi32(sum128, _mm_unpackhi_epi64(sum128,
sum128));

    // Perform additional horizontal addition to get a 32-bit sum
        __m128i sum32 = _mm_add_epi32(sum64, _mm_shuffle_epi32(sum64,
_MM_SHUFFLE(2, 3, 0, 1)));

    // Extract the 32-bit sum as an integer
    int sumf = _mm_cvtsi128_si32(sum32);
    return sumf;
}

/*****************************************    Counting    Integers    in
Binary File *****************************************/

/*
   Function: countIntegersInBinary

   Description:
   Counts the number of 8-bit integers in a binary file.

   Parameters:
   - fileName: Path to the binary file.

   Returns:
   - Number of 8-bit integers in the binary file.

   Note:
    This function opens the specified binary file in "read" mode and
seeks to the end of the file
    to determine its size in bytes. It then calculates the number of
8-bit integers in the file
      by dividing the total size by the size of each integer
(sizeof(int8_t)). The result represents
   the count of 8-bit integers in the binary file.
*/

long countIntegersInBinary(const char *fileName)
{
    FILE *file = fopen(fileName, "rb");
    // Check if the file was successfully opened
    if (file == NULL)
    {
        printf("Failed to open the binary file.\n");
        return -1;
    }

    // Seek to the end of the file
    fseek(file, 0, SEEK_END);

    // Get the size of the file in bytes
    long fileSize = ftell(file);

      // Calculate the number of integers (assuming each integer is
sizeof(int) bytes)
    long integerCount = fileSize / sizeof(int8_t);

    // Close the file
    fclose(file);
    return integerCount;
}

/************************************************ File Reading Function
************************************************/

/*
   Function: ReadFile

   Description:
    Reads integers from a text file and returns them as an array of
8-bit integers.

   Parameters:
   - filename: Path to the text file to be read.
    - size: Pointer to the variable where the size of the array will be
stored.

   Returns:
   - Pointer to the dynamically allocated array of 8-bit integers.

   Note:
    This function opens the specified file in "read" mode and counts the
number of lines in the file.
```

```
    It then allocates memory for the array, resets the file pointer to
the beginning, reads the lines,
   converts them to integers, and stores them in the array. The size of
the array is stored in the
    variable pointed to by 'size'. Memory is allocated dynamically and
should be freed by the caller
   after use.
*/

int8_t *ReadFile(char *filename, int *size)
{
    FILE *file = fopen(filename, "r");

    // Check if the file was successfully opened
    if (file == NULL)
    {
        printf("Failed to open the file.\n");
        return NULL;
    }

    // Count the number of lines in the file
    int count = 0;
    char buffer[10];

    while (fgets(buffer, sizeof(buffer), file) != NULL)
    {
        count++;
    }

    // Allocate memory for the array
    int8_t *data = (int8_t *)malloc(count * sizeof(int8_t));

    // Check if memory allocation was successful
    if (data == NULL)
    {
        printf("Memory allocation failed.\n");
        return NULL;
    }

    // Reset the file pointer to the beginning
    fseek(file, 0, SEEK_SET);

    // Read the lines and convert them to integers
    int i = 0;
    while (fgets(buffer, sizeof(buffer), file) != NULL)
    {
        int num = atoi(buffer);
        data[i] = (int8_t)num;
        i++;
    }

    // Set the size of the array
    *size = count;

    // Close the file
    fclose(file);

    return data;
}

/***************** WriteFile Function to Write 8-bit Integer Array into
a Text File *****************/

/*
   Function: WriteFile

   Description:
   Writes an 8-bit integer array into a text file.

   Parameters:
   - filename: Path to the text file where data will be written.
   - data: Pointer to the 8-bit integer array to be written.
   - length: Number of elements in the array.

   Note:
    This function opens the specified file in "append" mode and writes
each element
    of the 8-bit integer array on a new line using the fprintf function.
It then
   closes the file.
*/

void WriteFile(const char *filename, int8_t *data, int length)
{
    FILE *file = fopen(filename, "a");
```

```c
    // Check if the file was successfully opened
    if (file == NULL)
    {
        printf("Failed to open the file.\n");
        return;
    }

    // Iterate through the array and write each element on a new line
    for (int i = 0; i < length; i++)
    {
        fprintf(file, "%d\n", (int)data[i]);
    }

    // Close the file after writing
    fclose(file);
}

/**************************** Gaussian Noise Generator based on CLT
*********************************/

/*
   Function: generateGaussianNoise

   Description:
   Generates Gaussian noise based on the Central Limit Theorem.

   Parameters:
   - arr: Pointer to the array where Gaussian noise will be stored.
   - size: Size of the array.
   - sigma: Standard deviation of the Gaussian distribution.

   Note:
     This function generates Gaussian noise using the Central Limit
Theorem (CLT),
    summing 24 uniformly distributed random numbers and applying mean
and standard deviation.
   It uses the rand() function for random number generation.
*/

void generateGaussianNoise(int8_t arr[], int size, double sigma)
{
    srand(time(NULL)); // Seed the random number generator
    for (int i = 0; i < size; i++)
    {
        double sum = 0.0;

        // Sum 24 uniformly distributed random numbers between 0 and 1
        for (int j = 0; j < 24; j++)
        {
               sum += (double)rand() / RAND_MAX; // Generate 12 random
numbers between 0 and 1
        }

          // Apply the mean and standard deviation to generate Gaussian
noise
        double gaussian = (sum - 12) * sigma;

        // Round and cast the result to an 8-bit integer
        arr[i] = (int8_t)(floor(gaussian + 0.5));
    }
}

/*******************************            Calculates           median
**************************************/

/*
   Function: HistoMedian

   Description:
    Calculates the median of the input array using a histogram-based
approach.

   Parameters:
   - input_arr: Pointer to the input array.
   - len: Length of the input array.

   Returns:
   - Calculated median.

   Note:
    This function assumes that the input array contains signed 8-bit
data.
   It uses a histogram array to count the frequency of each data point
and
```

```
   calculates the median based on the cumulative sum of the histogram
array.
*/

int8_t HistoMedian(int8_t 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[(int)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 (int8_t)i;
        }
    }

     // Default return if the loop does not break (should not happen
under normal conditions)
    return 0;
}

/************************    Some    Vector    functions    from    c++
*************************/

/*
   Function: push_back

   Description:
   Appends elements to the end of an array.

   Parameters:
   - array: Pointer to the original array.
   - size: Pointer to the current size of the array.
   - element: Pointer to the elements to be appended.
   - push_size: Number of elements to append.

   Returns:
   - Pointer to the modified array.

   Note:
   This function assumes that the original array has enough space
   to accommodate the additional elements.
*/

int8_t *push_back(int8_t *array, long *size, int8_t *element, int
push_size)
{
    // Copy elements from 'element' to the end of 'array'
    for (int i = 0; i < push_size; i++)
    {
        array[*size + i] = element[i];
    }

    // Update the size of the array
    *size += push_size;

    // Return a pointer to the modified array
    return array;
}

/*
   Function: VectorSlice

   Description:
   Creates a new vector by slicing a portion of an existing vector.

   Parameters:
   - v: Pointer to the original vector.
   - X: Starting index for the slice.
   - Y: Ending index for the slice.
   Returns:
   - Pointer to the newly created sliced vector.
*/

int8_t *VectorSlice(int8_t *v, int X, int Y)
{
    // Calculate pointers to the first and last elements of the slice
    int8_t *first = v + X;
    int8_t *last = v + Y + 1;

    // Allocate memory for the sliced vector
    int8_t *vec = (int8_t *)malloc((last - first) * sizeof(int8_t));
    if (vec == NULL)
    {
        printf("Memory allocation failed.\n");
        return NULL;
    }

    // Copy elements from the original vector to the sliced vector
    int8_t *p = vec;
    while (first != last)
    {
        *p = *first;
        ++p;
        ++first;
    }

    // Return a pointer to the newly created sliced vector
    return vec;
}

//*************    Absolute    calculation    using    avx    instruction
*****************//

/*
   Function: abs_avx2

   Description:
    Calculates the absolute values of elements in an array using AVX2
instructions.

   Parameters:
```

```
   - arr: Pointer to the destination array where absolute values will
be stored.
   - a: Pointer to the source array containing input values.
   - WINDOW_SIZE: Number of elements to process in the arrays.

   Note:
     This function assumes that 'arr' and 'a' have sufficient memory
allocated.
     It utilizes AVX2 instructions to process elements efficiently in
chunks of 16.
    outputArray will contain absolute values of corresponding elements
in inputArray
*/

void abs_avx2(int8_t *arr, const int8_t *a, int WINDOW_SIZE)
{
     // Process the array in chunks of 16 elements (AVX2 registers hold
16 values)
    for (int j = 0; j < WINDOW_SIZE; j += 16)
    {
           // Load 16 integer values from the 'a' array into an AVX2
register
        __m256i a_values = _mm256_loadu_si256((__m256i *)&a[j]);

         // Calculate the absolute values of the 16 elements in the AVX2
register
        __m256i abs_values = _mm256_abs_epi8(a_values);

        // Store the result (absolute values) back into the 'arr' array
        _mm256_storeu_si256((__m256i *)&arr[j], abs_values);
    }
}

//**********************************           RFI           FILTER
**********************************//

/*
   Function: filter

   Description:
     Applies a filtering algorithm to an array of numbers based on
specified parameters.
```

```
   Parameters:
   - numbers: Input array of integers to be filtered.
   - filtered: Pointer to the destination array where filtered values will be stored.
   - data_count: Total number of elements in the 'numbers' array.
   - WINDOW_SIZE: Size of the window used for filtering.
   - MOM_WIN: Size of the window used for calculating the MOM.
   - N: Scaling factor for filtering.
   - RPL_OPTION: Option for replacement during filtering.
   - RPL_CONST: Constant value used for replacement (if RPL_OPTION is enabled).
   - thres: Pointer to an int8_t value representing the threshold for filtering.
   - flags: Pointer to an array of flags.

   Note:
   - The 'filtered' and 'flags' array should have sufficient memory allocated.
   - The filtering algorithm uses the specified parameters to process the 'numbers' array.
   - Replacement of values during filtering can be controlled by 'RPL_OPTION' and 'RPL_CONST'.
   - The 'thres' parameter is a pointer to an int8_t value representing the threshold for filtering.
*/

void filter(int8_t numbers[], int8_t *filtered, int data_count, int WINDOW_SIZE, int MOM_WIN, float N, int RPL_OPTION, int8_t RPL_CONST, int8_t *thres, int8_t flags[])
{
    long double avg_time = 0, avg_filter = 0, avg_mediantime = 0;
    long int flag_count = 0;
    int n_slices = data_count / WINDOW_SIZE;

    int8_t *filtered_output;
    int8_t *flag_output;
    filtered_output = (int8_t *)malloc(WINDOW_SIZE * sizeof(int8_t));
    flag_output = (int8_t *)malloc(WINDOW_SIZE * sizeof(int8_t));

    long filtered_size = 0;
    long flags_size = 0;

    // Making the MAD buffer array to calculate MOM value
    int8_t mad_buffer[n_slices];
    int start = 0;

    int8_t integer_noise[WINDOW_SIZE];

    float sigma;
    float upper_th;
    upper_th = 127;
    int8_t THRES = *thres;

    for (int i = 0; i < n_slices; i++)
    {
        int8_t *a;
        a = VectorSlice(numbers, start, start + WINDOW_SIZE - 1);
        // Starting the stopwatch for timer of current window

        clock_t start_time = clock();

            // Making the array to pass for median (directly taking
absolutes as first median is 0)
        int8_t arr[WINDOW_SIZE];
        abs_avx2(arr, a, WINDOW_SIZE);

        // Starting stopwatch for median calculation
        clock_t start_med = clock();
        // Adding the MAD value into buffer
        mad_buffer[i] = (int8_t)HistoMedian(arr, WINDOW_SIZE);
        clock_t stop_med = clock();

          // 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)
            int8_t 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;
            generateGaussianNoise(integer_noise, WINDOW_SIZE, sigma);
        }

          // 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
        clock_t start_filter = clock();

        //************** Filtering with switch case ****************//

        switch (RPL_OPTION)
        {
        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((THRES * (-1))); //
Constant vector of negative threshold values

                // Iterate over the input array in chunks of 32 (AVX2
register size)
            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]

                    // Compare each element in 'v' with the threshold,
generating masks
```

```
                __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

                // Combine the masks using logical OR
                __m256i cmp = _mm256_or_si256(cmp_gt, cmp_lt);

                // Square the mask values using multiplication
                cmp = _mm256_mullo_epi8(cmp, cmp);

                // Store the flags and the filtered output
                        _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 for the next iteration
            }
        }
        break;

        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)(THRES * (-1)));
// Constant vector of negative threshold values
                      __m256i  constant  =  _mm256_set1_epi8(RPL_CONST);
// constant vector of replacement constant values

                // Iterate over the input array in chunks of 32 (AVX2
register size)
            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]

                   // Compare each element in 'v' with the threshold,
generating masks
```

```
                    __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

                // Combine the masks using logical OR
                __m256i cmp = _mm256_or_si256(cmp_gt, cmp_lt);

                // Blend 'v' and 'constant' based on the mask
                __m256i result = _mm256_blendv_epi8(v, constant, cmp);

                // Square the mask values using multiplication
                cmp = _mm256_mullo_epi8(cmp, cmp);

                // Store the flags and the filtered output
                         _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 for the next iteration
            }
        }
        break;

        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)(THRES * (-1)));
// constant vector of negative threshold values

                // Iterate over the input array in chunks of 32 (AVX2
register size)
            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]

                    // Compare each element in 'v' with the threshold,
generating masks
```

```
                __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

                // Combine the masks using logical OR
                __m256i cmp = _mm256_or_si256(cmp_gt, cmp_lt);

                  // Blend 'v', 'threshold', and 'th_neg' based on the
masks
                    __m256i result = _mm256_blendv_epi8(v, threshold,
cmp_gt);
                result = _mm256_blendv_epi8(result, th_neg, cmp_lt);

                // Square the mask values using multiplication
                cmp = _mm256_mullo_epi8(cmp, cmp);

                // Store the flags and the filtered output
                        _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
for the next iteration
            }
        }
        break;

        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)(THRES * (-1)));
// constant vector of negative threshold values

                // Iterate over the input array in chunks of 32 (AVX2
register size)
            for (int8_t *p = a; p < a + WINDOW_SIZE; p = p + 32)
            {
                    // Load pregenerated noise and 32 numbers from the
input array into AVX2 registers
                    __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]

                    // Compare each element in 'v' with the threshold,
generating masks
                    __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

                // Combine the masks using logical OR
                __m256i cmp = _mm256_or_si256(cmp_gt, cmp_lt);

                // Blend 'v' and 'noise_v' based on the mask
                __m256i result = _mm256_blendv_epi8(v, noise_v, cmp);

                // Square the mask values using multiplication
                cmp = _mm256_mullo_epi8(cmp, cmp);

                // Store the flags and the filtered output
                    _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 for the next iteration
            }
        }
        break;
        }

        // Increasing the slicing position
        start = start + WINDOW_SIZE;
         for (int8_t *m = flag_output; m < flag_output + WINDOW_SIZE; m
= m + 32)
        {
            __m256i v2 = _mm256_loadu_si256((__m256i *)m);
            flag_count += horizontalSum(v2);
        }
        // Stopping timer for filtering
        clock_t stop_filter = clock();

        // Stopping the timer for total time
        clock_t stop_time = clock();

        // Getting average times for the window
              double  duration  =  (double)((stop_time  -  start_time))  /
((double)(1));
        avg_time = ((avg_time * i) + duration) / (i + 1);
        double duration_med = (stop_med - start_med);
         avg_mediantime  =  ((avg_mediantime  *  i)  +  duration_med)  /  (i  +
1);
        double duration_filter = (stop_filter - start_filter);
        avg_filter = ((avg_filter * i) + duration_filter) / (i + 1);
        *thres = upper_th;

         // Use push_back function to append 'WINDOW_SIZE' elements from
'filtered_output' to 'filtered' array
                 push_back(filtered,  &filtered_size,  filtered_output,
WINDOW_SIZE);

         // Use push_back function to append 'WINDOW_SIZE' elements from
'flag_output' to 'flags' array
        push_back(flags, &flags_size, flag_output, WINDOW_SIZE);

        free(a);
    }
    free(flag_output);
    free(filtered_output);
}

/**********************   Main   function   responsible   for   signal
processing and filtering **********************/

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

    const char *filename = "rfi_settings.hdr"; // Path to the settings
file
```

```
     RfiFilterSettings settings = readRfiFilterSettings(filename); //
Read RFI filter settings from the header file

    char *INPUT_FILENAME = argv[1];

    // Extract specific settings from the RFI filter configuration
    const int WINDOW_SIZE = settings.mad_win;        // Window Size for
all methods
     const int MOM_WIN = settings.mom_win;               // Number of
windows for MOM calculation
    const float N = settings.thresholdValue;          // Scaling factor
for filtering
      const int RPL_OPTION = settings.filteringOption; // Filtering
option (BYPASS, CONSTANT, THRESHOLD, DIGITAL_NOISE)
    const int RPL_CONST = settings.constantValue;     // Constant value
used for replacement (if RPL_OPTION is enabled)

    // OUTPUT FILE NAMES
    char op_flname1[50];
      sprintf(op_flname1, "d1_output_%d_%d_%d_4_mem_abs_avx.out", (N *
10), RPL_OPTION, RPL_CONST);
    char op_flname2[50];
      sprintf(op_flname2, "d2_output_%d_%d_%d_4_mem_abs_avx.out", (N *
10), RPL_OPTION, RPL_CONST);
    char op_flname3[50];
      sprintf(op_flname3, "d3_output_%d_%d_%d_4_mem_abs_avx.out", (N *
10), RPL_OPTION, RPL_CONST);
    char op_flname4[50];
      sprintf(op_flname4, "d4_output_%d_%d_%d_4_mem_abs_avx.out", (N *
10), RPL_OPTION, RPL_CONST);

    struct timespec start2, end2;

    int count = 0; // COUNT Variable

    // CALCULATE DATA LENGTH OF INPUT FILES TO BE FED
    long data_len1 = countIntegersInBinary(INPUT_FILENAME);
    long data_len2 = countIntegersInBinary(INPUT_FILENAME);
    long data_len3 = countIntegersInBinary(INPUT_FILENAME);
    long data_len4 = countIntegersInBinary(INPUT_FILENAME);

    const long chunk_size = WINDOW_SIZE * MOM_WIN;        // CHUNK SIZE
    int chunk_n = (long)data_len1 / chunk_size; // NUMBER OF CHUNKS IN
BINARY FILE

                      //*************************      Input      buffer
************************//
    // DECLARATION OF INPUT BUFFER
    int8_t **numbers;
    numbers = (int8_t **)malloc(8 * sizeof(int8_t *));

    // Allocate memory for each sub-array in 'numbers'
    numbers[0] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    numbers[1] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    numbers[2] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    numbers[3] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    numbers[4] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    numbers[5] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    numbers[6] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    numbers[7] = (int8_t *)malloc(chunk_size * sizeof(int8_t));

                      //*************************      Output      buffer
************************//
       int8_t  **filtered;  //  DECLARATION  OF  OUTPUT  BUFFER  OF  SIZE
2x(4*256M)
    filtered = (int8_t **)malloc(8 * sizeof(int8_t *));

    // Allocate memory for each sub-array in 'filtered'
    filtered[0] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    filtered[1] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    filtered[2] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    filtered[3] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    filtered[4] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    filtered[5] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    filtered[6] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    filtered[7] = (int8_t *)malloc(chunk_size * sizeof(int8_t));

                      //*************************      Flag      buffer
************************//
    int8_t **flags; // DECLARATION OF Flag BUFFER OF SIZE 2x(4*256M)
    flags = (int8_t **)malloc(8 * sizeof(int8_t *));

    // Allocate memory for each sub-array in 'flags'
    flags[0] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    flags[1] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    flags[2] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    flags[3] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    flags[4] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    flags[5] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    flags[6] = (int8_t *)malloc(chunk_size * sizeof(int8_t));
    flags[7] = (int8_t *)malloc(chunk_size * sizeof(int8_t));

    // Open the binary file multiple times for parallel processing
    FILE *binaryFile1 = fopen(INPUT_FILENAME, "rb"); // File pointer 1
    FILE *binaryFile2 = fopen(INPUT_FILENAME, "rb"); // File pointer 2
    FILE *binaryFile3 = fopen(INPUT_FILENAME, "rb"); // File pointer 3
    FILE *binaryFile4 = fopen(INPUT_FILENAME, "rb"); // File pointer 4

     // Check if any of the file pointers is equal to zero, indicating a
file opening error
     if (binaryFile1 == 0 || binaryFile2 == 0 || binaryFile3 == 0 ||
binaryFile4 == 0)
    {
        fputs("File error", stderr);
        exit(1);
    }

    // Variables to store the result of fread for each file
    int result1, result2, result3, result4;

      // Read data from binary files into different sections of the
'numbers' array
          result1  =  fread(numbers[4],  sizeof(int8_t),  chunk_size,
binaryFile1);
          result2  =  fread(numbers[5],  sizeof(int8_t),  chunk_size,
binaryFile2);
          result3  =  fread(numbers[6],  sizeof(int8_t),  chunk_size,
binaryFile3);
          result4  =  fread(numbers[7],  sizeof(int8_t),  chunk_size,
binaryFile4);

     // Check if the number of elements read is less than the expected
'chunk_size' for any file
     if (result1 < chunk_size || result2 < chunk_size || result3 <
chunk_size || result4 < chunk_size)
    {
        fprintf(stderr, "File end reached\n");
        exit(1);
    }
    // DECLARING THRESHOLDS AND INITIALISING THEM
    int8_t threshold1 = 127;
    int8_t threshold2 = 127;
    int8_t threshold3 = 127;
    int8_t threshold4 = 127;

     // Print a message to the standard output indicating the number of
iterations
      fprintf(stdout, "Near OMP threads. Number of iterations %d\n",
chunk_n);

    // Initialize variables for measuring time
    double avg_time = 0;
    clock_t start1 = clock();
    struct timeval shr1, shr2;
    float comp_time;

    int cr;
    for (int i = 0; i < 1000; i++)
    {
         // Read data from binary files into different sections of the
'numbers' array based on the value of 'count'
          result1 = fread(numbers[((count * 4) % 8)], sizeof(int8_t),
chunk_size, binaryFile1);
        result2 = fread(numbers[((count * 4) % 8) + 1], sizeof(int8_t),
chunk_size, binaryFile2);
        result3 = fread(numbers[((count * 4) % 8) + 2], sizeof(int8_t),
chunk_size, binaryFile3);
        result4 = fread(numbers[((count * 4) % 8) + 3], sizeof(int8_t),
chunk_size, binaryFile4);

           // Check if the number of elements read is less than the
expected 'chunk_size' for any file
         if (result1 < chunk_size || result2 < chunk_size || result3 <
chunk_size || result4 < chunk_size)
        {
            fprintf(stderr, "File end reached\n");
            exit(1);
        }

        comp_time = 0;
        // Stores current time at starting of pragma loop
        gettimeofday(&shr1, 0);

        // Set the number of OpenMP threads to 4
        omp_set_num_threads(4);

#pragma omp parallel for private(cr) schedule(static)
        for (cr = 0; cr < 4; cr++)
        {

            // Conditional statements for different values of 'cr'
            if (cr == 0)
            {
                filter(numbers[((count + 1) * 4) % 8], filtered[((count
+ 1) * 4) % 8], chunk_size, WINDOW_SIZE, MOM_WIN, N, RPL_OPTION,
RPL_CONST, &threshold1, flags[(count + 1) % 8]);
            }
            if (cr == 1)
            {
                      filter((numbers[(((count + 1) * 4) % 8) + 1]),
(filtered[(((count + 1) * 4) % 8) + 1]), chunk_size, WINDOW_SIZE,
MOM_WIN, N, RPL_OPTION, RPL_CONST, &threshold2, (flags[(((count + 1) *
4) % 8) + 1]));
            }
            if (cr == 2)
            {
                      filter((numbers[(((count + 1) * 4) % 8) + 2]),
(filtered[(((count + 1) * 4) % 8) + 2]), chunk_size, WINDOW_SIZE,
MOM_WIN, N, RPL_OPTION, RPL_CONST, &threshold3, (flags[(((count + 1) *
4) % 8) + 2]));
            }
            if (cr == 3)
            {
                      filter((numbers[(((count + 1) * 4) % 8) + 3]),
(filtered[(((count + 1) * 4) % 8 + 3)]), chunk_size, WINDOW_SIZE,
MOM_WIN, N, RPL_OPTION, RPL_CONST, &threshold4, (flags[(((count + 1) *
4) % 8) + 3]));
            }
        }

        // Stores time at the end of pragma loop
        gettimeofday(&shr2, 0);

        // Calculate the total computation time in milliseconds
```

```
           comp_time  =  (float)(1000.0  *  (shr2.tv_sec  -  shr1.tv_sec)  +
(0.001 * (shr2.tv_usec - shr1.tv_usec)));

     //*************************** Uncomment this commented section to
get the outputfiles*************************//
/***********
        // Write filtered data into text files using WriteFile function
           WriteFile(op_flname1, (filtered[(((count + 1) * 4) % 8)]),
chunk_size); // WRITES THE FILTERED DATA INTO TEXT FILES
           WriteFile(op_flname2, (filtered[((((count + 1) * 4) % 8) +
1)]), chunk_size);
           WriteFile(op_flname3, (filtered[((((count + 1) * 4) % 8) +
2)]), chunk_size);
           WriteFile(op_flname4, (filtered[((((count + 1) * 4) % 8) +
3)]), chunk_size);

        // Write flags data into text files using WriteFile function
         WriteFile(op_flname1, (flags[((count + 1) % 8)]), chunk_size);
// WRITES THE FLAGS DATA INTO TEXT FILES
         WriteFile(op_flname2, (flags[((((count + 1) * 4) % 8) + 1)]),
chunk_size);
         WriteFile(op_flname3, (flags[((((count + 1) * 4) % 8) + 2)]),
chunk_size);
         WriteFile(op_flname4, (flags[((((count + 1) * 4) % 8) + 3)]),
chunk_size);
***********/

        // Increment the count of outer loop
        count++;

        // Print information to the standard output
        fprintf(stdout, "%d\t%f\n", count, comp_time);
    }

    // Record the time at the end of the section
    clock_t stop1 = clock();

    // Calculate the duration of the section in clock cycles
    double duration1 = (double)(stop1 - start1);

    // Free memory for 'numbers' array
    for (int i = 0; i < 8; i++)
    {
        free(numbers[i]);
    }

    // Free memory for 'filtered' array
    for (int i = 0; i < 8; i++)
    {
        free(filtered[i]);
    }

    // Free memory for 'flags' array
    for (int i = 0; i < 8; i++)
    {
        free(flags[i]);
    }

    // Free the memory for the arrays of pointers
    free(numbers);
    free(filtered);
    free(flags);

    return 0;
}
```