Vrije Universiteit Amsterdam

Master Thesis

Removing Radio Frequency Interference in the LOFAR using GPUs

Author:
Linus Schoemaker

Supervisors:
Dr. Rob. V. van Nieuwpoort
Alessio Sclocco

A thesis submitted in fulfilment of the requirements for the degree of Master of Science

in the

Computer Systems Group
Department of Computer Science

May 2015

Declaration of Authorship

I, Linus Schoemaker, declare that this thesis titled, 'Removing Radio Frequency Interference in the LOFAR using GPUs' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a masters degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ____

Date: ____

"Computer Science is no more about computers than astronomy is about telescopes."

Edsger W. Dijkstra

VRIJE UNIVERSITEIT AMSTERDAM

Abstract

Faculteit Exacte Wetenschappen
Department of Computer Science

Master of Science

Removing Radio Frequency Interference in the LOFAR using GPUs

by Linus SCHOEMAKER

Radio Frequency Interference poses a large problem for analysing data gathered by modern, large radio telescopes such as LOFAR. Detecting and removing this RFI on-line is necessary as telescopes are in almost continuous operation. While at LOFAR a feasible solution existed in the form of an algorithm run on an IBM Blue Gene/P supercomputer, this is expensive. Porting this algorithm to a Graphical Processing Unit would drastically reduce costs.

This paper describes the process of porting the algorithm to a new architecture, and examines the results both in terms of speed and quality. Both were found to be acceptable: run times of the most successful implementation in terms of quality allowed a single GPU to process up to 200 radio receivers simultaneously in an on-line fashion.

Acknowledgements

Contents

List of Figures

List of Tables

Abbreviations

LOFAR	**Lo**w **F**requency **Ar**ray
RFI	**R**adio **F**requency **I**terference
GPU	**G**raphics **P**rocessing **U**nit
SKA	**S**quare **K**ilometer **A** rray

Chapter 1

Introduction

1.1 History

Radio Astronomy is a fantastic way of exploring and acquiring a better understanding of the vast universe we inhabit. It studies the radio waves emitted by sources far away from our home planet in order to better grasp what is out there.

Ever since Karl Jansky observed the Milky way emitting radio waves in the 1930s, we have come a long way, discovering many other sources like stars, galaxies, quasars and pulsars. Even confirmation of the Big Bang Theory was made possible by the discovery of comic microwave background radiation, encountered by radio astronomers Arno Penzias and Robert Wilson, who were later awarded with a Nobel Prize for this.

Ever since the early days of radio astronomy the tools we use to listen to space have become more and more advanced. While Jansky's first telescope - a 30 meter wide contraption rotating on a set of Ford Model-T tires - might have sufficed for picking up the Milky Way, to study the signals emitted by much fainter sources we need more precise instruments.

1.2 Radio telescopes

While radio astronomy can (and still is) done in single, large dish telescopes like the Arecibo radio telescope in Puerto Rico or the Five hundred meter Aperture Spherical

Telescope (FAST) currently being built in China, many modern day radio telescopes make use of a technique called radio interferometry. In this approach, instead of a single large dish, the data is collected by an array of telescopes of which the data is combined to form a single image. This technique increases the total signal collected, but more importantly vastly increases the resolution of the final result, allowing for much more detailed study of the sources of these radio waves.

Examples of these large, distributed telescopes are the VLA and VLBA, the Very Large Array and the Very Long Baseline Array in the United States, the Giant Metrewave Radio Telescope in India, and most notably the LOFAR, the Low-Frequency Array in the Netherlands. This last telescope, and solving its associated problems, is the focus of this thesis.

The LOFAR is currently the largest radio telescope ever built, consisting of about 25,000 small antennas concentrated in at least 48 larger stations, spanning an area of approximately 300,000 square meters. It is in almost constant operation, generating an enormous amount of data. Processing this data is no easy feat, and used to be done using an IBM Blue Gene/P supercomputer at the University of Groningen. (lofar-paper)

1.3 Issues

However, listening to space does present some issues: it is not the only thing making noise. While radio telescopes advanced and became larger, more powerful and more sensitive, sources of radio waves on earth also became louder, and more importantly, more numerable. Such sources of radio frequency interference - or RFI - can originate from deliberate radiation like radio and video broadcasting, but also from unintended sources like cars, planes, power lines or wind turbines (Bentum et Al. 2010 [zie offringa's thesis]). Unfortunately, when trying to study a pulsar many light years away, a plane passing by can easily drown out the weak signals coming from the other side of the galaxy.

While it is unavoidable to have a part of your measurements disturbed by interference caused by earthly sources in a telescope so vast, it is important to be able to detect and remove this interference in the final measurements.

Furthermore, simply having a way of detecting and removing interference will not suffice. The LOFAR is in almost constant operation, generating up to 200 Gigabits of data each second. If the method of RFI removal cannot be performed on-line, i.e. as fast as the telescope is generating data, a backlog of data will be created to which the removal algorithm will never catch up. This is obviously an undesirable situation.

Initially, removing interference was done off-line, using an algorithm called SumThreshold, devised by Andre Offringa. This was then ported for on-line operation on the IBM Blue Gene/P later.

While the Blue Gene was an extremely powerful machine at its time of purchase, a few years have passed since then. The machine has become slightly outdated, and difficult to maintain. Buying a new Blue Gene-type machine is an option, but a very costly one.

1.4 This thesis

In this thesis, we will investigate an alternative solution to this problem. With the advance of graphics processing units, which are widely available, relatively cheap and have enormous potential computing power, these architectures seemed like a very suitable replacement.

Therefore this thesis describes the conversion of the existing LOFAR algorithm for RFI removal from the original Blue Gene, to a new platform: the GPU. Because there are a number of substantial differences between these architectures, the original algorithm required a large overhaul.

The main research question is therefore whether it is possible to convert the SumThreshold algorithm from the Blue Gene to GPUs. Furthermore, the new algorithm has to be of sufficient quality, and has to be able to perform its operations on-line, i.e. as fast as data is generated.

The results of this thesis show that GPUs are indeed a prime candidate for replacing the Blue Gene in this part of the LOFAR pipeline. Both in speed and in accuracy the new algorithm performs sufficiently well, allowing for on-line processing of incoming astronomical data.

The rest of this thesis is structured as follows: chapter 2 provides necessary background information on the LOFAR and its data structures, different RFI removal methods and on the different architectures discussed in this work. Chapter 3 discusses the SumThreshold algorithm in general, while chapter 4 describes the steps and choices made while porting it from one architecture to the next. Chapter 5 contains the results of the tests performed on this new implementation, both concerning speed and quality. Chapter 6 discusses the results as well as related work, and in chapter 6 the conclusions of this thesis are presented.

Chapter 2

Background

2.1 RFI

RFI removal is a challenging problem within radio astronomy, but before trying to explain how to solve this problem, it is important to understand what exactly is RFI.

Interference could be described as the weeds of radio waves. It encompasses any radio waves that are unwanted in the final measurement. In the case of radio astronomy, RFI are radio waves caused by anything other than what we are trying to measure or study. More specifically, this means radio waves caused by sources from earth, instead of the part of space we are interested in.

2.2 Types of RFI

In order to deal with interference, it is important to understand the different shapes and forms in which it is manifested within the recorded data. Different sources of interference may create very different patterns in the data, which can require different detection methods.

Interference within the data collected by radio telescopes can be divided into three different categories:

1. Impulse-like bursts

2. Long narrow-band random oscillations

3. A superposition of the previous types

Furthermore, interference can originate from both stationary and non-stationary sources, both of which create different patterns in the data.

Impulse-like interference is interference that is only present for short amounts of time, usually repeated within a certain interval. The separate bursts occupy a large band of frequencies, but only a small slot of time. The interval of the bursts may be random, or fixed. A good example of a source of impulse-like interference are radar stations, as can be seen in figure 2.1 (c).

In impulse-like RFI there is little difference in the data patterns whether the source is stationary or non-stationary. A source that is moving towards the receiver might increase in power the closer it gets, but if the intervals between the pulses are regular they will not change. The range of frequencies affected might also increase slightly, but this is not of influence when trying to remove the affected data.

Long narrow-band interference is in some sense the opposite of impulse-like interference. Unlike the impulses, it is a constant stream and it is confined to a small frequency sub-band. Examples of sources of narrow-band RFI are cars and airplanes, or radio stations.

In RFI that has a longer duration, patterns do differ if the source is non-stationary or stationary. Stationary long narrow-band interference will occupy the same frequency sub-band throughout the measurement. Non-stationary sources however will drift through different frequencies as they move towards or away from the receiver, because of the Doppler effect.

Finally there exist combinations of the previous two types of interference. This is interference that exists within a narrow sub-band, but is not constant. The stationary or non-stationary nature of the source exhibit the same patterns as with long narrow-band interference.

Figure 2.1 shows examples of the different types of RFI encountered in the wild.

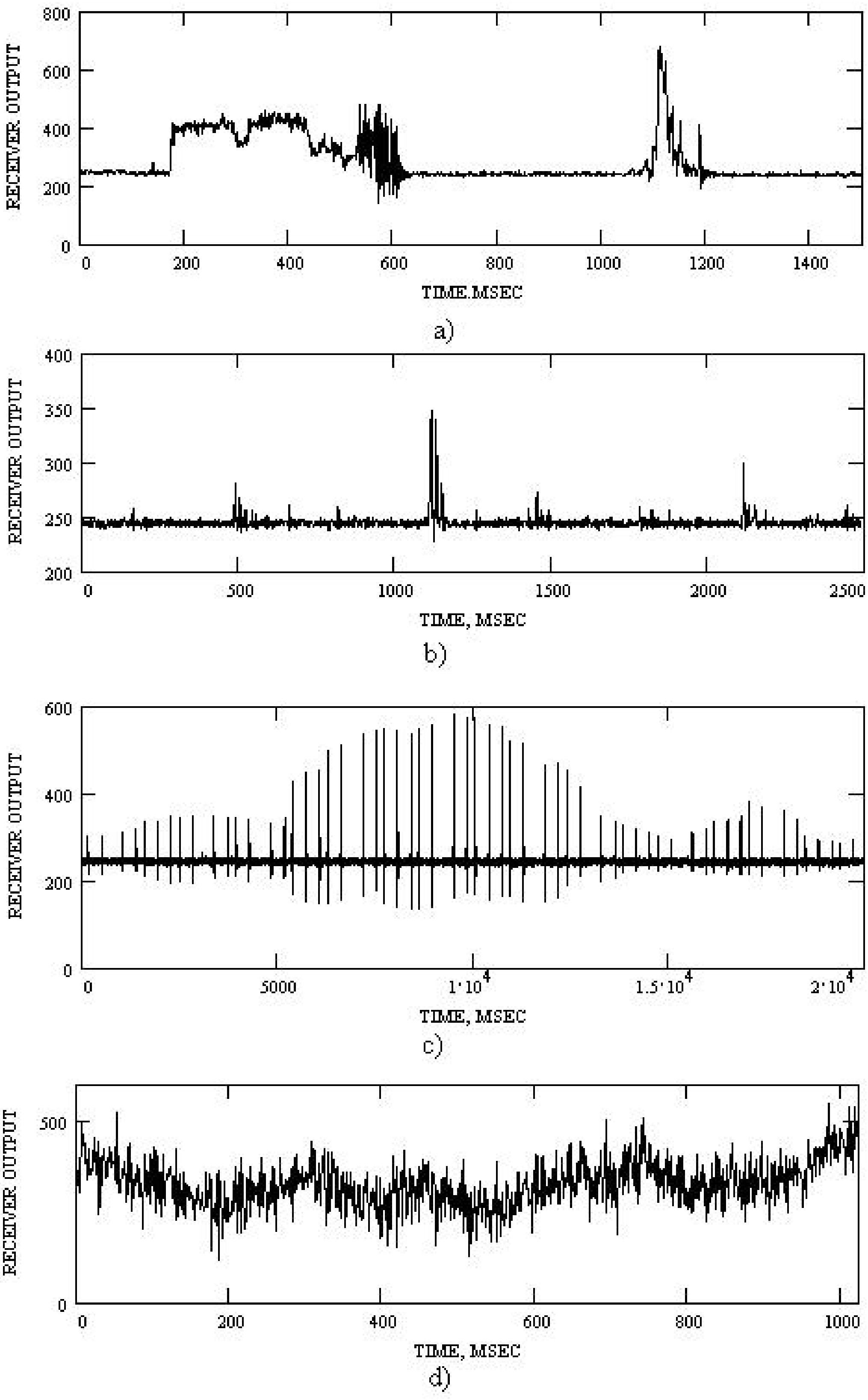

FIGURE 2.1: Examples of man made RFI: a) and b) are random impulse-like RFI, c) is impulse-like interference with a fixed interval, caused by a radar. d) shows an example of long narrow-band RFI. This image was taken from [1].

2.3 Data Structure

In order to understand the methods and algorithms for dealing with radio frequency interference, it is necessary to understand in what structure the data is presented to the different algorithms.

Our main focus will be on LOFAR, but the differences between the data structures of different radio telescope arrays is small to non-existent, save for the size and number of frequencies registered. Neither of these are important for RFI mitigation methods.

The LOFAR radio telescope consists of a large number of individual receivers. These receivers measure the intensity of radio waves on different frequency sub-bands over a period of time. Receivers also have the possibility of collecting different polarizations of radio waves. The collected measurements are then sent over to a central processing station.

Filtering out RFI can be done on several places within the LOFAR pipeline. The first opportunity is immediately after the data enters the pipeline. At this stage, the data is still uncompressed and uncorrelated. This means that flagging contaminated signals can be done extremely precisely, as the data is still at its highest resolution. This does of course imply a rather large processing cost.

After this, the data is transformed by a Fast Fourier Transform module. After this is done, another good opportunity arises for removing RFI. This is less precise then at the previous point, but therefore also faster.

The pipeline then makes a split: data either goes into the imaging pipeline, or the beam forming mode. In both the data is correlated. In correlation, a single data set is formed out of the many sets supplied by the different receivers. This includes transforming the data to account for the different positions of the receivers, as well as the rotation of the earth.

After correlation flagging RFI is also possible. Again, this results in a lower potential accuracy, but with a great improvement with regards to processing time.

Figure 2.2 shows a high level overview of the LOFAR pipeline and the possible points of RFI detection.

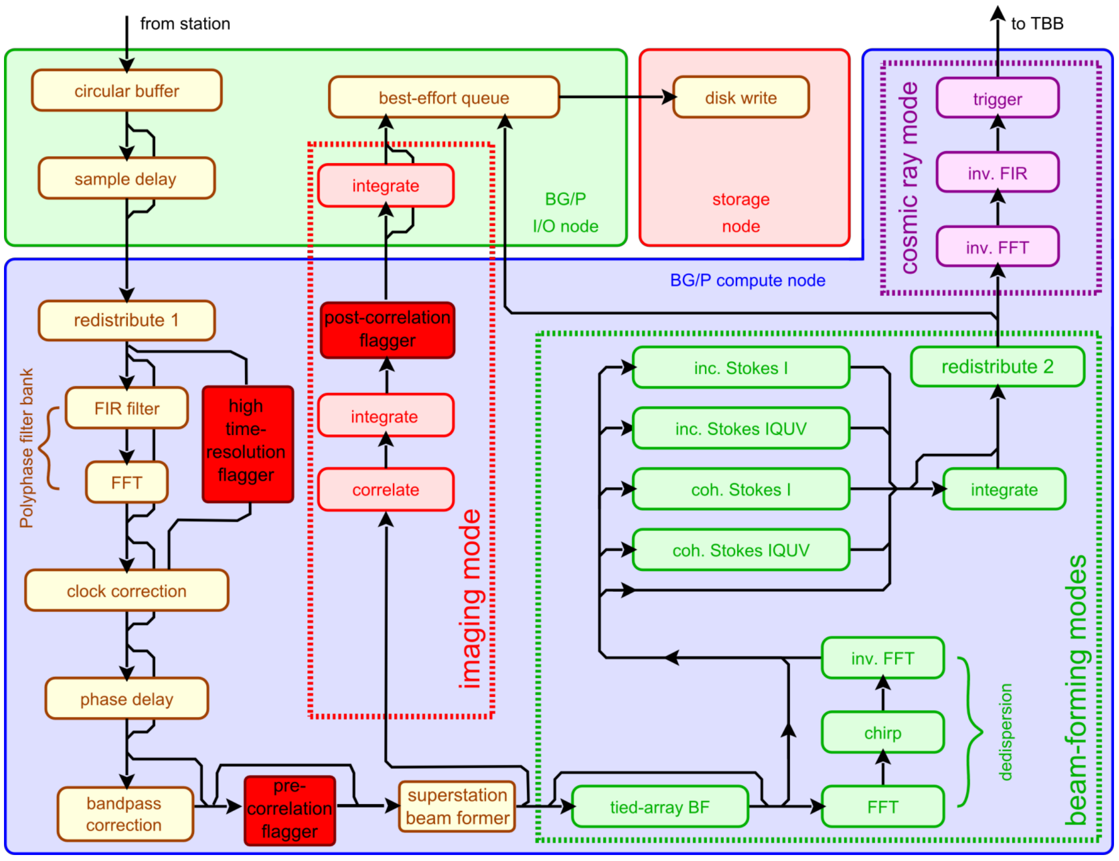

Figure 2.2: A high level overview of the LOFAR pipeline and the possible points of RFI detection.

Some RFI detection algorithms can only be performed pre-correlation, while some are only suitable after correlation. Some methods however can be applied at either stage.

The input given to any algorithm, be it pre- or post-correlation, consists of a matrix of values. The x-axis of the matrix corresponds to different timeslots. The size of these slots is dependent on the rate of sampling.

The y-axis of the matrix corresponds to different channels within a frequency sub-band. Usually in the LOFAR imaging mode, a single sub-band consists of 256 channels of 0.8 kHz resolutions. The values within the matrix represent the intensity of the signals received at that time, and in that frequency sub-band. A single matrix represents only one polarization.

The input data described above can be visualized in a so-called grey plot, in which the signal strength at a certain point corresponds to a tone of grey. The stronger the signal, the darker the point within the figure. An example is provided in figure 2.3.

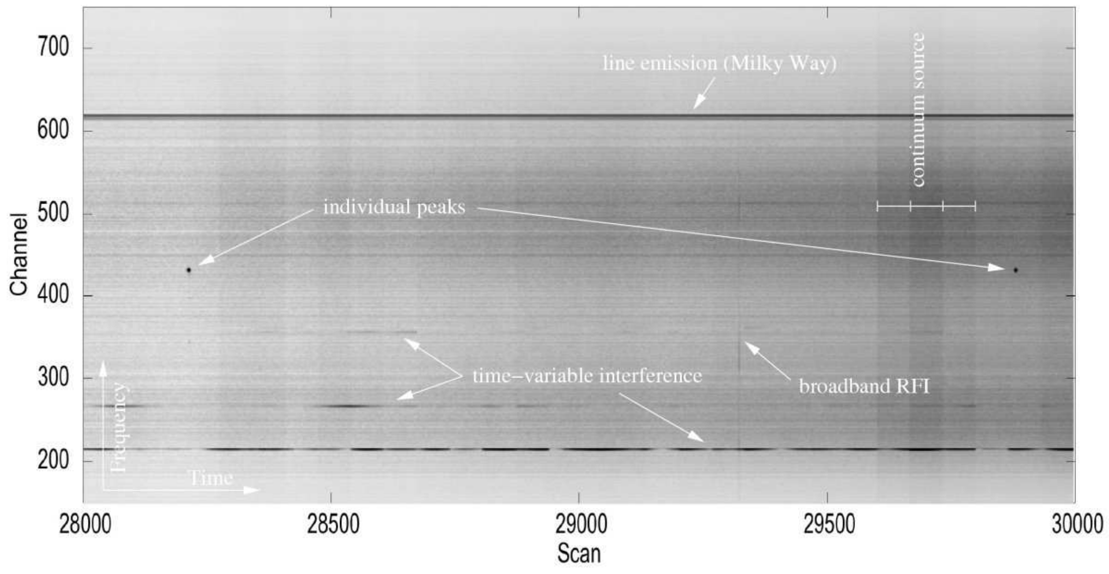

FIGURE 2.3: An example grey-plot showing radio-astronomical data as received by a radio telescope. Notice the RFI present in these measurements. This image was taken from [1]

2.4 RFI Mitigation algorithms

Since RFI has plagued radio astronomy for a long time, there have been a fairly large number of algorithms and methods created to deal with it.

For a complete taxonomy of current algorithms and their applications we would like to refer to [3], as a complete summary would be out of scope of this thesis.

We have however included a graph detailing the applicability of the algorithms and methods mentioned in [3] to give a brief overview in section 2.4.1. In section 2.4.2 we give a small introduction into thresholding algorithms, a category to which the algorithm ported in this thesis belongs.

2.4.1 Overview

Table 2.1 contains an overview of the different algorithms presented in [3] and their applicability to different types of RFI.

As this table shows, there are algorithms that are capable of dealing with all different kinds of RFI. As mentioned before, we have chosen to implement the SumThreshold algorithm for the LOFAR pipeline. A rationale for this choice is given in section 3.1

	Stationary impulse-like	Stationary long narrow-band	Stationary combined	Non-stationary impulse-like	Non-stationary constant
Thresholding	✓	✓†	✓†	✓	✓
Spatial filtering	✓*	✓	✓*		
Adaptive cancellation		✓	✓*	✓*	✓*
Fringe fitting	✓	✓	✓		
SVD			✓		
Combinatorial thresh.	✓	✓†	✓†	✓	✓
Surface fitting	✓	✓	✓	✓	✓
SumThreshold	✓	✓†	✓†	✓	✓

TABLE 2.1: *: This algorithm will work, but is likely to throw out significant amounts of actual data together with the RFI. †: Depending on the size of the frequency subbands.

2.4.2 Thresholding Algorithms

A large number of methods for dealing with RFI are in some way or another based on thresholding. The basic premise is simple: when a value exceeds a certain threshold, it is flagged as RFI and removed.

This type of detection is based on the assumption that RFI, in the case of radio astronomy emitted from sources on earth, is much stronger than the signals from space we are actually interested in. By looking at the values and singling out those that stray too far from the norm, we are likely to remove interference, while leaving actual data intact.

One of the difficulties in thresholding algorithms lies in the setting of the threshold. Set the threshold too high, and RFI will pass underneath it. Too low, and actual data will be removed along with the RFI.

Manual adjustment of the threshold is an option, but in large systems like the LOFAR unfeasible, especially if on-line processing is required. A common alternative is basing the threshold on the mean of the data. This however fails whenever there is too much RFI present, or the RFI is particularly strong, as this affects the mean too much.

An often used alternative is using the median, which is not affected by strong outliers. While it is slightly more computationally expensive to calculate, it is often a better candidate for filtering than the average or mean.

Simple Thresholding

The simplest thresholding algorithm, referred to simply as Thresholding, will loop over either the rows or the columns of the input matrix described in section 2.3. It will determine the median of the row or column, and set the threshold based on it. Then, it will compare all values within the current row or column, and flag any that exceed the threshold. These flagged values can later be dealt with in different ways. They can either be nulled (replaced with zero), or be replaced with an estimated sample, for example this median.

The simple thresholding algorithm works relatively well, and is sufficiently simple and fast for on-line processing. It does have a few short comings however, especially with regards to transient RFI. Transient RFI ramps up in strength over time until it peaks, and then decreases. While simple thresholding will detect and remove the peak, it often fails to detect the interference leading up to it.

Combinatorial Thresholding

An improvement of the simple thresholding algorithm called Combinatorial Thresholding offers a solution to the transient RFI problem. It was introduced by A. Offringa in [4], which in turn was based on [5]. Instead of comparing single values to the threshold, it will compare a set of neighbouring samples to a slightly lower threshold. This is done in iterations, with the sets of samples compared increasing in size while the threshold is lowered.

The samples in the set are summed together. When comparing n samples, the sum of the values is compared to $n \times \chi_n$. If the sum is greater than this threshold, all samples in the set are flagged.

SumThreshold

A final improvement on the Combinatorial Thresholding algorithm called SumThreshold was introduced by A. Offringa in [4]. The Combinatorial Thresholding algorithm has the tendency to flag too many samples when there is a very abrupt spike, as the sum of values will be very high, even though the ordinary low values surrounding the spike are not actually interference. To combat this, the SumThreshold algorithm replaces any values flagged in previous iterations with the current threshold when summing.

2.5 Hardware architectures

When porting an algorithm from one hardware architecture to another, it is important to understand the differences between them. Without a proper understanding it will be virtually impossible to implement an efficient algorithm. Due to the on-line requirements of this project, coaxing the most performance possible out of our program is critical.

This section gives a brief introduction in the different architectures on which the old and the new version of the RFI detection algorithm ran.

2.5.1 Blue Gene/P

The Blue Gene/P architecture used previously in the LOFAR is a supercomputer designed by IBM. It is a system composed of many small nodes, with several high-speed communication channels between them.

Important for our work is the fact that the Blue Gene/P falls in the MIMD (Multiple Instruction, Multiple Data) category of Flynn's taxonomy. This means that performance is not impacted whether different nodes perform exactly the same operation, or completely different ones. Naturally if there are dependencies between nodes this does become a consideration, but in general it is not a concern.

Due to this MIMD architecture, and the fact that individual nodes within the Blue Gene/P architecture are relatively powerful, the original RFI detection algorithm could be implemented using fairly coarse parallelization. A single matrix as introduced in section 2.3, of which different nodes create one for each combination of sub-channel and polarization, would be sent to a single node, which could then perform the detection algorithm without any dependencies to any other node.

2.5.2 GPUs

GPUs operate in a very different way from the MIMD architecture of the Blue Gene/P computer. They instead fall in the SIMD (Single Instruction, Multiple Data) category of Flynn's taxonomy, or at least should be treated as such in order to achieve the highest performance.

The greatest difference between MIMD and SIMD architectures is that in a SIMD setting, all nodes are performing the exact same operation, but on different data inputs In the MIMD however, it does not matter whether the different nodes do exactly the same, or something completely different.

Current generation GPUs consist of a large amount of separate cores. For this research, we have made use of CUDA, nVidia's parallel computing architecture. This allows for general purpose calculations on GPUs.

In CUDA, one can launch a great number of simultaneous threads. These threads exist within a hierarchy: threads are grouped into so-called blocks, while blocks are grouped into grids. See figure 2.4 for a visual example.

Thread blocks are mapped to Streaming Multiprocessors (SMs) on the GPU. An SM has 32 separate cores. The threads within the mapped thread block are grouped into groups of 32 called warps, and are executed simultaneously, as long as they perform the same operations. If different threads diverge by following different execution paths, their execution gets serialized. This is obviously detrimental to performance and should therefore be avoided.

Another consideration is the memory structure of a GPU. This is organized in a similar hierarchy as the threads. Threads have access to a few thread-local registers, the contents of which cannot be accessed by other threads. Access to this memory is the fastest within the hierarchy.

Threads within a block however have access to something called shared memory. This memory is, as the name implies, shared between all threads within a block, but cannot

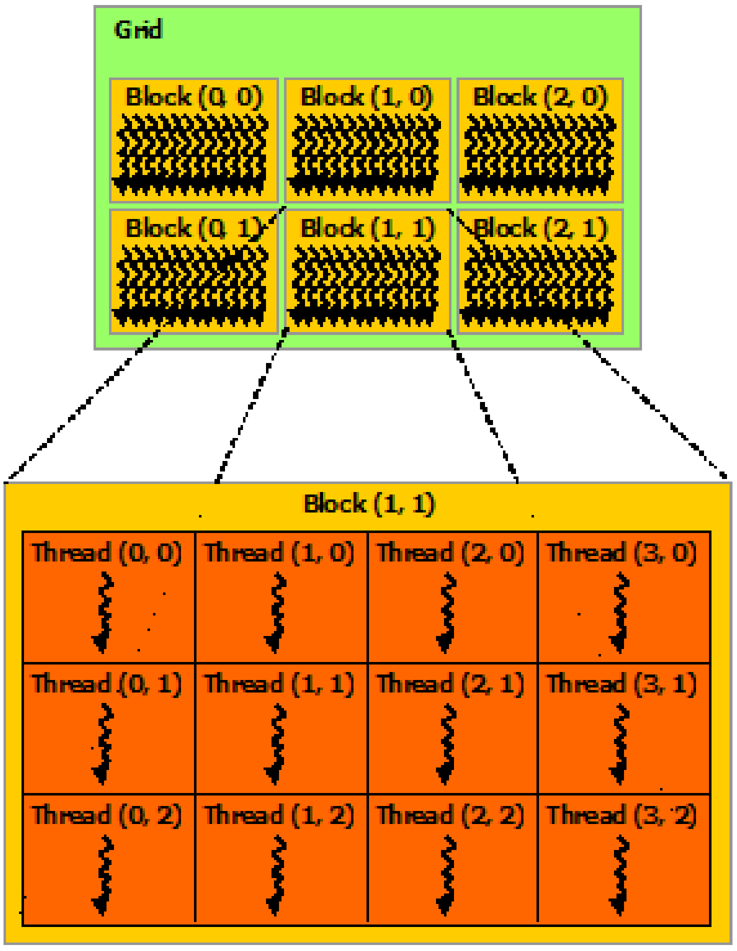

FIGURE 2.4: CUDA Thread Hierarchy. This image is taken from [2]

be accessed by threads outside this block. It is still relatively small (48 KB per SM for devices with a Compute Capability 2.0 or higher), but access is fast.

Finally, all threads regardless of block can access the global memory. This is the largest memory store available on the GPU, but it is the slowest memory within the hierarchy. In order for algorithms to perform optimally, it is important to structure memory accesses as efficiently as possible. Section 4.1 explains how this is done in our algorithm.

The CUDA memory hierarchy is illustrated in figure 2.5.

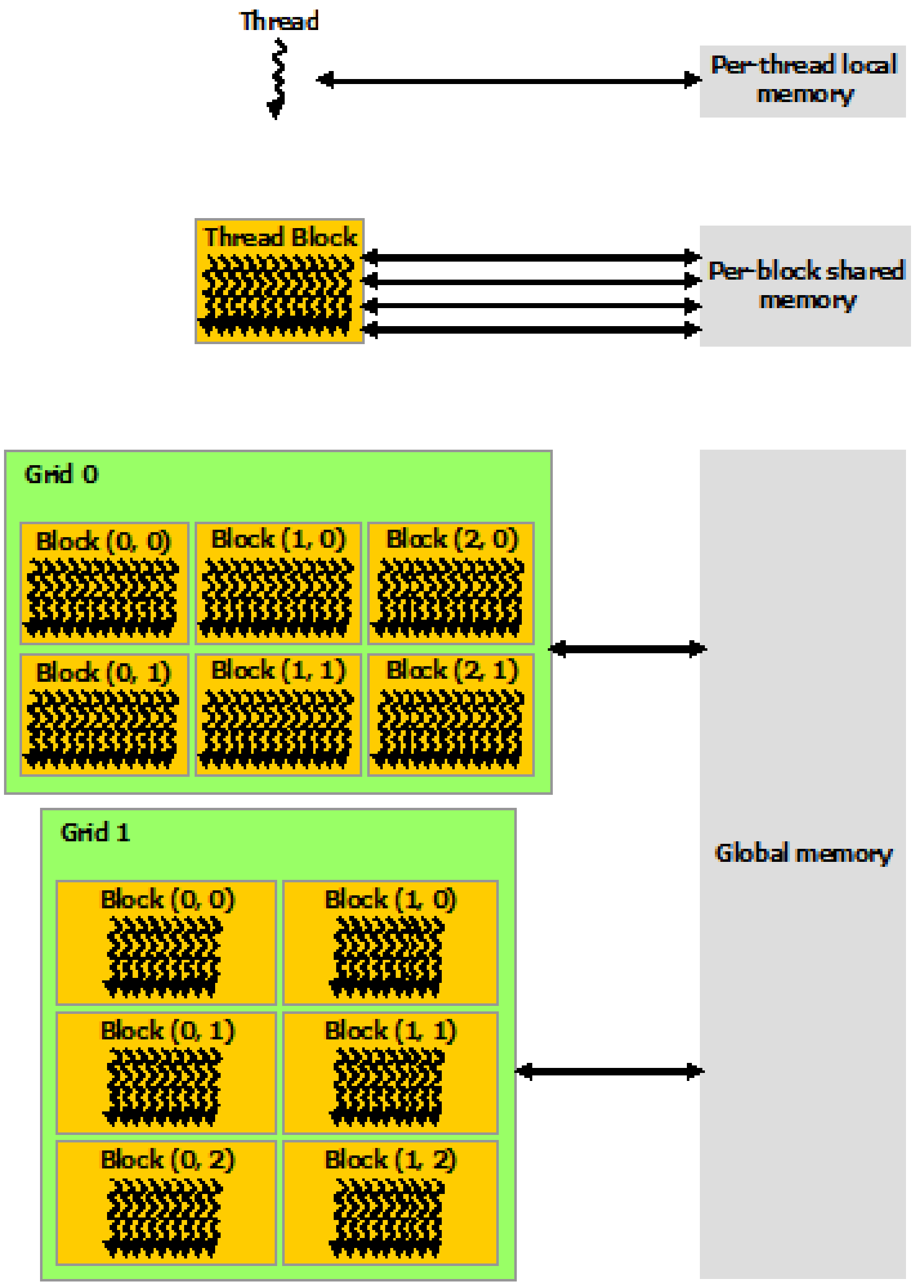

FIGURE 2.5: CUDA Memory Hierarchy. This image is taken from [2]

Chapter 3

The SumThreshold Algorithm

To deal with Radio Frequency Interference in the LOFAR, a suitable algorithm was required. We chose the SumThreshold algorithm [4].

This chapter starts of with a rationale for the choice of this algorithm. It then continues with a detailed explanation of the working of the basic algorithm. After this introduction, the porting of the algorithm from its original architecture to the GPU is discussed.

3.1 Why SumThreshold?

There are many different methods and algorithms for removing RFI in the context of radio telescopes. Different methods have different strengths and weaknesses, so the algorithm used requires a rationale.

The SumThreshold algorithm was used in this case for a few reasons:

- It is suited for all different types of RFI
- It runs in $\mathcal{O}(n)$, making it suitable for online processing
- It has already been implemented in the LOFAR and shown to work well

3.1.1 Applicability to RFI Types

As explained in chapter 2.2, RFI comes in many different shapes and sizes. From [3] however we learned that there are algorithms for dealing with all of these different types, and even that some algorithms are suited for dealing with any sort of RFI.

SumThreshold can be grouped among these algorithms, as can be seen in table 2.1. This obviously greatly simplifies things, as implementing a single algorithm is trivially easier than doing several different ones.

Furthermore, in the same paper ([4]), Offringa shows that SumThreshold has the greatest accuracy when compared to other RFI removal methods. This is naturally a great trait as well when selecting a method.

3.1.2 Performance

While there are many different methods for removing RFI, not all are suitable for online processing. Some methods require complex and expensive calculations. Because we are looking for a method for online processing, detecting the RFI cannot take longer than it takes to make the measurements.

SumThresholding runs in $\mathcal{O}(n)$, likely making it fast enough for online processing.

3.1.3 Existing versions

Maybe the most important reason for choosing the SumThreshold algorithm is that it has already proven to work in this context. The previous LOFAR pipeline ran a version of the algorithm on the IBM Blue Gene/P architecture, with satisfactory results.

All of these reasons combined convinced us that this method was very well suited for the LOFAR pipeline.

3.2 The Algorithm

The SumThreshold algorithm, as mentioned in section 2.4.2, is a variation on the standard thresholding algorithm. To reiterate, the data is given to the algorithm as a matrix, with timeslots on the x axis and frequency sub-bands on the y axis.

The algorithm loops over either the rows or the columns of the matrix. For each of these vectors, it first computes the threshold by calculating the median of the values in the vector.

SumThreshold performs multiple iterations over each vector in the input matrix. In each iteration, a number of samples is summed. The number of samples for each iteration is called the window-size, which increases with each new iteration.

In the first iteration, the window size is set to 1. This is effectively simple thresholding: each sample is compared to the set threshold. For every value that exceeds the threshold, a flag is set in the flag map. The flag map is a matrix of the same size as the input data. The elements consists of booleans (or more accurately, unsigned chars, which is the smallest data type in C [6]). The position of the flag in the map corresponds to a single value in the input data. If a value is unflagged, its corresponding flag in the map is set to zero. A flagged value is set to one.

In subsequent iterations, each value is summed together with its neighbouring values up to the window size. If a value has already been flagged in a previous iteration, its value in the sum is replaced with the value of the current threshold. The threshold is decreased every time the window size is increased.

If the sum of all the values within the window exceed the current threshold, all values in the window are flagged as RFI.

Below is a pseudo-code version of the basic SumThreshold algorithm:

```
sumThreshold(float[] samples,
  boolean[] flags, int window,
  float threshold) {

  for (int base = 1;
       base + window < samples.length;
       base++)
  {
    float sum = 0.0;

    for (int pos = base;
         pos < base + window;
         pos++)
    {
        if (!flags[pos]) {
          sum += samples[pos]
        }else{
          sum += threshold
        }
    }

    if (sum >= window * threshold) {
      // flag all samples
      // in the sequence!

      for (int pos = base;
           pos < base + window;
           pos++)
      {
        flags[pos] = true
      }
    }
  }
}
```

Calculating the threshold

As mentioned earlier, the threshold used within SumThreshold is computed from the median of the values within a slice (either in the time or the frequency direction). In practice, a slightly more complex formula is used: $\alpha \times \frac{1.5^i}{w} \times (\sigma \times \beta) + \eta$

in which α is a constant set to 6.0, i is the current iteration, w the current window size, σ the standard deviation of the values, β the base sensitivity, set to 1.0, and η the median.

While at first glance it might seem that the threshold *increases* with each iteration, as i will grow, note the it is divided by the window size w, which is doubled in each iteration, thereby growing faster than i, resulting in a lower threshold.

Chapter 4

Porting to the GPU

This chapter describes the changes required to the original algorithm in order to port it from the Blue Gene/P to CUDA.

As mentioned in section 2.5.1, the Blue Gene architecture consists of a number of relatively powerful nodes. Due to this configuration, it handles coarse-grained parallelization rather well. This means that in order to parallelize the SumThreshold algorithm, all that is needed is to dispatch different input data to different nodes. The nodes will then perform the entire calculation without any communication with other nodes, and return the result for central processing.

The GPU on the other hand, is a so called SIMT (Single Instruction, Multiple Thread) architecture (see section 2.5.2). While a SIMT architecture could in theory perform in exactly the same way as a MIMD architecture, approaching it as a SIMD (Single Data, Multiple Data) architecture will yield the highest performance. This means that in order for an algorithm to run efficiently, all threads will need to be performing the same operation at the same moment.

In an initial version of our ported algorithm, a single thread on the GPU was assigned a single row or column from the input matrix. This however turned out to be too coarse grained still. This is due to the divergence between different threads, meaning that different threads would still be performing different operations at different times. In a GPU architecture, divergent threads are paused until one branch of execution is completed, after which *they* are executed, which means any gains to be had from parallelizing are lost.

In the final version of the algorithm, a thread is launched for each separate value of the input matrix. The following sections describe all parts of the algorithm, detailing the adaptations made to get the maximum performance.

4.0.1 Calculating Statistics

Section 3.2 mentions how the threshold used in each iteration of the algorithm is calculated. As can be seen, a requirement for this calculation is knowing both the median and the standard deviation for the current row.

In the original Blue Gene/P version, calculating the median is a simple operation. A node would sort the input vector, and retrieve the $N/2$th value. The standard deviation is calculated by first calculating the mean, gotten by summing all values and then dividing by the number of values. Then, the difference between the mean and each value was squared and summed and again divided by the number of samples. The resulting value is called the variance, the square root of which is the standard deviation.

Below is a pseudo code sample showing this algorithm:

```
calculateStatistics(float[] samples, int n)
  samples = samples.sort() //Quick sort
  float mean = 0.0

  for (float sample in samples)
    mean += sample

  mean = mean / n

  median = samples[n / 2]

  stdDev = 0.0
  for (float sample in samples)
    float diff = sample - mean
    stdDev += diff * diff

  stdDev = stdDev / n
  stdDev = sqrt(stdDev)
```

While this works very well on the Blue Gene, especially the sorting part of this algorithm presents problems for the GPU. Most sorting algorithms, including the quick-sort used here, contain branching (if-else statements). If two threads on the GPU are working on different input vectors and sorting them, and one thread follows the if-branch while the other goes into the else-branch, parallelism is lost.

Fortunately, there are more efficients ways of sorting on the GPU. We chose the Bitonic sorting algorithm [7], which may not be particularly effective in sequential application, but maps very well to SIMD architectures. It allows many threads to cooperatively sort an array together.

Below is a pseudo code sample showing the bitonic sorting algorithm:

```
bitonicSort(float[] values, int n)
  int tid = threadId //Each thread has its own ID

  for(int k = 2; k <= n; k *= 2)
    for(int j = k/2; j>0;j /= 2)
      int ixj = tid ^ j; //Bitwise XOR
      if (ixj > tid)
        if((tid & k) == 0) //Bitwise AND
          if(values[tid] > values[ixj])
            swap(values[tid], values[ixj]);
        else
          if(values[tid] < values[ixj])
            swap(values[tid], values[ixj]);
```

In this algorithm, each value in the array gets assigned to its own thread. Due to the clever picking of the index (ixj), two threads will never inspect the same values, which prevents read and write conflicts. This results in a very efficient method for sorting the array of values.

While this algorithm at first glance seems to contain a lot of divergence still, due to the number of if-statements, all threads that do not get through the first if-statement are deactivated for the rest of the sorting procedure, as there is no else-statement. As for the second statement, which does have an if-statement, please note that the actions taken within both branches are identical, save for a slightly different check. This check again either disables a thread or lets it through.

Once the values are sorted, determining the median is trivial. It is simply the n/2-th value in the samples array.

Calculating the standard deviation is slightly more complicated. Again, the method used in the Blue Gene architecture is ill-suited, as it uses only one thread for the entire array. As we have used a thread per single value in the previous method, this would leave the great majority of threads unutilized in the GPU, wasting precious processing power.

As a solution, we have opted to use a sum of squares approach for calculating the standard deviation. Variance can be calculated using the following formula:

$\sqrt{\frac{\sum(x-\bar{x})^2}{n}}$ which can be rewritten to $\sqrt{\frac{\sum(x^2)}{n} - \frac{(\sum(x))^2}{n}}$

This method is better suited for the SIMT architecture, as it can be done in two *reduce* operations. The reduce-algorithm will cooperatively sum all the values in the array. The first pass will then calculate $\sum x$. Then, each thread squares its own assigned value, and the reduce is ran again, resulting in $\sum(x^2)$. Each thread can then calculate the standard deviation by performing $\sqrt{\frac{\sum(x^2)}{n} - \frac{(\sum(x))^2}{n}}$.

The reduce operation is described in pseudo code below:

```
float reduce(float[] values, int n)

  int tid = threadId

  for(int s=n/2; s > 0; s = s/2)
    if(tid < s)
      values[tid] += values[tid + s]

  return values[0]
```

4.0.2 SumThreshold

The SumThreshold algorithm and accompanying pseudo code in section 3.2 are very close to the actual algorithm for both the Blue Gene and the GPUs. Again however due to the small-grained parallelism on the GPU, some small changes had to be supplemented to achieve the goal performance.

While in the Blue Gene version each node had access to an entire row or column from the input data, remember that in the GPU each value has its own thread. In the first iteration of the SumThresholding, this is trivial: each thread compares its own value against the threshold.

In the second iteration, the window size is doubled. This means that only half of the threads need to inspect their original value and that of their immediate neighbour. This is accomplished by only running threads with even thread ID's.

This process continues, and in each iteration the amount of running threads is halved, until the specified amount of iterations is achieved.

Below is the pseudo code for the adapted SumThreshold algorithm:

```
sumThreshold(float[] samples,
  boolean[] flags, int window,
  float threshold) {

  int tid = threadId;

  float sum = 0.0;
  if(tid % window == 0){
    for (int pos = tid; pos < tid + window; pos++)
    {
        if (!flags[pos]) {
          sum += samples[pos]
        }else{
          sum += threshold
        }
    }

    if (sum >= window * threshold) {
      for (pos = tid; pos < base + window; pos++)
      {
        flags[pos] = true
      }
    }
  }
}
```

The only difference with the original algorithm is the fact that the index of the value(s) to inspect is derived from the thread's ID, and that a thread will only execute the algorithm if its ID modulo the window size equals zero. This makes sure that no values are inspected by more than one thread at a time, eliminating read and write conflicts.

4.1 Further Optimizations

Other than the basic SumThreshold algorithm, our version makes use of several optimizations. These are the ability to do multiple runs, use Windsorized statistics, and the SIR Operator. These optimizations are explained in the following sections.

4.1.1 Multiple Runs

In the original algorithm, the entire flagging process was run more than once. After each iteration, the flagged values were removed, and the statistics recalculated. Because of the relatively large amounts of memory available to the nodes in the Blue Gene, nodes could simply make a working copy of the samples starting each iteration.

As mentioned in section 2.5.2, memory on the GPU can be classified into three different categories: global memory, shared memory and local memory.

When performing calculations on a GPU, a certain number of threads are launched. These threads are grouped into so-called blocks, and blocks are arranged into a grid.

Global memory is memory that is available to all threads. It is the largest memory store on the GPU, 6 Gigabytes on the GTX Titan used for testing, but accessing it is relatively expensive. When the SumThreshold algorithm starts, all the necessary input data is stored in the global memory, and the output data needs to be stored there as well if other parts of the pipeline want to access it.

Because of the large amounts of reads and writes to the input data, this is not a very efficient solution. Blocks of threads therefore have access to something called per-block shared memory. At 48 Kilobytes per block, this memory is smaller than the global memory, but access is much faster.

Finally, there is per-thread local memory. Access to this memory is the fastest, but there is only a small amount (256 bytes) of it available.

In the SumThreshold algorithm therefore, each row or column of the input data to be inspected is assigned to a block of threads, containing as many threads as there are values in the input data. The first operation taken by the threads responsible is copying the input values from global to shared memory. The original data stays unchanged within in memory. This means that each block of threads has a local copy of the values which they can rearrange and alter, and a backup of the original values in the global memory. This global backup can be accessed whenever the original values in their original order were needed.

To be able to run multiple iterations of the entire algorithm while removing previously flagged values, the GPU version sets any of the flagged values to zero in shared memory. As the algorithm begins with a sort, these values will always be moved to the front of the input vector.

After each iteration, the number of flagged values is counted collectively. At the start of a new iteration, after the values get sorted, all threads with an ID lower than the amount of flagged values stop executing. This makes sure that any previously flagged values are not taken into account in the next iteration.

4.1.2 Windsorized statistics

A further optimization concerns using Winsorized statistics. Within statistical, real-world data one is likely to encounter outliers. These outliers tend to distort statistical measures like means and standard deviations.

Radio-astronomical data is no exception to this rule. To improve the accuracy of the standard deviation used for setting the threshold in the SumThreshold algorithm, Windsorized statistics are used.

Winsorizing is a process in which the most extreme values in a data set are not taken into account when calculating other statistics. In our case, we chose to exclude the top and bottom 10% of values. This leads to a more accurate standard deviation and mean, which in turn improves the overall quality of the algorithm.

In the Blue Gene version of the algorithm, a so called $high - value$ and $low - value$ were computed. These values correspond to the borders of the middle 80% window: any value above or below them would not be taken into account. This was achieved by sorting the data, looking up the $n \times 0.1$ and $n \times 0.9$-th value in the vector, and replacing any value above or below them with this value.

The GPU version uses a similar approach. After sorting, any thread with an ID lower or higher than the bottom or top 10% of unflagged values would replace its value with the value on the border. This results in a data set with a much smaller amount of outliers, improving the statistics.

As soon as the actual SumThreshold algorithm starts however, the values in the per-block shared memory are restored from global memory, making sure that the algorithm deals with the real data instead of the altered, sorted data.

4.1.3 Directions

Up until this point, we have only discussed running the SumThreshold algorithm in only one direction. To clarify briefly, the SumThreshold algorithm compares neighbouring samples to a threshold based on the median of a row or column. Whether we are looking at rows or columns determines the direction of the algorithm.

Recall the shape of the input data: a matrix, with time on the x-axis and frequencies on the y-axis. There are three options when it comes to flagging this data with the SumThreshold algorithm: we can scan in the frequency direction, taking the median of the rows, or in the time direction, iterating over the columns, or we could do both.

Running over the data in the frequency direction is more likely to detect long narrow-band RFI. Running over the time direction however, is much better suited for detecting short broadband bursts. Doing both will likely result in the greatest accuracy, but will also increase the runtime of the algorithm substantially.

All three tactics were implemented and evaluated. The results can be found in Chapter 5.

4.1.4 Reducing the data

Another optimization comes from reducing the data before flagging with SumThreshold. Instead of inspecting each sample individually, a number of samples, either in the time or frequency direction are summed together. Then, if the flagging algorithm flags this meta-sample, all samples of which it was originally composed of are flagged in the final flag map.

This optimization, which can easily be performed in parallel using the reduce operation mentioned in section 4.0.1, is likely to speed up performance considerably, as the number of samples to be scanned is greatly reduced. Accuracy however may suffer, as whole blocks of samples are flagged at once, instead of flagged single samples.

The trade-off between accuracy and speed for this optimization is further examined in chapter 5.

4.1.5 SIR Operator

After flagging using the SumThreshold algorithm is completed, another algorithm may be used to further increase the quality of the results. This algorithm, known as the Scale Invariant Rank operator, or SIR, was published first by A. Offringa in [8].

This algorithm makes use of the assumption that interference often creates straight lines within the final data, meaning it is either concentrated within a small sub-band, or a small but broad-band burst in time. While this may not always be the case theoretically, it does conform to LOFAR measurements.

The SIR operator will scan a row or column of the flag map after flagging, first from left to right and then from right to left. With each positive flag encountered, a variable called *credit* is increased. With each negative flag, it is decreased. Then, based on a fixed value set in advance, unflagged values might be flipped if enough previous values have increased the *credit* variable.

A pseudo-code version of the SIR operator is given below:

```
void sir_operator(char[] flags, int n){
  float credit = 0.0
  float w
  float max_credit
  for(int i = 0; i < n;i++){
    w = flags[i] ? SIR_VALUE : SIR_VALUE - 1.0
    max_credit = credit > 0.0 ? credit : 0.0
    credit = max_credit + w
    flags[i] = credit >= 0.0
  }
  credit = 0;
  for(int i = n-1; i > 0;i--){
    w = flags[i] ? SIR_VALUE : SIR_VALUE - 1.0
    max_credit = credit > 0.0 ? credit : 0.0
    credit = max_credit + w
    flags[i] = credit >= 0.0 | flags[i]
  }
}
```

In our GPU adaptation, this algorithm is implemented by a different kernel than the SumThreshold algorithm. This allows us to reconfigure the number of blocks and threads. In this case, as there is a linear dependency it is impossible (or at least impractical) to assign a thread to each value. We therefore chose to allocate a thread for each row or column to be processed. While the SIR operator is likely to increase the accuracy of the algorithm, it also obviously increases the runtime. The final results for both accuracy and performance can be found in chapter 4.

4.2 Position in the LOFAR pipeline

After data is collected at the individual nodes, it is sent to the central LOFAR processing pipeline. The flagger described in the previous sections needs to be placed somewhere in this pipeline. The precise location, or the order in which operations in the pipeline take place, is of importance.

Generally speaking, the data collected at the nodes is transferred to the central processing pipeline. It is first filtered and correlated. Then, it continues into either the imaging pipeline, or the beam-former, depending on the desired output format.

In the imaging pipeline, the data is correlated and integrated. Finally, it is stored to disk. In the beam-former, the data is beam-formed and then stored to disk.

Throughout the processing pipeline, the dimensions of the data are subject to change, as different operations reduce the data. However, the general shape of the data, a matrix mapping time to frequency, stays the same.

As a result of this, the SumThreshold algorithm can be applied at any stage within the LOFAR pipeline. Different locations result in different accuracies, as flagging unreduced data is likely to yield more accurate results at the expense of more processing time.

Chapter 5

Results

This chapter details the results obtained with the ported SumThreshold algorithm. It is divided into two sections: one detailing the performance of the algorithm with regards to runtime, and one assessing the quality of the algorithm.

The runtime performance of the algorithm is important due to the strict time restraints of running the flagging algorithm on-line, i.e. processing data as quickly as it is generated. This requirement poses a very simple goal to be met: a second of recorded data should take less than a second to process.

The quality of the algorithm is of obvious importance as well. Writing a fast algorithm that does not effectively filter out RFI is of very little practical use.

5.1 Runtime performance

In order to accurately assess the runtime of the algorithm, the algorithm in all its different configurations is run on randomly generated data. As the results are not important, random data is easier to generate in the various sizes required for performance testing.

Also note that for our measurements the height and width of the input matrix are of little importance. The total number of values is what actually matters in this case. This is due to how the LOFAR transforms data gathered at the receiver: each receiver records an analogue signal, which is transformed into the matrix to be processed by a Fourier transform. The parameters of this transform determine the height and width of the matrix generated, but are a trade-off: a higher time resolution results in a lower frequency resolution. The total number of samples stays constant.

The algorithm was run in several configurations, each detailed in the subsections below. The same configurations were used for testing the quality of the algorithm, allowing us to make a cost-benefit assessment for these different settings.

All runs were performed on an Nvidia GTX Titan GPU. Only the kernel runtime was measured, as copying data to and from the GPU will not occur during the processing in the LOFAR pipeline. This is based on the assumption that flagging is not the first operation within the pipeline, meaning previous operations have already transferred the data into the GPU's global memory.

5.1.1 Frequency direction, without SIR

In this section we will look at the performance of the default algorithm, which scans in the frequency direction but does not use the SIR operator.

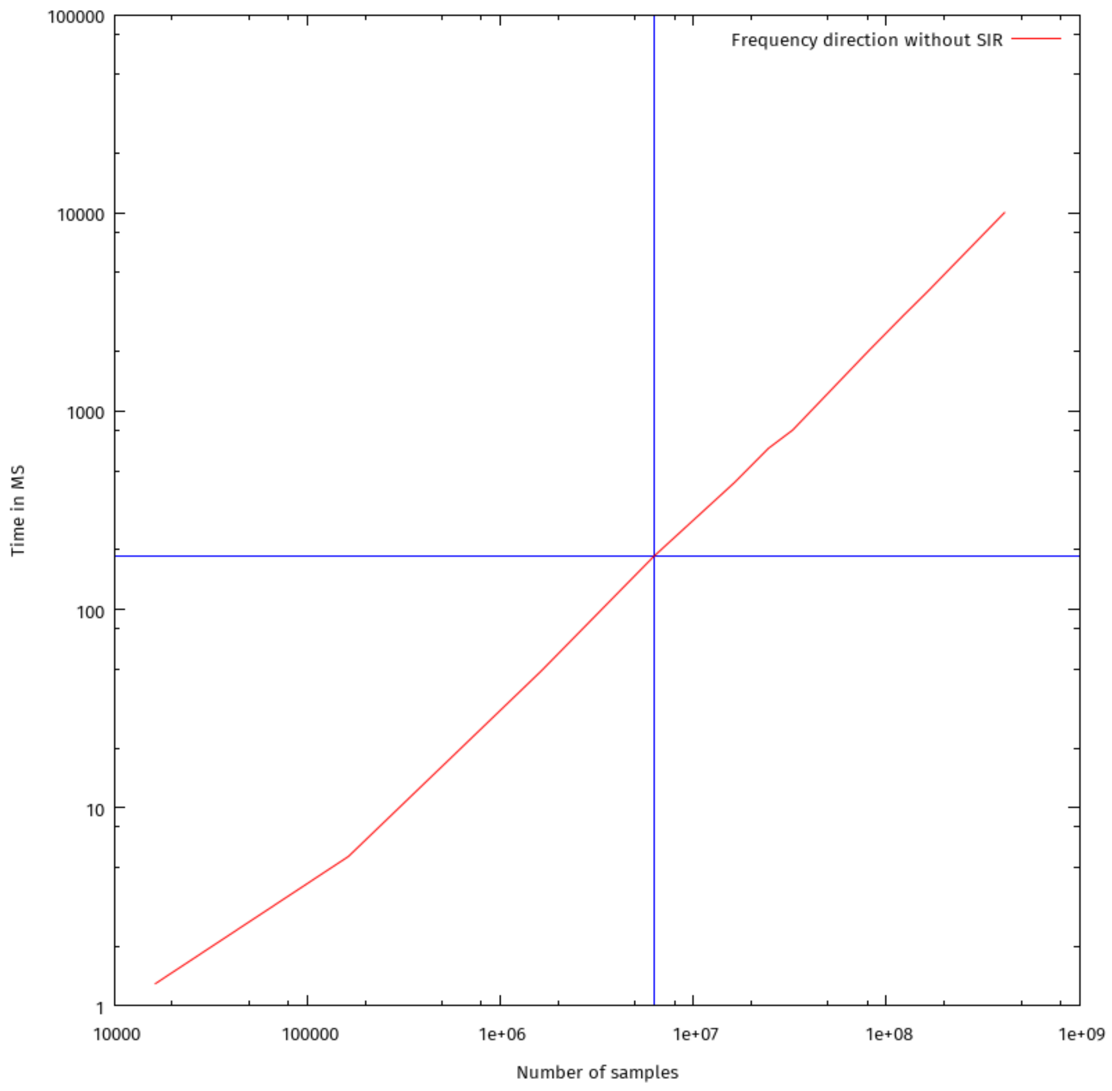

Figure 5.1: Performance of the SumThreshold algorithm. The blue lines mark the number of samples contained in a second of collected data.

Figure 5.1 immediately answers the research question pertaining to on-line performance. Recall that in order for the algorithm to be able to run on-line, i.e. process data as quickly as it is generated, it needs to be able to flag the amount of data generated in a second in less than a second.

To illustrate this point, the blue lines are added to the graph. These lines show the number of samples generated in a single second. The horizontal blue line marks the amount of time required to process this second of data, consisting of 2 polarizations, 32 subbands and 256 channels, and measuring about 380 samples per second.

As can be read from the graph, the algorithm scanning in the frequency direction without using the SIR operator or a reduction of the data can process a second of generated data in 185 milliseconds, i.e. about 5.4 times faster than real time. This is well within the bounds of our on-line requirements.

As can be seen in figure 5.1, regardless of the input size, the algorithm continues to perform in a linear fashion. This is in line with our expectations, as all of the operations in the algorithm, except for sorting the values, have linear computational complexity.

5.1.2 Frequency direction, with SIR

As discussed in section 4.1.5, the SIR operator is likely to increase the accuracy of the flagger, at the cost of a slightly higher runtime. The following figures show the exact increase in runtime.

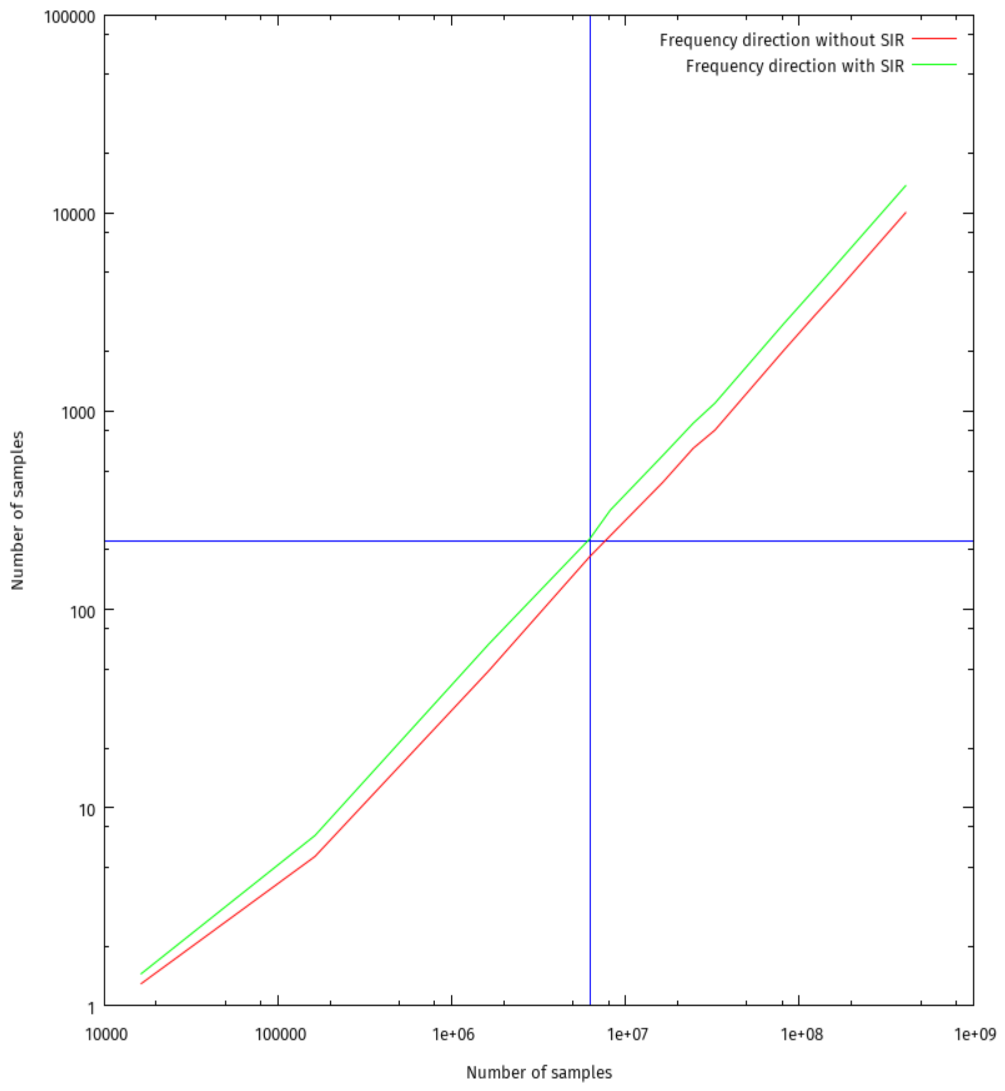

FIGURE 5.2: Comparison of performance with and without SIR

As can be seen in figure 5.2, adding the SIR operator does increase the runtime slightly. However, the algorithm is still sufficiently fast for on-line processing and still scales linearly.

5.1.3 Frequency direction with SIR after reduction

As explained in chapter 3, a possible optimization of the SumThreshold algorithm, especially in regards to runtime would be to reduce the data before flagging. This entails summing together values in either the time or the frequency direction. This is a trade-off however, as reducing the data will result in coarser flagging which will affect the quality of the flagging.

In the following figure we display the runtime of the SumThreshold algorithm after each frequency channel has been reduced to a single value, compared to the regular algorithm and the regular algorithm plus the SIR operator. Note that the reduced algorithm also includes the SIR operator.

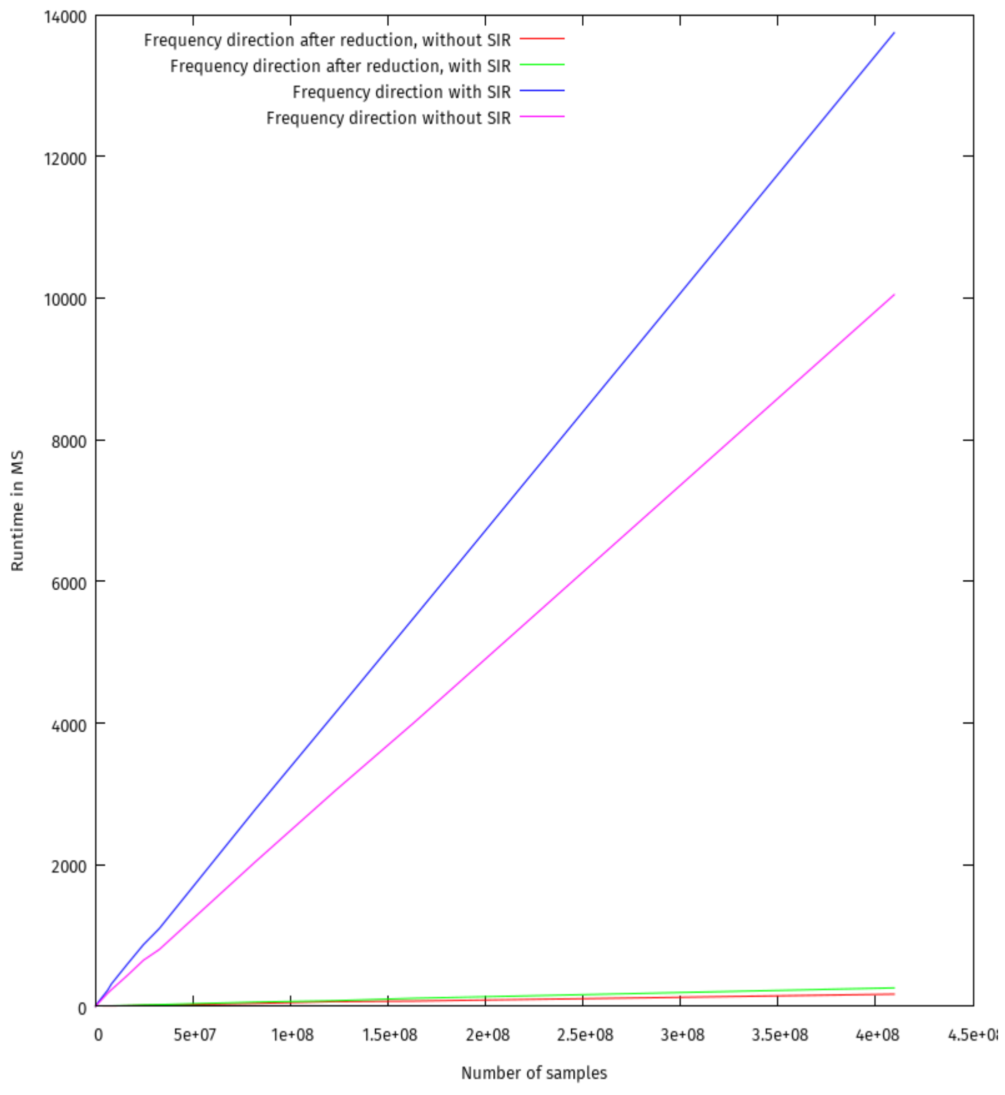

FIGURE 5.3: Comparison of performance of different configurations of the SumThreshold algorithm in the frequency direction.

Looking at the time values in figure 5.3 we can see that the runtime of this version of the algorithm is much lower that that of the original algorithm. Note that this figure uses a linear scale as opposed to the log scales used in figures 5.1 and 5.2, to emphasize the enormous differences in runtime.

The reduced version was able to process a second of data in around 5 milliseconds, making it 200 times faster than real time. This would allow a single GPU to process the input of 200 stations sampling 380 times per second on 2 polarizations, with 32 subbands of 256 channels.

This version of the algorithm removes any doubt whether or not the GPU version would be efficient enough for on-line processing of the LOFAR data. The only question that remains is whether or not the quality of flagging is sufficient for actual use. The results of quality assessments of the different versions of the algorithm are presented in the next section.

5.1.4 Time direction

As mentioned in section 4.1.3, the algorithm can be performed both in the time and the frequency direction. Both options were implemented while porting the algorithm. Due to the nature of the GPU however, there are slight differences in runtime when comparing scanning in the frequency with scanning in the time direction. This is caused by the data structure in memory: neighbouring samples in memory correspond to neighbouring sub-frequencies. Recall that each sample of a row or column to be inspected was given a separate thread. Each thread would copy its sample to shared block memory first.

When scanning in the frequency direction, threads in a block all access adjacent samples. The GPU is optimized for this type of memory access, also known as coalesced memory access.

When comparing neighbouring samples in the time direction however, threads will have to access memory values offset by the number of sub-frequencies per time slot. This means that copying the memory to inspect to shared block memory will be slower, affecting the overall runtime.

Note that just like in the frequency direction the amount of samples can be reduced before flagging.

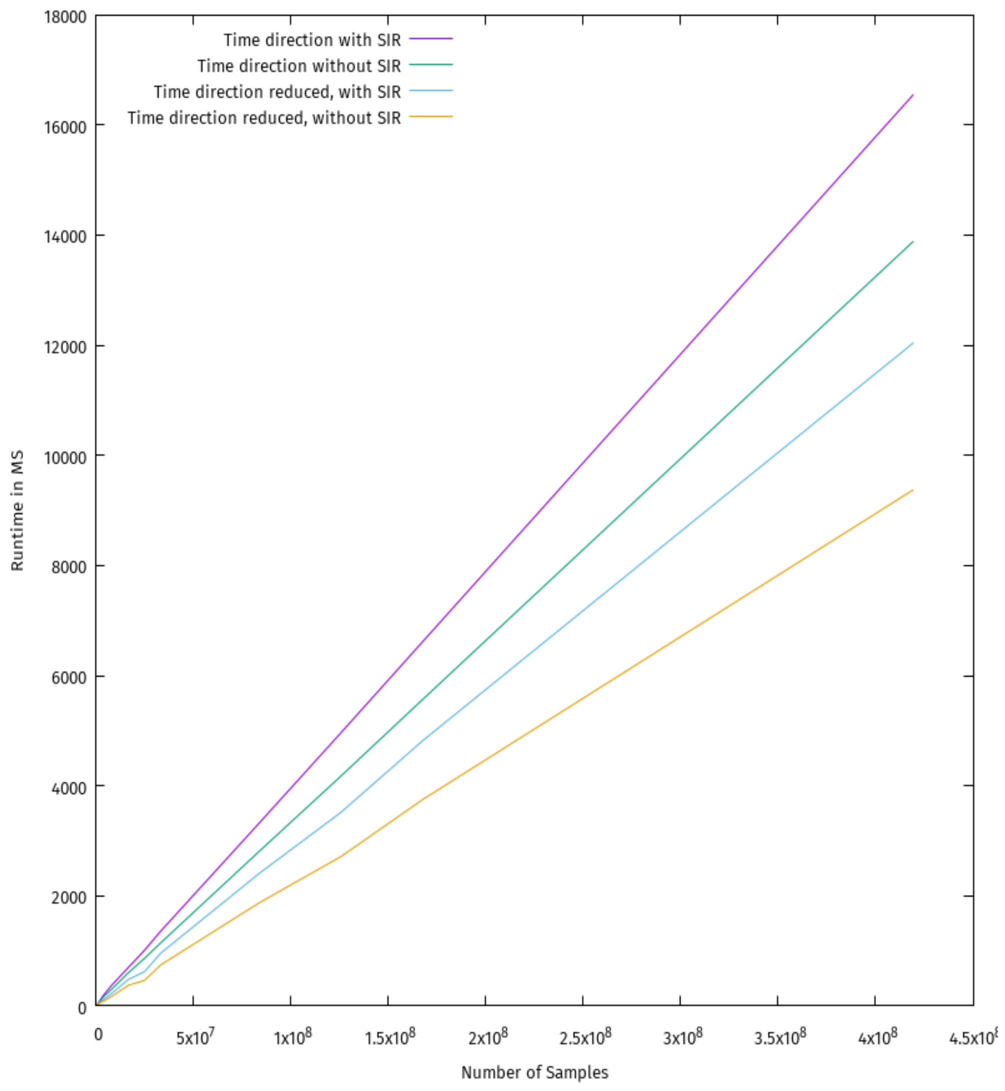

FIGURE 5.4: Comparison of performance of different configurations of the SumThreshold algorithm, in the time direction

Figure 5.4 shows a comparison of the different runtimes for different configurations of scanning in the time direction. Note that all versions of the algorithm are still within the limits of real time processing.

Reducing in this instance yields much smaller performance gains than reducing in the frequency direction. This can be explained by the fact there is much less room for reduction in the time direction: while each timeslice contained 2 polarizations of 32 channels with 256 subbands each, the total number of time slices (especially when considering a lower number of samples) is much smaller. Because of this, the amount of data that can be reduced in the time direction is necessarily also much smaller.

5.1.5 Time and Frequency direction

As a final option, flagging in both the time and frequency direction can be done. This should result in the greatest accuracy when flagging, at the cost of a higher runtime.

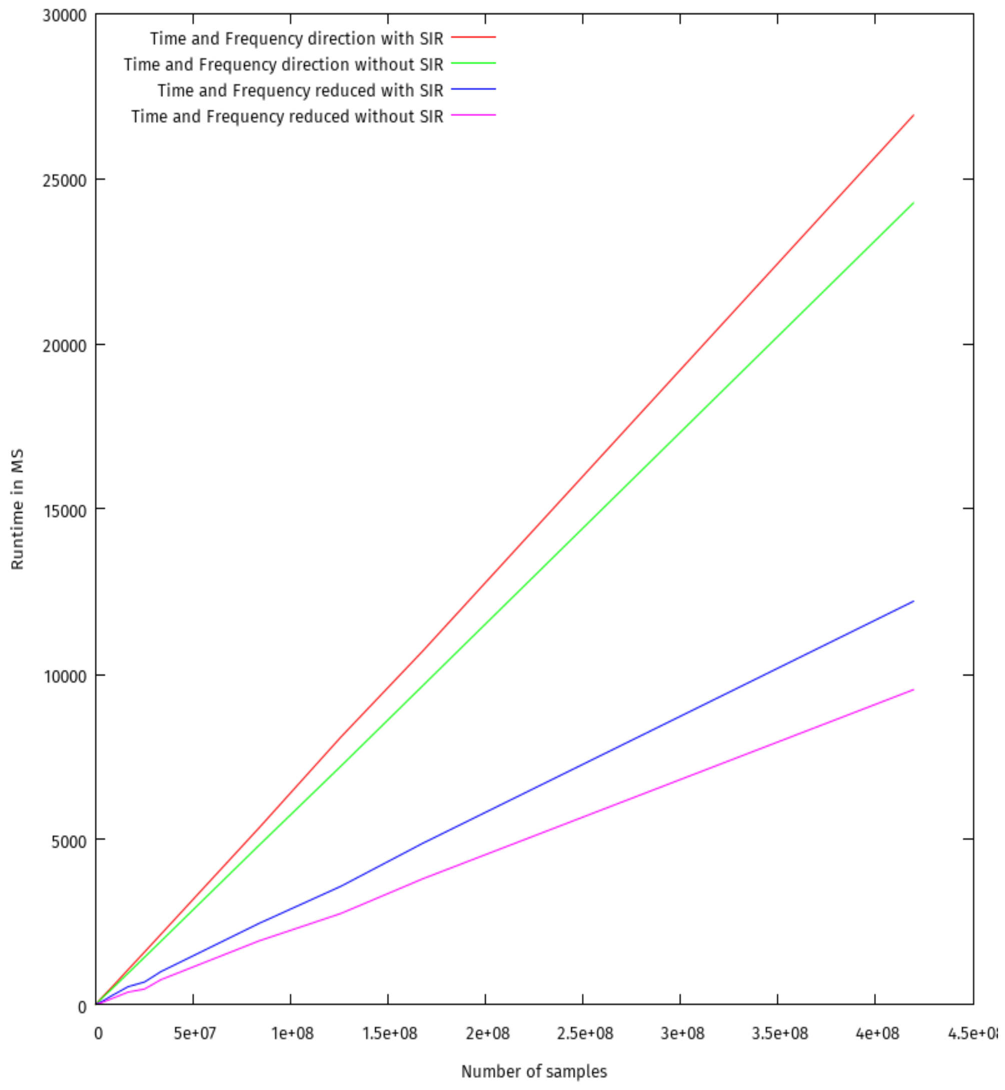

FIGURE 5.5: Comparison of performance of different configurations of the SumThreshold algorithm, in the time and frequency direction

Figure 5.5 shows a comparison of all possible options when running in both the time and frequency direction. Note that again, all versions are within the confines of online processing.

5.2 Algorithm quality

In this section we present the measurements and results concerning the quality of the flagging algorithm. The quality of a flagging algorithm is measured simply by seeing if the samples flagged were indeed caused by RFI.

To achieve this, we have run the algorithm on a real dataset, which is known to be fairly heavily contaminated with all kinds of RFI. This dataset also contains measurements for a well-known pulsar. Pulsars, as is implied in the name, are stars that send out pulses of radio waves. [9]

Because we know the period and intensity of the pulsar contained in the measurements, we can compare the output of the flagging algorithm with this known pulsar. We can then calculate the Signal-to-Noise ratio of the output, which is the ratio between the peaks and the baseline of the measurements.

Generally speaking, the higher the Signal-to-Noise ratio, the better. In order for the results to be considered statistically significant however, a ratio of 5 or greater is necessary. This is based on Chebychev's inequality, which is:
$Pr(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$
The SNR is the distance of the maximum value to the mean of the values of the distribution. Chebychev's inequality helps us to figure out the probability whether this distance is a random fluctuation on noise, or is actually part of the distribution. At an SNR of 5, this probability dips below the statistical line of significance of 5%.

Table 5.1 shows the SNR and the percentage of flagged values for different configurations of the SumThreshold algorithm.

Algorithm Configuration	SNR	% flagged
No flagging	1.08	0.0
Frequency direction without SIR	5.41	6.72
Frequency direction with SIR	5.72	11.71
Time direction without SIR	0.86	6.96
Time direction with SIR	1.07	8.81
Frequency then Time direction without SIR	3.47	11.87
Time then Frequency direction without SIR	3.72	11.62
Frequency then Time direction with SIR	3.47	18.79
Time then Frequency direction with SIR	3.35	19.61
Frequency direction reduced, without SIR	5.59	5.34
Frequency direction reduced, with SIR	5.59	5.55
Time direction reduced, without SIR	2.82	0.22
Time direction reduced, with SIR	3.49	0.43
Frequency then Time direction reduced, without SIR	4.36	17.01
Time then Frequency direction reduced, without SIR	5.28	5.65
Frequency then Time direction reduced, with SIR	5.63	8.78
Time then Frequency direction reduced, with SIR	5.68	5.70

Table 5.1: Comparison of the SNR and percentage of values flagged for different settings of the SumThreshold algorithm

As can be seen from table 5.1, flagging in the frequency direction is much more successful in our case than flagging in the time direction. This is likely due to the specific manifestation of RFI in our dataset. We can also see that combining scanning in the time and frequency direction, regardless of order, is not a successful strategy. This is likely caused by too much flagging, which accidentally flags real data as well.

Surprisingly, flagging in the frequency direction after reducing the data performs remarkably well. It flags only half the amount samples compared to the regular version, but the SNR decreases with not much more than 0.1. Results for the time direction after reduction are markedly better than flagging without reducing. This can be attributed to the fact that without reduction, the vector inspected contains too few values. In this case, RFI contaminating the complete vector will throw off the threshold too much, allowing RFI to pass through. In general reducing increases sensitivity, as areas with high amounts of RFI will stick out more. It is a trade off however, as flagging reduced data will be less fine-grained.

Below we present figures plotting the different pulse profiles with regards to the pulsar after flagging the data. The period of the graphs is the exact period of the pulsar. This means that in the optimal case, a single peak is shown. The power of the profile is influenced by the flagging: the less values flagged, the higher the power. A higher power is positive if the pulse profile is correct, as it means the RFI was identified correctly, and only those values were flagged. A lower power is preferable however if it means more RFI was removed. The SNRs in table 5.1 correspond to the height of the peak compared to the baseline.

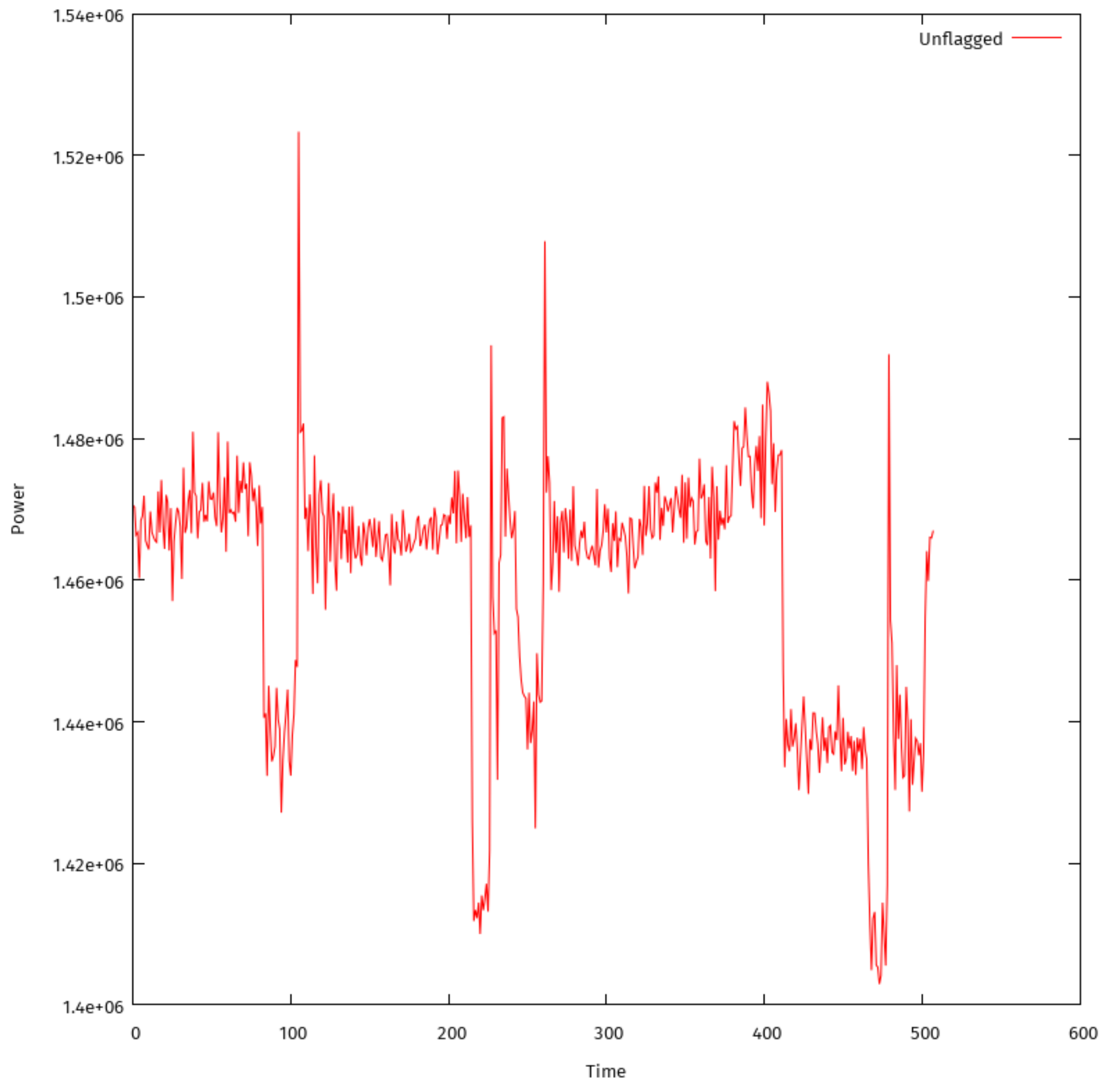

FIGURE 5.6: Pulse profile of the dataset without flagging

Figure 5.6 shows the pulse profile of the data set before flagging. This is clearly not what we are after: the pulse contains many peaks and dips, while we are expecting only a single pulse.

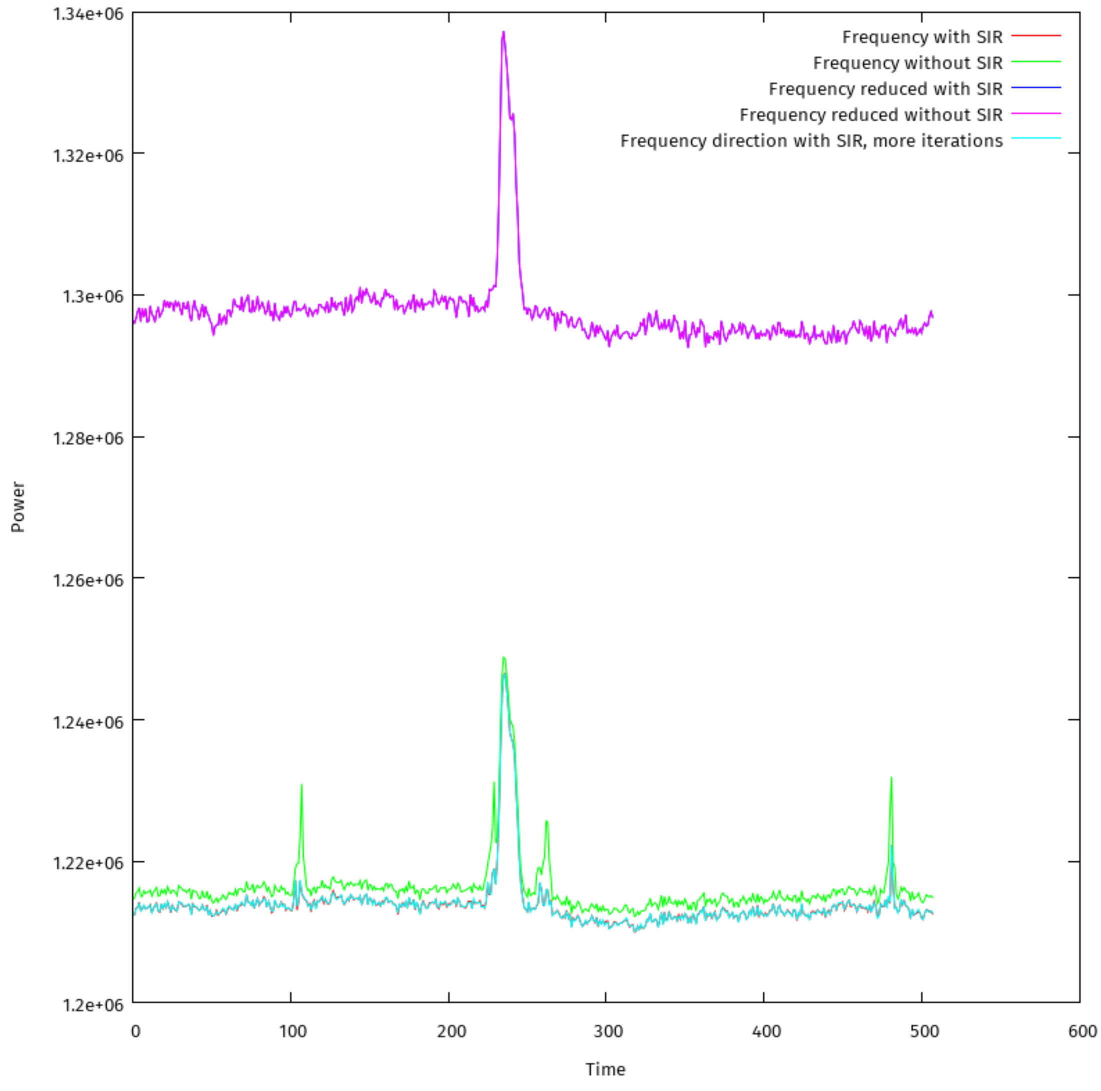

FIGURE 5.7: Pulse profiles of the different SumThreshold versions in the frequency direction

Figure 5.7 contains the pulse profiles for different settings in the frequency direction. Note that the algorithm without SIR (the green line) leaves several peaks caused by RFI. While the SNR on its own might indicate that the quality is satisfactory, the pulse profile invalidates this assumption. Applying the SIR operator after running the algorithm (the red line) in the frequency direction reduces the RFI peaks, but still leaves them, as can be seen by the small peaks still present around t=100 and t=500.

This is slightly difficult to see, as the red line is almost completely masked by the light blue line. This line corresponds to the same algorithm, but with more iterations in the SumThreshold portion (7 instead of 5). It is clear from this image that running more iterations offers no improvements on the quality.

The reduced version in the frequency direction however shows the best pulse profile: the pulsar is clearly visible, and contains no false positives. It also has the highest power. Note here that running the SIR operator after reducing in the frequency direction seems to have no effect: the purple line almost perfectly overlaps the dark blue one.

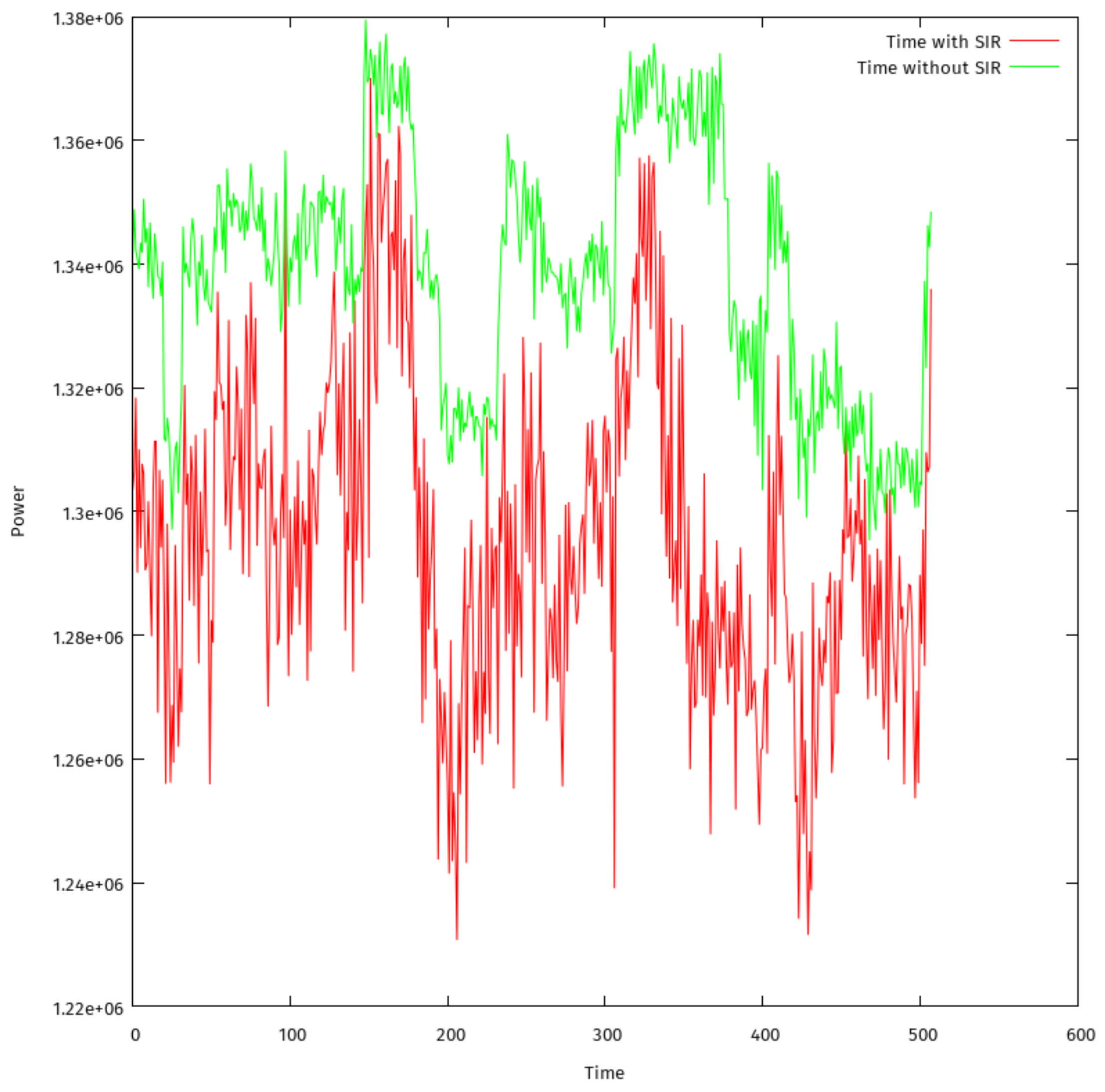

FIGURE 5.8: Pulse profiles of the different SumThreshold versions in the time direction

Running in the time direction shows that this is clearly not effective for eliminating the types of RFI encountered in this data set. This is no surprise after inspecting the SNR. The reduced pulse profiles were plot separately, as the regular pulse profiles obscured a clear view of the reduced ones.

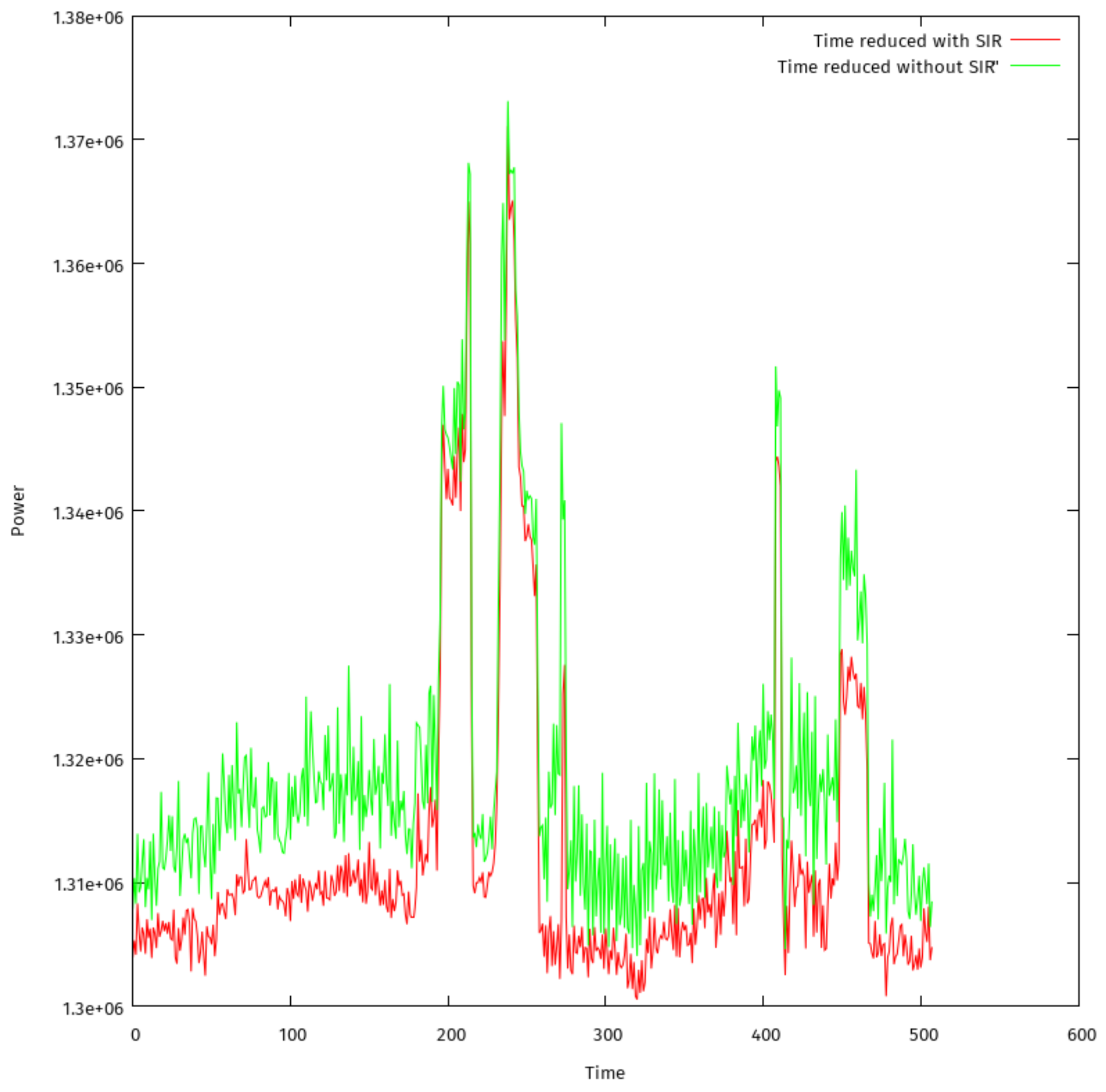

FIGURE 5.9: Pulse profiles of the different SumThreshold versions in the time direction, after reduction

While the reduced pulse profiles show a much more promising picture than the regular versions, it is clear that they are nowhere near as effective as those generated by scanning in the frequency direction.

As mentioned before, this is caused by two things. The dataset is likely tainted mostly with long narrow-band RFI as opposed to short broadband bursts. Scanning in the frequency direction is much more effective for eliminating this kind of RFI. Secondly, the number of values compared without reduction is likely too small, allowing entire vectors to be tainted throwing off the threshold. This explains why a reduction, which allows for comparison of a larger number of values, performs better.

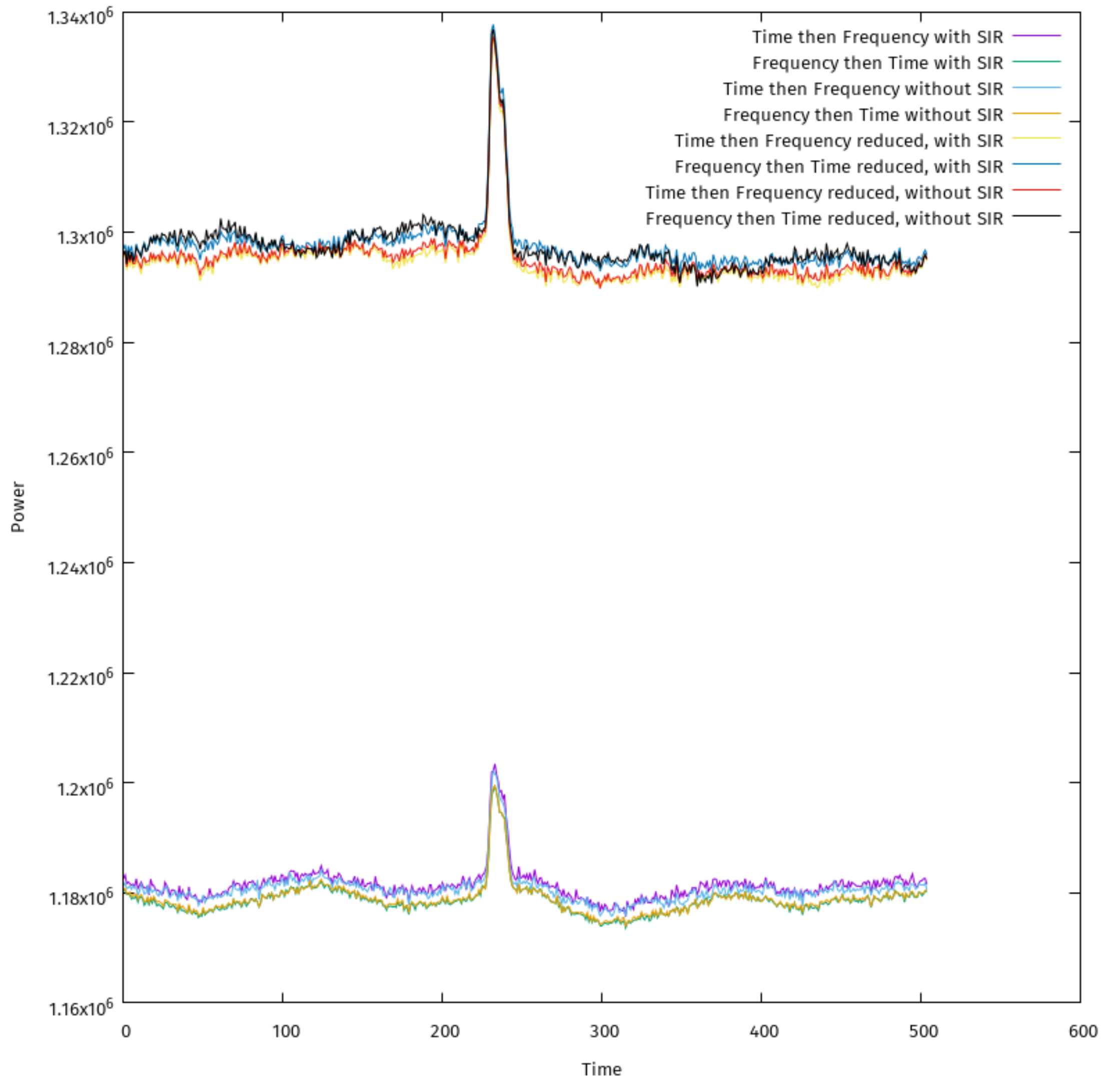

FIGURE 5.10: Pulse profiles of the different SumThreshold versions in the time and frequency direction

Figure 5.10 shows the pulse profiles for scanning in both the time and frequency direction, in either order. This is an obvious improvement to scanning in the time direction only. Without reduction however, the signal-to-noise ratio suffers, resulting in a very small peak (the lower four lines). In this case as well, the SIR operator seems to make very little difference. This can be attributed to the fact that combining both flaggers causes them to flag too many samples, some of which are not actually RFI.

The reduced versions perform better. When compared to the reduced version of just the frequency direction however, the quality is very similar.

5.2.1 Quality Comparison with the Blue Gene Implementation

Another good test to see whether the quality of this algorithm is sufficient is to compare it to the SNR achieved by the previous Blue Gene algorithm. A straight comparison was however impossible: the only datasets available for comparison were flagged, and then beamformed. Because the beamforming software was not available to us, we chose to flag a sample set that had already been beamformed. This will result in a small difference in accuracy, as there is more data available before beamforming. This is because the beamformer uses the sum of the values of all stations, with as consequence that if a single stations contains RFI, it will be present in all data after summation. Flagging prior to this summation will therefore likely result in a much higher accuracy.

The dataset flagged by the Blue Gene algorithm managed to reach an SNR of 6.99. The difference between this result and ours is not negligible, we believe it to be the result of the ordering of the flagging and beamforming. Actual tests within the LOFAR pipeline need to be done to determine whether this is the case or not.

Besides the placement within the pipeline the previous Blue Gene/P implementation also made use of a so-called history flagger. This feature adjusted the threshold based on values found in previous iterations. This allows the flagger to adapt quickly to changing baselines in the data, resulting in a more accurate result.

The same history flagger could arguably be implemented in the CUDA version of the SumThresholding algorithm. This is something that could be examined in a future version.

Chapter 6

Discussion

The aim of this research was to determine whether the flagging algorithm currently in use in the LOFAR could be ported to GPUs, while retaining sufficient performance for on-line processing and be of acceptable quality.

The measurements performed on algorithm after successful porting show that this is indeed possible. All of the different versions of the algorithm were able to process generated data in significantly less time than required to collect the data. Reducing the input data by summing certain frequency sub-bands especially delivered an extremely high performing method, allowing a single GPU to process 200 stations in real time.

The quality of most of the settings of the algorithm were sufficient as well. While not all configurations reached a statistically significant Signal-to-Noise ratio, flagging in the frequency direction did. Also, while there is a difference in quality between the old Blue Gene algorithm and the new GPU version, this is likely caused by the unfortunate placement of the flagger within the pipeline and the inability to make a straight comparison between the two methods.

Moreover, the Blue Gene/P version implemented a history flagger, which allowed the threshold to be updated based on values found in previous iterations, allowing for a continuously improving flagger. This is something that could likely be implemented in the CUDA version of the SumThreshold algorithm as well. This needs to be researched further in future versions.

Chapter 7

Conclusions

In this thesis we investigated the porting of the SumThreshold algorithm for the detection of RFI in radio astronomy from the IBM Blue Gene/P to GPUs. Furthermore, we researched the performance and quality of the ported algorithm, assuring that the detection could be performed in real time, and that the quality was acceptable.

We conclude that the porting is indeed possible. Furthermore, the quality of several of the different variations of the algorithm yielded acceptable results. The algorithm also met its criterion of on line performance, in some cases by over 200 times.

We believe that this result has large implications, both for current generation radio telescopes such as LOFAR, and those yet to be realized like the SKA. GPUs provide an affordable and easily obtainable alternative to large supercomputers like the IBM Blue Gene/P.

The quality of the algorithm discussed here needs further research: as unprocessesed datasets were unavailable to us during our research, it is not yet clear whether the GPU version of the SumThreshold algorithm is actually capable of matching the Blue Gene/P's version. While the quality was certainly acceptable, a straight comparison would be desirable.

Furthermore, testing the algorithm's quality on different data sets would yield more information on whether flagging in the time direction has its uses. In the data available to us the time direction flagger performed far worse than the frequency direction. It is unclear whether this is caused by the input data or something else entirely.

Bibliography

[1] P.A. Fridman and W.A. Baan. Rfi mitigation methods in radio astronomy. *Astronomy & Astrophysics*, 378:327–344, 2001.

[2] Cuda c programming guide, 2014.

[3] L. Schoemaker. A taxonomy of rfi mitigation and excision algorithms in radio astronomy, 2014. URL http://clipperhq.com/thesis/literature_study2.pdf.

[4] A.R. Offringa, A.G. de Bruyn, M. Biehl, S. Zaroubi, G. Bernardi, and V.N. Pandey. Post-correlation radio frequency interference classification method. *Monthly Notices of the Royal Astronomical Society*, 405:155–167, 2010.

[5] I.V. Nikiforov M. Basseville. *Detection of Abrupt Changes: Theory and Application*. Prentice Hall, Inc, 1993.

[6] Iso/iec 9899:1999 specification, tc3, 2007.

[7] K.E. Batcher. Sorting networks and their applications. *Proceeding AFIPS '68 (Spring) Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[8] J.B.T.M. Roerdink A. Offringa, J.J. van de Gronde. A morphological algorithm for improving radio-frequency interference detection. *Astronomy & Astrophysics*, 539, 2012.

[9] J.H. Taylor R.N. Manchester. *Pulsars*. W.H. Freeman and Company, San Francisco, 1977.