Master's Thesis: Accelerating radio transient detection using the Bispectrum algorithm and GPGPU

Tsu-Shiuan Lin
Supervisor: James Gain
Co-Supervisor: Richard Armstrong

Department of Computer Science
University of Cape Town

October 3, 2015

Acknowledgements

A very big thank you to both my supervisors, Prof. James Gain and Dr. Richard Armstrong for all your inspirational talks, guidance and hard work throughout these two years. All your timeously suggestions and enthusiasm, particularly during the last 2 weeks to allow me to finish on time, are very much appreciated. Thanks to my parents for their continued support during this time. Also, a thank you to SKA for funding this Masters. Lastly, thank you to all my colleagues in the lab for the interesting conversations, advice and technical support.

A final, but certainly not least thank you to my best friend Nikasha Armoogam for always being there to push me, motivate me, inspire me and having faith in me to finish my masters.

Contents

Abstract

Modern radio interferometers such as those in the Square Kilometre Array (SKA) project are powerful tools to discover completely new classes of astronomical phenomena. Amongst these phenomena are radio transients. Transients are bursts of electromagnetic radiation and is an exciting area of research as localizing pulsars (transient emitters) allow physicists to test and formulate theories on strong gravitational forces. Current methods for detecting transients requires an image of the sky to be produced at every time step. Since interferometers have more information available to them, the computational demands for producing images becomes infeasible due to the larger data sets provided by larger interferometers. Law and Bower (2012) formulated a different approach by using a closure quantity known as the "bispectrum": the product of visibilities around a closed loop of antennae. The proposed algorithm has been shown to be easily parallelized and suitable for Graphics processing units (GPUs).

Recent advancements in the field of many core technology such as GPUs has demonstrated significant performance enhancements to many scientific applications. A GPU implementation of the bispectrum algorithm has yet to be explored. In this thesis, we present a number of modified implementations of the bispectrum algorithm, allowing both instruction-level and data-level parallelism. Firstly, a multi-threaded CPU version is developed in C++ using OpenMP and then compared to a GPU version developed using Compute Unified Device Architecture (CUDA).

In order to verify validity of the implementations presented, the implementations were firstly run on simulated data created from MeqTrees: a tool for simulating transients developed by the SKA. Thereafter, data from the Karl Jansky Very Large Array (JVLA) containing the B0355+54 pulsar was used to test the implementation on real data.

This research concludes that the bispectrum algorithm is well suited for both CPU and GPU implementations as we achieved a 3.2x speed up on a 4-core multi-threaded CPU implementation over a single thread implementation. The GPU implementation on a GTX670, achieved about a 20 times speed-up over the multi-threaded CPU implementation. These results show that the bispectrum algorithm will open doors to a series of efficient transient surveys suitable for modern data-intensive radio interferometers.

Chapter 1

Introduction

Transients are bursts of electromagnetic radiation which are associated with explosive or dynamic events across the universe. Localizing the sources of these transients provide enormous potential to uncover a wide range of new astrophysics. In this thesis, an algorithm used to detect radio transients, called the bispectrum algorithm, is implemented on two processing architectures. In particular, an optimised multi-threaded *Central Processing Unit* (CPU) implementation is compared to an implementation using a *Graphics Processing Unit* (GPU). This is used to demonstrate potential performance enhancements in the area of astronomical computations, specifically for transient detection software with interferometers.

This chapter serves to define the context and motivation behind a GPU accelerated implementation of the bispectrum algorithm. Several problems that arise with new generation interferometers are described and the relevance of high performance computing in solving these problems is highlighted. These factors lead to several research objectives that are explored throughout this work.

1.1 Motivation: Radio astronomy

Recent advancements in the field of multi-core and many-core architectures, particularly Graphics Processing Units (GPUs), allow many scientific applications to benefit from high performance computing. Amongst these applications, the *Square Kilometre Array* (SKA)[1] project requires the processing and storage of very large amounts of data collected by radio interferometers [SKA14]. Each radio dish produces data at an estimated rate of 60GB/s, implying the instantaneous data rates of the SKA will be ∼100TB/s [SKA14]. The problem is further magnified if post-image formation of the data is taken into account. Current implementations of sequential programs are no longer able to analyse such large datasets in real-time. Without the aid of well tuned software that utilizes parallelism, together with new hardware architectures, data management and analysis in a real-time context is unattainable for the SKA project.

Radio telescopes generate many times more data than optical telescopes due to the wide and deep fields of the sky probed, particularly with the spatial and frequency resolution they are able to achieve [Kok13]. Although the resolution achievable by a single-dish telescope is determined by the diameter of its effective aperture, radio interferometric techniques allow a collection of smaller dishes (an interferometer) to synthesize much larger apertures by combining the signals from each dish. These interferometers are able to view large fields of view and survey more efficiently than a large single-dish telescope at the cost of higher computational and signal-processing complexity [Bed06]. Consequently, issues of scalability arise as the amount of data produced by interferometers scales quadratically to the number of dishes. In particular, the

[1] `http://www.skatelescope.org`

SKA project aims to have over 2000 dishes by 2024 [SKA14], which will generate data volumes of orders of magnitude greater than other interferometers around the world.

Transient detection is an important field in astrophysics and radio interferometry. Exploration of the transient (or 'time-variable') universe using next-generation radio interferometers has become an exciting area of research, as technological limitations have previously left a large portion of the radio sky unexplored.

Transients are associated with explosive or dynamic events across the universe, and analysing them provides potential opportunities to explore a wide range of new astrophysics. For instance, *pulsars*, which are highly magnetized rotating neutron stars that emits beams of electromagnetic radiation, were one of the earliest-discovered transients. Amongst other interesting properties, analysis of objects surrounding pulsars are currently the best means to test theories of gravity in strong gravitational fields [Kok13].

There are other known radio transients such as the enigmatic fast radio bursts (FRBS) [BBD$^+$09], which, upon discovery and analysis, may help us probe the unconstrained properties of the intergalactic medium (IGM). Other examples include radio emission from supernovae (SNe) and novae outbursts, tidal disruption events (TDEs), where stars emit radiation before they merge with supermassive black holes at the centre of galaxies. Furthermore, it is probable that there exist other objects not yet discovered, because we have not yet properly looked. The search for radio transients is therefore a directed search for such unknown objects, as well as known and established sources of transient radio emission.

Visibility data produced by interferometers are used to make images of the sky. However, elements of this process have been regarded as unintuitive and difficult to analyze [LB11]. Creating images from visibilities require algorithms such as the Fast Fourier Transform (FFT) and repeated deconvolution operations (a full description of imaging is beyond the scope of this thesis and further details may be found in [ART01]) to process the data into images as required, before insightful observations can be made. However, Law and Bower (2011) demonstrate a new technique for detecting radio transients based on interferometric closure quantities that avoids creating images in the first stage, and thus does not require highly non-linear imaging algorithms.

Using the product of visibilities around a closed-loop of baselines, a statistical quantity related to the brightness of any source in the field of view is formed. These quantities or bispectra, are calibration independant, resistant to interference and computationally efficient [LB11]. Although the bispectrum algorithm has been successfully implemented in Python at the *Karl G. Jansky Very Large Array* (JVLA), parallelizing this algorithm is required to use multi-core CPU and GPU technologies to acquire large speed-ups. This potentially allows the bispectrum algorithm to achieve the performance improvements required for real-time transient detection on next-generation interferometers such as the SKA.

1.2 Motivation: Graphics Processing Units

Modern GPUs have floating-point computation capabilities that far exceed current CPU architectures [nVi14]. These devices contain large numbers of simple Single-Instruction-Multiple-Data (SIMD) processors that are suitable for the bispectrum algorithm as well as for interferometric data. Historically, software developers have relied on advances in hardware capabilities to sustain the growing computational needs required for complex tasks. However, due to energy consumption and heat-dissipation issues that arise in faster processors [Far11], GPUs have become a popular choice in the field of high performance computing [Bax13]. This popularity

is largely due to their relatively low monetary costs when compared to CPU clusters [Far11], whilst exhibiting significant performance gains.

Utilizing GPUs for *general purpose processing* has become prominent in recent years, particularly in the field of radio astronomy. For example, a CUDA GPU implementation of gravitational lensing performs two orders of magnitude faster than a single core CPU implementation [Bax13]. Monte Carlo dust temperature simulations have reported speed-ups of 69 times over CPU implementations [JP09]. Due to the "embarrassingly parallel" nature of these algorithms, as well as the independently separable data these algorithms operate on [Lue08], they are well suited to GPU computing. The bispectrum algorithm is a prime example of such an "embarrassingly parallel" algorithm as shown in Chapter 4.

Although GPUs were designed primarily for the rendering of images and geometry in applications such as computer games, the increasing need for more programmable GPUs spawned new graphics programming APIs. nVidia, a vendor of GPUs, has developed architectures that no longer restrict developers to a fixed function pipeline through the use of the *Compute Unified Device Architecture* (CUDA) [nVi14]. CUDA allows programmers to exploit parallelism on GPUs using several programming abstractions. As a result, with the ubiquitous nature of GPUs and the development of programming APIs, particularly CUDA, programmers are now able to fully exploit the capabilities of a GPU and achieve maximum performance.

1.3 Problem statement

Next-generation interferometers such as the SKA will generate data at rates incomparable to any other current interferometers. Consequently, current sequential implementations are no longer sufficient to process and analyse observations in real-time. Without access to large amounts of computational power, data analysis will be limited to partial and less meaningful observations.

In this work, an investigation into the applicability of using GPUs to optimize the bispectrum transient detection algorithm is explored. Specifically, a CUDA implementation is used to leverage the massively parallel processing capabilities of commodity graphics hardware to efficiently process large data volumes.

1.4 Aims and Approach

The foremost aim of this project is to develop an initial prototype for detecting transients in simulated data. This implementation aims to produce correct results through a variety of simulated tests using this prototype of the bispectrum algorithm. Thereafter, the prototype will be tested on known transients in real interferometric data to ensure the validity of the implementation and the bispectrum algorithm.

Once validation of the implementation has been completed, a multi-threaded CPU implementation of the algorithm will be developed by parallelizing the program using OpenMP: a standard API for writing parallel C/C++ code [TM11]. Optimizing this implementation provides a performance benchmark for the GPU implementation. Furthermore, it will determine critical performance factors and provides insight into optimising the GPU implementation.

The majority of this work will focus on the optimization of a GPU implementation using several parallel programming techniques. In particular, using the CUDA programming model, the GPU implementation aims to utilize the full computational capacity of GPUs for best performance. The final GPU implementation aims to find the optimal performance configuration that

is scalable for the SKA, as well as its precursors such as the MeerKAT.

1.5 Research Questions

By parallelizing the bispectrum algorithm for execution on the GPU, we aim to answer the following research questions:

1. Can the bispectrum algorithm be accelerated using *General Purpose Computation on Graphics Processing Units* (GPGPU)?
2. If so, how, and by how much can we speed it up?
3. How does the performance of the GPU implementation scale to the number of antennae, frequency bands and longer observations?
4. What are the limitations of the bispectrum algorithm?
5. What are the limitations of the GPU implementation?
6. How does the OpenMP implementation compare with the CUDA implementation?

1.6 Thesis Outline

The remaining chapters of this thesis are organised as follows: Chapter 2 introduces relevant background and foundational concepts in radio interferometry, as well as a detailed analysis of the bispectrum algorithm. Chapter 3 substantiates the applicability of using GPUs to accelerate the bispectrum algorithm with a detailed introduction to CUDA and its programming principles. Chapter 4 describes the individual components of the bispectrum algorithm. Chapters 5 and 6 detail the CPU and GPU implementations, respectively, with particular emphasis on optimizing these implementations. In Chapter 7, we analyse the experimental results acquired from performance tests, which are aimed at answering the aforementioned research questions. Conclusions towards these research questions are presented in chapter 8, together with a general discussion of future work.

Chapter 2

Background Research: Radio Astronomy

This chapter introduces relevant background and foundational concepts in Radio Astronomy. In particular, emphasis is given to the history of radio interferometry observational techniques and the scalability issues that arise when the number of dishes in interferometers increases. Pulsars and transients are described and current methods of detecting these transients are discussed. A new technique for detecting radio transients called the bispectrum technique is introduced and analysed, while highlighting the relevance of GPUs to the bispectrum algorithm.

2.1 Radio Interferometry Observation Techniques

Astronomical radio sources are objects in the Universe that emit electromagnetic radiation. Radio emission is generated by a wide variety of sources and reveals some of the most extreme energetic physical processes in the Universe. For ground-based radio telescopes, the electromagnetic radiation entering the antenna is converted into electrical currents [Kok13].

From the surface, the earth's atmosphere appears to be transparent to most radio waves. However, the atmosphere's constituents are able to absorb a range of wavelengths. Typically, any wavelength (λ) approximately longer than $\lambda = 0.2$mm and shorter than $\lambda = 20$m will be visible from the surface [BBD$^+$09]. These correspond to a frequency limit of $\nu = 1.5$THz and $\nu = 15$MHz, respectively. These limits are only guidelines as variations occur with different altitude, geographical position and time [BBD$^+$09].

Traditionally, radio observations of the sky were conducted using individual parabolic dishes and receivers. However, the cost of very large dishes quickly becomes infeasible since larger dishes are not only more difficult to build, but also more expensive.

Radio interferometry attempts to solve this problem by using a combination of smaller radio-receiving dishes to achieve better resolution at lower cost. Arrays of telescopes can be used to enhance the resolution of astronomical observations compared to single dishes. This is done by measuring interference patterns between the receiving antennas [Bed06].

2.1.1 Single Dish Observations

For single dishes, the resolving power of the instrument is directly related to the diameter of the aperture presented to the sky by the paraboloid dish. The following relation connects these three quantities:

$$R \sim D/\lambda \tag{2.1}$$

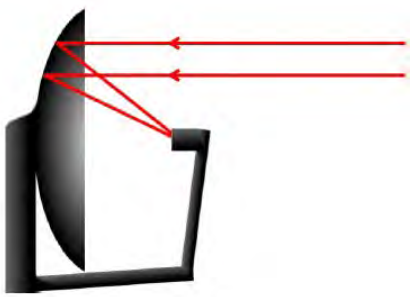

Figure 2.1: **Paraboloid dish with a feed horn**: Incoming electromagnetic radiation are reflected and focused onto a feed-horn which converts the signal into a voltage. Credit: Baxter [Bax13].

where R is the angular resolution, D is the diameter of the dish and λ is the wavelength of the radio wave [Bed06].

To achieve higher angular resolution, larger telescopes may be built. Unfortunately, the engineering and monetary requirements for creating significantly larger dishes are becoming increasingly infeasible as costs scale $D^{\sim 2.7}$ [DFH09]. To date, the largest steerable radio telescopes in the world are about 100m in diameter, such as the Effelsberg 100m Radio Telescope and Green bank Telescope (which is slightly larger than 100m) [Bax13]. The latest ongoing developments include the Five hundred meter Aperture Spherical Telescope (known as FAST) currently being built in China.

2.1.2 Interferometry Theory

Radio interferometry, through the technique of *aperture synthesis*, allows an observer to significantly increase the achievable resolution compared to a monolithic dish. This is done by using arrays of telescopes to synthesize a virtual aperture of a size corresponding instead to the largest separation between dishes (baseline distance) [BBD$^+$09]. The key concept behind an interferometer is effectively simulating a large single radio telescope, with a diameter equivalent to the longest distance (baseline distance) between them. This is achieved by linking smaller dishes together and combining their signals.

The simplest interferometer consists of two dishes. Readings from each telescope are correlated by multiplying the voltage reading on each telescope and averaging them [BBD$^+$09]. In general, an interferometer with N_a dishes has $N_a(N_a-1)/2$ possible correlations. Essentially, an interferometer can be broken down into $N_a(N_a-1)/2$ two-element arrays of telescopes, as shown in Figure 2.2. A pair of dishes in an interferometer is known as a *baseline* and their correlated voltage reading is called a *visibility* [BBD$^+$09]. The distance between a pair of dishes is called their *baseline length*.

Consider two identical telescopes shown in Figure 2.1 separated by the baseline vector $\vec{b}$. Both telescopes are pointing in the same direction denoted by $\vec{s}$. Radio waves that manage to enter through the earth's atmosphere from this direction will reach the telescopes at different times. As shown in Figure 2.2, radio waves travelling to antenna 1 have an extra distance of:

$$\vec{b} \cdot \vec{s} = ||b|| \cos(\theta). \tag{2.2}$$

Therefore observed signals from antenna 1 lag by a geometric delay:

$$\tau_g = \frac{\vec{b}.\vec{s}}{c}, \tag{2.3}$$

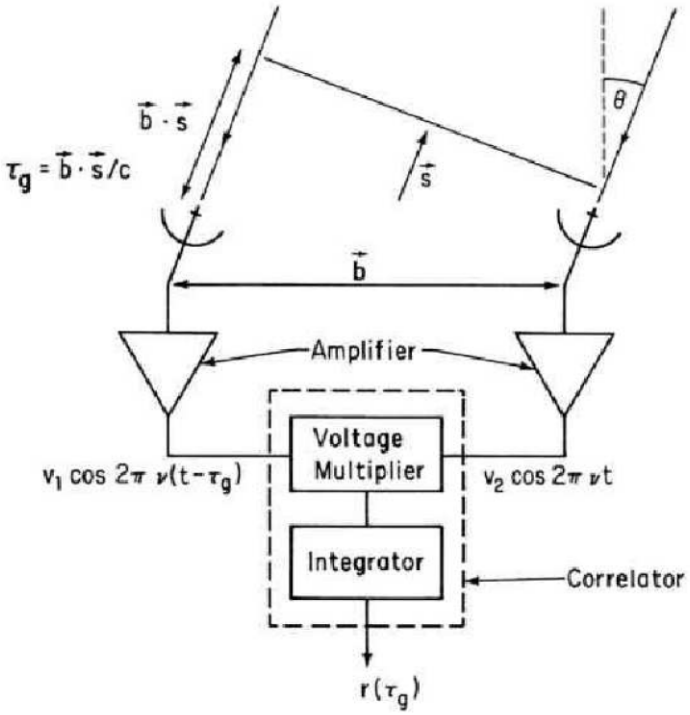

Figure 2.2: **A Simple two-element interferometer**: As shown in the figure above, a wavefront from a source will arrive at different times at each of the antennas. A correlation solution is applied by the receivers for this geometric time delay. Credit: Thompson [ART01].

where c $= 3 \times 10^8 m/s$, the speed of light. Let us assume that our interferometer is quasi-monochromatic, i.e. it responds only to radiation in a very narrow band centered on frequency $\nu = \omega/2\pi$. The electromagnetic waves propagating from direction $\vec{s}$ will induce the following voltages V_1 and V_2 in each antenna receiver [BBD$^+$09] [Kok13]:

$$V_1 = V\cos\left[\omega(t-\tau_g)\right] \tag{2.4}$$

$$V_2 = V\cos\left[\omega t\right], \tag{2.5}$$

where t denotes the time and V is the voltage amplitude. Since readings from each pair of telescopes are multiplied, combining equation 2.4 and 2.5 (and using the cosine rule) gives us:

$$V_1V_2 = V^2\cos\left[\omega(t-\tau_g)\right]\cos\left[\omega t\right] = \frac{V^2}{2}(\cos\left[\omega(t-\tau_g)\right] + \cos\left[\omega\tau_g\right]) \tag{2.6}$$

Thereafter, a time averaging function (where $\Delta t \gg 1/2\omega$, where Δt is the time average) is applied to remove the high frequency term $\cos\left[\omega(t-\tau_g)\right]$ as the average will tend to zero [Kok13]. Equation 2.6 is thereafter simplified to:

$$V_1V_2 = \frac{V^2}{2}\cos\left[\omega\tau_g\right] \tag{2.7}$$

The uncorrelated noise is eliminated through the correlation process and does not appear in the correlator output. Compared to the observations with a single dish, this leads to significantly decreased fluctuations due to receiver gain and atmospheric emission [Kok13].

2.1.3 Radio Frequency Interference (RFI)

Radio frequency interference (RFI) is the radiation or conduction of radio frequency energy or electronic noise produced by electrical devices in the vicinity, at levels that obscure the astronomical signals collected by receivers [Cur09], i.e., unwanted radio signals. Common sources of

electrical RFI include components such as power lines, neon signs, nearby computers and other computing devices. Electrically radiated RFI is usually controlled by providing proper shielding to ensure the radiated signals are attenuated to satisfactory levels [Cur09].

Another major source of RFI is electromagnetic transmitters on and around the Earth, such as cellular base stations and satellites [wik14a]. These sources can be many times stronger than the astronomical signals of interest. Several methods such as filters in hardware and advanced algorithms have been developed to mitigate these signals. Spectrum management regulates the use of radio frequencies from *3kHz* to *300GHz* [Cur09] by restricting the usage of certain frequency bands. For example, some frequency bands that are very important for radio astronomy, such as the 21-cm Hl line at 1420MHz, are reserved exclusively for radio astronomy [wik14a].

Overcoming these RFI signals that obscure celestial information poses a challenge to radio astronomers, particularly in the field of transient detection. A common technique to mitigate RFI requires RFI detection algorithms to be installed in software. Such software flags samples of observations that have been contaminated by an interfering source [wik14a]. These samples are thereafter ignored in further analysis. In Section 10, we give details of how the bispectrum algorithm can not only reject local interference, but also proves resistant to RFI to a certain extent because it differentiates visibilities in time and maximizes the output SNR.

2.1.4 Current Interferometry Architectures

Astronomical observation techniques have greatly improved, with radio observations in particular, becoming one of the most productive means of astronomical research. Some of the most innovative telescopes still include large single dish telescopes such as the Arecibo Radio telescope in Puerto Rico with a 305m diameter [nas97]. The FAST telescope (mentioned in section 2.1.1) is 500m in diameter and is currently still under construction. However, interferometers have become more popular, with the SKA (Square Kilometre Array) at the forefront with approximately 3000 dishes planned [Tre13], totalling *one square kilometre* of surface area [SKA14]. Table 2.1 below describes some of the modern interferometers.

2.1.5 Scalability issues

As interferometers achieve larger sizes, (and thus with quadratically more baselines), they push the boundaries of what was previously possible. This presents new challenges in the form of data management and data processing, particularly given that a single dish can produce data at a rate of around 160GB/s [Tre13]. It is evident that these new generation of interferometers require novel techniques to overcome the challenge of larger datasets. In the field of transient

Name	N_a	Location	N_{bl}	b_{max} (km)
WRST	14	Netherlands	91	2.7
GMRT	30	India	435	25
VLA	27	U.S.	351	36
$MeerKAT^*$	80	South Africa	3160	20
e-MERLIN	7	U.K.	21	220

Table 2.1: **Inteferometer examples**: N_a represents the number of antennas, N_{bl} the total number of baselines and b_{max} the maximum baseline distance. Different interferometers observe at different frequencies and at different time-scales [Mio14]. $^*MeerKAT$ will have 80 dishes after "phase two".

Figure 2.3: **VLA interferometer**: Image of the Very Large Array (VLA) interferometer in the United States. The VLA has a total of 27 radio antennae in a Y-shaped formation.

detection, current imaging algorithms will not be fast enough for real-time transient detection [LBP+12] as next-generation interferometers will have more antennas, wider bandwidths and larger fields of view. In section 10, we present a technique using interferometric closure quantities with more moderate computational demands that can reduce the flow of data to a manageable size [LBP+12]. Other strengths of this technique include many independent processes that can be readily parallelized. Furthermore, these independent processes operate on independent data which is well-suited for GPU [nVi14], and thus, speed-ups can be expected on a GPU implementation.

2.2 Pulsars and Transients

The radio sky is relatively unexplored for transient signals and this has remained a major astrophysics frontier over the past few decades [MLL+06]. Transient phenomena have time scales ranging from sub-nano seconds to years or longer and thus span almost 20 orders of magnitude [BCC+13]. Transient are associated with explosive or dynamic events across the universe and hence provide enormous potential to uncover a wide range of new astrophysics.

The optical sky is (to some extent) routinely monitored for transient phenomena [BCC+13] by a number of wide field of view instruments. These transient phenomena are typically with algorithms that take advantage of their repeating nature. Historically, transient detection has been performed using high-sensitivity radio surveys using large single dishes which, by definition, would have relatively narrow fields of view. Next-generation radio interferometers (discussed in section 2.1.4) will open up new avenues for exciting radio transient science, particularly in the field of pulsar detection.

Pulsars are highly magnetized rotating neutron stars that emit beams of electromagnetic radiation, or transients, and are observed from earth as periodic pulses across a range of frequencies (see Figure 2.4) [LK07]. Currently known radio pulsars emit pulses with periods between 1.4 milliseconds and 8.5 seconds [nra14]. In the simple case, the radio beam emitted from the poles of these stars are actually continuous, however, pulsars rotate with short regular periods [wik14b]. This beam of radiation can only be observed when pointed in the earth's direction. This can be compared to a lighthouse emitting beams of light towards an observer.

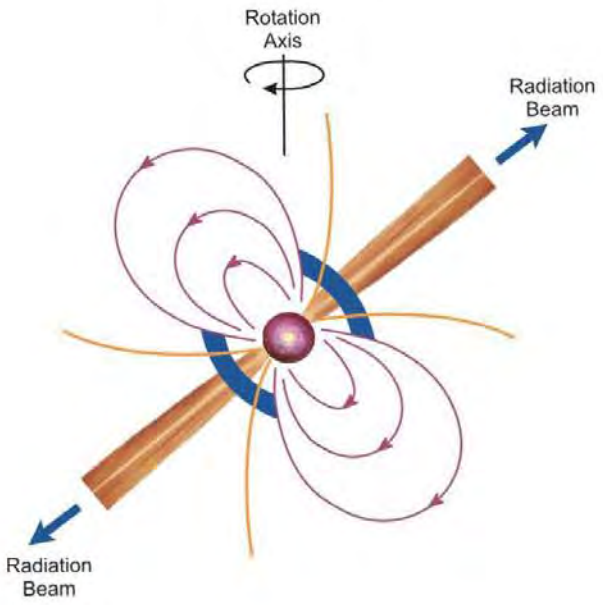

Figure 2.4: **Radiation emitted by a pulsar** : A diagram of the traditional magnetic dipole model of a pulsar. Beams of electromagnetic radiation are emitted through the poles and observed as radio pulses. Credit: NRAO website [nra14].

2.2.1 Importance of pulsars

Since the first discovery of pulsars by Jocelyn Bell and Anthony Hewish in 1967 [BBD$^+$09], astronomers have detected over 1300 pulsars in our galaxy alone and predict that billions of pulsars exist in the universe. The analysis of relativistic binary pulsars is currently the best means to test theories of gravity in strong gravitational fields. The precise periods of pulsars makes them useful tools, as certain types of pulsars rival atomic clocks in their accuracy [wik14b, nas97]. Furthermore, the first extrasolar planets were discovered around a pulsar opening up many observations of unanticipated phenomena that occur in the vicinity of pulsars [BBD$^+$09].

2.2.2 Transient detection pipeline (TraP)

The transient detection pipeline used in the LOFAR radio transient software pipeline (TraP) (see Figure 2.5) is a near real-time pipeline that monitors an incoming steam of images for both known transients and unknown variable sources [Kok13]. Currently transient detection techniques require that images of the sky be produced before any form of source detection can begin. Much research has been invested into the optimisation of imaging [Tre13]. Furthermore, optimisations of transient detection algorithms in the TraP pipeline have improved significantly. For example, the TraP's source detection algorithms improved image processing rates from 0.14 images per second per core to 0.85 [Mol13]. Currently, the TraP pipeline performs source detection at a rate of 48 images/s over 57 cores.

Although TraP includes a quality control component in its pipeline that automatically rejects bad or dirty images, much computational effort has already been expended to produce these images. A majority of incoming images do not contain any significant sources. Thus, a preprocessing of the raw data would be useful to reduce unnecessary work. Fundamentally, the bispectrum algorithm developed by Law and Bower [LBP$^+$12] allows statistical analysis of visibilities to take place prior to imaging (imaging is at least twice as computationally demanding as the bispectrum algorithm [LBP$^+$12]). This allows the bispectrum to efficiently filter out timesteps in observations where transients are improbable. This not only heavily reduces the amount of computation required to image an observation but also naturally reduces the number

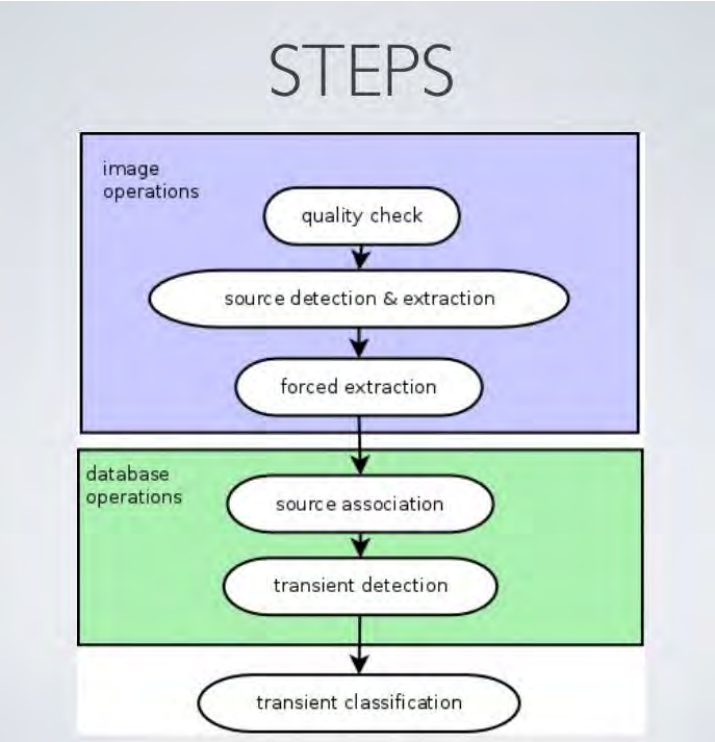

Figure 2.5: **TraP pipeline** : Schematic flowchart of the transient detection pipeline (TraP). Data from interferometers are initially pre-processed by an imaging step and transferred to TraP. Transient analysis is performed using a combination of source finding and analysis routines and a high performance database [Kok13]. Credit: Gijs Molenaar [Mol13]

of unnecessary images to be processed by TraP.

2.2.3 Dedispersion

As electromagnetic waves propagate through the ionized interstellar medium, they become dispersed over time. Since radiation is delayed less at higher frequencies, we observe a pulse at higher frequencies first [LK07]. Since observations usually span different frequency channels, the delay in the observed transient is different across each frequency channel. The time delay in seconds between two observing frequencies is given by:

$$\Delta t = 4.15 \times 10^3 \dot{DM}(\frac{1}{f_1} - \frac{1}{f_2}), \tag{2.8}$$

where frequency is in MHz and DM is the dispersion measure in pc/cm^3 (parsec per cubic centimetre) [Kok13] [BBD$^+$09]. The dispersion measure depends on the distance between the object emitting radiation and the receiver. Thus an advantageous strategy for searching the radio sky for transients is to iterate through many different DM values (blind detection) until there is a peak in SNR. One can imagine a transient signal smeared across many time steps over the different channels, and de-dispersion is the process of re-aligning them back together for a strong signal. De-dispersion is also suited for GPU computing as different DM trials can be processed in parallel [WA12].

2.3 Bispectrum theory and analysis

Much research has been invested into the development of a computationally efficient method of transient detection [Kok13, Ahu05]. Amongst these methods is a new technique for detecting

radio transients based on interferometric closure quantities [LBP+12]. This method of transient detection uses the bispectrum, a statistical quantity that is formed by combining baseline visibilities around a closed loop of an interferometer.

The bispectrum is calibration independent, resistant to interference and computationally efficient compared to other imaging techniques (mentioned in Section 2.2.2), and can therefore be built into correlators for real-time transient detection [LB11]. It has been shown that the bispectrum method can reject local interference and detect dispersed pulses by dedispersing visibilities [LBP+12]. The bispectrum method not only requires less computation but is also an "embarrassingly parallel" algorithm that is well suited to parallel computing. This technique is a promising step towards the development of real-time transient detection for the new generation of radio interferometers.

2.3.1 Closure Quantities

A closure quantity is a mathematical combination of visibilities from antennae that form a closed loop [CGB08]. That is, a quantity formed by combining the three unique baseline visibilities associated with three different antennae. An example of a closure quantity relevant to interferometric data is the triple phase or the closure phase, whereby the sum of the visibility phases from a closed loop of antennas are summed. Assuming there are three antennas *i, j, k*, their relative phases are expressed as follows:

$$\Phi_{ij} = \phi_{ij} + \psi_i - \psi_j + \epsilon_{ij} \tag{2.9}$$

$$\Phi_{jk} = \phi_{jk} + \psi_j - \psi_k + \epsilon_{jk} \tag{2.10}$$

$$\Phi_{ki} = \phi_{ki} + \psi_k - \psi_i + \epsilon_{ki}. \tag{2.11}$$

The phase Φ_{ij}, measured on the baseline between antenna i and j, is the true phase for this baseline ϕ_{ij} plus systematic errors ψ_i, ψ_j introduced above the two antennas, plus random errors ϵ_{ij} related to this baseline. By summing the three baselines above we get the equation below:

$$\Phi_{ijk} = \phi_{ij} + \phi_{jk} + \phi_{ki} + \epsilon, \tag{2.12}$$

where $\epsilon = \epsilon_{ij}+\epsilon_{jk}+\epsilon_{ki}$, from equations 2.9, 2.10, 2.11, is the combined random error of the visibilities.

The triple phase Φ_{ij} has the useful property that it is independent of systematic phase errors associated with any individual antenna due to correlation, as shown in equation 2.12. Another important property of the triple phase is its sensitivity to point sources anywhere in the field of view [LB11, LBP+12]. Closure quantities have been primarily developed for long baseline interferometry since it is possible to detect a source even when phase calibrations are unstable [AR95].

A related closure quantity is the product of visibilities that form a closed loop, known as the triple product or bispectrum. This quantity should not be confused with the Fourier transform of a third-order cumulant-generating function, also known as the bispectrum [AMDM85]. In this thesis, the bispectrum refers to the triple product and can be written as:

$$b_{ijk} = a_{ij}a_{jk}a_{ki}e^{i\phi_{ijk}} \tag{2.13}$$

where b_{ijk} is the bispectrum, a_{ij} is the magnitude of the *ij* baseline visibility and ϕ_{ij} the triple phase of antennas *ijk*.

It can be seen from equation 2.13 that the bispectrum inherits useful properties from the closure phase. The bispectrum is independent of atmospheric phase errors and is also sensitive to point

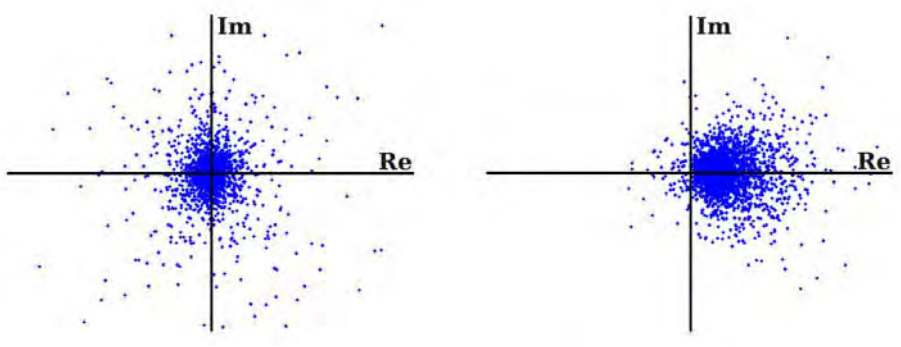

Figure 2.6: **Distribution of bipsectra on a complex plane** : Each point represents the bispectrum of a particular triple of antennas. Left panel shows the absence of a transient signal. Right panel demonstrates how the bispectra migrate to positive real values when a new celestial point source appears in the field. Credit: Georgi Kokotanekov. [Kok13]

sources anywhere in the field of view. Furthermore, in the absence of signal, the bispectra across all baseline triplets have a mean at the origin of the real-imaginary plane (shown on the left in Figure 2.6) [AMDM85, AR95]. Celestial point sources in the field of view will move the mean of the bispectra towards positive real values (shown on the right in Figure 2.6). The mean of the bispectra is also proportional to the cube of the point sources' brightness [AR95]. This allows the bispectrum to be a useful quantity for statistical detection of point sources.

2.3.2 Fringe considerations

In the first step of calculating bispectra, the visibilities need to be differenced in time (see Section 4.3). This subtracts constant emission from the visibilities to maximise the SNR of the bispectrum. Proper removal requires that the visibility fringe or interference visibility does not change over the background subtraction time. More precisely, the error from subtracting visibilities leaves residual errors proportional to the brightness of the sources in the field of view.

The interferometer coherence pattern (observed by the radio antennae) has a spacing λ/b_{max} where λ is the wavelength and b_{max} is the maximum distance between two baselines [Kok13]. Different sources in the sky observed by an interferometer rotate about the celestial pole at angular rate $\omega = 7.3 \times 10^{-5}$ rad/sec. The total amount of time for a source to move by λ/b_{max} in the worst case, assuming the source is at the edge of the field of view, can be calculated as follows:

$$t = \frac{\lambda}{b_{max}.\omega.\theta_{FoV}} \tag{2.14}$$

where θ_{FoV} is the field of view of the interferometer.

For accurate subtracting of constant emission, the window for calculating the mean visibility should not exceed t for a given set of parameters of an interferometer. For example, the MeerKAT interferometer with longest baseline distance $b_{max} = 20$km and $\theta_{FoV} = 1$ degrees has $t_{max} = 0.2$ seconds. If MeerKAT has time integrations of 10ms, calculations of the mean should not exceed 20 time steps. However, equation 2.14 is rather optimistic, since proper subtraction of visibilities also depends on the brightness of the source. For implementations in Chapters 5 and 6, window sizes were made substantially smaller, where possible.

2.3.3 Standard deviation of Bispectrum

In the absence of signal, the mean of the bispectra will be centred at the origin of the complex plane (see Figure 2.6). Once a point source enters the field of view, the point source shifts the

mean of the bispectra towards real positive values in the complex plane. It has been shown that the standard deviation of the complex bispectra is given by:

$$\sigma_B = \sqrt{3}Q^3 s^2, \tag{2.15}$$

where Q is the noise per baseline [Kul89] and s the signal strength.

The standard deviation of the bispectrum can be used to differentiate between detections from radio frequency interference (RFI) and celestial sources. In contrast to a celestial source, interference is often in the near-field or subject to multi-path propagation. This is likely to produce a random triple phase and hence a larger variance. Therefore, RFI increases the variance of the bispectra. This allows the algorithm to classify all bispectra with standard deviation higher than σ_B as RFI [LBP$^+$12].

2.3.4 Scalability

Each antenna from an interferometer is correlated with every other antenna to form baselines. For an interferometer of N_a antennas the number of baselines is expressed by the following equation:

$$N_{bl} = \frac{N_a(N_a - 1)}{2} \tag{2.16}$$

and the number of closed triplets formed by the following equation:

$$N_{tr} = \frac{N_a(N_a - 1)(N_a - 2)}{6} \approx \mathbb{O}(N^3) \tag{2.17}$$

However, it can also be shown that some of these triplets are redundant [AR95], e.g. $\phi_{123} = \phi_{124} + \phi_{143} + \phi_{234}$ as we can pair the conjugates of each term to cancel with another. Hence, equation 2.17 can be reduced to the following quadratic equation:

$$N_{tr} = \frac{(N_a - 1)(N_a - 2)}{2} \approx \mathbb{O}(N^2) \tag{2.18}$$

Kulkarnial [Kul89], showed that the bispectra formed from triples sharing a baseline are correlated only when there is a strong source seen by all baselines. Hence, they showed that in the limit of low values of s (after the removal of constant sources), all triplets are independent and can be coherently combined to improve SNR [LBP$^+$12]. To achieve low s for the calculation of the bispectrum, the bispectrum algorithm uses visibilities that have been differenced in time, which leads to a very low signal, and thus all baselines are usable.

2.3.5 Bispectrum SNR

By coherently combining signals over all the baselines (beamforming), SNR is improved from s to $s\sqrt{N_{bl}}$ [LBP$^+$12]. One can think of imaging techniques as coherently beamforming all possible beams across the interferometer. The bispectrum on the other hand, combines visibilities across three independent baselines which improves SNR in the following way:

$$SNR_{bisp} = \frac{1}{2}s^3\sqrt{n_{tr}} \tag{2.19}$$

This shows the bispectrum has stronger SNR compared to imaging technique and also scales well to the size of the interferometer [LB11].

According to Kulkarni [Kul89], Equation 2.19 is only valid in the limit of small s ($s < 1$).

In his constant source analysis, he assumes the object would be present in all timesteps, i.e. in all background noise measurements. This means that in the case of a strong source, not all the triplets (e.g, ϕ_{123}) are independent. This effectively adds noise proportional to the flux of the source, called "self-noise", which lowers the SNR of the bispectrum as the relation in equation 2.19 breaks down.

The bispectrum algorithm uses visibilities that have constant emissions subtracted off. Consecutive visibilities are averaged and subtracted off in blocks (see Section 4.3.1). Thus when a pulse appears, its strength is calculated relative to an empty sky, implying there are no "self-noise" effects that occur to reduce SNR. Thus, for the bispectrum algorithm, the scaling of s^3 is not limited to $s < 1$.

Sigma (σ) Threshold

According to Gaussian statistics, the bispectrum SNR is related to the false positive rate as $\frac{1}{2}(1-erf(SNR/\sqrt{2}))$ [AR95] where *erf* is the "error function". Thus a 5σ deviation in the bispectrum would correspond to a false positive rate of 2.87×10^{-7} [LB11]. However, a slight complication to this SNR analysis is that individual bispectrum measurements are not Gaussian distributed. Visibilities are Gaussian distributed, so multiplying three noise-like complex visibilities produces a distribution with a long tail [LB11]. However, since each individual bispectrum is independent (see refined equation 2.9), the Central Limit Theorem states the the mean of the bispectra will approach a Gaussian distribution with larger arrays of bispectra. This is particularly applicable since interferometers are constantly expanding to include more antenna and the number of triplets increases quadratically with the number of antennae (Equation 2.18).

2.3.6 Computational efficiency

Another strength of the bispectrum is its ability to find transients in real time throughout the field of view. The algorithm is efficient enough to be built into correlators for real-time transient detection. The bispectrum is thus useful since it is computationally simple with several independent parallel components (see section 4.2), resistant to interference and requiring no phase calibration [LBP+12].

In a recent work by Kokotanekov [Kok13], the computational efficiency of the bispectrum was compared to imaging methods using 660-second MSSS observations $N_t = 660$ and 26 antennas. Using equations 2.16 and 2.18 with 26 antennas yields $N_{bl} = 325$ baselines and $N_{tr} = 2600$ triplets. He combined 4 subbands (grouping of several channels), for which all channels were averaged. This lead to the equivalent of $N_{ch} = 4$ separate frequency channels. Each visibility also had N_{pol} polarizations to produce images with N_{pix} pixels per side.

For the MSSS observation given above, the subtraction of visibilities needs approximately $N_{sub} = 10 N_{bl} N_{pol} N_{ch} N_t = 17$ million operations (Mop). This is required by both the imaging techniques and the bispectrum algorithm. The imaging FFT (Fast Fourier Transform) requires $N_{FFT} = N_{bl} N_{ch} + 5 N_{pix}^2 \log_2 {N_{pix}}^2 = 24$ Mop per image. Therefore, the total number of calculations required to produce the images is:

$$N_{imaging} = N_{sub} + N_{FFT} N_t \approx 1600 Mop. \tag{2.20}$$

Whereas, to run the bispectrum algorithm for the same dataset requires:

$$N_{bispectrum} = N_{sub} + 16 N_{tr} N_{pol} N_t \approx 72 Mop. \tag{2.21}$$

Illustrated in a straight-forward comparison, where the number of operations for imaging excludes image cleaning and calibration, it is evident that the bispectrum algorithm is far more

Computational Demand per Integration for VLA Fast Transient Search Algorithms[a]

Process	Bispectrum		Coherent Beamforming		Imaging	
	Scaling	Demand	Scaling	Demand	Scaling	Demand
Subtraction	$10N_{\rm bl}N_{\rm pol}N_{\rm ch}N_{\rm t}$	14 Mop	$10N_{\rm bl}N_{\rm pol}N_{\rm ch}N_{\rm t}$	14 Mop	$10N_{\rm bl}N_{\rm pol}N_{\rm ch}N_{\rm t}$	14 Mop
Beamform	–	–	$3N_{\rm b}N_{\rm bl}N_{\rm pol}N_{\rm ch}N_{\rm t}$	28 Gop	–	–
Dedisperse	$N_{\rm bl}N_{\rm pol}N_{\rm ch}N_{\rm DM}N_{\rm t}$	287 Mop	$N_{\rm b}N_{\rm ch}N_{\rm DM}N_{\rm t}$	2.8 Gop	$N_{\rm bl}N_{\rm pol}N_{\rm ch}N_{\rm DM}N_{\rm t}$	287 Mop
Image[a]	–	–	–	–	$1.5e6N_{\rm DM}N_{\rm t}$	602 Mop
Bispectrum	$16\ N_{\rm tr}N_{\rm pol}N_{\rm DM}N_{\rm t}$	37 Mop	–	–	–	–
Total[b]		339 Mop		33 Gop		904 Mop

[a]Excluding image cleaning, source identification, and application of calibration.

[b]Assuming high-demand case: $N_{\rm bl} = 351$, $N_{\rm tr} = 2925$, $N_{\rm SB} = 16$, $N_{\rm pol} = 2$, $N_{\rm ch} = 1024$, $N_{\rm b} = 7000$, and $N_{\rm DM} = 200$. This produces spectra with 2 GHz of bandwidth and 2 MHz channel size. The number of dispersion trials, $N_{\rm DM} = 200$, is appropriate for a high-demand case of $DM = 1000$ pc cm^{-3} for 1.2–2.0 GHz band and 10 ms integration time. The number of time scales probed is effectively $N_{\rm t} = 2$, assuming that the number of trials scales as the inverse of the time scale. Imaging and beamforming assume the D configuration, which has a longest baseline of 1 km. The A configuration requires 900 times more pixels and beams.

Figure 2.7: **Computational demands**: A comparison of the different computational demands for the different components of the bispectrum algorithm.Credit: Law and Bower. [LBP+12]

.

computationally efficient for coherent pulse detection. Although the above does not consider localization and imaging candidate pulses, the bispectrum algorithm only requires these operations to be performed occasionally (that occur above 5σ). The total computation time for the bispectrum is therefore still dominated by steps before calibrating and imaging candidate events. See Figure 2.7 for Law and Bower's comparison of computational demands.

2.4 Bispectrum in the real world

Law and Bower [LBP+12] demonstrate their transient detection technique using the Allen Telescope Array (ATA) and the Karl G. Jansky Very Large Array (VLA). They show that the bispectrum algorithm is suitable for new generation interferometers and can detect dispersed pulses and reject local interference [LBP+12]. Figure 2.8 shows how the bispectrum and coherent beamforming techniques can detect pulses in PoCo (Pocket correlator - one of the observing modes of the VLA) data. For both techniques, Figure 2.8 shows the apparent pulse SNR when the pulsar is at the phase center, away from the phase center and at the phase center but uncalibrated. The bispectrum detects the pulse regardless of the calibration of the data and location of the pulse. Figure 2.9 shows that the bispectrum also responds well to multiple pulses in the same window.

Following the observations of B0329+54, Law and Bower presented the first blind interferometric detection and imaging of a millisecond radio transient with an observation of transient pulsar J0628+0909 [LBP+12]. Using the bispectrum algorithm, Law and Bower searched 16minutes of VLA data with 10ms integrations and three DM trials. Figure 2.10 shows the bispectrum algorithm detecting J0628+0909 at around time integration 300.

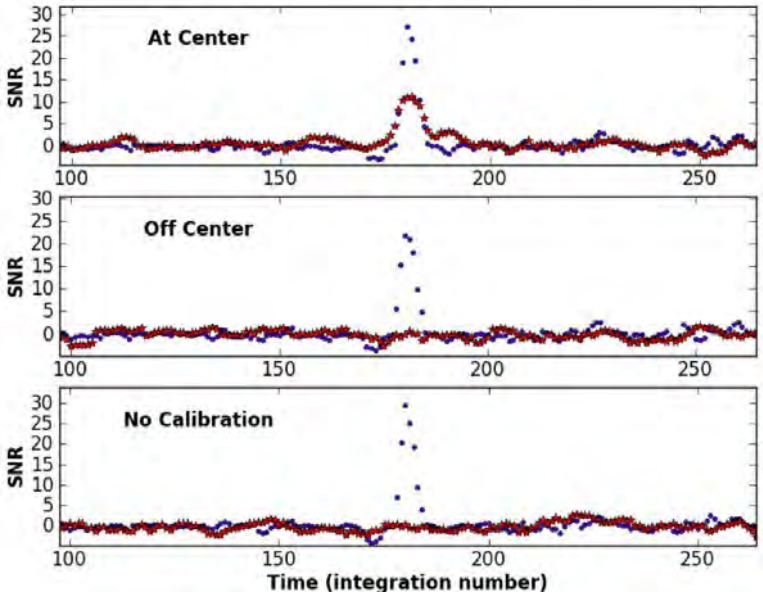

Figure 2.8: **Bispectrum in practice** : Time series of the apparent SNR measured by the bispectrum (blue) and with coherent beamforming (red) in 5-antenna PoCo data of pulsar B0329+54. The PoCo data have a time resolution of 1.2ms and have been dedispersed, assuming the known DM of B0329+54. *Top*: Calibrated data with pulse at phase center. *Middle*: The same plot but with the pulse off the phase center. *Bottom*: Same plot with no phase calibration. Credit: Law and Bower. [LBP+12]

.

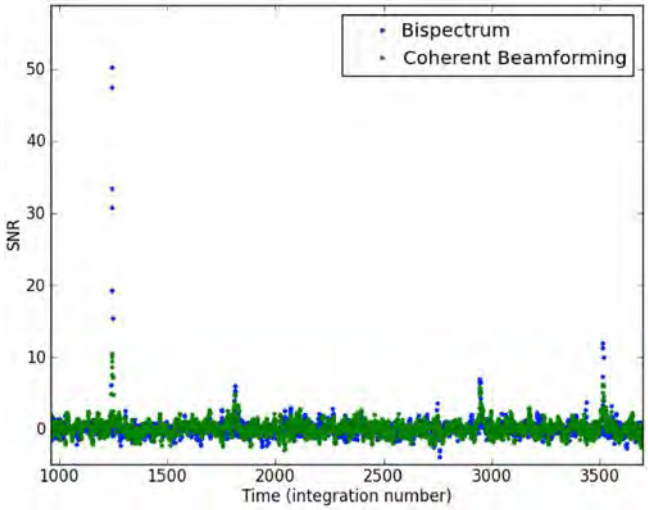

Figure 2.9: **Bispectrum in practice** : A time series of PoCo data observing B0329+54. The rotation period of B0329+54 is about 600 integrations and four of the five pulses in this window are detected. Credit: Law and Bower. [LB11]

.

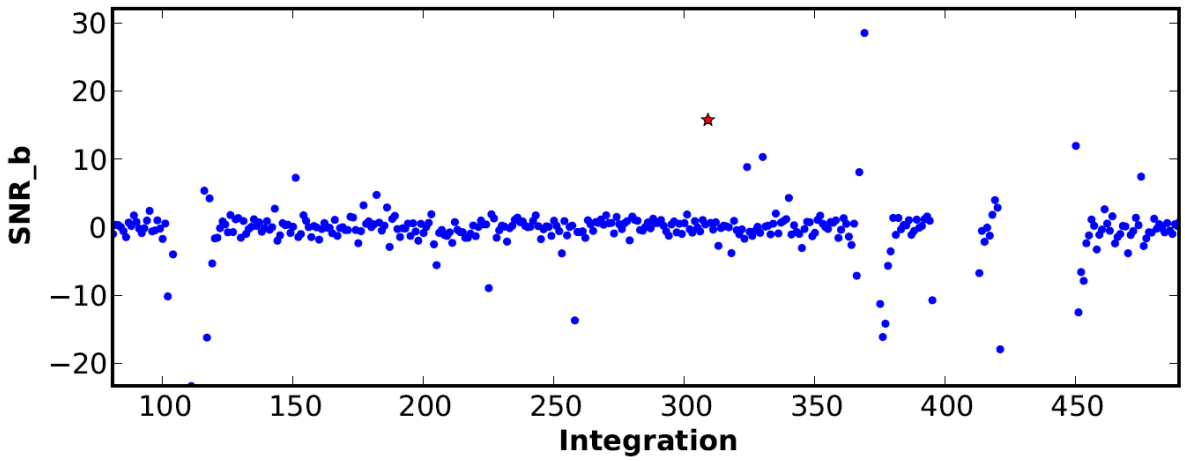

Figure 2.10: **Bispectrum in practice** : The red star shows the pulse detected by the algorithm. Other SNR above 10σ were classified as RFI using the source differentiation technique described in Section 2.3.3. Credit: Law and Bower. [LBP+12]

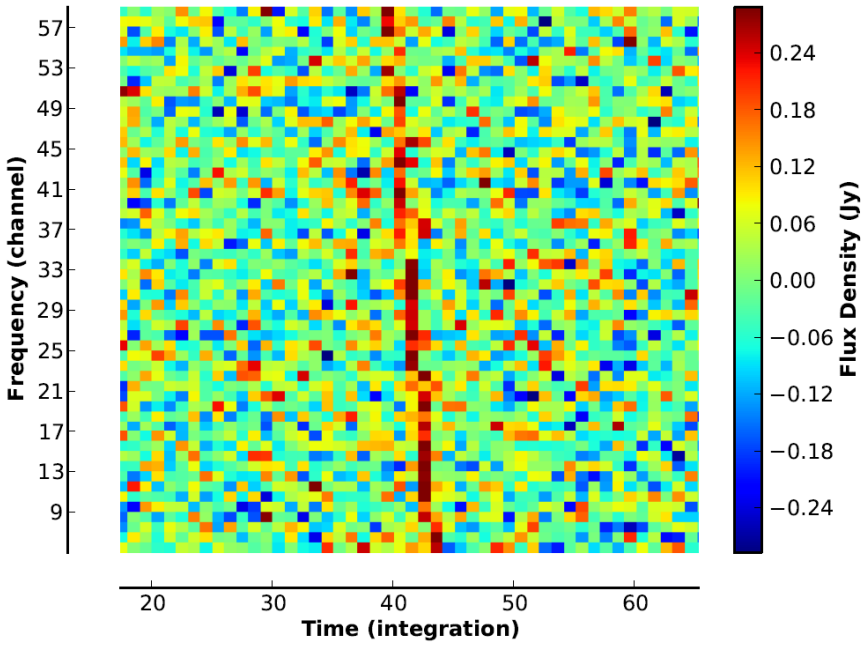

Figure 2.11: **Bispectrum in practice** : Spectrogram showing the dispersed pulse from RRAT (Rotating radio transient) J0628+0909 shown in Figure 2.10. The spectrogram reads the flux density or the intensity of the pulse. As shown in the figure, the pulse is seen first by higher frequency channels due to dispersion. Credit: Law and Bower.[LBP+12]

2.5 Summary

In this chapter we introduced foundational concepts in Radio Astronomy and relevant background in interferometry. In particular, emphasis was given to the history of radio interferometry observational techniques and the scalability issues that arise when interferometers increase in size. Pulsars and transients were introduced with current transient detection pipelines such as TraP outlined. A new technique for detecting radio transients called the bispectrum was introduced and analysed while highlighting the relevance of GPUs to the parallel nature of the bispectrum algorithm. We concluded that the bispectrum algorithm requires less computation when compared to imaging and beamforming techniques. Lastly, we introduced Law and Bower's recent work on detecting transients using the bispectrum.

Chapter 3

Background: CUDA

Modern Graphics Processing Units (GPUs) contain a large number of simple Single-Instruction-Multiple-Data (SIMD) processors that can be used for general purpose computing. Programming these GPUs has become simpler in recent years with the development of general application programming interfaces for GPUs such as nVidia's Compute Unified Device Architecture (CUDA) technology: a SIMD model for general purpose computation on nVidia's commodity GPU hardware [Bax13]. CUDA runs on all current NVIDIA GPUs including the HPC-orientated Tesla product line. The ubiquitous nature of these GPUs makes them a compelling platform for accelerating high performance applications. Furthermore, GPUs have a notably lower price to performance ratio. That is, GPUs are capable of more floating point operations (FLOPS) at lower costs, when compared to CPUs.

This chapter serves to firstly substantiate the choice of using GPUs and CUDA for the implementation of the bispectrum algorithm. Thereafter, CUDA's hardware and execution models, with their corresponding programming principles, such as memory management and optimisation strategies, are introduced. These programming principles are critical to the development of a fully optimised GPU implementation.

3.1 Applicability

The use of GPUs to accelerate computationally intensive algorithms (in the field of radio astronomy) has become prominent in recent years. For example, a CUDA GPU implementation of gravitational lensing software runs two orders of magnitude faster when compared to a single-core CPU implementation [Bax13]; Monte Carlo dust temperature simulations by Jonsson and Patrik [JP09], produce speed-ups of up to 69 times over CPU implementations; and many diverse astronomy algorithms have reported speed-ups of approximately two orders of magnitude. Due to the "embarrassingly parallel" nature of these computations, it is evident that, in a similar fashion, the bispectrum algorithm may benefit from the use of GPUs, as independent components are well suited for GPU computing.

GPUs have also excelled in other domains such as computational chemistry, gene sequencing and even biomedical imaging. Researchers and companies alike are reporting speed ups typically between 10× to 100× compared to optimised CPU implementations. Speedups of this magnitude open exciting avenues in these particular fields. Processes that were previously evaluative (e.g., awaiting results) may now be interactive (i.e., ready on the fly). For example, a speed-up of 50× reduces a simulation from 1 year runtime to approximately 1 week, decreasing the waiting time when simulating highly complex systems [Tun11].

In particular, the challenge of processing incoming streams of big data from interferometers

using the bispectrum algorithm (detailed in Section 10), is well suited for GPU [Lue08]. Interferometric data contains highly independent separable data in the form of different frequency channels, different baselines and many columns of time-step readings. This allows *data-level parallelism* (see Section 3.4.1) where computation is free to operate on different chunks of independent data. Furthermore, the bispectrum algorithm has different separable operations such as iterations over different dispersion measures and calculations of bispectra over different triplets. This allows *instruction-level parallelism* (see Section 3.4.1) where different sections of the algorithm can be executed in parallel without data dependency issues.

3.2 Graphics Processing Units

Historically, software developers have relied heavily on advances in hardware to increase the speed of their applications. New generations of processors are introduced and are able to complete tasks much faster via faster clock cycles. However, due to energy consumption and heat-dissipation issues that arise in faster processors [Far11], new architectures containing multiple processing units have been introduced. In particular, Graphics processing units (GPUs) have become increasingly popular for computationally intensive algorithms in a wide range of scientific fields [Bax13]. This is largely because they are relatively cheap when compared to multi-core CPUs and supercomputers.

In response to the ever-growing performance needs of complex software with more features and capabilities, multi-core (CPUs) and many-core architectures (GPUs) has become prominent in the hardware industry. Multi-core architectures typically have between 2-8 individual cores with one logic unit assigned to each core. In contrast, many-core architectures typically have hundreds of cores assigned to a single logic unit. These are large independent processing units that execute the same instruction, over different data by indexing data through abstractions such as thread hierarchies (see section 3.3.2).

The escalating computational demands of new programs provided the incentive for parallel program developments also known as the *concurrency revolution* [DBK10]. The idea of parallel programming, however, is by no means new. The high performance computing community has been developing parallel programs for decades. These programs typically run on large-scale, expensive computers [DBK10]. Nowadays, due to the ubiquitous nature of GPUs, they have become the frontier hardware used for parallel computing.

GPUs were designed primarily for the rendering of images and geometry in applications such as computer games [Far11]. The steady increase in the performance of these GPUs has been driven by the growth in the gaming industry. The gaming industry requires real-time, high definition 3D graphics. Under these circumstances, GPUs have become highly parallel, high performance commodity hardware [nVi14] (See Figure 3.1).

Algorithms that have a higher ratio of floating point operations to memory operations with independent parallel components are particularly well suited to GPUs. Data parallel computation is ideally embarrassingly parallel (operations can execute on many data elements separately) lacking the requirement for sophisticated flow control hardware found on CPUs [nVi14] (see Figure 3.2). This is particularly evident in 3D rendering where large sets of pixels and vertices are mapped to parallel threads [nVi14]. Other examples include image and media processing applications, such as video encoding and decoding or image scaling, where image blocks and pixels are mapped to parallel processing threads.

In essence, GPUs are designed as numeric computing engines, performing as many floating-

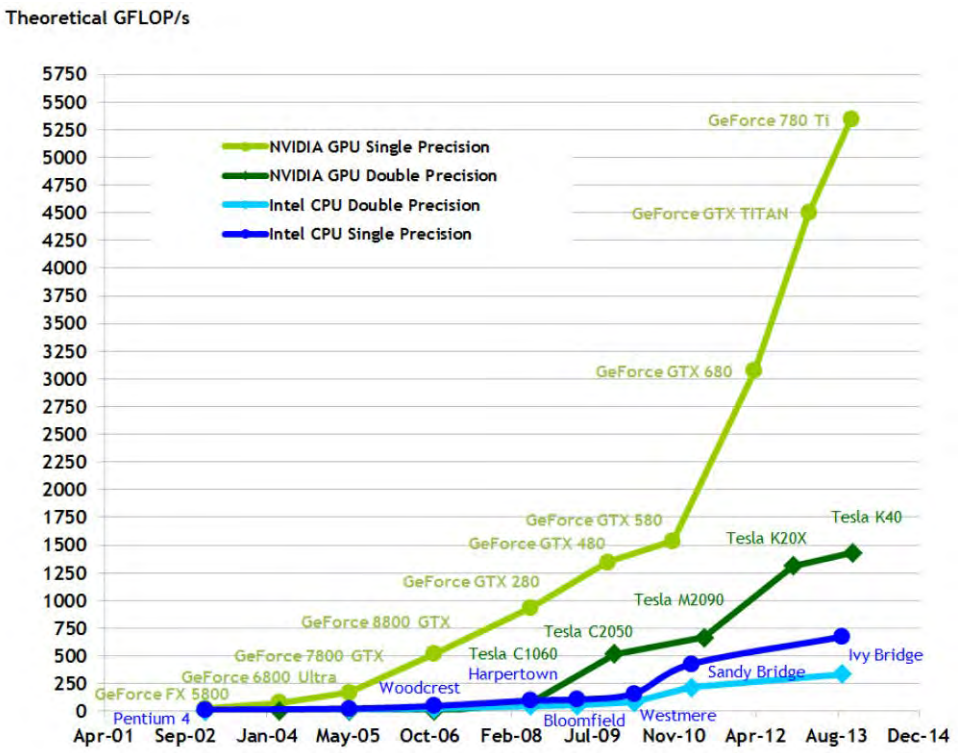

Figure 3.1: **Floating-point operations per second for the CPU and GPU** : Unlike the CPU, the GPU does not use large data caches to decrease memory latency since spatially local caches only benefit a few threads. Instead, GPUs use fast context switching and massive multi-threading to hide the performance gap between on-chip memory access [Tun11]. Note the number of GFLOPs are theoretical maximums and output GFLOPs is dependent on the problem at hand and optimisations. These are usually limited to memory bandwidth as memory accesses are very slow relative to floating point operations on the GPU. Credit: [nVi14]

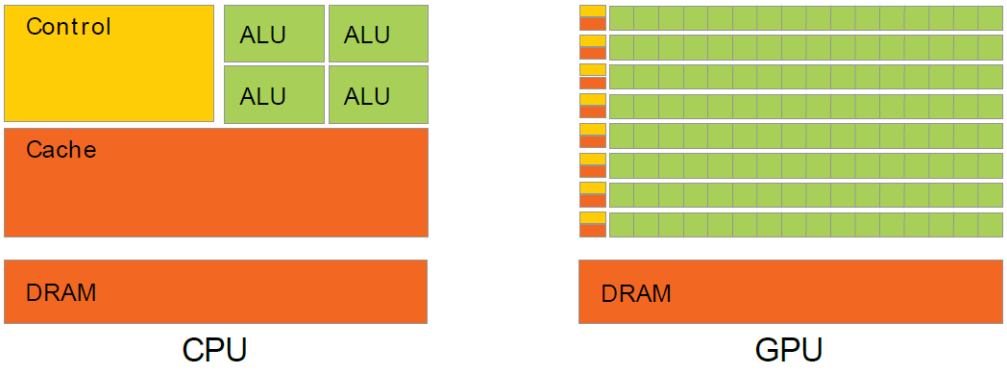

Figure 3.2: **Comparison of CPU and GPU architectures** : The need for highly parallel computation in the form of graphics rendering resulted in a design (GPU) with more transistors for data processing rather than data caching and flow control. Credit: [nVi14]

point calculations per second as possible [Woo14]. However, focusing design on this type of architecture to accomplish a specific goal typically causes disadvantages in other areas. Consequently, although GPUs perform well in many areas, some tasks are better suited to CPUs. Most successful applications use both CPUs and GPUs to accomplish acceleration, executing sections of the code that are not amenable to parallelization on a CPU (such as frequent memory access conflict components), and numerically intensive parallelizable parts on the GPU (such as independent calculations with few or no data dependencies).

3.2.1 General purpose processing using Graphics Processing Units

Initially GPUs could only perform fixed rendering operations and specific 3D geometry transformations [Bax13]. They were later extended into programmable *shaders*, which enabled programmers to control the GPU via OpenGL or DirectX API calls [Pee14]. Thereafter, APIs such as Cg, GLSL, HLSL enabled shaders to be written in a high level, C-like language. Although initial tests showed significant performance gains when shader processors were deployed to perform general purpose computing on the GPU, it forced programmers to have an understanding of the latest graphics hardware and express their algorithms in terms of graphics primitives, i.e. polygons or textures [Tun11]. Furthermore, programmers had no control over memory allocations as they had little or no access to the underlying memory hierarchy of shader GPUs [IBH04].

Much effort was invested to abstract the GPU programming model so that it no longer needed a graphics API. This resulted in the development of the *BrookGPU*, an early influential attempt to enable general purpose computing on GPUs. *BrookGPU* facilitated the use of GPUs as streaming processors by using *streams*, *kernels* and *reduction* operators (explained in Section 3.3) [IBH04]. These allow programmers to write their programs in Brook, which compiles and runs on any hardware that is compatible.

The increasing need for more programmable GPUs spawned new graphics programming APIs that exposed GPUs at a low level, bypassing the graphics programming APIs. Vendors of GPUs such as nVidia and ATI developed architectures that no longer restrict developers to a fixed function pipeline. CUDA from nVidia and CTM from ATI allowed general computation to be deployed onto GPUs [nVi14]. More general approaches such as OpenCL focused on general computation on any GPU regardless of vendor [ope09a]. With GPUs becoming ubiquitous in high performance computing and a number of new programming interfaces, the number of general purpose applications ported to GPUs have increased, resulting in performance benefits across many fields and applications.

3.3 CUDA Programming Model

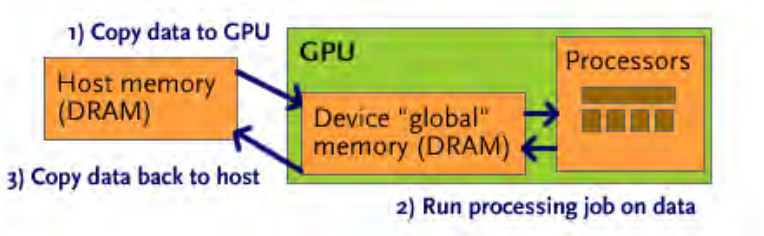

Figure 3.3: ***CUDA programming model**: Once preprocessing has been completed on the CPU, data is copied to the GPU. Kernels are thereafter executed on the GPU and, once completed, the results are copied back to the CPU.*

The purpose of this section is to introduce and differentiate CUDA programming practices from conventional programming for both single-core and multi-core processors. Ultimately, increasing the performance of an application is the reason for using GPUs. Most of this section is dedicated to strategies for optimising a CUDA program and understanding the memory architectures of the GPU. These concepts are pivotal to exploiting the full capabilities of a GPU and achieving maximum performance.

3.3.1 Hardware specifications

The hardware we have chosen to run our GPU implementation is the GTX670, and will be used as an example for this chapter. Some of the important hardware specifications are given below:

- GPU: GTX670
- Number of cores: 7
- Warp size: 32
- Shared memory: 49152bytes
- Concurrent copy and kernel execution: Yes

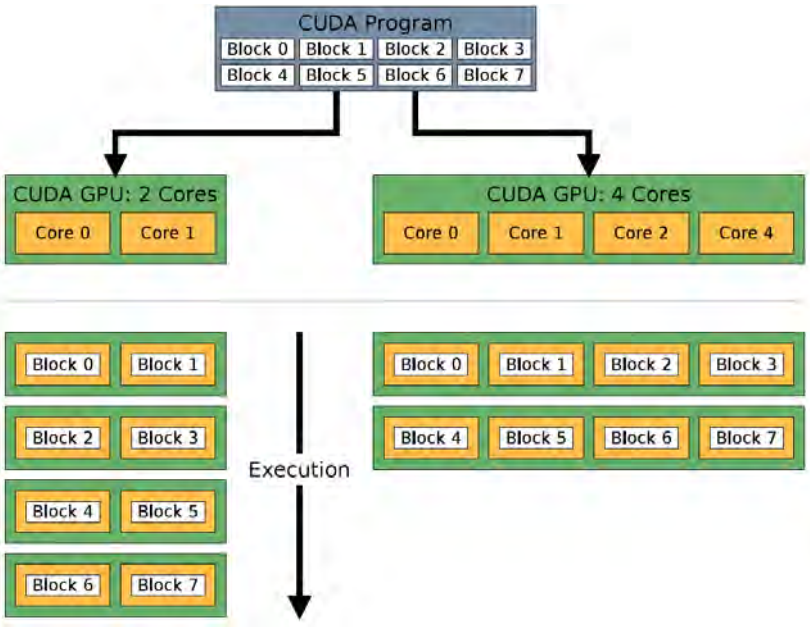

Figure 3.4: ***CUDA hardware scheduling*** *: The data parallelism of CUDA programs allows identical program executions across any number of streaming multiprocessors (SMs). Thread blocks are scheduled on the available hardware and thereby ensure that CUDA applications can execute on any CUDA capable device without modifications. A multithreaded program is partitioned into blocks of threads that execute independently from each other, resulting in GPUs with more multiprocessors automatically executing programs in less time. Credit: [nVi14]*

3.3.2 The Compute Unified Device Architecture

CUDA allows programmers to exploit parallelism on GPUs by writing straight forward C/C++ code that will potentially run in thousands or millions of invocations, or threads on the GPU. This is achievable through three key abstractions [nVi14]: a hierarchy of thread groups, shared memory and barrier synchronization. CUDA exposes these abstractions as a minimal set of

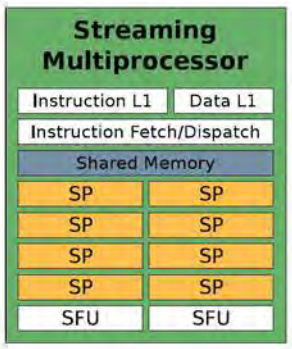

Figure 3.5: ***Streaming Multi-processor*** *: Streaming multiprocessors contain an array of scalar processors (SPs) together with instructions, data, local caches and special function units (SFU). Shared memory is also located on the SM. Shared memory provides fast memory access to threads in the same block. Each SM maps each thread to one SP to be executed independently with its own instruction address and register state. Credit: [nVi14]*

language extensions, which provides fine-grained data parallelism and thread parallelism mechanisms. Parallel thread blocks allow problems to be decomposed into smaller subdivisions that have independently solvable parts. These threads can thereafter be independently scheduled and run in parallel, which enables automatic scalability (See Figure 3.4).

Programs that are easily parallelizable will initially exhibit significant performance, gains when run on the GPU. This is because programming in CUDA requires no understanding of the GPU hardware in order to attain excellent performance because of the level of abstraction the API provides [nVi14]. However, fine tuning a CUDA program to optimise performance, requires utilizing as much of the available resources possible on the GPU. This requires in-depth knowledge of the architecture of a modern GPU in order to understand the different bottlenecks and memory constraints associated with this architecture.

Streaming Multiprocessors (SMs) create, manage and execute concurrent threads in hardware with no scheduling overhead [Tun11]. Threads executing on the same SM can communicate via shared memory and hundreds of threads can be managed by an SM and mapped to different scalar processors (SPs) (see Figure 3.5) by executing these threads in groups, called warps, of 32 parallel threads [DBK10, nVi14]. Each individual thread in a warp starts with the same initial state, but can diverge and branch out independently. Thus, warps are most efficient when all 32 threads follow the same path of execution, since divergence is handled by executing each branch serially.

Thread hierarchy

CUDA organises groups of threads into *thread blocks* which in turn are assigned to *a grid* of thread blocks (see Figure 3.6) [DBK10]. Thread blocks are collections of threads processing spatially-local data and can have up to three dimensions (recall that thread blocks are also split into warps of 32 threads). These thread blocks are assigned to a grid of up to two dimensions and thus can be distributed as work load units onto the GPU. For example, the nVidia GTX670, thread blocks are limited to a maximum number of 1024 threads per block. Each dimension of the block is limited to (1024,1024,64) for its respective x, y, z co-ordinates. Therefore, the GTX670 can schedule up to $1024^3 \times 64$ threads. Unlike the CPU, which would thrash when this many threads contend for the processor, the GPU is specifically designed to handle massively multi-threaded usage.

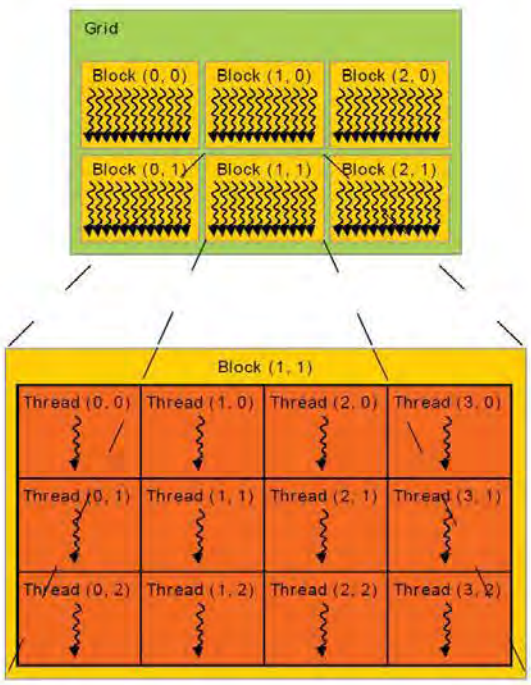

Figure 3.6: ***Thread Hierarchy*** *: CUDA threads are organised into 2 dimensional blocks, which belong to a two dimensional grid. Credit: [nVi14]*

Kernels

In CUDA, a kernel function specifies the code to be executed by all threads during a parallel phase [DBK10]. Since all threads execute the same code, CUDA programming becomes an instance of single-program, multiple data (SPMD) parallel programming style. Since all threads in a grid execute the same kernel function, they rely on unique coordinates to distinguish themselves from each other and to identify the appropriate portion of the data to process. The motivation behind using this kind of abstraction, is that it provides a means to decompose and efficiently map complex algorithms to the GPU [Far11].

These coordinates can be accessed by the kernel via *blockId.x* (and *blockId.y* for 2 dimensional blocks) for block index and *threadIdx* for thread index. These are built-in, pre-initialized variables and can be combined with *blockDim* (dimensions of thread blocks) and *gridDim* (dimensions of grid) to uniquely index data access. Below is a snippet of a possible definition of threadID:

$$threadID = blockID.x * blockDim.x + threadIdx \tag{3.1}$$

where (illustrated in figure 3.6) blockID.x and threadIdx are unique identifiers for their respective block and thread.

The CUDA programming model assumes that CUDA threads execute on a physically separate device that operates as a coprocessor to the host [Tun11]. For a given program, only the kernel is executed on the GPU whereas the rest of the program is executed on the CPU. CUDA also assumes the host (CPU) and the device (GPU) have separate memory, referred to as *host memory* and *device memory*, respectively.

The nature of the thread hierarchy, specifically thread blocks and grids, allows CUDA programs to implicitly loop over elements in an array. Serially, a 1D thread block corresponds to a single *for loop*, and a 2D thread block to a nested loop. Threads within the same block are able to communicate with each other via shared memory. Shared memory can be seen as an explicit

Type	Location	Cached	Access	Scope	Lifetime	Characteristics
Register	On-chip	—	r/w	Thread	Thread	—
Local	Off-chip	No*	r/w	Thread	Thread	Used for registry spills
Shared	On-chip	—	r/w	Block	Block	Shared by threads in a block
Global	Off-chip	No*	r/w	Global	Application	Coalesced access
Constant	Off-chip	Yes	r	Global	Application	—
Texture	Off-chip	Yes	r	Global	Application	1D/2D/3D Spatial Caching

Figure 3.7: **CUDA memory hierarchy** : Register and Shared memory have very fast, on-chip access but are limited in size. Global, constant and texture Memory are all off-chip, which is consequently slower but are larger. Local memory is utilized when an over-allocation occurs for register memory. Source: [Bax13]

low-latency cache to each SM, much like L1 cache on a conventional CPU [Tun11].

Occupancy and latency

Although a large number of threads can be scheduled for the GPU, a relatively lower number 2048×7 (2048 for GTX670 for each of its 7 SMs) can be concurrently executed. In practice, however, only a fraction of these threads are actively processing at a given time. Occupancy is defined as the ratio of the number of active warps per multiprocessor (SM) to the maximum number of possible active warps [nVi14]. Occupancy is a measure of how well threads are able to hide the latency of comparatively slower memory transactions. A lightweight scheduling system can efficiently switch thread context/warps, ensuring that latency is hidden and the GPU is kept as busy as possible.

Thread block sizes should be tuned to achieve an optimal balance between speed and occupancy. Although maximizing occupancy is not the goal of optimizing a GPU program (as higher occupancies may lead to lower performances per thread), it serves as a useful metric since occupancy corresponds directly to latency hiding.

Utilizing register and shared memory can drastically increase the occupancy of kernels, and in turn, increase the performance of an application. However, register and shared memory are relatively small, and should be software constrained accordingly to avoid register spilling. Data that cannot fit into shared or register memory, "spills" into global memory, which can be detrimental to application performance.

3.3.3 CUDA memory

In addition to a programming model, CUDA also offers a variety of memory types, each with its own limitations. Correctly utilizing the most suitable memory is critical for latency hiding and ultimately application performance. During execution, each thread has access to its own private local memory. Each thread in the same thread block has access to shared memory and all threads from all blocks have access to global, constant and texture memory. This is summarized in Figure 3.7.

Memory optimisations have the largest impact on application performance [nVi14]. Specifically, the goal of memory optimisation is maximizing memory bandwidth by utilizing as much (fast) on-chip memory as possible and as little (slow) off-chip as possible. Understanding the characteristics of the different memory types and how they can be utilized by threads is critical to achieving memory optimisations.

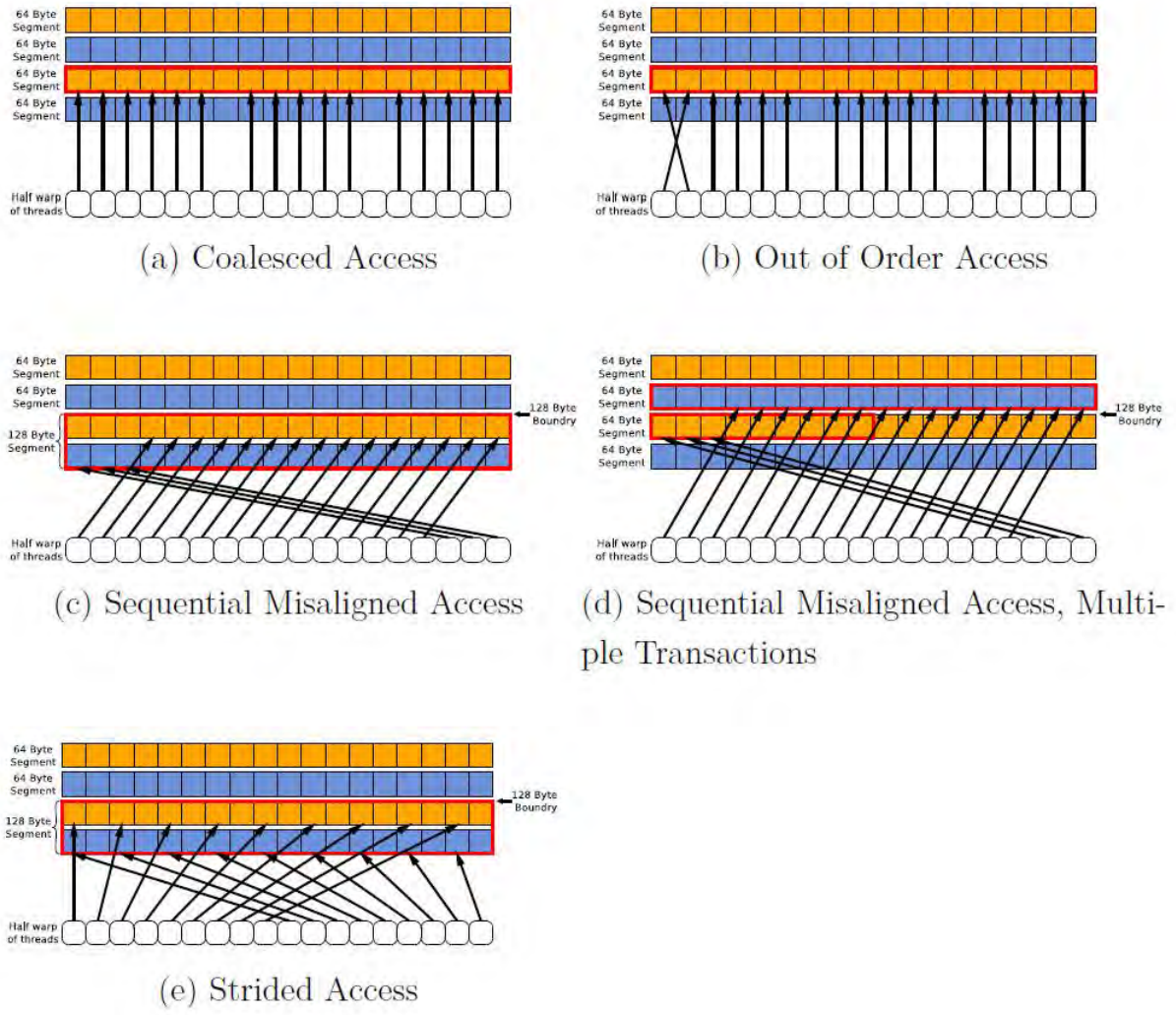

Figure 3.8: **Memory coalescing** : Coalesced global access is pivotal to achieving high memory bandwidth. Since global memory is partitioned into 32, 64 and 128 byte addressable segments, it is advantageous for threads to access consecutive memory addresses. This minimizes the number of memory transactions which have high latency. Source: [Tun11]

Global memory

Global memory, synonymous to system RAM, residing in device memory (DRAM), is the most commonly used memory as it has the greatest capacity (2GB for GTX670). However, accessing global memory is extremely slow relative to other memory types. This high latency access can be slightly alleviated through *memory coalescing*, a memory distribution technique whereby consecutive threads are able to access consecutive memory addresses (see figure 3.8)[nVi14] and load in a single operation. Although global memory is where most of the data resides initially, sub-data can be assigned to other memory types, such as shared memory, to reduce latency.

Shared memory

Shared memory resides on the SM and can be accessed by all threads in a block. Shared memory is relatively small (49KB for the GTX670) but offers very low latency (around 100x faster compared to global memory). Since shared memory is on-chip, it naturally has lower latency compared to global memory. Furthermore, shared memory also provides high bandwidth by dividing memory into equally-sized memory modules, called banks, which can be accessed simultaneously. Any memory request (read or write) consisting of n addresses that fall in n

distinct memory banks, can therefore be accessed simultaneously, yielding an overall bandwidth that is n times higher relative to a single module. Programs that suffer from global memory latency when accessing the same data multiple times can vastly increase performance by utilizing shared memory. However, if two memory requests fall in the same bank, a conflict occurs and is resolved serially, thereby increasing latency [nVi14].

Registers and Local memory

For the GTX670, every SM on the GPU houses 65,536 32-bit registers, which are allocated to threads when the kernel is launched [nVi14]. Registers are fast and store local variables during the runtime of the application, specifically during execution of threads. Registers are considered to have zero latency when used optimally, that is, when no bank conflicts or read-after-write dependencies occur. The most critical performance consideration for registers is to avoid storing more variables than can be contained in registers, per thread block. This causes register values to "spill" into local memory, which is located off-chip and consequently results in a 2 order of magnitude read/write latency compared to registers.

3.3.4 Single precision vs. double precision

Although current GPUs can perform both single and double precision computations, single precision computations are significantly faster than double precision computations. The limitations of GPU floating point accuracy stem from GPUs being designed to favour speed over accuracy. The penalty of performance for double precision calculations limits performance as double precision computations take approximately 8 times longer to compute [Far11]. Due to the nature of the bispectrum algorithm, the SNR differs significantly from each time-step implying single precision on the GTX670 is sufficiently capable of producing correct results.

3.4 Optimisations

Generally, naive first prototypes of GPU implementations will have marked speed-ups over a multi-threaded CPU implementation, provided the program is well suited for parallel computing. However, several optimisation techniques allow GPU implementations to be tweaked to fully utilize the computational powers of the GPU. These careful optimisation strategies require special considerations and a thorough understandings of the CUDA hardware and execution model. The subsequent sections detail the different methodologies used to optimise GPU implementations.

3.4.1 Parallelising techniques

Although understanding GPU hardware characteristics and their influences toward application performance provide insight into optimisations, algorithmic design are far more important [nVi14]. Choosing the manner in which data or instructions are divided into parallel components is the most important task above any other optimisation techniques. That is, maximizing the ratio of parallel to sequential code to exploit the computational capabilities of any high performance device will potentially exhibit the most performance gains. This is done by instruction-level and data-level parallelism.

Instruction-level parallelism

Instruction-level parallelism is a measure of how many operations can be executed simultaneously. Specifically for GPUs, this refers to how many instructions the kernel can execute simultaneously. Data dependencies are one of the factors that prevents instruction-level parallelism but in some occurrences can be rectified by restructuring the order of code execution as

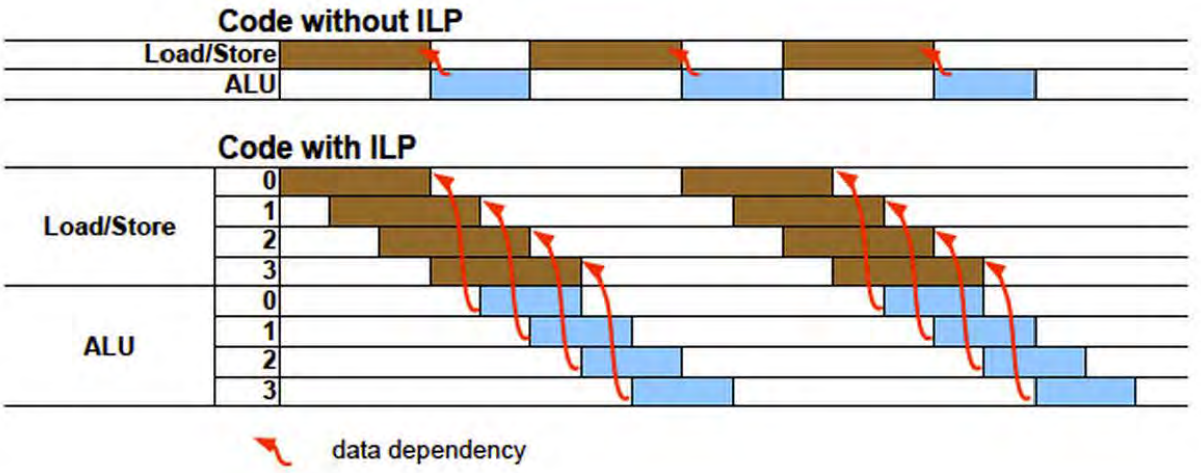

Figure 3.9: ***Instruction level parallelism:*** *More operations can be completed in the same number of clock cycles by rearranging the order of instructions to minimize the total waiting time caused by data dependencies. Restructuring the order of code can greatly improve performance. Credit:[Lam13]*

shown below:

$$C = A + B \tag{3.2}$$
$$E = C + D \tag{3.3}$$
$$F = A + D \tag{3.4}$$
$$G = A + F \tag{3.5}$$

In the pseudocode above, processing equation 3.3 requires equation 3.2 to be processed first. However, by swapping equation 3.4 and 3.3, the value of F can be computed in parallel to the value of C, resulting in a reduced number of clock cycles to execute all four lines of code. The above example is illustrated in figure 3.9 on a larger set of code. These forms of optimisations are controlled by the programmer, and are imperative towards maximizing occupancy.

Data-level parallelism

In the context of parallel computing, data-level parallelism refers to a set of instructions for a multi-processor system that can operate on multiple-data elements in parallel. There are many methods to accomplishing this, but determining which method that will be most efficient requires creativity and understanding the specific application at hand [Woo14]. For example, interferometric data are typically composed of many independent components, i.e., separate baselines, different frequency channels, large number of time-steps. In particular, when calculating the mean of bispectra (see Section 10), each individual bispectrum can be calculated in parallel to frequency channels and baselines and thereafter summed and averaged to determine the mean. More generally, interferometric data is well suited for data-level parallelism.

3.4.2 Asynchronous transfer

In figure 3.7, different GPU memory types were analysed according to their bandwidth capabilities, latency and capacity. The largest communication overhead, however, occurs when transferring data between the host (CPU) and the device (GPU). The key to hiding this transfer latency is overlapping computation on the GPU with the data transfer. This is achieved by calling *cudaMemcpyAsync()*, known as asynchronous transfer. This allows the programmer to invoke *streams* or asynchronous job queues whereby program control is returned to the host

once data transfer has been initiated. nVidia even recommends using kernels that are of no performance benefit if it means minimizing this host-device bottleneck [Tun11].

3.4.3 Memory coalescing

Memory coalescing refers to memory transactions in which consecutive threads access consecutive memory addresses. Since all threads in a warp execute the same instruction, when a load or store instruction is called, the hardware detects whether or not the threads access consecutive memory locations. Since global memory is partitioned into 32, 64 and 128 byte addressable segments, accessing consecutive memory addresses allows a single consolidated access. This minimizes the number of memory transactions, which in turn, improves memory access latency (see figure 3.8). For example, by flattening a two-dimensional array that contains baseline visibilities with their corresponding time steps, there are typically two ways to do this. Either all the baselines occupy consecutive indexes in the array as shown below:

$$array = [t_0b_0, t_0b_1, t_0b_2, ...t_1b_0, t_1b_1...] \tag{3.6}$$

or alternatively, the same baseline for each time step an occupy consecutive indexes shown below:

$$array = [t_0b_0, t_1b_0, t_2b_0, ...t_0b_1, t_1b_1...] \tag{3.7}$$

This is particularly important for algorithm 4.2 (in Chapter 4), where the average over the same baseline is calculated. Flattening the two dimensional array using equation 3.7, memory accesses are coalesced when calculating the mean of the baselines.

3.4.4 Occupancy

Low occupancy (defined in section 3.3.2) is typically caused by a low number of resident thread blocks on the SM and therefore limits the number of warps that can be scheduled and utilised by an SM's warp scheduler. The maximum number of warps per SM is limited by shared memory limitations, register sizes and most importantly (and controlled by the programmer), thread-block size.

Although there are no simple rules to specify thread and block size [Bax13], these parameters usually have a dramatic influence on GPU performance. Utilizing larger thread blocks puts strain on the limited memory capacity of register and shared memory as they are divided amongst more threads. This may reduce occupancy and can also lead to *register spilling*. In contrast, utilizing smaller thread blocks means the SM might be under-utilized and consequently exposed to the relatively high latency of global memory accesses.

It is possible to iteratively experiment with different combinations of thread and block sizes to find the optimum solution. Two main considerations should be taken into account:

- The number of threads in a block should be a multiple of the warp size, otherwise the last warp in each block will under-utilize the SM.
- More blocks with lower numbers of threads are usually better to occupy all the SMs as this ensures the grid has enough blocks to allocate to each SM.

3.4.5 Shared memory

At the finest level of GPU optimisation, minimizing the use of global memory can be accomplished by explicitly caching data via shared memory. This replaces global access latency with a substantially lower shared memory access latency. However, shared memory should only be

utilized in cases where same memory banks are accessed multiple times (see section 3.3.3) since the amount of overhead copying data from global memory to shared memory should not exceed the decreased memory access latency.

3.4.6 nVidia Visual profiler

Many tools have recently become available to assist developers in analysing CUDA applications. The nVidia visual profiler generates a time-line for the lifetime of the program and enables simple detections of bottlenecks that the programmer can thereafter address. Furthermore, occupancy statistics are also shown allowing the programmer to analyse the different effects of *load balancing*, i.e. the process of iterating through different numbers of thread blocks and thread block sizes to achieve optimum application performance.

3.5 Summary of chapter

In this chapter we introduced GPUs and their applicability to the computational demands of radio astronomy. Thereafter, we detailed the CUDA hardware and execution model focused particularly on an understanding of optimisation techniques. We also described different CUDA memory types and the general guidelines of utilizing these memory types. Lastly, some of the most critical optimisation strategies were discussed and general guidelines for these strategies were explored.

Chapter 4

Bispectrum Algorithm

This chapter provides an overview of the bispectrum algorithm and the different approaches that were explored, not only to optimise the efficiency of the algorithm, but also to improve the sensitivity of the bispectrum algorithm to transient radio signals. We also describe the key design choices that will later influence our multi-threaded CPU and GPU implementations. Finally, we conclude the Chapter with preliminary validations of our approach using simulated and real data.

4.1 Description of Algorithm

To use the bispectrum (as discussed in Section 2.3) for transient detection, we make the fair assumption that the signal-to-noise-ratio (SNR) of the bispectrum is directly related to the brightness of sources in the field of view (see section 2.3.1). The bispectrum algorithm used to detect pulses consists of the following steps [LB11]:

1. Load the dataset: The interferometric visibilities must be read from the data files provided, which are stored as measurement sets (MS) and loaded into appropriate data structures for processing.

2. Subtract the mean visibility in time: Differencing visibilities removes all constant emissions over a pre-defined time scale.

3. Dedisperse visibilities: The frequency-dependent arrival time of the transient (due to dispersion by cold plasma along the line of sight [Kok13]) must be corrected to maximize the SNR per baseline before calculating the bispectrum.

4. Calculate the mean and standard deviation of bispectra: The mean of the bispectra gives a single value related to the significance of the transient, while the standard deviation is related to the source's spatial structure. Both the mean and the standard deviation have dimensions of DM (dispersion measure) and time [LB11], so they can be used to set thresholds just as for a single-dish transient search.

5. Calibrate and image candidate pulses: Bispectra can be used to determine candidate time steps to image, but calibrated visibilities are needed for an accurate localization [ART01]. This step is not implemented since it is beyond the scope of this thesis. Furthermore, much work on imaging calibrated data is currently being explored elsewhere [Tre13].

Algorithm 1 loadData

```
1: procedure LOADDATA(fileName, nrow, startrow)          ▷ Using PYRAP
2:     List dataArray, dataTemp
3:     dataTemp ← getMS(fileName, nrow, startrow)        ▷ numpy array
4:     dataArray ← format(dataArray)                     ▷ 3D complex float array
5:     return dataArray
6: end procedure
```

4.2 Loading Data

At the beginning of the program, PYRAP (a python library used to read MS files) provides the tools needed to load the measurement set into the Python program. The metadata for the measurement set, such as number of baselines and channel frequencies, are read separately. This metadata, located in the header of the file, allows the program to accurately navigate around a rather complex dataset.

Unfortunately, a general solution to loading MS files into a specific format of choice is not possible as different interferometers store visibility data in different forms. These forms are further complicated by dependencies such as the number of antennae, multiple spectral windows and non-observing idle antennas. For example, the Very Large Array (VLA) precedes all cross-correlation data with self-correlation data (see figure 4.1). This indexing irregularity is overcome by using a *startrow* parameter to ignore initial self-correlation data, since it is not required for forming bispectra (see Algorithm 1). Here, the choice of *nrow* (number of rows) allows loading a set number of time steps in cases where the observation is extremely long.

Once MS data has been loaded into *numpy arrays*, the visibility data is thereafter extracted and re-formed into a three dimensional list of complex floats. This data preparation allows for straight-forward, effective parallel computation when loaded into the C++ multi-threaded implementation. The structure of the visibility data is shown below:

$$dataArray = [numChannels][numTimeSteps][numBaselines] \tag{4.1}$$

where *numChannels, numTimeSteps, numBaselines* can be extracted from the MS metadata and are the number of channels, number of time steps and number of baselines in the observation, respectively. The VLA observation in Figure 4.1 is composed of 128 channels, 206 time steps and 351 baselines. In the case where there is more than one polarization, they are averaged over the sum of their squares as shown in Equation 5.2.

4.2.1 Python to C++

Unfortunately, there is no equivalent library to PYRAP for C++ and consequently the visibilities needed to be re-written into binary files in a format of choice (although it is possible to write a loader for C++, this is more convenient). Thereafter, a data loader was developed in C++ to read these binary files.

4.3 Subtract the Mean Visibility in Time

To remove constant emission from persistent radio sources in the field of view, different methods of differencing visibilities were explored. Subtracting the mean visibility in time prepares the 3D complex array for bispectrum calculation, and thereafter returns the differenced data. Three main methods were implemented and tested to determine which was the most suitable. In this

	UVW	FLAG	'LAG_CATEGOR'	WEIGHT	SIGMA	ANTENNA1	ANTENNA2	
5562	[-112.247, -1...	[2, 128] Bool...	[0, 0, 0] Bool...	[0.0196693, ...	[1, 1]	0	1	0
5563	[-3990.79, 1...	[2, 128] Bool...	[0, 0, 0] Bool...	[0.00486944,...	[1, 1]	0	2	0
5564	[-3878.54, 1...	[2, 128] Bool...	[0, 0, 0] Bool...	[0.00533648,...	[1, 1]	1	2	0
5565	[31.6621, 11...	[2, 128] Bool...	[0, 0, 0] Bool...	[0.0150321, ...	[1, 1]	0	3	0
5566	[143.909, 30...	[2, 128] Bool...	[0, 0, 0] Bool...	[0.0164739, ...	[1, 1]	1	3	0
5567	[4022.45, -1...	[2, 128] Bool...	[0, 0, 0] Bool...	[0.00407835,...	[1, 1]	2	3	0
5568	[370.54, -20....	[2, 128] Bool...	[0, 0, 0] Bool...	[0.0214765, ...	[1, 1]	0	4	0
5569	[482.787, 17...	[2, 128] Bool...	[0, 0, 0] Bool...	[0.0235364, ...	[1, 1]	1	4	0
5570	[4361.33, -1...	[2, 128] Bool...	[0, 0, 0] Bool...	[0.00582678,...	[1, 1]	2	4	0
5571	[338.878, -1...	[2, 128] Bool...	[0, 0, 0] Bool...	[0.0179874, ...	[1, 1]	3	4	0
5572	[-76.3524, -1...	[2, 128] Bool...	[0, 0, 0] Bool...	[0.0176372, ...	[1, 1]	0	5	0
5573	[35.8944, 61....	[2, 128] Bool...	[0, 0, 0] Bool...	[0.0193289, ...	[1, 1]	1	5	0

Figure 4.1: **VLA MS data format** : Example of visibility data from the VLA interferometer. All self-correlations precede cross-correlations.

chapter, we refer to these methods as: *Block subtraction*, *rolling mean* and *sliding window*. The following metrics were used to determine their suitability to the bispectrum algorithm:

- Bispectrum SNR: The best method for removing constant emission will produce the maximum bispectrum SNR.
- Potential parallelism: If bispectrum SNR is not affected, a highly parallel method that is more suitable for GPU implementation will be favoured.
- Scalability: A scalable method for subtracting the mean is required to reduce the overall amount of computation for large problem sizes.

4.3.1 Block subtraction

The initial naïve implementation for subtracting the mean, averaged baseline visibilities over every channel and time step for the entire observation. However (as shown in section 2.3.2), residual errors occur from subtracting visibilities that are proportional to the brightness of the sources in the field of view. This fringe error consideration inspired the use of blocks, whereby the mean was calculated and subtracted over a pre-defined *blockSize*, instead of the entire observation (see Algorithm 2). Recall the maximum amount of time that visibilities should be averaged over is defined by:

$$t = \frac{\lambda}{b_{max}.\omega.\theta_{FoV}} \tag{4.2}$$

Thus, for relatively short observations, the choice of block size may contain the entire observation. However, realistically, observations are substantially longer and contain long baselines. Consequently, this leads to a limitation of $blockSize < t \times timeStep$, where *timeStep* is time integration of the observation defined by $timeStep = t_n - t_{n-1}$.

Suitability

As seen in Figure 4.2, the SNR of the bispectrum is largely unaffected by the choice of algorithm to remove constant emission. Since using blocks requires a natural separation of data into time-step groupings, it is well suited for parallel computing because the block subtraction can be done

Algorithm 2 blockSubtraction

```
1: procedure SUBTRACTMEAN(dataArray, blockSize)          ▷ default blockSize = 50
2:     vector meanArray
3:     for each channel do
4:         for each baseline do
5:             for each timestep, timesteps+=blockSize do
6:                 for t in range (0, blockSize) do
7:                     meanArray[baseline]+ = dataArray[channel][time + t][baseline]
8:                 end for
9:                 meanArray[baseline] ← meanArray[baseline]/(blockSize ∗ numBaselines)
10:                for t in range (0, blockSize) do
11:                    dataArray[channel][time + t][baseline]− = meanArray[baseline]
12:                end for
13:            end for
14:        end for
15:    end for
16:    return dataArray                                  ▷ Differenced data
17: end procedure
```

independently. Although it requires a moderate number of operations, the expected performance gain from a parallel solution makes it a suitable method. Furthermore, the flexibility of using block subtraction allows the method to adapt to different interferometers. Given the various parameters in equation 2.3.2, the block size can be adjusted to avoid fringe errors.

4.3.2 Rolling mean

Algorithm 3 rollingSubtraction

```
1: procedure SUBTRACTMEAN(dataArray, dataArrayTemp, k)
2:     for each channel do
3:         for each baseline do
4:             for time in timeBlock do
5:                 avg ← 0
6:                 for n in range[-k,k], n!=0 do          ▷ dataArrayTemp padded
7:                     avg+= dataArrayTemp[channel][time+n][baseline]
8:                 end for
9:                 avg ← avg/2k
10:                dataArray[channel][time][baseline]− = avg
11:            end for
12:        end for
13:    end for
14:    return dataArray                                  ▷ Differenced data
15: end procedure
```

The second implementation termed the *rolling mean*, calculates the mean over a very short time-interval, centred about a time step T. Thereafter, the result is subtracted from T and this process iterates through all visibilities v as shown below:

$$v_2 \leftarrow v_2 - (v_1 + v_3)/2 \tag{4.3}$$

This method can be generalized to include k elements before and k elements after T as shown:

$$v_n \leftarrow v_n - \frac{(v_{n-1} + v_{n-2}... + v_{n-k} + v_{n+1} + v_{n+2}... + v_{n+k})}{2k} \tag{4.4}$$

This method has previously been suggested by Law and Bower (using k=1) [LB11] and Kokotanekov (using k=2) [Kok13].

Suitability

As with using blocks, the *rolling mean* algorithm is able to avoid fringe errors by adapting the value of k. As shown in Figure 4.2, the SNR is slightly lower compared to the other methods. Unfortunately, since the mean value is calculated over short intervals, it will also remove some transient pulse signal when subtracted, resulting in lower SNR for the bispectrum.

It can be seen from equation 4.4 that the mean subtraction on v_{n+1} relies on $v_{n-k+1}, v_{n-k+2}..., v_n$ and these values cannot be altered and written back into memory concurrently, which imposes thread synchronization and occupancy issues for a GPU implementation. Consequently, this method cannot be easily parallelized across baselines unless a shadow copy of the entire observation is also stored. This would effectively double the amount of memory needed. Furthermore, each visibility v_n is accessed *2k+1* times (see equation 4.4), which greatly increases memory access latency.

4.3.3 Sliding window

Algorithm 4 slidingWindow

```
1:  procedure SUBTRACTMEAN(dataArray, blockSize, dataElementIndex)        ▷ default
    blockSize = 50
2:      vector meanArray
3:      for each channel do
4:          for each baseline do
5:              for time in range (0,blockSize do
6:                  meanArray[baseline] += dataArray[channel][time][baseline]
7:              end for
8:              for each timeStep do                                      ▷ dataArray padded
9:                  index = dataElementIndex + timeStep
10:                 meanArray[baseline] ← meanArray[baseline]/(blockSize * numBaselines)
11:                 dataArray[channel][index][baseline] -= meanArray[baseline]
12:                 meanArray[baseline] -= dataArray[channel][timeStep][baseline]
13:                 newElement = timeStep + blockSize
14:                 meanArray[baseline] += dataArray[channel][newElement][baseline]
15:             end for
16:         end for
17:     end for
18:     return dataArray                                                  ▷ Differenced data
19: end procedure
```

This method, as with *block subtraction*, calculates the mean baseline visibility per *blockSize*. However, instead of subtracting the mean from all visibilities in the current block, the mean is only subtracted off a specified element within the block. In the special case where the centre element is chosen, it is almost equivalent to the rolling mean subtraction method, however the "*blockSize*" is always $2k + 1$ in the rolling mean. This makes the sliding window a more general

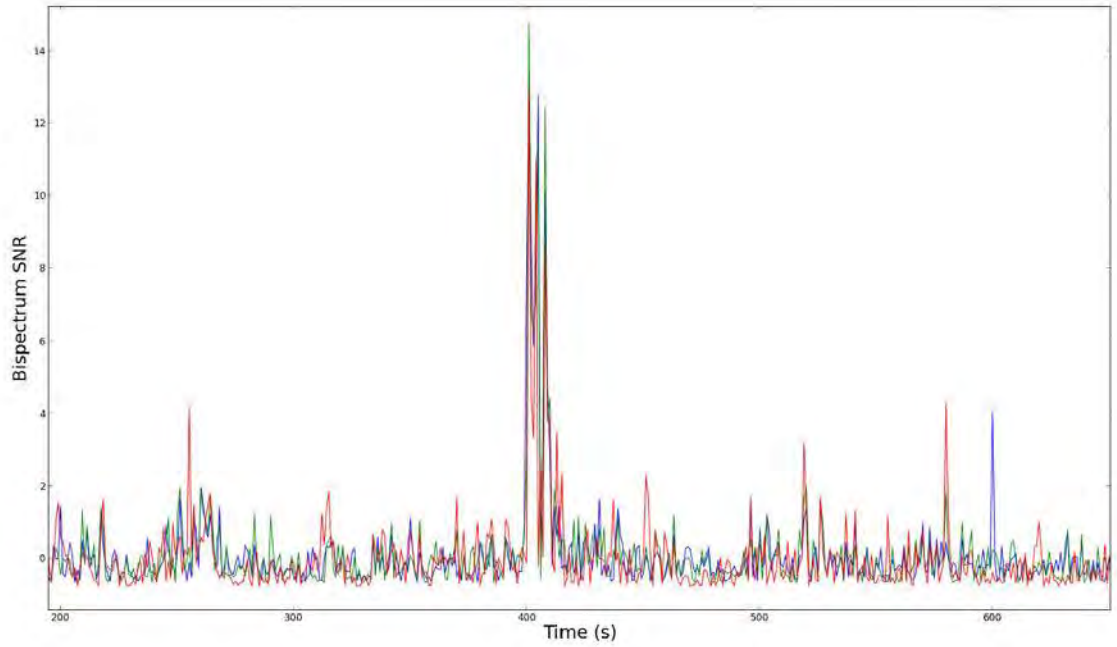

Figure 4.2: **Mean subtraction techniques** : ***Red****: Rolling mean,* ***Green****: Sliding window,* ***Blu****e: Block subtraction. As shown in the figure above, all subtraction methods successfully detect the transient at time step* $\approx$ *400. The maximum SNR is achieved using the sliding window with a 10.4% increase over the block subtraction method and 15.2% over the rolling mean method. Furthermore, it is important to note that the bispectrum SNR in the sliding window method does not exceed* 2σ *for time steps where the transient is not present. In contrast, both the rolling mean and block subtraction show bispectrum SNR signals of above* 4σ*. This motivates the use of the sliding window approach, as it is less likely to have false detections.*

solution to the mean subtraction problem.

Once the mean is calculated for the first time step, the block iteratively moves by a single step and recalculates the mean for the next element. This is done by subtracting off the first element of the previous block, and adding the last element of the new block (to avoid extra computation). This is finally subtracted off the specified element within the block.

Suitability

Once again, this method is flexible in allowing different block sizes. Similar to the rolling mean subtraction, it is not easily parallelizable as each subsequent mean calculation depends on the previous time step. However, the benefits in SNR may motivate the use of this algorithm (10.2% and 15.2% increase over rolling mean and block subtraction methods, respectively, as shown in figure 4.2).

To restructure the algorithm for GPU, the mean can be recalculated at each time step (this is detailed in Chapter 6, algorithm 13). However, the computational demands grow with *blockSize* as the number of operations required to calculate the mean scales linearly with block size. Although, it is highly parallelizable, the number of memory accesses scale up terribly to block size (*blockSize* number of look-ups per element). The use of this algorithm for a GPU implementation is motivated in section 6.1.3.

4.4 De-disperse Visibilities

Algorithm 5 dedisperse

```
1: procedure DEDISPERSE(DM, offsetList)
2:     float topChannel                          ▷ Usually reference channel
3:     float channelWidth                                          ▷ in MHz
4:     float timeStep                                  ▷ in milliseconds
5:     int numChannels
6:     float K ← 4.15 * 10^3
7:     for i in range numChannels do
8:         f1 ← 1/(topChannel)^2
9:         f2 ← 1/[top − (width * (numChannels − i))]^2
10:        offsetList[channel] ← round(DM * K * (f1 − f2)/timeStep)
11:    end for
12:    return offsetList          ▷ Data index adjustment values per channel
13: end procedure
```

Recall that the interstellar medium creates a delay in the arrival times of electromagnetic radiation. This frequency dependent arrival time can be determined by the de-dispersion equation:

$$\Delta t = 4.15 \times 10^3 D\dot{M}(\frac{1}{f_1} - \frac{1}{f_2}) \tag{4.5}$$

where Δt is the difference in arrival times of radio signals respective to two frequency channels f_1, f_2, and DM the dispersion measure.

Our initial implementation of de-dispersion (Algorithm 5 above) calculates the total time delay per channel using the top channel as the reference point. Thereafter, the time delay is expressed as a time-step offset which is stored into *offsetList*. When calculating bispectra, the offset list is parsed into the method and *dataArray* is referenced as follows:

$$dataArray[channel][timeStep + \textbf{\textit{offsetList[channel]}}][baseline] \tag{4.6}$$

Our second implementation of de-dispersion corrected the actual data in *dataArray* by shifting the array using *offsetArray* as shown in Algorithm 6:

Algorithm 6 dedisperseData

```
1: procedure DEDISPERSEDATA(dataArray, offsetList)          ▷ dataArray padded
2:     for each channel do
3:         for each baseline do
4:             for each timeStep do
5:                 dataTemp ← dataArray[channel][timeStep + offsetList[channel]][baseline]
6:                 dataArray[channel][timeStep][baseline] ← dataTemp
7:             end for
8:         end for
9:     end for
10:    return dataArray
11: end procedure
```

Although the first implementation requires de-dispersion to effectively be applied every time the data is accessed, it is not computationally expensive. Ultimately, a blind transient search requires the data to be de-dispersed over many DM values. This makes it impractical to store an

additional copy of the data for each DM. Furthermore, shifting the data to apply de-dispersion requires a large number of memory accesses (*O(n)*, where n is the total number of data points).

4.4.1 Blind searches

Algorithm 7 blindSearch

```
1: procedure BLINDSEARCH(dataArray, DMList)                    ▷ dataArray padded
2:     vector bispectrumArray
3:     n ← 0
4:     for DM in DMList do
5:         offsetList ← dedisperse(DM, offsetList)
6:         bispectrumArray[n] ←bispectrum(dataArray,offsetList)    ▷ parses offsetList
7:         n++
8:     end for
9:     return bispectrumArray
10: end procedure
```

For blind transient searches, different DM trials attempt to maximize SNR by assuming the transient is a certain distance away from the point of observation. This corrects the delay in radio signals at different distances away from the observation, as dispersion depends on this distance. Each of these DM values creates a new *offsetList* that is parsed by the bispectrum method, and represents a search of transients a certain fixed distance away. This is synonymous to adjusting the lens of an optical telescope to achieve a clear image of an object a certain distance away.

4.4.2 Linear interpolation

In Algorithm 5, line 10, we use a simple rounding function to provide an integral number by which the data needs to be shifted. In practice, however, most of these offset values contain significant decimal figures that may increase SNR if unaccounted for. This lead us to use a *linear interpolation* of dispersion (we assume here that *offsetList* has not been rounded):

Algorithm 8 linearInterpolation

```
1: procedure LINEARINTERPOLATION(dataArray, offsetList)        ▷ dataArray padded
2:     for each channel do
3:         for each baseline do
4:             for each timeStep do
5:                 offset ← round(offsetList[channel])
6:                 r ← offsetList[channel])-offset                  ▷ Remainder
7:                 dataTempA ← dataArray[channel][timeStep + offset][baseline]
8:                 dataTempB ← dataArray[channel][timeStep + offset + 1][baseline]
9:                 dataTemp ← dataTempA + (dataTempB − dataTempA) ∗ r   ▷ Interpolation
10:                dataArray[channel][timeStep][baseline] ← dataTemp
11:            end for
12:        end for
13:    end for
14:    return dataArray
15: end procedure
```

In essence, the linear interpolation accounts for decimal figures by using a straight line to approximate visibility readings between time steps. This is calculated by multiplying the

difference between two consecutive visibilities, and the decimal figures of the offset:

$$v_n \leftarrow v_n + (v_{n+1} - v_n) * R \tag{4.7}$$

where R is the decimal figures of the offset. However, this method is not suitable for GPUs, except for the condition where *DMList* is very large.

4.5 Calculate mean of Bispectra

After differencing de-dispersed visibilities, the mean of bispectra can be computed. This is achieved by firstly determining the indexes of all closed triplets in the dataset. Thereafter, using these indexes, bispectra can be formed and averaged over all channels and triplets. Calculating bispectra is the most computationally expensive component of the bispectrum algorithm, thus obtaining a potential parallelizable solution is essential for a sizeable performance gain on the GPU.

4.5.1 Triplets

Data storage conventions differ between different interferometers and even observations, which make it difficult to implement a general solution. Algorithm 9 below assumes VLA data format with all 27 antennas operational, but can be easily extended to other interferometers.

The first step towards calculating the indexes of closure triplets is to simulate the form in which baselines are stored (see Figure 4.1 for VLA data). Each row in *indexArray* contains the two antennas used to create the correlated baseline visibility. These rows map directly to the *antenna1*, and *antenna2* columns in Figure 4.1. The second step calculates all combinations of possible closure triplets and stores the antennas numbers in each *triplesArray*. Lastly, using *triplesArray* in conjunction with *indexArray*, the indexes for the baselines each triplet contains is determined and stored into *triplesIndex*. For example, the closure triplet "4-5-6", requires the baseline indexes of bl_{45}, bl_{56} and bl_{46}, which corresponds to baseline indexes *(14,20,**19**)*. Note the conjugate of b_{46} (index 19 in the example) is used for bispectrum computation.

4.5.2 Calculating the bispectrum

Notice the closure triplet "i-j-k", requires the baselines bl_{ij}, bl_{jk} and bl_{ik}. However, the definition of closure quantities (see 2.13) requires bl_{ki} instead of bl_{ik}. Consequently, the *conjugate* of bl_{ik} used to form bispectra instead as the conjugate of $bl_{ik} = bl_{ki}$.

The bispectrum is calculated using the *offsetList* to correct the data for dispersion and tripleArray to index the baselines to form closed triplets. Each bispectrum is calculated and summed into *bispectrumArray* and thereafter averaged over the total number of channels and triplets.

4.5.3 Parallelism overview

Calculations of bispectra contain a large number of independent parallelizable components, i.e. *timeSteps, channels, triplets and DM values*. The primary choice of dimension to split the data into different components depends on the size of the parameter. In general, the best parameter for long observations would be *timeSteps* as the data contains large number of time steps. It may be advantageous to use triplets for a large interferometer with many baselines and hence many triplets (VLA has 27 antennas, hence 2925 closure triplets). Typically, a blind transient search iterates over approximately 500 DM values, which could also be a possible parameter of choice.

Algorithm 9 makeTriplets

```
1: procedure MAKETRIPLES(numAntennas)
2:     int [ ] [ ] indexArray                          ▷ Stores index of baselines
3:     int n ← 0
4:     for i in range (0,numAntennas-1) do
5:         for j in range (0,i+1) do
6:             indexArray[n][0] ← j
7:             indexArray[n][1] ← (i + 1)
8:             n+ = 1
9:         end for
10:    end for
11:    int[ ] [ ] triplesArray                         ▷ Stores triplet combinations
12:    int n ← 0
13:    for i in range (0,numAntennas-2) do
14:        for j in range (i+1,numAntennas-1) do
15:            for k in range (j+1,numAntennas) do
16:                triplesArray[n][0] ← i
17:                triplesArray[n][1] ← j
18:                triplesArray[n][2] ← k
19:                n+ = 1
20:            end for
21:        end for
22:    end for
23:    vector triplesIndex
24:    n ← 0
25:    for triples in triplesArray do
26:        indexTemp ← getIndex(triples,indexArray)
27:        triplesIndex[n] ← indexTemp)
28:        n+ = 1
29:    end for
30:    return triplesIndex                             ▷ Index of closure triplets
31: end procedure
```

Algorithm 10 bispectrum

```
1: procedure BISPECTRUM(dataArray, tripleIndex, offsetList)
2:     vector bispectrumArray
3:     for each timeStep do
4:         for each channel do
5:             offset ← offsetList[channel]
6:             for each triplet in tripleArray do
7:                 A ← dataArray[channel][time + offset][triplet[0]]
8:                 B ← dataArray[channel][time + offset][triplet[1]]
9:                 C ← dataArray[channel][time + offset][triplet[2]]
10:                bispectrumArray[timeStep]+ = [A ∗ B ∗ (C.conjugate())].real   ▷ real
11:            end for
12:        end for
13:        N ← numTriplets ∗ numChannels
14:        bispectrumArray[timeStep] ← bispectrumArray[timeStep]/N)
15:    end for
16:    return bispectrumArray                          ▷ Mean Bispectra
17: end procedure
```

4.6 Calculate SNR

Algorithm 11 bispectrumSNR

```
1:  procedure BISPECTRUMSNR(bispectrumArray)
2:      float mean
3:      float sum ← 0
4:      float sigma ← 0
5:      int n ← bispectrumArray.size
6:      for bispectrum in bispectrumArray do
7:          sum+ = bispectrum
8:      end for
9:      mean ← sum/n
10:     for i in range n do
11:         sigma += (bispectrumArray[i]-mean)**2
12:     end for
13:     sigma ← sqrt(sigma/n)                          ▷ Standard deviation
14:     float SNR
15:     for i in range n do
16:         SNR ← (meanBispectra[i]-mean)/sigma
17:         bispectrumArray[i] ← SNR                   ▷ Apparent SNR
18:     end for
19:     return bispectrumArray
20: end procedure
```

The SNR is calculated by firstly calculating the mean of the bispectra across all time steps. Thereafter, using the mean, the standard deviation for the bispectrum is calculated. To compute the bispectrum SNR, we firstly normalize the bispectra around the x-axis, by expressing the bispectra as the number of standard deviations from the mean. We define bispectrum SNR to be the number of standard deviations above zero for each bispectra over the entire observation. Thus the SNR for a bispectrum at time B_t is given by:

$$SNR_{bisp} = \frac{(B_t - mean)}{\sigma} \tag{4.8}$$

where *mean* is the mean of the bispectra across the whole observation and σ is the standard deviation (also see Algorithm 11 line 16).

Although it is possible to compute SNR in parallel, the computational demands are relatively low compared to mean subtraction and computing bispectra. For GPU implementations, *bispectrumArray* can be copied from device back to host, and the SNR can be computed on the CPU.

4.6.1 Sensitivity and thresholds

The distribution of SNR for a typical simulated transient will typically take the form of figure 4.2. Most of the SNR points are below 5σ, which correspond to noise and weaker non-transient sources. According to Law and Bower [LB11], (see section 2.3.5) 5σ is a reasonably accurate threshold. That is, the algorithm deems any SNR above 5σ (or 5 standard deviations) as a likely transient detection.

4.6.2 Standard deviation of bispectra

Calculating the standard deviation of all bispectra over each time step is not only computationally expensive, but it also requires each bispectrum to be stored in memory. The amount of extra memory needed to compute the standard deviation (during bispectra calculation) scales linearly with the number of triplets. Due to these factors, this process is particularly difficult to implement on a GPU. As a result, we propose Algorithm 12, whereby the standard deviation is only computed for each candidate detection time step. Typically, this computation is needed relatively rarely. Once computed, a threshold (according to Equation 2.15) is used to determine whether the detection is a transient or RFI.

Algorithm 12 standardDeviation

```
1:  procedure STANDARDDEVIATION(dataArray, time, tripletArray, offsetList)
2:      vector bispectrumArray
3:      int n ← 0
4:      for each channel do
5:          offset ← offsetList[channel]
6:          for each triplet in tripleArray do
7:              A ← dataArray[channel][time + offset][triplet[0]]
8:              B ← dataArray[channel][time + offset][triplet[1]]
9:              C ← dataArray[channel][time + offset][triplet[2]]
10:             bispectrumArray[n] = [A * B * (C.conjugate())].real          ▷ real
11:             n++
12:         end for
13:     end for
14:     float mean
15:     float sum ← 0
16:     float sigma ← 0
17:     for bispectrum in bispectrumArray do
18:         sum+ = bispectrum
19:     end for
20:     mean ← sum/n
21:     for i in range n do
22:         sigma += (bispectrumArray[i]-mean)**2
23:     end for
24:     sigma ← sqrt(sigma/n)                                  ▷ Standard deviation
25:     return sigma
26: end procedure
```

4.7 Initial Validation

In order to validate initial implementation, we begin by using MeqTrees[1], a software package for implementing Measurement Equations (an extensive validation is detailed in Chapter 7). Meqtrees is suited to simulations of radio astronomical data, specifically transients with variable parameters. These tests are also important to validate several implementation deviations, such as different mean subtraction methods. As a result, both simulated and real data is used for the validation.

Validation of our implementation of the bispectrum algorithm using simulated data, required

[1] `http://meqtrees.net/`

a transient to be inserted into an existing MS dataset. Figure 4.3 shows that the bispectrum algorithm successfully detected the transient placed at time step 800. For the validation of real data, a VLA observation of pulsar B0355+54 was loaded and run through our bispectrum implementation. We conclude that our implementation successfully detects B0355+54 at time step 170 as shown in figure 4.4.

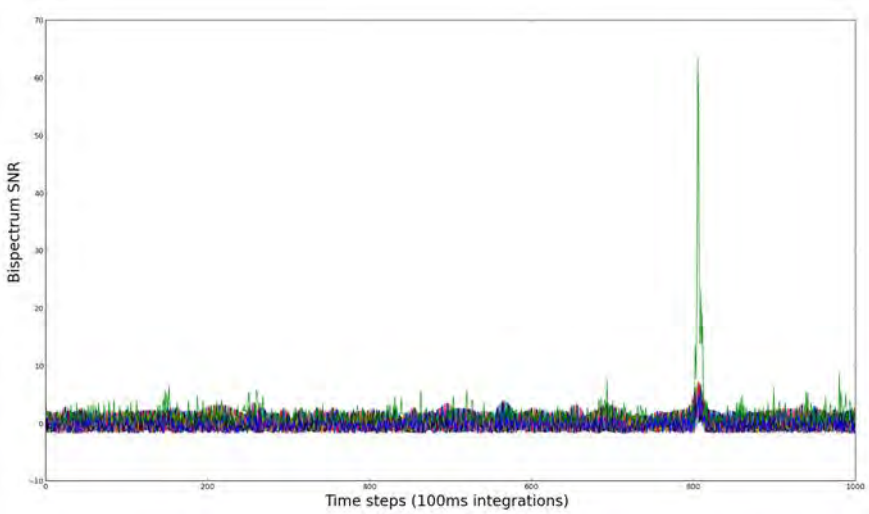

Figure 4.3: **Simulated transient** : *Green: Bispectrum SNR. Others: Baseline averages (beamforming). As shown in the figure above, the bispectrum SNR visibly responds to the transient at time step 170.*

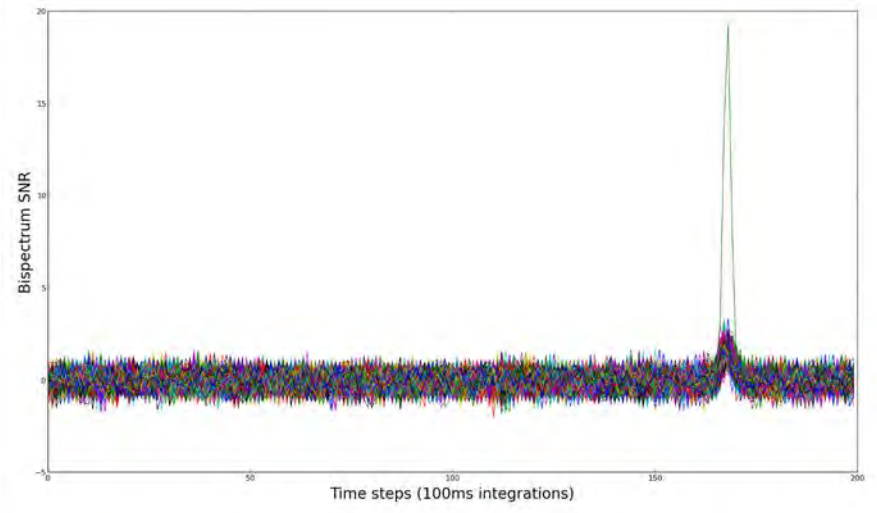

Figure 4.4: **Real VLA data of B0355+54** : *Green: Bispectrum SNR. Others: Baseline averages (beamforming). As shown in the figure above, the bispectrum SNR visibly responds to the transient at time step 800.*

4.8 Summary of chapter

In this chapter, the bispectrum algorithm, with its different design choices and algorithms is discussed in detail. Special consideration is given to potentially parallelizing the bispectrum algorithm by separating independent components that can be computed in parallel. Different

design choices are substantiated using metrics such as maximizing SNR and computational efficiency. Lastly, a validation process using both simulated and real transients aided the validation of the serial implementation.

Chapter 5

CPU Implementations

In this chapter we present various CPU implementations of the bispectrum algorithm. Firstly, a Python prototype is developed, as PYRAP (Python library for loading MS files) is required to load MS files. Secondly, a single-threaded CPU implementation in C++ is developed. Thereafter, a multi-threaded CPU version is developed to benchmark performance for the GPU implementation. Our initial multi-threaded implementation utilized P-threads to parallelize the bispectrum algorithm. However, this proved infeasible due to large thread overheads. As a result, a second multi-threaded version using OpenMP was developed.

5.1 First Python Prototype

The Python prototype has two main purposes. Firstly, it is used to ensure the algorithm is implemented correctly. This is achieved by testing the prototype with simulated data created by Meqtrees. Since transients can be placed at predefined time steps in an MS file, we test the validity of the program by checking whether the prototype successfully detects the transient at the specified time step and nowhere else. Secondly, the visibility data is written into binary files in a format of choice, as there are no existing libraries equivalent to PYRAP in C++ to load MS files.

5.1.1 Data formats and PYRAP

The python prototype begins by utilizing the PYRAP library[1] to load visibility data into the program from a specified MS file. Although different interferometers store and index their data in unique ways, they are typically loaded from the MS files into *numpy* arrays of complex numbers in the following form:

$$dataArray[x, y, z] = [numTimeSteps * numBaselines][numChannels][polarizations] \quad (5.1)$$

where the x-dimension indexes the flattened array of baselines over the entire observation, i.e. $x_0 = t_0b_0, x_1 = t_0b_1, x_2 = t_0b_2$.... The y-dimension indexes the frequency channel and the z-dimension the separate polarizations (radio telescopes can record two orthogonal polarizations [Pol15]). When there are two polarizations, namely p_1 and p_2, they are averaged over the sum of their squares (and similarly for multiple polarizations):

$$P_{new} = \sqrt{(p_1^2 + p_2^2)} \quad (5.2)$$

[1] `https://code.google.com/p/pyrap/`

5.1.2 MeqTrees

The software package *MeqTrees* is used to populate existing MS files with transients. MeqTrees provides these transients with the following relevant adjustable parameters:

- *Source flux (Jy)*: How bright or intense the observed transient is.
- *Start time (s)*: How long into the observation the transient appears.
- *Noise (Jy)*: How much background noise is added to the observation.

Adjusting the *source flux* gives an indication of how sensitive the bispectrum algorithm is and its limitations for faint transients (low source flux). Specifying the *start time* for the transient verifies whether the program is detecting it at the right time step. Finally, background *noise* (RFI) is optionally added to the observation to test whether the algorithm is resistant to local interference.

5.1.3 Program flow

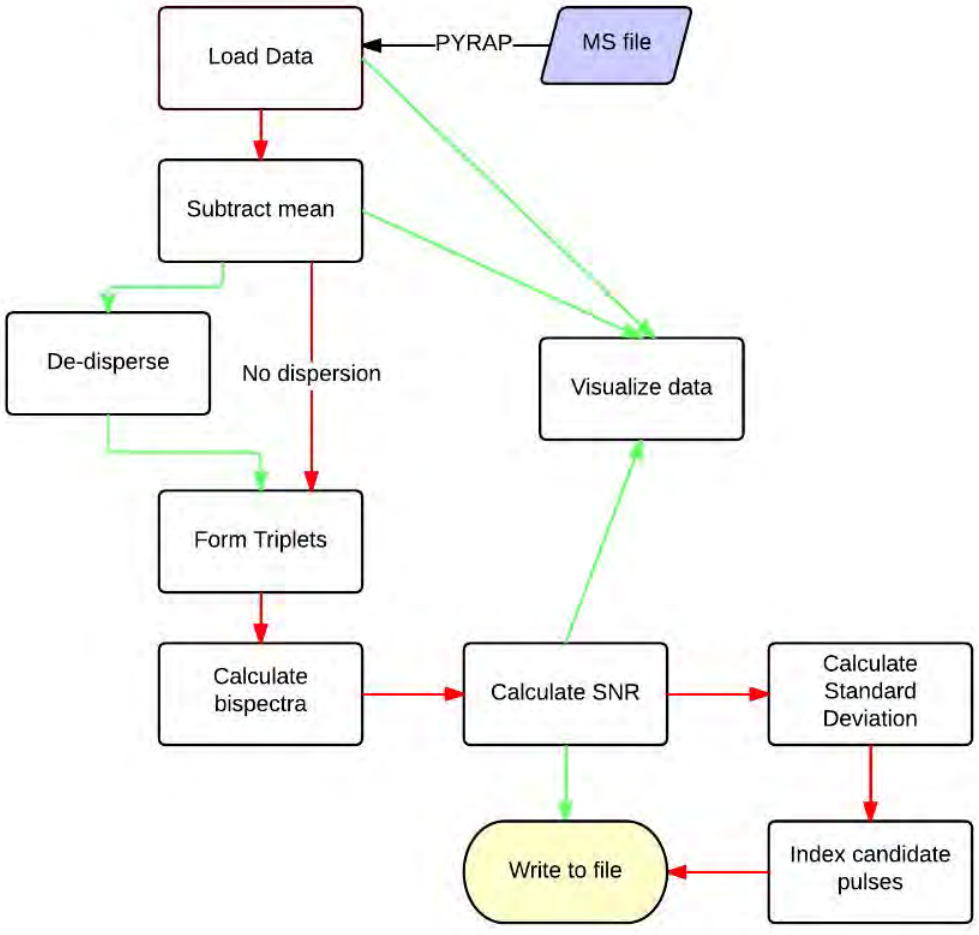

Figure 5.1: **Python flowchart**: (Flowchart showing the different components of the first Python prototype). *Green arrows: Optional non-terminating paths. Red arrows: Critical paths. These components and their algorithms are detailed in Chapter 4. Data is loaded from MS files acquired from the Karoo Array Telescope (KAT-7), the Very Large Array (VLA) or simulated data using MeqTrees. These are loaded using the PYRAP library.*

The remainder of the program executes the different components according to the critical path defined by the red arrows shown in figure 5.1. The mean visibility value per baseline,

per channel is subtracted off per time step using algorithm 4.2. Thereafter, each unique triplet is formed and the de-dispersion *offsetList* is calculated. This prepares and collects the data required to execute the *Calculate bispectra* component in Figure 5.1. Following the bispectra calculation, the SNR of the bispetra is calculated to determine the indices of candidate pulses. The standard deviation of the bispectra of corresponding candidates is calculated to distinguish detections between RFI and real sources (see Section 2.3.5). Lastly, indices of the detections that have qualified as real sources are written to file.

Visualizations of visibility data are produced at specific significant points in the program, namely: right after the data has loaded, after the mean has been subtracted and after bispectra SNR have been calculated. This allows step-by-step visual insight into how raw visibility data is transformed into bispectra SNR. It also aids the experimental phases (such as adjusting several *MeqTrees* parameters), where different methods of differencing visibilities are analysed.

The bispectra SNR from the *Calculate SNR* component is also optionally written to file. This is used to store results that may be graphed and analysed at a later stage.

5.1.4 Python limitations

The initial attempt to parallelize the bispectrum algorithm in Python raised several performance issues. Firstly, Python utilizes the *Global Interpreter Lock* (GIL), which is a *mutex* that prevents multiple native threads from executing Python code simultaneously [Bea10]. This lock is necessary as Python's memory management is not thread-safe (currently at the time of writing). Consequently, Python uses thread switching techniques to simulate multi-tasking programs that are not focused on performance. This prevents multi-threaded Python programs from utilizing any extra computational power from multi-core CPUs [Bea10]. Although there have been attempts to work around the GIL, these methods are complex and risky in terms of memory management. It is evident that, for the purpose of this work, multi-threaded implementations in a language such as c++, effectively supports multi-threading and provides a fair comparison of performance with the GPU implementation.

5.1.5 Python to C++

As there is no equivalent library to PYRAP in C++, the raw visibility data (as read by PYRAP), is written from the Python prototype to file. Although it may be advantageous to re-shape the data into a different form, i.e. separating the individual baselines from the x-dimension (see equation 5.1), this may produce biased performance comparisons. Any changes to the visibility data structure is made in the C++ program since these may contain significant memory overheads that affect performance. Although file formats are configurable (to interchange columns in visibility data) from interferometers such as the VLA, this may not be a fair assumption for interferometers in general.

5.2 C++ Single-Threaded Implementation

There are two main purposes for the single-threaded implementation. Firstly, it loads the data into the C++ program and tests that it has been correctly loaded as data inconsistencies may arise between MS files and C++ data. Secondly, it provides a performance benchmark for the multi-threaded implementation. Since the multi-threaded implementation uses OpenMP to parallelize the program, the single-threaded implementation is a required preliminary.

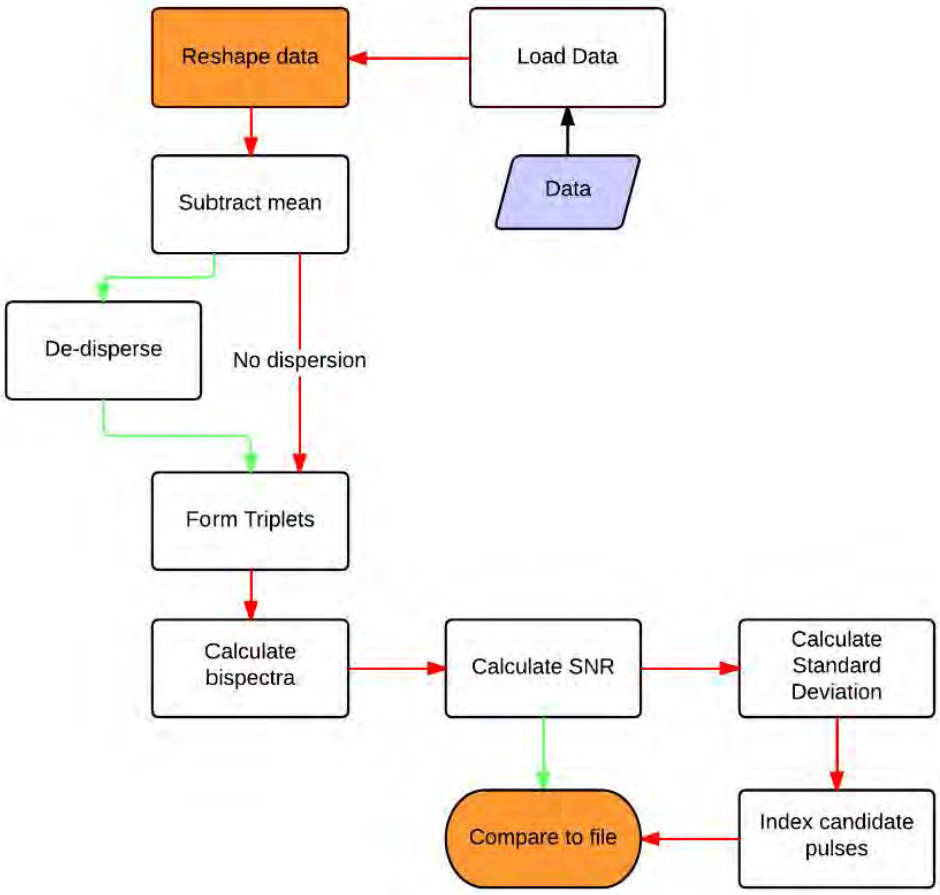

Figure 5.2: **Single-threaded C++ flowchart** : *Green arrows: Optional non-terminating paths. Red arrows: Critical paths. Flowchart showing the different components of the single-threaded C++ implementation. These components and their algorithms are detailed in Chapter 4. Data is loaded from binary file outputted by the Python prototype.*

5.2.1 Loading and reshaping data

The program firstly loads the raw visibility data by reading the data file produced by the Python prototype. This data is stored into a 3-dimensional *vector* of *complex floats*:

$$vector[x, y, z] = [numTimeSteps * numBaselines][numChannels][polarizations] \tag{5.3}$$

where the x, y and z-dimensions are equivalent to those in equation 5.1. However, due to the complex memory access patterns of the x-dimension as well as preparing the data for the multi-threaded implementation, the vector is restructured. When applicable, the polarizations are combined using equation 5.2. The time steps and baselines are separated into two columns by traversing through the vector. Since we remove one column by combining polarizations, and add another column splitting the x-dimension into two components, the resultant vector is as follows:

$$vector[x, y, z] = [numTimeSteps][numBaselines][numChannels] \tag{5.4}$$

In general, each of the columns may be shuffled accordingly to ensure data parallelism is enforced in the optimum way. That is, for short observations with a large number of baselines, it may be advantageous to swap the baseline and time step columns:

$$vector[x, y, z] = [\boldsymbol{numBaselines}][\boldsymbol{numTimeSteps}][numChannels] \tag{5.5}$$

5.2.2 Program flow

The remainder of the program executes similarly to the Python prototype which are defined by the critical paths shown by the red arrows shown in Figure 5.2. However, at the end of the program the indices of the detections that have qualified as real sources are compared to validated output files from the Python prototype.

5.2.3 Validation against Python prototype

The single-threaded implementation was initially validated by comparing the successful candidate indices in the output file with the indices outputted from the Python prototype. However, this comparison may ignore small inconsistencies that have occurred during calculations or the writing and loading of the visibility data. Therefore, the bispectra SNR is also optionally compared to the output files of the Python prototype to ensure the program will produce the correct results across all data files.

5.3 C++ Multi-Threaded Implementation

The main purpose of the multi-threaded implementation is to provide a performance benchmark for the GPU implementation. A fair performance comparison between CPU and GPU requires the full computing capabilities of the CPU be harnessed through multi-threading.

Parallelizing the algorithm on CPU provides insight into determining the different separable components that are suitable for the GPU implementation. These separable components provide different methods of parallelizing the bispectrum algorithm. The optimal choice of how to separate the data and how to separate independent computations ultimately determines how much performance gain can be expected from a parallel solution. In the following multi-threaded implementation, OpenMP (a standard API for writing parallel C++ code) is used to implement the multi-threaded C++ program.

5.3.1 Hardware specifications

Our multi-threaded implementation is executed on an *Intel Core i7-3820 CPU @ 3.60GHz*. The specifications of this CPU as follows:

- *Number of cores*: 4
- *Number of threads*: 8 (hyperthreading)
- *Prcessor base frequency*: 3.60GHz
- *Price*: $305

Although this CPU can concurrently run 8 threads, this is due to *Hyper-threading*: a technology used by some Intel microprocessors that allows a single microprocessor to appear as two separate processors to the operating system [hyp14]. Therefore, in terms of performance, we expect a theoretical maximum of 4x speed up on the CPU, and not 8x. The total number of threads utilized in the multi-threaded implementation should be a multiple of the number of cores to maximize performance [TM11].

5.3.2 OpenMP overview

OpenMP is a standard API for writing parallel, multi-threaded code for C/C++ and Fortran [TM11]. It is used in this context to parallelize C++ code, and hence the bispectrum algorithm. Figure 5.3 shows the openMP programming model for parallelizing different components of C++

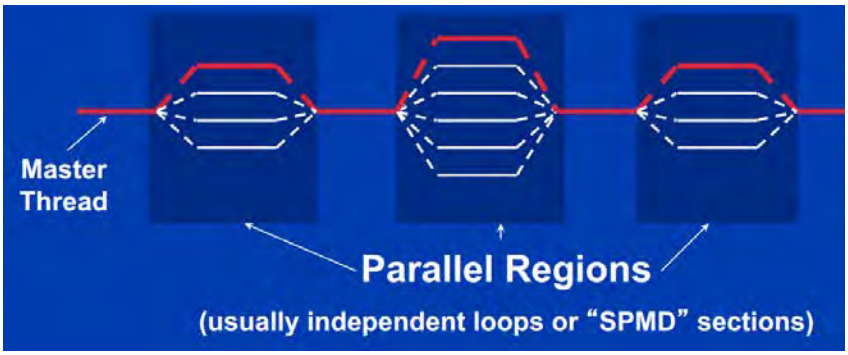

Figure 5.3: **OpenMP programming model** : *The master thread spawns a team of threads needed to parallelize the program. It uses a Fork-Join technique where parallelism is added incrementally, and thereafter joined back into the serial components of the program. Credit: [TM11]*

code.

OpenMP is used to parallelize specific components (or blocks of code) of a program by using compiler directives before loops. This is achieved as complier directives split loops into separate components, and thereafter, multiple threads are spawned and assigned to these components and these are executed by threads on separable CPU cores in parallel. As a result, performance gains are obtained that scale to the number of CPU processors. Specifically, the bispectrum algorithm is well suited to openMP due to the vast number of loops and nested loops required to prepare data and compute bispectra SNR. Although openMP implementations typically require synchronization techniques to protect data conflicts caused by uncoordinated access to shared data, interferometric data is highly separable and can be modified to avoid these data conflicts. In the event that data conflicts cannot be resolved, openMP allows the programmer to synchronize threads. However, thread synchronizations are relatively expensive and should be avoided where possible [TM11].

One of the main advantages of OpenMP is the relatively little programming effort required to parallelize code. In C++, this is achieved by using *pragmas*, which precede loops that need to be parallelized. This is illustrated in the code snippet below:

```
OMP_NUM_THREADS = 5;
int n = 1000;
#pragma omp parallel for
for (int i=0;i<n;i++)
   c[i] = a[i] + b[i];
```

In the above example, iterations in the *for loop* are sub-divided and executed by 5 independently executed threads. These sub-divisions are illustrated in figure 5.4.

5.3.3 Program flow

The remainder of the program executes similarly to the single threaded implementation, following the critical paths defined by the red arrows shown in figure 5.5. However, the computationally expensive components of the algorithm (highlighted in light blue in figure 5.5) have been parallelized using openMP. The mean value per baseline or channel (see code Snippet A and Snippet B), is subtracted off per time step by assigning each channel or baseline to a separate thread. Since each channel and baseline is independent, threads need not be synchronized to avoid race

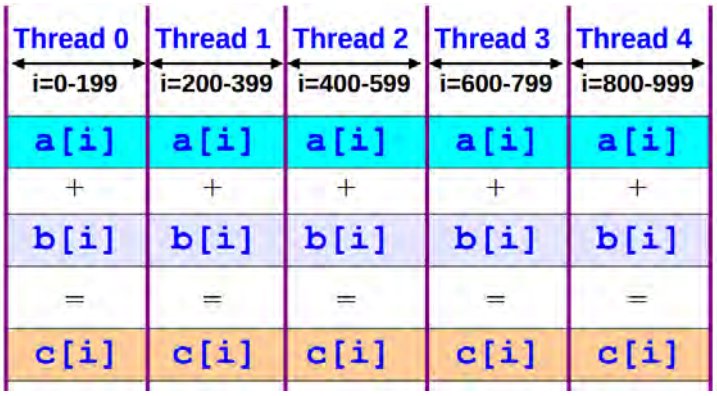

Figure 5.4: **OpenMP thread work distribution** : *Example showing how OpenMP initializes 5 threads that executes in parallel. Threads in this parallel region are assigned to different subdivisions of the loop. Credit: [ope09b]*

conditions. The code snippet below shows the two possible ways to parallelize the *Subtract mean* component:

```
//Snippet A
OMP_NUM_THREADS = numChannels;
#pragma omp parallel for
for (int channel = 0; channel<numChannels; channel++)
{
   for (int baseline = 0; baseline < numBaselines; baseline++)
   {
      //DO subtractMean(data,channel,baseline);
   }
}
```

```
//Snippet B
OMP_NUM_THREADS = numBaselines;
#pragma omp parallel for
for (int baseline = 0; baseline < numBaselines; baseline++)
{
   for (int channel = 0; channel<numChannels; channel++)
   {

      //DO subtractMean(data,channel,baseline);
   }
}
```

Snippet A shows how the code block is parallelized by assigning a thread to each channel. In contrast, snippet B shows how the code block is parallelized by assigning a thread to each baseline instead. For our multi-threaded implementation, by default, the code is parallelized by using the former. Interferometric data typically have either 64, 128, 256 or 512 channels (powers of 2) which are suitable for a 4-core CPU. The number of baselines are generally not a multiple of 4, making it a slightly less favourable choice.

The *Calculate bispectra* and *Calculate SNR* component is also parallelized. Similar to *mean subtraction*, this is achieved by assigning a thread to each channel. Thereafter, indexes of the detections that have qualified as real sources are compared to validated output files from the single threaded implementation. This is to ensure the correct results are produced as incorrect results may occur after parallelization.

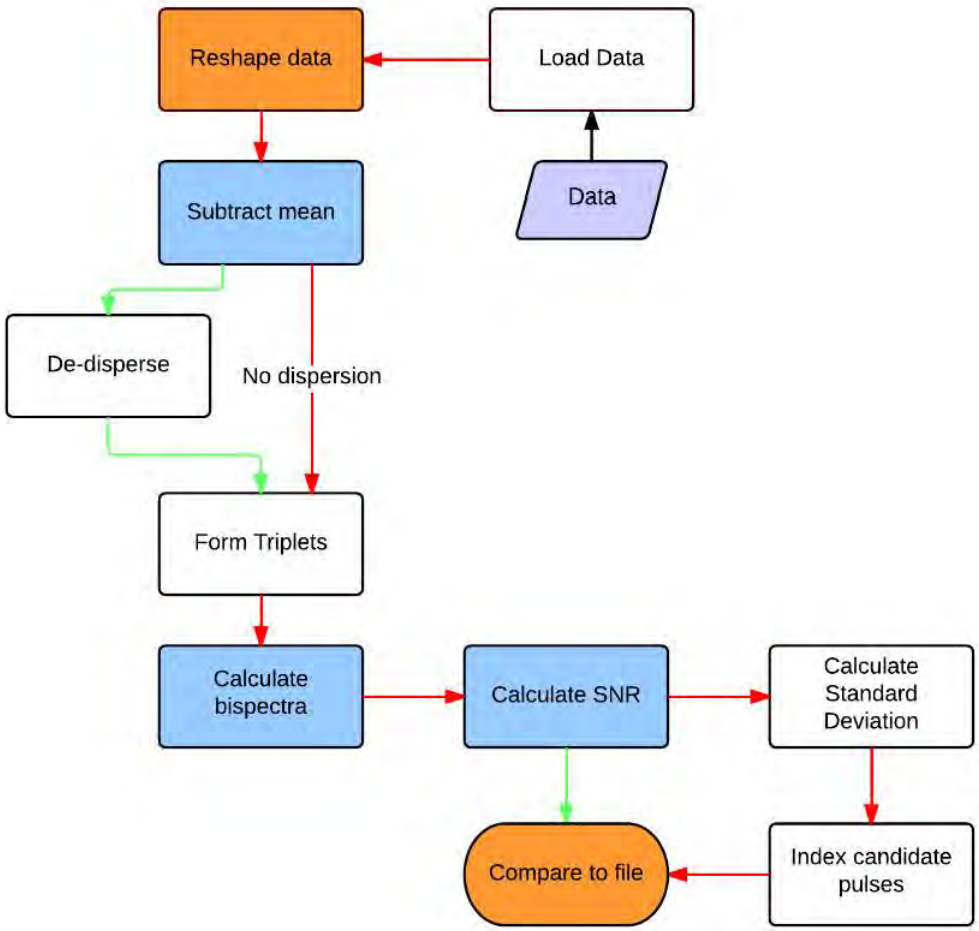

Figure 5.5: **Multi-threaded flowchart** : *Green arrows: Optional non-terminating paths. Red arrows: Critical paths. Flowchart showing the different components of the multi-threaded implementation. The components filled with light blue are parallelized using OpenMP.*

5.3.4 Summary

In this chapter, we presented 3 different implementations of the bispectrum algorithm:

- *Python prototype*
- *C++ Single threaded*
- *C++ Multi-threaded*

Our initial implementation in Python served to verify the proper behaviour of the bispectrum algorithm and to write MS data into a different format. This allowed the C++ single threaded implementation to read the visibility data without PYRAP. The single threaded implementation serves as a performance benchmark for the multi-threaded implementation. Using OpenMP, the multi-threaded implementation aims to provide insight into parallelizing the different components of the bispectrum algorithm. Furthermore, this provides a performance benchmark for the GPU implementation.

Chapter 6

GPU Implementation

In this chapter, we present iterative improvements of an initial GPU implementation of the bispectrum algorithm. Firstly, we implement a brute force prototype in CUDA and thereafter modify and restructure the program for optimisations. This prototype primarily serves to confirm that the bispectrum algorithm is able to execute successfully on the GPU with performance improvements over a multi-threaded CPU version. A detailed discussion of the methods used and decisions taken to accelerate the initial prototype are presented. These discussions specifically highlight where code differs from the initial prototype. This provides practical insight into determining additional variables and factors that influence performance.

Optimisation of each subsequent GPU implementation closely follows the optimisation techniques detailed in Chapter 3 as well as the nVidia programming guide [nVi14] and the nVidia best practices guide [Gui14]. In summary, the 3 most important techniques used to optimise our GPU implementation are: carefully utilising *shared memory* to optimise memory access, *asynchronous transfer* of data from the CPU to the GPU to minimize the GPU's total idle duration, and finding an optimal distribution of work loads amongst threads, in order to achieve high *occupancy*. The nVidia visual profiler [nVi14] is used to generate and analyse the time-line of the program to remove bottlenecks that affect performance.

6.1 Brute force implementation

The GPU in our implementation is used to compute the most computationally expensive parts of the bispectrum algorithm. This consists of subtracting the mean visibility over time and calculating the bispectra (algorithms 4 and 10 respectively). These are the most computationally expensive components of the bispectrum algorithm and are the most probable components that will benefit (in terms of performance) by using the GPU. Furthermore, these are also well suited to the GPU due to their independent components. Therefore, the less significant components of the algorithm such as generating triplet indexes and calculating SNR are computed on the CPU where performance benefits achieved would be minimal.

In a similar fashion to the CPU multi-threaded implementation, the mean subtraction and bispectrum calculations are executed on the GPU by assigning individual threads to sub-divisions of the mean subtraction process. To achieve this on the GPU, the following tasks are iteratively performed until the bispectrum Signal to Noise Ratio (SNR) has been computed for an entire observation:

1. *Prepare the data on the CPU*: Preparation of data is required as only 1-dimensional data can be transferred to the GPU. This presents the opportunity to restructure the data in order for better memory coalescing on the GPU.

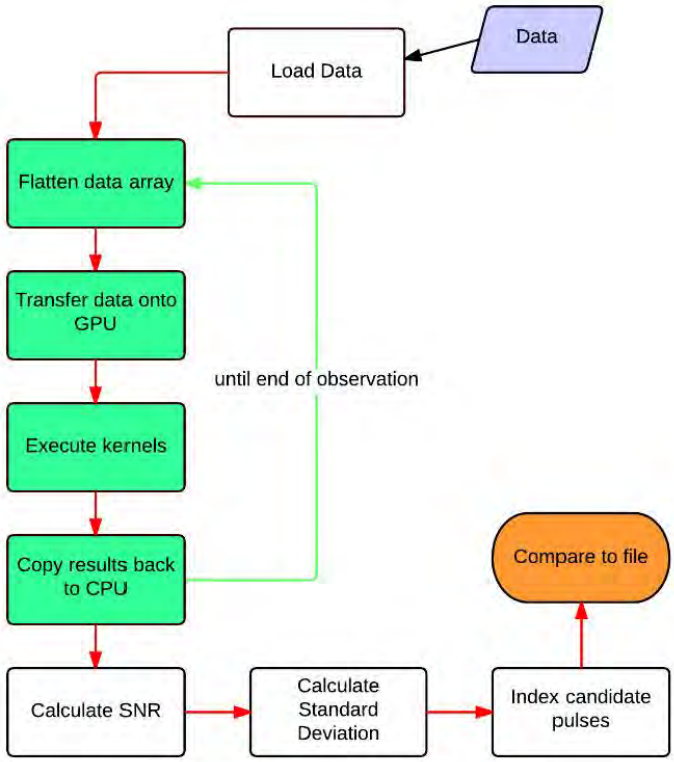

Figure 6.1: **GPU implementation flowchart** : *Green arrow: Iterative execution of GPU components of the program where data is copied to the GPU and kernels are executed until the end of the observation is reached. Red arrows: Critical paths of the algorithm. The components filled with green are the GPU components of the program. Subtraction of the mean, de-dispersion and calculations of bispectra are computed on the GPU using kernels. Finally, the SNR is generated on the CPU and candidate pulses are optionally compared to results from the python prototype.*

2. *Allocate memory on the GPU*: The exact amount of memory needs to be allocated prior to transferring the data onto the GPU. Furthermore, any additional memory space that is required during the execution of a kernel (such as the de-dispersion *offsetList* and *tripletsArray*) needs to be allocated.

3. *Transfer data onto the GPU*: The flattened, 1-dimensional array of data is transferred to the GPU.

4. *Execute kernels*: Computations of the bispectrum algorithm are performed on the data transferred to the GPU.

5. *Transfer data from GPU to CPU*: The mean of the bispectra per channel is copied back to the CPU.

6. *Free GPU memory*: Once the transfer from GPU to CPU is complete, any memory allocations that are no longer needed are freed on the GPU.

Our first implementation (or brute force implementation) merely serves to confirm that the bispectrum algorithm is able to successfully execute on the GPU. This implementation uses a naïve approach to structuring the CUDA kernel, which calculates the computationally-expensive components without any optimisations.

6.1.1 Data structures

The GPU implementation uses the same loader as the multi-threaded implementation. However, since the data will ultimately be loaded onto the GPU, using a generic *complex double* will not suffice, since the nVidia compiler does not recognise c++ structs. Complex numbers are required to use the *cuComplex* class supported by CUDA. Hence, our data takes the following form:

$$cuComplex[x, y, z] = [numChannels][numTimestamps][numBaselines] \tag{6.1}$$

Generally, cuComplex arrays are required to be 1-dimensional when transferred onto the GPU. This requires the flattening of the *timestamps* and *baseline* columns in equation 6.1. Typically, this can be done in one of two ways: namely, using an array of *structs*, or a struct of *arrays* [Ama15] shown below:

$$X[0] = t_0b_0, X[1] = t_0b_1, X[2] = t_0b_2...X[T] = t_1b_0, X[T+1] = t_1b_1... \tag{6.2}$$

or

$$X[0] = b_0t_0, X[1] = b_0t_1, X[2] = b_0t_2...X[B] = b_1t_0, X[B+1] = b_1t_1... \tag{6.3}$$

where X is the flattened array containing timestamps and baselines, T represents the total number of timestamps and B the total number of baselines. Our first prototype used equation 6.2 since this required no processing on the CPU to flatten the array in C++. This is due to 2-dimensional arrays in C++ using row-major ordering for arrays (which is not always the case for other programming languages such as fortran) [Row15]. That is, in C++ the 2-dimensional array is already stored in the form 6.2. Furthermore, this also exploits caching, as upon fetching a certain element from an array, elements near it will be cached.

Although flattening the channels to alter the data into one big 1-dimensional array is possible, our initial implementation that uses this array (detailed in algorithm 13), transfers a single channel of data on every iteration. Increasing the number of channels transferred per iteration is also explored later in the asynchronous transfer implementation.

From a different perspective, sub-dividing the data array into the aforementioned smaller divisions can be done in different ways. That is, instead of transferring one channel per iteration, a different approach where we flatten the time steps with the channels could have been implemented. However, observations are typically stored with channels as the first column and this would require extra work on the CPU to restructure the data. Theoretically speaking, as long as there are enough number of channels to exploit concurrent computation on the GPU and transfer of data to the GPU (asynchronous transfer, see section 6.3), it may not be beneficial to restructure the data on the CPU to flatten the array differently.

6.1.2 Transfer from CPU to GPU

Since subtracting the mean is independent of each channel, these can be calculated separately. Our initial implementation transfers one channel at a time onto the GPU (instead of the entire dataset all at once), anticipating the use of asynchronous transfer for optimisations in later implementations (this is also the case for the transfer of the bispectra results back to the CPU). That is, a flattened array of timestamps and baselines are transferred to the GPU and a single kernel is executed per transfer (see figure 6.2). However, in subsequent implementations, we account for kernel invocation overheads, and use a number of channels (or channel size) per transfer to the GPU, as an adjustable performance parameter.
Apart from the data array, calculating the bispectra requires memory allocations for the following variables:

- *arrayTriples - indexes of the baseline data that form triplets.*

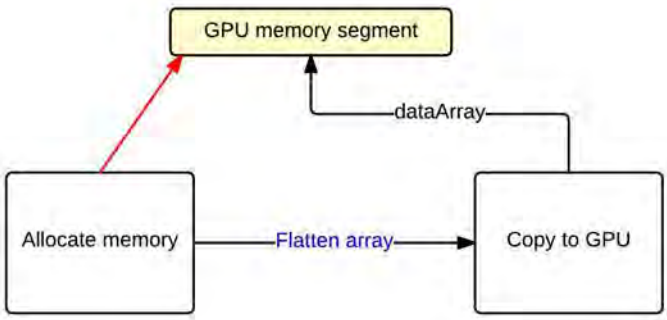

Figure 6.2: **GPU memory flowchart** : *Firstly, memory is allocated on the GPU prior to transferring data onto the GPU. The data is flattened into a 1-dimensional array and copied to the previously allocated memory.*

- *bispectraMean - The mean of bispectra across all timestamps.*
- *differencedData - used for storing visibility data with the mean subtracted*

6.1.3 Subtracting mean

Referring to algorithm 4 and fringe considerations detailed in section 2.3.2, our first kernel uses the sliding window subtraction methods with a block size of 20 time integrations. Since the data transferred to the GPU has dimensions $numTimestamps \times numBaselines$, we assign 1 block of threads per timestamp, and 1 thread per baseline, as follows:

$$meanSubKernel <<< numTimestamps, numBaselines, 0 >>> ... \tag{6.4}$$

Following Algorithm 13, the brute force kernel distributes the total work load by assigning one thread to each baseline per time step. Each thread is therefore responsible for calculating the sum of its baseline over 20 (*blockSize*) time steps. Thereafter, the average of this sum is calculated is subtracted off the original *dataArray*. This replaces the sum values originally stored in the *differencedData* array. Since the differenced data array has the same dimensions as the original data array, either array is suitable to store the differenced data. A separate kernel is used to calculate the bispectra, which uses this *differencedData* array. This array has the same structure as dataArray:

$$D[0] = t_0b_0 - mean, D[1] = t_0b_1 - mean, D[2] = t_0b_2 - mean... \tag{6.5}$$
$$D[T] = t_1b_0 - mean, D[T+1] = t_1b_1 - mean..., \tag{6.6}$$

where D is the differenced data array.

It is important to notice that our initial method of subtracting the mean does not have coalesced access to memory (specifically for *dataArray*). In particular, as each thread is executed, access to consecutive baseline data is separated by a leap of size *numBaselines* (see Algorithm 13, line 6). This is later corrected in Section 6.2. However, it is also important to note that our mean subtraction kernel has coalesced write to the *differencedData* array.

6.1.4 Calculating bispectra

Following the subtraction of the mean, the bispectrum values for each triplet need to be calculated. This is done by assigning all triplets per time step to each thread. Since the warp size is 32, we define $threadBlockSize = 32$, and assign $numTimestamps/threadBlockSize$ number of

Algorithm 13 BruteForceMeanSubKernel

```
1: procedure BruteForceMeanSubKernel(dataArray, differencedData, blockSize)
2:     int tid = threadIdx.x+blockIdx.x*blockDim.x
3:     int timestampId = blockIdx.x;
4:     if (timestampId ≤ numTimestamps-blockSize) then          ▷ Boundary condition
5:         for i in range blockSize do
6:             differencedData[tid]+=dataArray[tid+i*numBaselines]
7:         end for
8:         differencedData[tid] = differencedData/blockSize
9:         differencedData[tid] = dataArray[tid]-differencedData[tid]
10:    end if
11: end procedure
```

Algorithm 14 BruteForceBispectraKernel

```
1: procedure BruteForceBispectraKernel(diffData, meanBispectra, triples)
2:     int timeStamp = blockIdx.x
3:     int tid = threadIdx.x+blockIdx.x*blockDim.x
4:     meanBispectra[tid] ← 0
5:     if (tid≤numTimestamps) then
6:         for i in range triples.size do
7:             A ← diffData[tid*numBaseLines+triples[i*3+0]]
8:             B ← diffData[tid*numBaseLines+triples[i*3+1]]
9:             C ← diffData[tid*numBaseLines+triples[i*3+2]].conjugate()
10:            meanBispectra[tid] ← (A*B*C).real()
11:        end for
12:    end if
13:    meanBispectra[tid] = meanBispectra[tid]/triples.size
14: end procedure
```

blocks as shown in Equation 6.7 below:

$$bispectrumKernel <<< numTimestamps/threadBlockSize, threadBlockSize, 0 >>> ... \tag{6.7}$$

In the case of VLA data, 27 antennae equates to 2925 triplets and this will grow substantially for the SKA. The primary reason behind choosing this work distribution for our brute force implementation, is that no *reduction* (summing elements in an array), is required.

As shown in Algorithm 14, each thread loops through the *triples* array and calculates the bispectrum and finds the mean. Since each thread represents a single time step of the observation, no reduction is needed across threads. As a naïve initial implementation, we keep in mind that reduction is a necessary optimisation technique, and this is explored in our shared memory implementation in Section 6.4.

6.1.5 Calculating bispectrum SNR

Once both kernels have been completed per channel, the *meanBispectra* array is copied back from the GPU. These bispectra values are then averaged over all channels, and bispectra SNR is calculated as detailed in Section 4.6.

6.2 Implementation with Memory Coalescence

Flattening the data array using Equation 6.2, imposes a strided memory access pattern when executing the mean subtraction kernel. This can be seen in Algorithm 13, line 6. However, using equation 6.8 below to flatten the data array, we allow coalesced memory access. We modify the mean subtraction kernel as shown in Algorithm 15 to group our baselines first.

$$X[0] = b_0t_0, X[1] = b_0t_1, X[2] = b_0t_2...X[B] = b_1t_0, X[B+1] = b_1t_1... \tag{6.8}$$

Algorithm 15 coalescedMean

```
1:  procedure COALESCEDMEAN(dataArray, differencedData, blockSize)
2:      int tid = threadIdx.x+blockIdx.x*blockDim.x
3:      int index = threadIdx.x*gridDim.x+blockIdx.x
4:      int timestampId = threadIdx.x
5:      if (timestampId ≤ numTimestamps-blockSize) then        ▷ Boundary condition
6:          for i in range blockSize do
7:              differencedData[index]+=dataArray[tid+i]
8:          end for
9:          differencedData[index] = differencedData[index]/blockSize
10:         differencedData[index] = dataArray[tid]-differencedData[index]
11:     end if
12: end procedure
```

In general terms, the data array has all data for a particular baseline, over the entire observation, grouped together in contiguous elements in the data array. Hence, when calculating the mean of a baseline over a specified *blockSize*, at a particular time step, (see Algorithm 15, line 6), memory coalescing takes place. However, our previous kernel parameters need to be modified to account for this change. The mean subtraction kernel parameters are now as follows:

$$meanSubKernel <<< \textbf{numBaselines,numTimestamps}, 0 >>> ... \tag{6.9}$$

where each thread is now assigned to a particular time step, and each thread block to a baseline.

It is important to note, that our initial implementation of the bispectrum calculation kernel already has coalesced access to memory, and is therefore not modified. As seen in Algorithm 14, calculations of the bispectrum over all triplets, requires memory accesses to the data array within one time step block, which are contiguous elements in *dataArray* of length *numBaselines*. However, since our parameters have changed for executing the mean subtraction kernel, we need to adjust the index, to reflect which threads write to the differenced data array. As seen in Algorithm 15, line 6, we define the index to which threads write as:

$$int\ index = threadIdx.x*gridDim.x+blockIdx.x \tag{6.10}$$

where gridDim.x is the x-dimension of the grid (*numBaselines*).

Using Equation 6.10, no changes needs to be made to the bispectrum kernel. However, due to new indexing, writing to the differenced data array is no longer coalesced. Although this may not seem a fair trade off initially, as more un-coalesced writes to the differenced data array occur than coalesced reads from *dataArray*, in later implementations with shared memory, only one un-coalesced write is required to the differenced data array (see Section 6.4).

6.3 Implementation with Asynchronous Transfer

Pre-processing of the data on the CPU, to allow for memory coalescing, has considerable overhead compared to the computations on the GPU. Furthermore (as discussed in section 3.4.2) the largest communication overhead occurs while transferring data between the CPU and the GPU. The key to hiding this transfer latency is to overlap computations on the GPU, with the transfer of data from the CPU to the GPU. The two aforementioned overheads, should be overlapped with computation on the GPU to the greatest extent possible, to minimize GPU idle time.

Overlapping computation on the GPU, with the transfer of data and pre-processing of data on the CPU, uses a CUDA function call to allow for asynchronous transfer of data to the GPU, namely:

$$cudaMemcpyAsync() \tag{6.11}$$

where the above function takes an additional *stream* parameter, that specifies the stream to which the transfer belongs. These streams are declared and initialized by using the *cudaStream_t* keyword. A stream allows the programmer to specify a sequence of operations that execute in issue-order on the GPU [nVi14]. Initially, since we have a transfer of data to the GPU per channel, we initialize an array of streams of size *numChannels*. This stream parameter is also passed to the kernel call, that is associated with the asynchronous transfer, as shown below:

$$meanSubKernel <<< numBaselines, numTimestamps, 0, stream[i] >>> ... \tag{6.12}$$

For our initial asynchronous transfer implementation, no changes to the mean subtraction and bispectrum kernel are necessary. However, the pointers used to define the memory spaces used by the GPU, needed to be redefined. As shown in Figure 6.3, for our initial brute force implementation, one pointer was used for the entire dataset. Since each execution of the GPU kernel is synchronous, the memory space is re-written with new channel data once results have been copied back to the CPU.

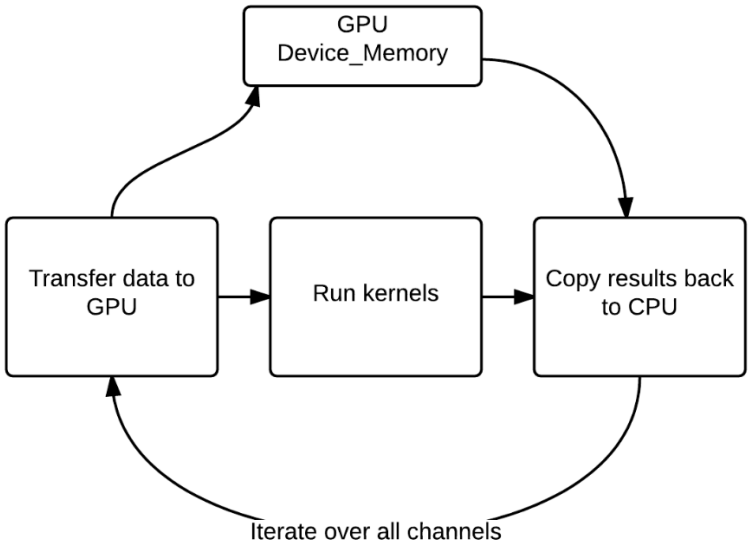

Figure 6.3: **Memory management of the brute force implementation** : *The same memory space is used for all the channels since the kernels are executed synchronously.*

In contrast, for our asynchronous implementation, each channel of data is required to be written to a different memory space as shown in Figure 6.4.. We assume initially, for the case of simplicity, that the entire observation data (which is under 1GB for most VLA observations), can

fit onto the GTX670 (up to 2GB of memory). In the case where the entire observation cannot fit onto the GPU, we reuse memory spaces by dividing the observation into manageable chunks accordingly.

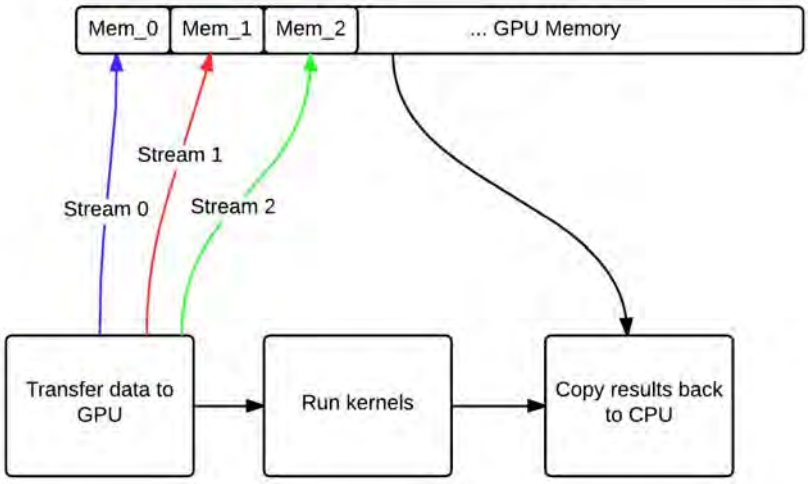

Figure 6.4: **Memory management of the asynchronous implementation** : *A different memory space is required to be allocated per channel of data. This is to facilitate the asynchronous addition to the implementation.*

6.3.1 Number of channels per transfer

Ideally, our asynchronous implementation needs to account for kernel invocation overheads (up to 14 microseconds per invocation [Ker15]) whilst balancing the number of kernels that can be executed asynchronously. That is, we can increase the number of channels each kernel works on, by flattening the dataset over multiple channels. The benefits from such an implementation, include fewer kernel invocations, which may increase performance, as well as giving each kernel more work, which may increase its occupancy. For example: transferring two channels per iteration would require half the number of kernel invocations and double the total amount of work for each kernel. This is achieved by defining a *channelSize*, and flattening the data array as follows:

$$X[0] = ch_0b_0t_0, X[1] = ch_0b_0t_1, X[2] = ch_0b_0t_2 ... X[CH] = ch_1b_0t_0, X[CH+1] = ch_1b_0t_1 ... \tag{6.13}$$

where CH = *numBaselines*numTimsteps* and indexes the array to account for multiple channels per transfer to the GPU. Several further modifications are required, as the number of device memory allocations and streams depend on the pre-defined channel size.

Specifically, both the mean subtraction kernel and bispectrum kernel require modification to work on the new flattened array of multiple channels. Keeping in mind that memory coalescing is an important performance consideration, we modify the kernel by adding an offset index of size CH = *numBaselines*numTimsteps*. These modifications can be seen in algorithm 16 for the mean subtraction kernel.

6.4 Implementation with Shared Memory

As discussed in section 3.3.3, shared memory offers low latency compared to global memory (around a factor of 100x improvement). GPU implementations can benefit significantly from

Algorithm 16 meanSubMultipleChannel

```
1: procedure MEANSUBMULTIPLECHANNEL(dataArray, differencedData, blockSize, chSize)
2:     int tid ← threadIdx.x+blockIdx.x*blockDim.x
3:     int index ← threadIdx.x*gridDim.x+blockIdx.x
4:     int timestampId ← threadIdx.x
5:     if (timestampId ≤ numTimestamps-blockSize) then          ▷ Boundary condition
6:         for ch in range chSize do
7:             index ← index+ch*numBaselines*numTimesteps
8:             for i in range blockSize do
9:                 differencedData[index]+=dataArray[tid+i]
10:            end for
11:            differencedData[index] ← differencedData[index]/blockSize
12:            differencedData[index] ← dataArray[tid]-differencedData[index]
13:        end for
14:    end if
15: end procedure
```

using shared memory, particularly in the case where threads in the same block need to access the same data multiple times. Specifically, both the mean subtraction and bispectrum kernels require multiple accesses to the same data. The mean subtraction kernel accesses the same data *blockSize* number of times, and the bispectrum kernel significantly more as each baseline belongs to a group of triplets.

6.4.1 Shared memory management

Shared memory resides on the SM and can be accessed by all threads in a block. Due to the restricted nature of shared memory access by threads, several modifications are required to our previous mean subtraction and bispectrum kernels. The GTX670 has a capacity of 49KB, and can store up to approximately 6000 *cuComplex* variables, which is sufficient for VLA observations, as shown in section 7.3.6.

Mean subtraction kernel

CUDA allow programmers to explicitly specify how much shared memory is required for a kernel. This is specified in the kernel invocation step. Recall that the mean subtraction kernel contains *numTimestamps* number of threads and *numBaselines* number of blocks as shown below:

$$meanSubKernel <<< numBaselines, numTimestamps, 0, stream[i] >>> ... \tag{6.14}$$

The kernel executes by assigning each thread to a particular time step, and iterates through the data array *blockSize* times. We utilize shared memory by copying the baseline assigned to a particular block of threads from global memory into shared memory. Therefore, we require *numTimestamps* number of *cuComplex* data elements in shared memory, and this is allocated as shown below:

$$meanSubKernel <<< numBaselines, numTimestamps, \textbf{sharedSize*2}, stream[i] >>> ... \tag{6.15}$$

where $\boldsymbol{sharedSize} = numTimestamps * sizeof(cuComplex)$. We also double the amount of shared memory used, in order to retain the result that is calculated per thread.

In the mean subtraction kernel, shared memory is specified using the *__shared__* keyword as shown below:

```
extern __shared__ cuFloatComplex sharedArray[];
```

Since each thread in a block is assigned to a time step, we use the threads to copy one data point from global memory into shared memory as follows:

```
int tid = threadIdx.x+blockIdx.x*blockDim.x;
sharedArray[threadIdx.x] = dataArrayGlobal[tid];
sharedArray[threadIdx.x+numTimestamps] = dataArrayGlobal[tid];
__syncthreads();
```

We use the *__syncthreads()* command to synchronize all threads within a block. This is to prevent calculations of the mean from being executed before all the data has been copied into shared memory. Another copy of the data is stored to shared memory (with an offset of *numTimestamps*), where the result is stored. This is advantageous in terms of performance, since the memory space would otherwise have had to be initialized to zero to calculate the mean, and we thereby save one addition operation in the calculation of the mean (see Algorithm 17).

Algorithm 17 sharedMemMeanSubKernel

```
1:  procedure SHAREDMEMMEANSUBKERNEL(dataArray, differencedData, blockSize, chSize)
2:      extern __shared__ cuFloatComplex sharedArray[]
3:      int tid ← threadIdx.x+blockIdx.x*blockDim.x
4:      int index ← threadIdx.x*gridDim.x+blockIdx.x
5:      int timestampId ← threadIdx.x
6:      for ch in range chSize do
7:          tid ← tid+ch*numTimestamps*numBaselines
8:          index ← index+ch*numBaselines*numTimesteps
9:          sharedArray[timestampId] ← dataArray[tid]
10:         __syncthreads();
11:         if (timestampId ≤ numTimestamps-blockSize) then          ▷ Boundary condition
12:             result ← timestampId+numTimestamps;
13:             for i in range (1,blockSize) do
14:                 sharedArray[result]+=sharedArray[timestampId+i]
15:             end for
16:             sharedArray[result] ← sharedArray[result]/blockSize
17:             differencedData[index]=sharedArray[result]-sharedArray[timestampId]
18:         end if
19:     end for
20: end procedure
```

As seen in Algorithm 17, once the data has been copied to shared memory, the entire kernel uses shared memory for the subtraction of the mean. It is only in line 17, that the result is copied back to global memory. As before, each sub-component of the mean subtraction kernel makes use of memory coalesced reads and writes, with the exception of the final write to global memory in line 17. This is, however, necessary since it allows memory coalescing in the bispectrum kernel.

Bispectrum kernel

Utilizing shared memory for the bispectrum kernel, requires a few changes. In particular, this subsection will detail the changes in the distribution of work amongst threads. Furthermore, we also implement reduction, to sum up the bispectrum elements.

Initially, the bispectrum kernel was invoked as follows:

$$bispectrumKernel <<< numTimestamps/threadBlockSize, threadBlockSize, 0 >>> ... \tag{6.16}$$

where each thread calculated and summed the bispectra values. In contrast, in order to utilize shared memory, we copy and share each unique baseline per time step amongst all threads in a block. Thereafter, we assign each thread of the block to calculate the bispectra in the same time step separately, and finally use a reduction step to sum the elements. These changes are detailed step by step below:

1. Update the bispectrum kernel by assigning one thread per baseline and allocating twice the number of baselines as shared memory (one section for the baseline data and one for the result).

   $$bispectrumKernel <<< \textbf{numTimestamps}, \textbf{numBaselines}, \textbf{numBaselines*2} >>> ... \tag{6.17}$$

2. Copy the differenced data array into shared memory.

```
extern __shared__ cuFloatComplex sharedArray[];
int tid = threadIdx.x+blockIdx.x*blockDim.x;
sharedArray[threadIdx.x] = differencedArray[tid];
sharedArray[threadIdx.x+numBaselines] = make_cuComplex(0.0,0.0)
__syncthreads();
```

3. Modify the kernel to account for threads calculating bispectra for a number of triplets.

```
int calcNum = numTriplets/blockDim.x.round(); //Rounded up
if (threadIdx.x*calcNum<numTriplets)
{
   calcBispectrum();
}
```

4. Reduce the bispectrum results across all threads in a block.

```
for (int s=1;s<blockDim.x;s*=2)
{
   int index = 2*s*threadIdx.x;
   if (index < blockDim.x)
   {
   sharedArray[index]+=sharedArray[index+s];
   }
   __syncthreads();
}
```

The modified bispectrum kernel (see Algorithm 18), typically assigns multiple triplets per thread. Since we are allocating *numBaselines* number of threads per block, there will be more triplets than the number of threads. Recall from section 2.3.4 that the number of baselines is given by:

$$N_{bl} = \frac{N_a(N_a - 1)}{2} \tag{6.18}$$

and the number of closed triplets formed by the following equation:

$$N_{tr} = \frac{N_a(N_a - 1)(N_a - 2)}{6} \approx \mathbb{O}(N^3) \tag{6.19}$$

Algorithm 18 bispKernelShared

```
1: procedure BISPKERNELSHARED(diffData, meanBispectra, triples)
2:     int timeStamp ← blockIdx.x
3:     int tid ← threadIdx.x+blockIdx.x*blockDim.x
4:     extern __shared__ cuFloatComplex sharedArray[];
5:     sharedArray[threadIdx.x] ← differencedArray[tid];
6:     sharedArray[threadIdx.x+numBaselines] ← make_cuComplex(0.0,0.0)
7:     __syncthreads()
8:     int calcNum ← numTriplets/blockDim.x.round();
9:     if (baselineId*calcNum≤numTriplets) then
10:        for (i in range calcNum) do
11:            A ← sharedArray[triples[i*3+0]]
12:            B ← sharedArray[triples[i*3+1]]
13:            C ← sharedArray[triples[i*3+2]].conjugate()
14:            sharedArray[baselineId+numBaselines] ← (A*B*C).real()
15:        end for
16:    end if
17:    __syncthreads()
18:    for (int s=1;s<blockDim.x;s*=2;) do
19:        int index ← 2*s*threadIdx.x+numBaselines;
20:        if (index<blockDim.x) then
21:            sharedArray[index]+=sharedArray[index+s]
22:        end if
23:        __syncthreads()
24:    end for
25:    if (threadIdx.x==0) then
26:        sharedArray[numBaselines] ←sharedArray[numBaselines]/numTriplets;
27:        meanBispectra[timeStamp] ← sharedArray[numBaselines];
28:    end if
29: end procedure
```

Therefore, each triplet is required to calculate $\frac{(N_a-2)}{6}$ number of triplets. These are calculated and summed per thread and written into shared memory. Thereafter, the array in shared memory in reduced (see Step 4 above), and written back into global memory. As with the asynchronous transfer implementation, this is also calculated over a set number of channels.

6.5 Further optimisations

Our last optimisation seeks to exclude any unnecessary computation in the bispectrum kernel. Since we are only interested in the real part of the bispectrum, we can exclude calculations of the imaginary component, which is automatically calculated when using the *cuComplex* class. Theoretically, this will save approximately a third of the major computation in the bispectrum kernel as the imaginary component contains the same amount of computation (with different terms). However, we still need to calculate the imaginary component of complex multiplication of the first two baselines. To implement a new complex number multiplication method, we first analyse the bispectrum expression in algorithm 18, line 14:

$$A = a_r + a_i i, B = b_r + b_i i, C.conj() = c_r - c_i i, \tag{6.20}$$

where a_r, b_r, c_r are the real and a_i, b_i, c_i are the corresponding imaginary components. We multiply the three complex numbers and express the bispectrum in terms of their real and

imaginary components:

$$bispectrum = A * B * C = (a_r b_r - a_i b_i + a_r b_i i + a_i b_r i)(c_r - c_i i), \tag{6.21}$$

and thereafter group the terms for the real components below:

$$bispectrum.real() = a_r b_r c_r - a_i b_i c_r + a_r b_i c_i + a_i b_r c_i, \tag{6.22}$$

and finally by grouping like terms to lower the number of operations, we get:

$$bispectrum.real() = c_r(a_r b_r - a_i b_i) + c_i(a_r b_i - a_i b_r). \tag{6.23}$$

6.5.1 Extending the cuComplex class

We implement the alterations to the previous bispectrum kernel, by extending the *cuComplex* class and writing a separate method to calculate the bispectrum, given three *cuComplex* variables. This is shown in the code snippet below:

```
public float bispectrum(cuComplex A, cuComplex B, cuComplex C)
{
//Calculates the returns the real part of the bispectrum given three complex numbers
return (C.r)*[(A.r)*(b.r)-(A.i)*(B.i)]+(C.i)*[(A.r)*(B.i)-(A.i)*(B.r)]
}
```

Lastly, we modify line 14 of the bispectrum kernel (see algorithm 18) to:

```
sharedArray[baselineId+numBaselines] = bispectrum(A,B,C);
```

6.6 Summary

In this chapter, we firstly introduce an initial brute force GPU implementation of the bispectrum algorithm and detail the implementation of the mean subtraction and bispectrum kernel. This is followed by iterative improvements using several optimisation techniques detailed in Chapter 3. These include memory coalescing, asynchronous transfer and utilizing shared memory. Finally, we modify the *cuComplex* class to reduce unnecessary computation in the bispectrum kernel, to achieve higher performance.

Chapter 7

Experimentation and Results

In this chapter, we firstly present validation results by testing the bispectrum algorithm on both simulated and real data. This is required to verify the correctness of our implementations, and to test certain properties of the bispectrum algorithm. Secondly, we present performance evaluations of the various GPU implementations detailed in Chapter 6. Finally, we compare the performance of a final CPU implementation against a final GPU implementation. These performance tests are initially run using only data from the Jansky Very Large Array (JVLA). However, we later extend evaluations to include larger simulated datasets (with a higher number of baselines and hence more triplets), to account for growth in the size of interferometers that is expected in the near future.

Multiple instances of each test case was run, and where applicable, the mean of the results were taken. Unless otherwise stated, all test cases were run using the following hardware setup:

- CPU: Intel Core i7-3820 CPU @ 3.60GHz, which contains 4 cores and utilizes 8GB of RAM.
- GPU: GTX670, which contains 7 SMs (1344 cores) and 2GB global memory.

7.1 Validation Using Simulated Data

To validate our initial implementation of the bispectrum algorithm, we use the *MeqTrees* software package to simulate transients in a measurement set. Thereafter, we adjust several parameters in *MeqTrees* to ensure that the algorithm works as expected. These parameters include:

- *Location of the transient:* Adjusting the time step where the transient is inserted provides a basic initial test that validates whether the algorithm is implemented correctly.
- *Adjusting the source flux intensity*: Adjusting the intensity of the transient gives insight into the sensitivity of the bispectrum algorithm.
- *Adding noise*: The bispectrum algorithm is expected to successfully detect transients regardless of noise (local interference).
- *Transients off the phase centre*: Transients off the phase centre cannot be detected using traditional beamforming techniques [LBP+12]. However, the bispectrum is expected to be capable of detecting transients that are off the phase centre in this case.

7.1.1 Location of transient

The initial test sets out to verify that our implementation is able to successfully detect transients in the simplest case. No artificial noise or other complex parameters (explored later), were

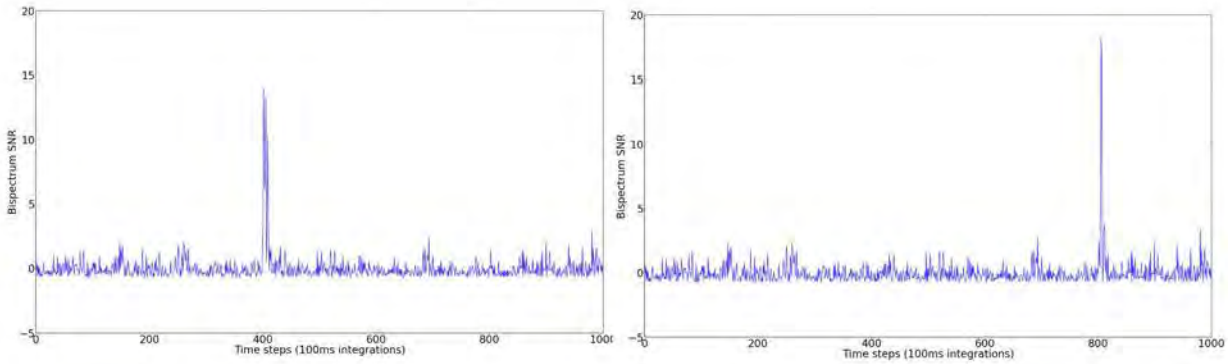

Figure 7.1: **Left**: *A transient of flux intensity 5Jy is placed at the 400th time step of an observation of 1000 time steps. As seen in the graph, the bispectrum SNR responds significantly to the transient, reaching an SNR of 14σ. The bispectrum SNR does not peak above 5σ (the threshold used for possible detections) for any of the remaining time steps.* **Right**: *A transient with the same properties as the transient on the left is placed at the 800th time step. Similarly, the bispectrum SNR responds well and achieves an SNR of above 15σ with the remaining time steps well below 5σ.*

added to the observation. This straight-forward test, uses two datasets, each with a transient with relatively high flux intensity (5Jy). These were placed at two different time step locations, namely 400 and 800, in a KAT-7 observation with 1000 time steps. The results of these two test cases are shown in figure 7.1

Upon examination of the graphs in figure 7.1, it is evident that the bispectrum SNR is relatively unaffected by background sources that are simulated with the observation. This can be attributed to the mean subtraction process that happens prior to the bispectrum calculations [LBP$^+$12]. Furthermore, it is important to notice the accuracy of the bispectrum algorithm. That is, detections occur at the exact time step where the transient is placed.

7.1.2 Flux intensity of transient

The second experiment sets out to determine the sensitivity of the bispectrum algorithm, and to provide insight into the behaviour of the bispectrum SNR. That is, finding the limitations of the bispectrum algorithm to define the scope of where the algorithm is applicable. More specifically, it attempts to answer two questions:

1. What is the lower limit for flux intensity that the bispectrum algorithm can detect?
2. How does the bispectrum SNR change with lower flux intensity?

Analysis of the results in figure 7.2 show that a flux intensity of 3Jy is a lower limit for successful transient detections in this dataset. Although this may vary between datasets, it provides insight into the sensitivity and accuracy of the algorithm. It is evident that although the bispectrum SNR produces few false detections in data where a transient is not prominent (where SNR is above 5σ), the algorithm is still able to flag these as noise. Furthermore, despite having a lower flux intensity, the transient is still detected at the exact time step.

False detections at certain time steps in the observations are more likely to occur, where transients have lower flux intensities. A comparison of Figure 7.1 and Figure 7.2 shows that this can be attributed to the bispectrum SNR becoming somewhat "suppressed" by noise-related false positives, with higher flux intensities than the inserted transient.

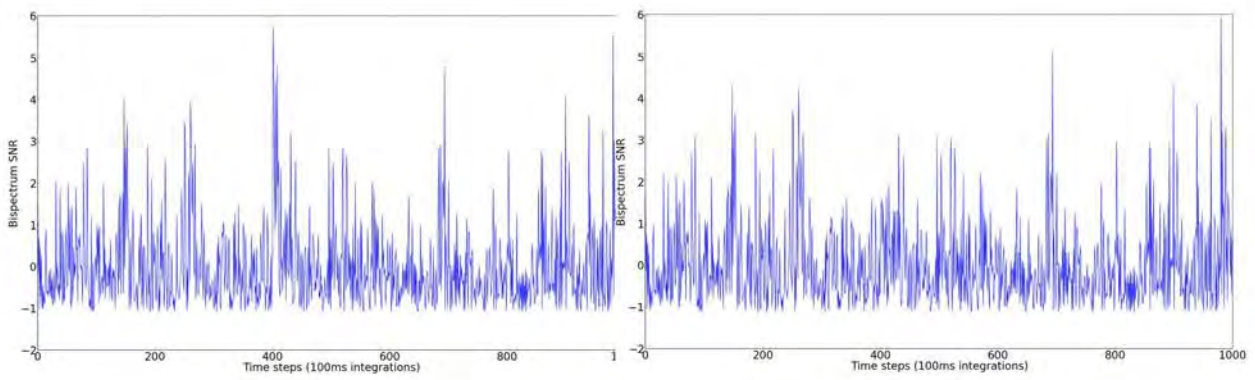

Figure 7.2: **Left**: *A transient of flux intensity 3Jy is placed at the 400th time step. At this lower limit, the bispectrum can still successfully detect the transient, despite having another detection at the 980th time step. The standard deviation of the triplets for the 980th time step detection is much higher and is flagged as noise.* **Right**: *A transient of flux intensity 2Jy is placed at the 400th time step. It can be seen from the graph that the bispectrum fails to detect the transient. At two different time steps (697 and 980), the bispectrum SNR is above 5σ but is also flagged as noise.*

7.1.3 Noisy data and transients off the phase centre

We expand our experimentations by analysing the influence of adding noise to the simulated data, and analyse the response of the bispectrum SNR to this noise. Furthermore, we attempt to verify the property that the bispectrum algorithm is able to detect transients that are away from its phase centre.

From Figure 7.3 (left), it is evident that the bispectrum algorithm is capable of successfully detecting transients off the phase centre. Similar to earlier tests, the accuracy of the algorithm is shown, as the algorithm detects the transient at exactly the 400th time step. The detection at time step 180 is flagged as noise. Figure 7.3 (right), shows a transient of 4Jy becoming obscured by noise levels of the same intensity. Although this is to be expected, it is also highly unlikely that noise levels would reach 4Jy [Bed06]. Although the bispectrum algorithm is unsuccessful here, beamforming and imaging algorithms would also be obscured by this intensity of noise, as it increases the likelihood that the transient would be indistinguishable from background noise.

7.2 Validation Using Real Data

Our final validations step uses real data from the VLA to test our implementation. This provides a more concrete validation, as testing data from a real context may introduce unexpected complexities that cannot be simulated by *MeqTrees*. The VLA dataset used was generously provided by Casey Law.

In Figure 7.4, it can be seen that the bispectrum algorithm behaves as expected and successfully detects B0355+54 at the 135th time step. This figure also displays one of the powerful properties of the bispectrum, as the pulsar is detected regardless of whether the data is calibrated. Other transient detection techniques such as beamforming and imaging require the data to be calibrated (see section 10). It can be observed that although the bispectrum SNR in the calibrated data is more apparent (above 7σ), the SNR largely unaffected by uncalibrated data (slightly below 7σ).

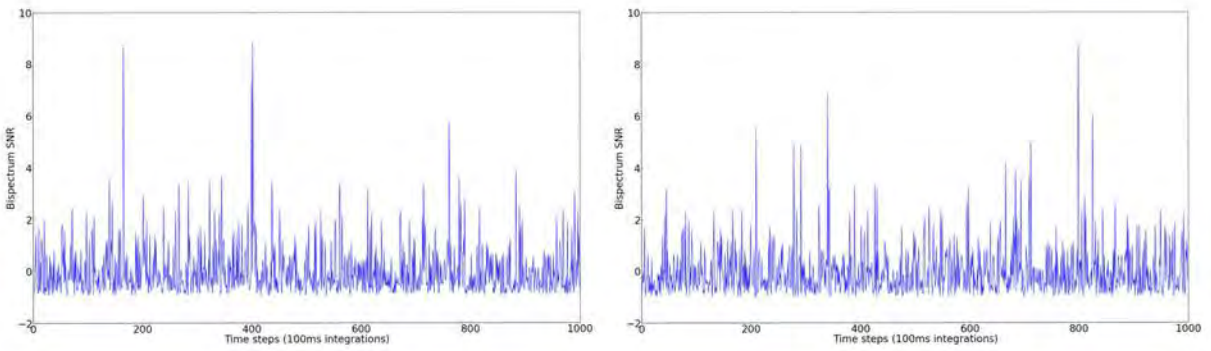

Figure 7.3: **Left**: *A transient off the phase centre of 4Jy is placed at the 400th time step. Simulated noise of 2Jy is also added to this observation. It can be seen that the algorithm successfully detects the transient, with an SNR value above 8σ.* **Right**: *A transient of 4Jy is placed at the 400th time step. Noise levels were incremented until the algorithm cannot detect the transient. This simulation has added noise of 4Jy across all channels (unlikely to be this high, in a real context).*

It is also easily seen that there are no false detections. The bispectrum SNR is generally below 2σ for most of the observation, except for where the transient is present. This further substantiates the use of the bispectrum algorithm in transient detection systems.

A second VLA dataset also containing B0355+54 is also used. The results of the validation can be seen in Figure 4.4 in Chapter 4, where the transient is detected at the 170th time step.

7.3 Performance Results

In this section, the performance results between two CPU implementations is presented. Thereafter, the effect of optimisations to the GPU implementation is analysed. The run times of each optimisation are graphed followed by a discussion on how each optimisation has reduced the total run-time of the bispectrum algorithm.

7.3.1 CPU performance

Analysing the performance of the two CPU implementations in figure 7.5 shows a consistent speed-up of between 3.0x and 3.2x for the datasets used. Theoretically, the maximum amount of speed-up that can be achieved through multi-threading on a 4-core CPU is 4x, there are thread invocation overheads and other components in the bispectrum algorithm that cannot be parallelized. This accounts of the reduced amount of speed-up from the theoretical maximum. Figure 7.5 also demonstrates the superior scalability of the multi-threaded implementation, despite both implementations scaling linearly to the number of time steps.

Since no efforts were made to optimise the initial Python prototype in Section 5.1, no performance analysis is done between Python and C++. However, we do note that the Python implementation had a total run-time of 267 seconds for 200 time steps.

7.3.2 GPU Experimentation outline

To analyze the performance of the various GPU implementations, the optimized CPU implementation in Chapter 5 is used as a performance benchmark. Furthermore, the cumulative performance benefits from subsequent GPU implementations are analyzed. These GPU implementations (as detailed in Chapter 6) include:

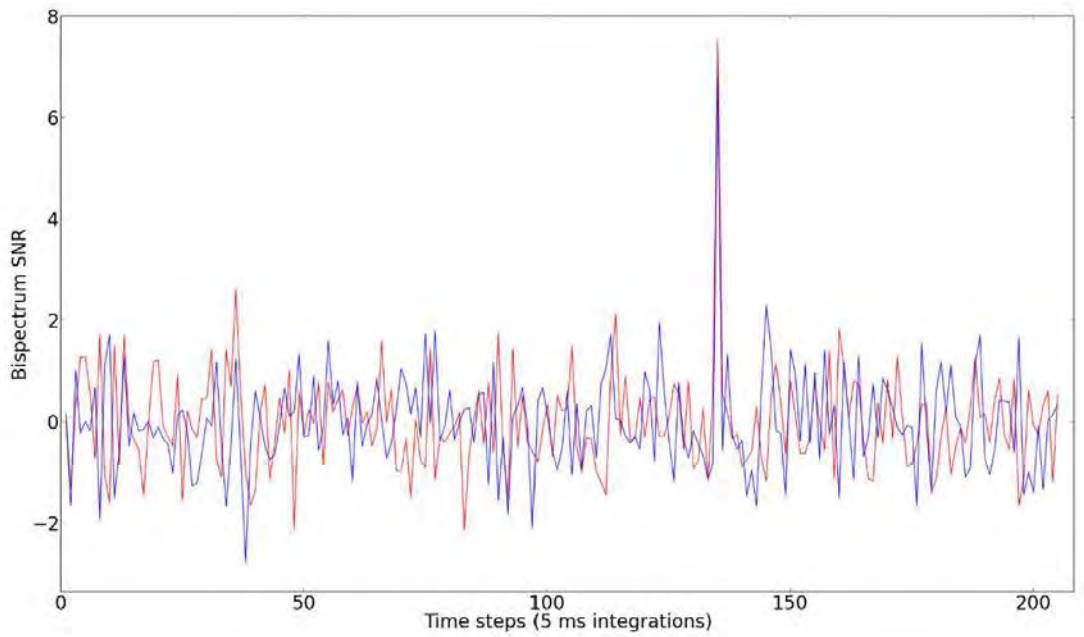

Figure 7.4: **Red**: *Calibrated data.* **Blue**: *Uncalibrated data. This graph shows that the bispectrum algorithm is calibration independent. That is, it can detect transients regardless of whether the data has been calibrated beforehand. Both bispectrum SNRs peak at around 7σ at the 135th time step. It is also important to note that this is real data from the VLA, and contains a transient emitted by pulsar B0355+54. Credit: Data supplied by Casey Law.*

1. *Brute force implementation*
2. *Memory coalescing*
3. *Asynchronous transfer*
4. *Shared memory*
5. *A modified cuComplex class*

The data used to test our GPU implementation is VLA data, containing 128 channels and 351 baselines and assumes a single DM value. However, the size of the dataset is increased artificially by duplicating the time samples to a specified size. This gives insight into how well the algorithm scales as the dataset size increases. In final test runs, we also increase the number of baselines artificially as above, to analyse how the algorithm scales for interferometers containing more dishes (weak scaling versus strong scaling).

Multiple instances (typically 10) of each test were run, and the mean of the time taken was calculated and plotted. This is intended to average out any outlier tests and external factors that may cause tests to run significantly slower or faster.

7.3.3 Brute force implementation results

The first performance test is a comparison of the total run-time of the optimised CPU multi-threaded implementation, with the brute force GPU implementation. The run-time results are shown in Figure 7.7, where it can be seen that even an initial brute force implementation on the GPU can yield substantial speed-up compared to a fully optimised CPU implementation. This is credited to the highly parallel nature of the bispectrum algorithm and how well it is suited to the GPU. Initially, at 200 time steps, the GPU implementation yields almost 4x speed up.

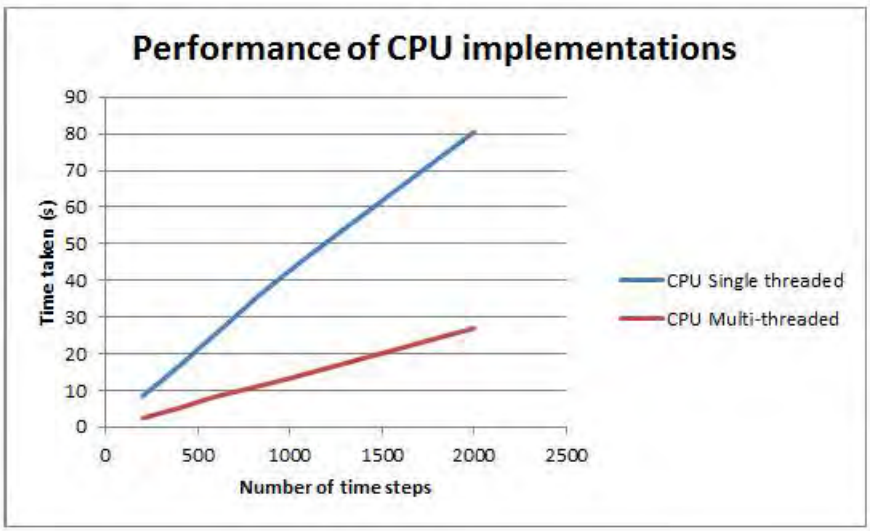

Figure 7.5: *Performance comparison of the single threaded and multi-threaded CPU implementation. 5 VLA datasets with time steps up to 2000 time steps is used.*

Thereafter, the GPU implementation scales better, yielding above 10x speed up with observations of 2000 time steps (2000 time steps being the upper limit due to memory limitations).

Although Figure 7.7 illustrates significant speed-ups, we can expect improvements from further optimisations. Using the nVidia profiler to analyse the GPU implementation, it is clear that the bispectrum kernel dominates the total execution time of the program (2.3% for mean subtraction, 97.7% for bispectrum kernel). For this implementation, we only achieve 1.6% occupancy for the bispectrum kernel. This implies that much more work can be done towards an optimised GPU implementation.

Analysis of the time-line of the program, shows that the GPU is often idle. This is due to the GPU waiting on data to be transferred from the CPU to the GPU, as well as overheads in flattening the data array.

7.3.4 Memory coalescing results

Now that we have a benchmark for a GPU implementation, we continue tests by comparing the performance of each iterative GPU implementation. Here, in Figure 7.6, we compare the performance of the brute force GPU implementation with an implementation with coalesced access to memory.

In Section 6.2, it was shown that strategic alterations to the GPU implementation focused on allowing coalesced access to memory in the mean subtraction kernel. Since the total run time is dominated by the bispectrum kernel, there is not much total performance benefit to be gained from memory coalescence.

However, if we analyse the total run-time of the mean subtraction kernel using the profiler, we see that the occupancy increases as expected for the mean subtraction kernel (from 69.4% to 74.2%). This is due to lower data transfer latencies from coalesced memory access. Although we do not have coalesced writes to global memory for the differenced data any more, there are more coalesced reads from global memory for the initial data array (As stated in Section 6.2). This justifies the alterations made to the mean subtraction kernel.

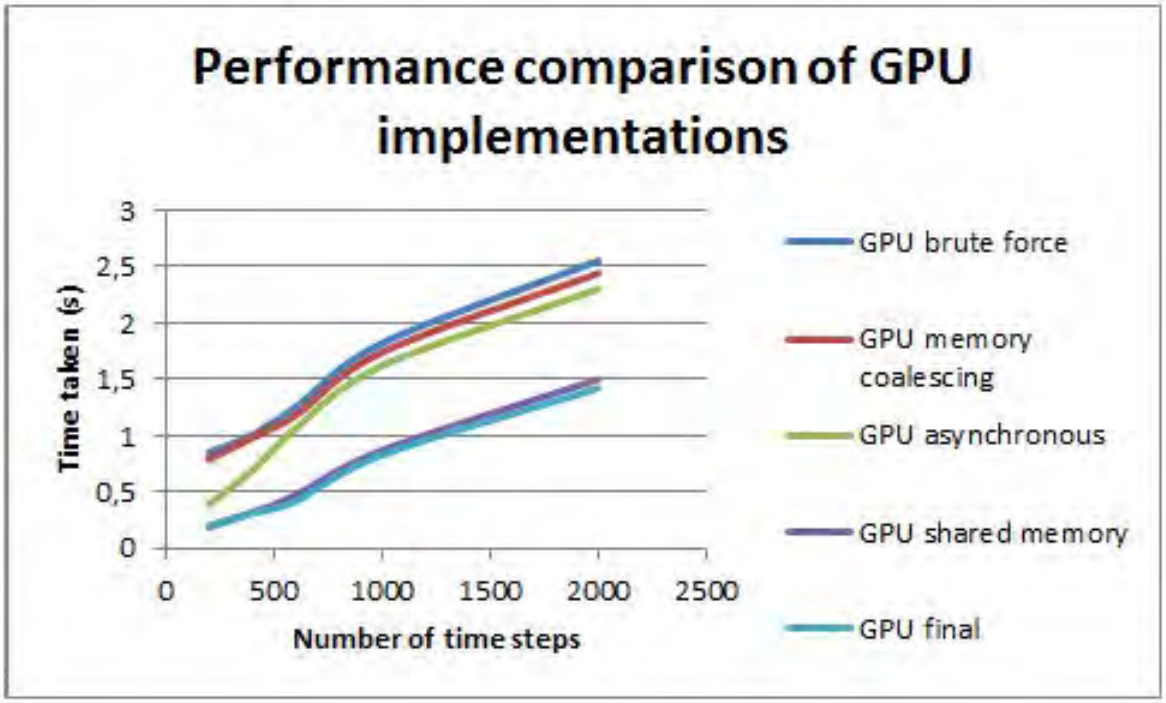

Figure 7.6: *Performance comparison of the the 5 GPU implementations. The same datasets are used (from CPU performance comparison) here.*

Although there is no significant speed up (only 5% of the total run time at best), having coalesced memory access is necessary for the use of shared memory. Thus, the full impact of coalesced memory access is not fully evident here, but the benefit is realised for the shared-memory implementation in section 6.4.

7.3.5 Asynchronous transfer results

The next result shows the performance benefits from asynchronous transfer of data from the CPU to the GPU.
In figure 7.8, we observe speed ups across all data sizes. Overlapping computation on the GPU with the transfers of data from the CPU to the GPU, lowers the amount of time that the GPU is idle. Furthermore, it allows the GPU to overlap computation with the preprocessing of data on the CPU.

It can be seen that the asynchronous transfer implementation yields greater speed-up for smaller datasets. For 200 time steps, the asynchronous transfer implementation yields a 2x speed-up. However, for 2000 time steps, only a 10% speed up is observed. Recall in Section 6.3 that the grid size of the bispectrum kernel is limited by the number of time steps. At 200 time steps, we only have 8 blocks (with 32 threads in a block), which is insufficient to fully occupy the GPU. This lowers the occupancy of the bispectrum kernel significantly. Initial analysis of the asynchronous transfer implementation, showed that the GPU was executing multiple bispectrum kernels simultaneously, further solidifying the hypothesis that the performance is limited by the block size.

At 2000 time steps, the memory coalesced implementation runs the bispectrum kernel with 80 grids. This increases the occupancy of the kernel to (4.4%), which explains the reduced speed up, when compared to the asynchronous transfer implementation.

Another reason for a lower speed-up, is that the overheads of pre-processing data on the CPU

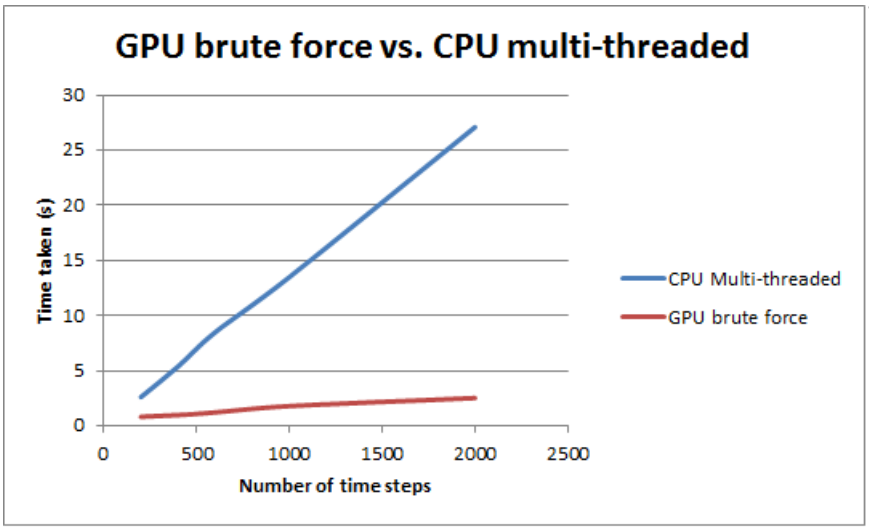

Figure 7.7: *Performance comparison of the multi-threaded CPU implementation and the GPU brute force implementation.*

becoming less significant for larger datasets. Furthermore, a higher bandwidth between the CPU and the GPU is achieved for larger datasets. This lowers the ratio of GPU idle time to the total GPU computation time, hence speed-ups of the asynchronous transfer implementation are less significant in larger datasets.

From a different perspective, this does not mean the asynchronous transfer implementation performs poorly for larger datasets. Rather, the GPU implementation without asynchronous transfer (our GPU memory coalesced implementation), has a higher ratio of computation to data transfer for larger datasets, since the portion of GPU idle duration is lower.

Analysing the program using the nVidia profiler shows that although the bispectrum kernel still has low occupancy (6%) for the asynchronous transfer implementation, multiple bispectrum kernels can now be executed simultaneously. This implies that the asynchronous transfer implementation is suited for both small and large datasets.

7.3.6 Shared memory results

Our next GPU implementation, which yields the most amount of cumulative speed-up, uses shared memory to lower memory access latency. Since both the mean subtraction and the bispectrum kernel require multiple accesses to the same data, we copy data to shared memory first, and only write the results back to global memory once all computation has completed. The results are shown in Figure 7.9 below: It is no surprise that the most significant speed up comes from using shared memory. Access to global memory is extremely expensive relative to access to shared memory (shared memory is 100x faster). For the mean subtraction kernel, access to the same data occurs a total of *blockSize* times, and for the bispectrum kernel, the total number of triplets is given by:

$$N_{tr} = \frac{N_a(N_a - 1)(N_a - 2)}{6} \tag{7.1}$$

Since each triplet contains 3 separate baselines, it can be seen that access to the same baseline occurs a total of $N_{tr}/3$ times. Even for current interferometers such as the VLA with 27 antennae as an example, 2925 triplets equates to 975 accesses to each baseline data element.

Reducing global memory latencies improves the occupancy of both the mean subtraction and

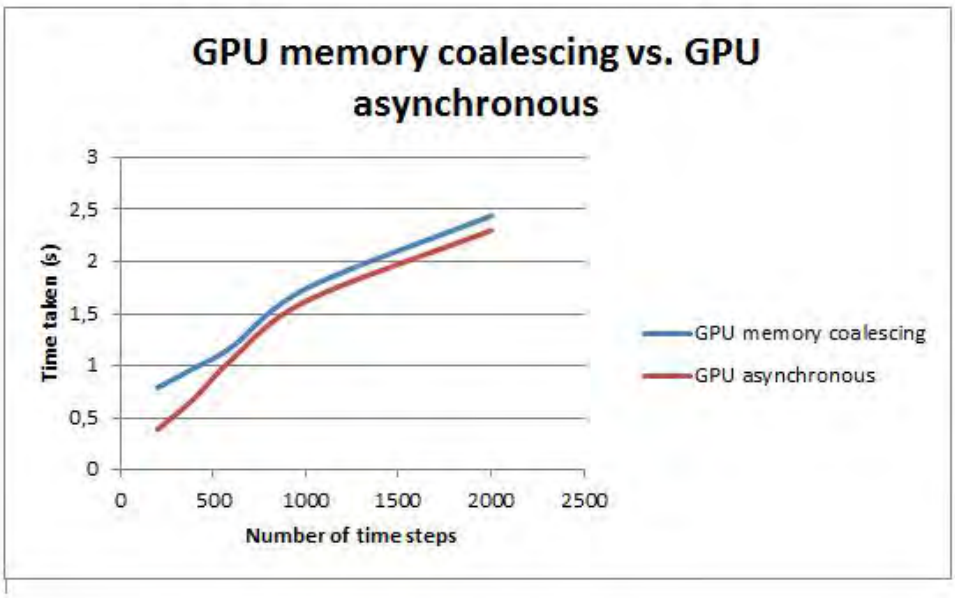

Figure 7.8: *Performance comparison of the GPU memory coalescence implementation and the GPU asynchronous transfer implementation.*

bispectrum kernels. The occupancy of the mean subtraction kernel improved from 74% to 85.9%, and the bispectrum kernel, improved from **4.4%** to **83.7%**. Figure 7.9 shows significant speed ups across all time step sizes (up to 2.2x at time step 600).

Although the speed-up may seem underwhelming for such a large increase in occupancy in the bispectrum kernel, recall that for our asynchronous transfer implementation, multiple kernels were being executed concurrently. This is no longer possible due to the use of shared memory, as each block of threads needs to finish executing to free the shared memory, before another block can execute. That is, it was previously possible for two thread blocks to occupy one SM on the GPU, but shared memory prohibits this, as each thread block have exclusive usage of the shared memory on the SM.

7.3.7 Modifying the cuComplex class

The last optimisation, reduces the amount of computation in the bispectrum kernel by modifying the cuComplex class to only calculate the real component of the bispectrum.

On examination of the results in Figure 7.6, it is clear that there is a consistent amount of speed up across all datasets (8%). Despite being a slightly lower speed-up compared to previous optimisations, 8% is relative to the GPU shared memory implementation. This is already a few times faster than our initial brute force implementation.

7.3.8 Overall performance comparisons

Here, the overall speed-ups are highlighted by comparing the total run times for the CPU multi-threaded implementation, the brute force GPU implementation, and our final GPU implementation.

As shown in Figure 7.10, the bispectrum algorithm is well suited to GPU with regular significantly lower run times compared to CPU implementations. Further, GPU implementations generally also scale better as the problem size gets larger. In figure 7.11 we can see that our final GPU implementation yields at best a 20x speed-up over the CPU implementation, and scales

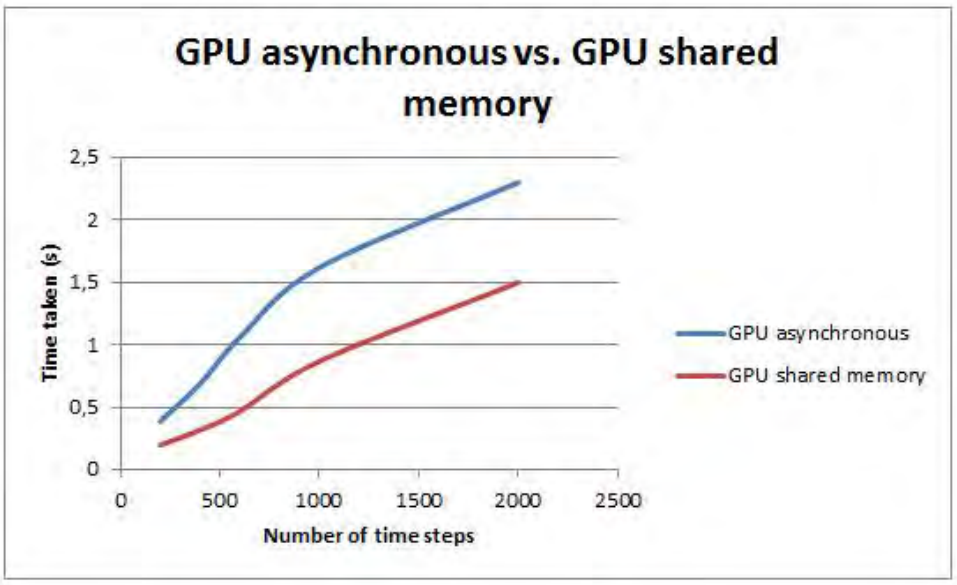

Figure 7.9: *Performance comparison of the GPU asynchronous transfer implementation and the GPU shared memory implementation.*

well as datasets get larger.

Similar to the comparison between pre-asynchronous and post-asynchronous transfer implementations, the brute force implementation suffers from low occupancy in the bisepectrum kernel due to the limitation of grid size. This is shown at time step 200, where the final GPU implementation performs 4.7x faster. In contrast, with 1000 and 2000 time steps, the final GPU implementation is only 2.2x and 1.8x faster, respectively.

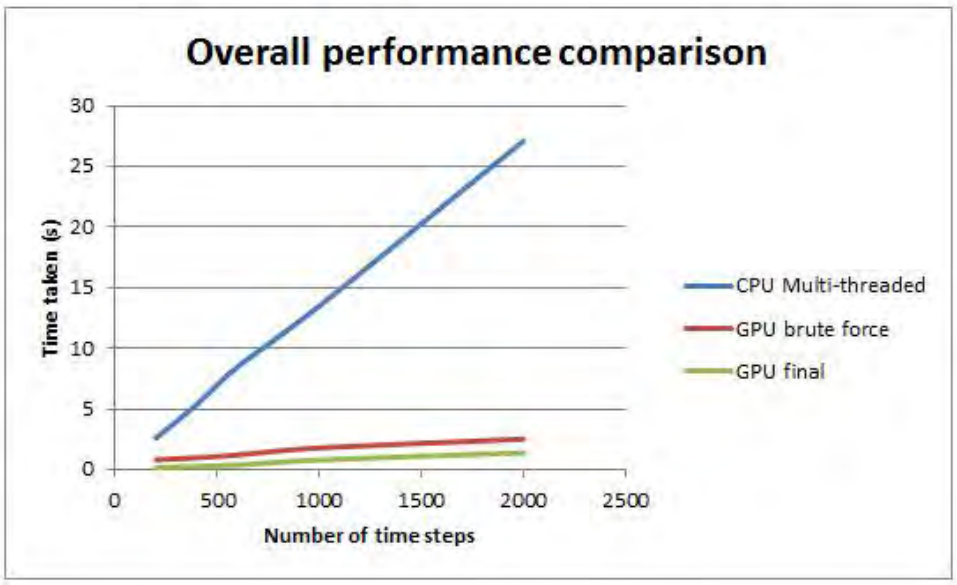

Figure 7.10: *Performance comparison of the CPU implementation against both the brute force GPU implementation and the final GPU implementation.*

7.3.9 Larger interferometers

It is clear that the performance of the bispectrum algorithm will scale linearly with the number of channels. We are interested in evaluating the performance of the algorithm with increasing number of antennas. The number of antennae is varied between 27 (VLA) and 64 (meerKAT) by artificially increasing the dataset size and the results are shown in figure 7.10.

It can be seen from figure 7.12 that our GPU implementation is likely to scale approximately quadratically with the number of antennae. Unfortunately, due to memory limitations, simulations of larger datasets are not possible. This is due to 64 antennae corresponding to 2016 baselines and 41664 triplets.

7.4 Summary of chapter

In this chapter, we firstly show validation results of our initial prototype. Validation of our implementation involved using *MeqTrees* to simulate transients with specific properties and thereafter running our implementation on this simulated data. Secondly, we detail and analyse the performance benefits achieved from different GPU optimisation techniques. Thereafter, we looked at the total speed-ups of the final GPU implementation compared to the CPU implementation and our initial brute force GPU implementation. Results show a 20x speed-up between our final GPU implementation compared to the CPU implementation. Finally, we ran the final GPU implementation on datasets with more antennae, and conclude that the implementation scales quadratically.

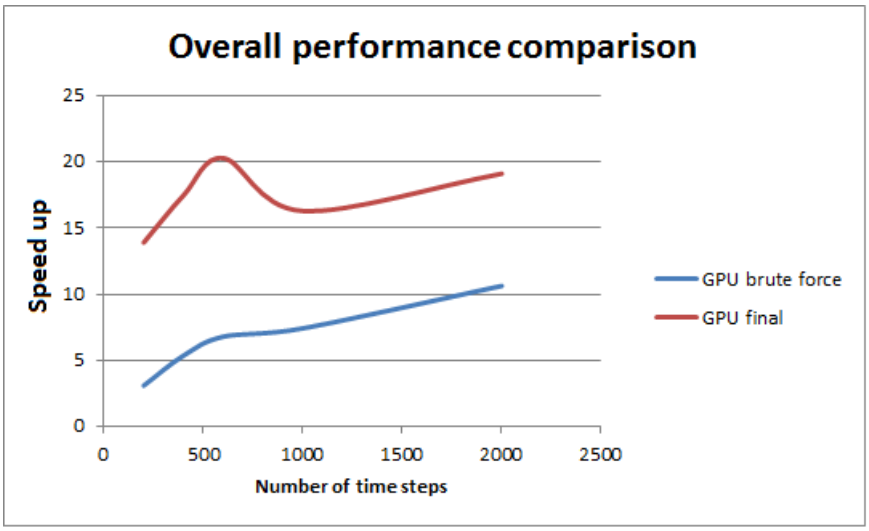

Figure 7.11: *Speed ups of both the brute force GPU implementation and the final GPU implementation against the CPU implementation.*

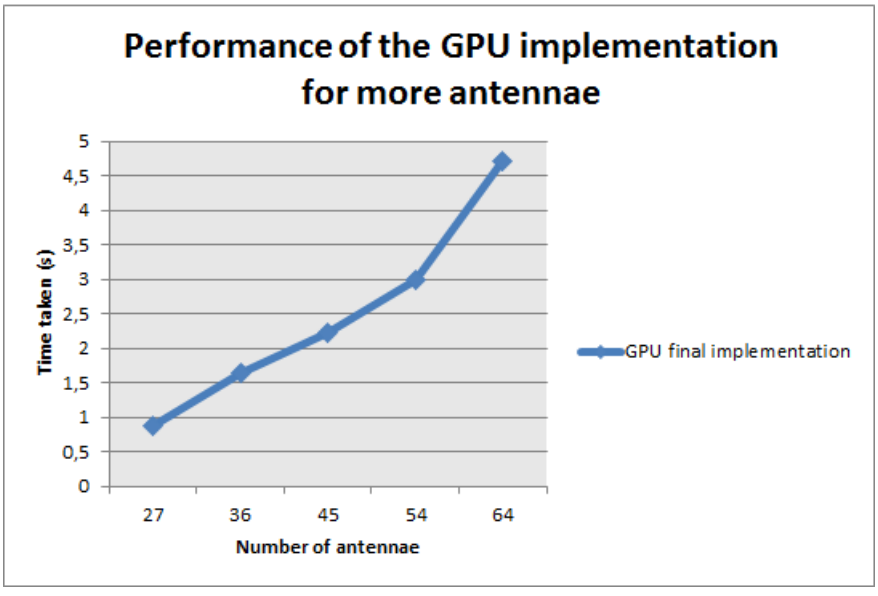

Figure 7.12: *Performance comparison of the GPU shared memory implementation implementation and the final GPU implementation with optimisations to the bispectrum kernel.*

Chapter 8

Conclusion

The SKA project creates many opportunities for developing novel techniques to tackle the challenge of analysing large amounts of data in a real-time context. Amongst many other data analysis algorithms for SKA, transient detection algorithms in the form of beamforming and imaging are incapable of real-time analysis due to the ever-growing size of interferometers, particularly the SKA [LBP+12]. Correspondingly, much research has been invested into the development of a computationally efficient method of transient detection, namely the bispectrum algorithm. The bispectrum algorithm can be accelerated using commodity GPUs by parallelizing its individual components, and the speed-ups gained from GPU implementations (shown in Chapter 7) motivate the use of GPUs for future transient detection systems.

This thesis set out to investigate the applicability of GPUs to accelerate the bispectrum algorithm, and hence transient detection. The GPU implementations in this thesis were designed specifically to exploit the computational capabilities of the GPU to achieve speed-ups and scalability over CPU implementations. An initial brute force implementation for the GPU was compared to an optimised CPU multi-threaded implementation and showed approximately a 10x performance gain. However, an analysis of the GPU implementation showed many optimisation opportunities which were later explored.

Some of the main limiting factors in the brute force implementation include:

- Low occupancy for the bispectrum kernel: The GPU s idle during most of the bispectrum kernel, as a result of a limited grid size. Increasing the number of blocks shows performance benefits, particularly for larger datasets as larger datasets have larger grid sizes.
- GPU waiting on the CPU: Pre-processing of data on the CPU and the transfer of data to the GPU is not concurrently executed with subsequent computations on the GPU. This is resolved by utilizing streams and asynchronous transfer to enable concurrent execution on CPU and GPU. This further lowers the total GPU idle duration.
- Access to global memory: Using the VLA data as an example, with 27 antennas, the bispectrum kernel needs 975 accesses to the same data element in global memory. This motivates the use of shared memory, which is $\sim$100x faster.

Testing of each subsequent GPU implementation involves analysing how each implementation scales to longer observations. Datasets of up to 2000 time steps were artificially created and analysis of the results show that the performance scales linearly with the number of time steps. This is expected as the amount of computation for both GPU kernels (mean subtraction and bispectrum) scales linearly to more time steps. It was also evident that computations for both kernels also scale linearly when increasing the number of frequency bands or channels.

After a series of iterative improvements and optimisations, the final GPU implementation represents better utilization of the computing capabilities offered by the GPU. A speed-up of 20x over the multi-threaded CPU implementation (intel i7 CPU with 4 cores) is achieved for the final GPU implementation (GTX670 with 7 cores) using a VLA dataset with 600 time steps. It can be shown that only minor optimisations remain possible as both the mean subtraction and bispectrum kernels run at very high occupancy (85.9% and 83.7%, respectively). Furthermore, computation on the GPU is overlapped with data transfers and pre-processing of data on the CPU through the use of asynchronous transfers.

Once the final GPU implementation was developed, experiments were conducted to analyse performance, particularly to gain insight into how it scales with the number of antennae. Results from Chapter 7 show the GPU implementation scales quadratically with the number of antennae. Again, this is as expected, since the number of baselines and unique triplets also scale quadratically to the number of antennas, and hence the amount of computation required also scale quadratically.

The CPU implementation uses both simulated and real data to verify validity. Using the software package *MeqTrees*, we were able to simulate transients at specified time steps, with adjustable parameters to define their properties. The CPU implementation succeeded in locating all transients above 3Jy. This includes transients off the phase centre, as well as datasets with noise added. Thereafter, the implementation was run using real VLA data, to locate known transients. This test was successful in locating the transient B0355+54 for both calibrated and uncalibrated data (a powerful property of the bispectrum algorithm) in two separate observations. Lastly, in order to validate the GPU implementation, results acquired from the GPU was compared to results of the CPU implementation. This is achieved by comparing the bispectrum SNR values written to file.

This research demonstrates both the computational efficiency and accuracy of the bispectrum algorithm for detecting transients, as well as the applicability of using GPUs to accelerate the bispectrum algorithm. These two factors together show a bright future, in the area of transient detection software, particularly for the SKA and other large interferometers.

8.1 Future work

Analysis of the results in Chapter 7, shows much potential for the bispectrum algorithm to evolve into a real-time transient detection system. We consider the following areas for future research:

- To overcome the memory limitations of a single GPU processor, particularly in shared memory, the algorithm could be extended to run on multiple GPUs or a GPU cluster. Furthermore, experiments with techniques to optimally utilize shared memory can be conducted to push the boundary of memory limitations, such as freeing baseline elements that are no longer needed in the bispectrum kernel.

- Having more GPUs opens opportunities to test larger datasets of up to 2000 antennas (SKA phase 2). This can provide insight into unexpected performance drops in larger datasets, as the tests conducted in this thesis may not have hit certain memory or computational limitations that might exist when dealing with larger data.

- Our implementation serves as a transient detector, but does not image the time steps where the transient is detected. An automated imaging component could be integrated with the bispectrum algorithm as localization of the transient is needed.

- Intel has recently released their own many core architecture, called the Xeon Phi co-processor [Tre13]. Performance comparisons on this hardware against GPUs will provide insight into whether the Xeon Phi is more suitable in terms of scalability, power consumption and cost for the problem at hand

Bibliography

[Ahu05] Mitra Kembhavi Ahuja, Gupta. A novel technique for estimating pulsar dispersion measures. pages 4–7, 2005.

[Ama15] Intel AmandaS. Memory layout transformations. `https://software.intel.com/en-us/articles/memory-layout-transformations`, 2015. Accessed: 2015-02-11.

[AMDM85] Henry Abarbanel, Gordon Macdonald, Russ Davis, and Gordon J Macdonald. "Bispectra". 1985.

[AR95] Sheperd Doeleman Alan Rogers. Fringe detection for very long baseline arrays. March, 1995.

[ART01] George W. Swensn Jr. A. RIchard Thompson, James M. Morgan. *"Interferometry and Synthesis in Radio Astronomy, 2nd Edition"*. 2001.

[Bax13] Richard Baxter. GPU-based Acceleration of Radio Interferometry Point Source Visibility Simulations in the MeqTrees Framework. 2013.

[BBD+09] A Burkert, W B Burton, M A Dopita, A Eckart, T Encrenaz, E K Grebel, B Leibundgut, J Lequeux, A Maeder, and V Trimble. *Radio Astronomy 5th ed.* 2009.

[BCC+13] N D R Bhat, J N Chengalur, P J Cox, Y Gupta, J Prasad, J Roy, S S Kudale, and W Van Straten. DETECTION OF FAST TRANSIENTS WITH RADIO INTERFEROMETRIC ARRAYS. 2013.

[Bea10] David Beazley. Understanding the Python GIL. 2010.

[Bed06] Timothy R Bedding. Introduction to Interferometry. (Figure 1), 2006.

[CGB08] T. J. Cornwell, K. Golap, and S. Bhatnagar. The non-coplanar baselines effect in radio interferometry: The W-Projection algorithm. pages 1–12, July 2008.

[Cur09] I. Curtis. Q . What exactly is RFI ? . 2009.

[DBK10] Wen-mei W. Hwu David B. Kirk. *Programming Massively Parallel Processors, A Hands-on Approach.* 2010.

[DFH09] Tim Colegate Paul Alexander Dominic Ford, Rosie Bolton and Peter Hall. The SKA Costing and Design Tool . 2009.

[Far11] Rob Farber. *CUDA application design and development.* 2011.

[Gui14] nvidia cuda c best practices guide: Design guide. `docs.nvidia.com/cuda/cuda-c-best-practices-guide/`, 2014. Accessed: 2014-11-14.

[hyp14] Hyper-threading. `http://en.wikipedia.org/wiki/Hyper-threading`, 2014. Accessed: 2014-12-14.

[IBH04] D. Horn J. Sugerman K. Fatahalian M. Houston I. Buck, T. Foley and P. Hanrahan. Brook for GPUs : Stream Computing on Graphics Hardware. 2004.

[JP09] Patrik Jonsson and Joel R Primack. Accelerating Dust Temperature Calculations with Graphics Processing Units. 2009.

[Ker15] Cuda kernel overhead. `http://www.cs.virginia.edu/~mwb7w/cuda_support/kernel_overhead.html`, 2015. Accessed: 2015-02-11.

[Kok13] Georgi Kokotanekov. LOFAR Transient Detection with a Bispectrum Algorithm. pages 47–62, 2013.

[Kul89] Shrinivas R Kulkarnial. Self-noise in interferometers: Radio and infrared. The Astronomical Journal vol.98(3):1112–1130, 1989.

[KWB12] G. J. Madsen K. W Bannister. A Galactic Origin for the Fast Radio Burst FRB010621. 2012.

[Lam13] Siu Kwan Lam. Cuda performance: Maximizing instruction-level parallelism. `http://continuum.io/blog/cudapy_ilp_opt`, 2013. Accessed: 2014-11-19.

[LB11] Casey J. Law and Geoffrey C. Bower. All Transients, All the Time: Real-Time Radio Transient Detection with Interferometric Closure Quantities. December 2011.

[LBP+12] Casey J. Law, Geoffrey C. Bower, Martin Pokorny, Michael P. Rupen, and Ken Sowinski. The RRAT Trap: Interferometric Localization of Radio Pulses from J0628+0909. August 2012.

[LK07] Christopher Laidler and Michelle M Kuttel. Detection of binary pulsars with GPU-accelerated sinusoidal Hough transformations . (Aulbert 2005):1–4, 2007.

[Lue08] David Luebke. CUDA: Scalable parallel programming for high-performance scientific computing. *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pages 836–838, May 2008.

[Mio14] Amy J. Mioduszewski. Radio interferometers around the world. 2014. Accessed: 2014-11-12.

[MLL+06] M a McLaughlin, a G Lyne, D R Lorimer, M Kramer, a J Faulkner, R N Manchester, J M Cordes, F Camilo, a Possenti, I H Stairs, G Hobbs, N D'Amico, M Burgay, and J T O'Brien. Transient radio bursts from rotating neutron stars. *Nature*, 439(7078):817–20, February 2006.

[Mol13] Gijs Molenaar. Trap status update. `http://www.slideshare.net/gijzelaerr1/ska-trap`, 2013. Accessed: 2014-11-14.

[nas97] Why do we study pulsars? `http://imagine.gsfc.nasa.gov/docs/ask_astro/answers/971112c.html`, Nov 1997. Accessed: 2014-11-14.

[nra14] Pulsar properties. `http://www.cv.nrao.edu/course/astr534/Pulsars.html`, 2014. Accessed: 2014-11-14.

[nVi14] nVidia. nvidia cuda c programming guide: Design guide. `docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2014. Accessed: 2014-11-14.

[ope09a] Opencl overview. `http://www.khronos.org/opencl`, 2009. Accessed: 2014-11-16.

[ope09b] An overview of openmp. http://openmp.org/mp-documents/ntu-vanderpas.pdf, Jan 2009. Accessed: 2014-12-06.

[Pee14] M. S. Peercy. Interactive multi-pass programmable shading. proceedings of the 27th annual conference on computer graphics and interactive techniques. http://portal.acm.org/citation.cfm?doid=344779.344976, 2014. Accessed: 2014-11-16.

[Pol15] The theory of interferometry and aperture synthesis. http://ned.ipac.caltech.edu/level5/March12/Middelberg/Middelberg2.html, 2015. Accessed: 2015-02-14.

[Row15] Row major ordering. http://en.wikipedia.org/wiki/Row-major_order, 2015. Accessed: 2015-02-11.

[SKA14] Everything you wanted to know about the ska. http://www.ska.ac.za/qa/index.php, 2014. Accessed: 2014-11-12.

[TM11] Rudolf Eigenmann Tim Mattson. OpenMP Overview. 2011.

[Tre13] Timothy Trewartha. Project Proposal Acceleration of AW-Projection Kernel Generation in Radio Astronomy. pages 1–7, 2013.

[Tun11] Ian William Tunbridge. Graphics Processing Unit Accelerated Coarse-Grained Protein-Protein Docking. (January), 2011.

[WA12] Mike Giles Wes Armour. Real-time de-dispersion in astrophysics. 2012.

[wik14a] Electromagnetic interference. http://en.wikipedia.org/wiki/Electromagnetic_interference, 2014. Accessed: 2014-11-13.

[wik14b] Pulsar. http://en.wikipedia.org/wiki/Pulsar, 2014. Accessed: 2014-11-14.

[Woo14] Daniel Wood. Master Thesis : Fast Galactic Structure Finding using Graphics Processing Units. pages 12–21, 2014.