New techniques in wide-field interferometric imaging

Benjamin James Mort

St. John's College

DPhil Thesis

Hilary term, 2011

Declaration

I declare that no part of this thesis has been accepted, or is currently being submitted, for any degree or diploma or certificate or any other qualification in this University or elsewhere.

This thesis is the result of my own work unless otherwise stated.

New techniques in wide-field interferometric imaging

Benjamin James Mort
St. John's College

DPhil Thesis

Hilary term, 2011

Abstract

With each new telescope that comes online, modern astronomy is ever moving towards a point where processing the resultant data deluge is arguably as much of a challenge as interpreting the science for which the instruments are designed. To this aim I have investigated the application of new processing techniques to two common problems in computationally intensive astronomical data reduction.

I present a comparative study of the CLEAN deconvolution algorithm applied to the spectral analysis of jet speed data from the microquasar SS 433 recorded over the last 26 years to investigate a claim made in the recent literature that there was a phase shift in one of the key periodicities observed. This investigation demonstrates how the CLEAN algorithm can be used to great effect for the spectral analysis of unevenly sampled time-series data, revealing characteristics that would otherwise be unattainable. While using this technique it was possible to rule out any strong evidence for a steady systematic phase variation in the jet speed periodicity, due to the well known relationship between frequency instability and phase variation. It was not possible with the current data set to definitively exclude the possibility that a variable frequency dependence of the jet speed on orbital period is at play in this microquasar.

Modern interferometric telescopes mandate innovative use of computers for data processing because (i) huge data rates and sensitive detectors require complex algorithms to optimally extract the valuable science results and (ii) the rising costs of electricity and the environmental impact of excessive power consumption means that simply allocating more and more computing resources to solve the problem is impractical. To this context I describe my implementation of the w-projection algorithm for imaging wide-field interferometric data on graphics processing units (GPU). In this study I identify the advantages and pitfalls of using GPUs for the processing of astronomical data and demonstrate that GPUs present a fast, cost effective and energy efficient solution to imaging for the next generation of radio interferometers where the value of FLOPS per watt is increasingly an important factor in developing new processing solutions. While investigating the application of imaging on the GPU, I have developed a modular, flexible testbed application for developing, studying and testing the deployment of GPU algorithms for interferometric imaging can I present this code.

Acknowledgements

Over the many years it has taken me to complete this thesis I have been extremely fortunate to have had the support of my supervisor, colleagues, friends and family.

My supervisor Katherine Blundell has been a fantastic advisor and I have been extremely fortunate for the generous support and encouragement she has always given me. In this project, which had a largely technical focus, deciding on what to pursue has not always been easy and Katherine's clear vision and guidance throughout has been invaluable. I could not ask for a better advisor and it is difficult to express the debt I feel.

I am very grateful to Eric Greissen of NRAO with whom, early on in my thesis, valuable discussions about the imaging software in *AIPS* taught me a great deal about synthesis imaging and software development. It was these discussions that played a large part in directing me towards the subject which I eventually pursued.

I'd like to thank Ian Heywood who generated simulated data which I used to test my imaging code. I am very grateful for his generous and enthusiastic help on using *AIPS* as well as for discussions I had with him about interferometric imaging from which I have learned a great deal.

I would also like to thank my colleagues at the Oxford e-Research Centre, in particular Fred Dulwich and Stef Salvini along with the Director of the department Anne Trefethen - without their enduring support and encouragement finishing of this thesis would have been difficult.

My studentship has been funded by PPARC's e-science budget, and I feel very lucky that this has allowed me to study some technical computing solutions towards the challenges that will come with the next generation of instrument, a subject which I have grown to be particularly passionate about.

Finally I'd like to thank my parents without whose support, over the seemingly never-

ending course of this study, none of this would have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With each new telescope that comes online, modern astronomy is ever moving towards a point where processing the resultant data deluge is arguably as much of a challenge as interpreting the science for which the instruments are designed. To a certain extent, this is already the case with the data rates from LOFAR[1], ALMA[2] and the EVLA[3] already pushing the limits of what current generations of radio astronomy software can deliver. When the first phase of the SKA[4] comes online in ∼2020 however, the expected data rate of ∼1 terabyte per minute (Taylor 2008) from the imaging correlator will require a paradigm shift in the way astronomy is carried out.

In addition to the huge computing resources required to handle the large data rates, in order to take full advantage of the data generated from increasingly sophisticated detectors and instruments (e.g. Ekers 2001), new analytical and numerical mathematical techniques will be required. For example, in order to obtain imaging dynamic ranges of greater than $\sim 10^6$ from interferometers constructed using aperture arrays, imaging will need to take into account the primary beam asymmetries and subtract out sources from a region of the sky corresponding to pretty much the entire hemispere above the detectors. Therefore, the

[1]http://www.lofar.org/
[2]http://www.almaobservatory.org/
[3]http://www.aoc.nrao.edu/evla/
[4]http://www.skatelescope.org/

combination of increasing data rates and the need for sophisticated algorithms means that the study of new data processing techniques from both a mathematical and computational context is ever pressing.

Much of the analysis of astronomy data can be classified as signal processing and much can be learnt from creative signal processing techniques applied in new ways. Data retrieval of subtle temporal signals using Fourier inversion of time series data followed by removal of artifacts due to incomplete sampling by deconvolution, a technique which has its origins in imaging, explored in Chapter 3, is such an example.

As the potential to make both redshift (HI) and radio continuum surveys that will revolutionise cosmological studies is one of the main science drivers of building new instruments (e.g. Blake et al. 2004), telescopes are increasingly being designed to observe with multiple instantaneous large fields of view to match the required survey speed. While wide-field imaging techniques, particularly at low frequencies, have been studied and used for the last $\sim$ 10 years (Cornwell & Perley 1992), traditional techniques are computationally expensive and not particularly suited to providing the high image fidelity, high dynamic range images required for using new instruments at all, still less to their full potential.

Over the last few decades the solution to increasingly demanding computing problems has been underpinned by the semiconductor industry's ability to cram ever increasing numbers of transistors onto silicon, an observation made by Moore (1965). Over the years keeping up this trend has been increasingly difficult as each semiconductor die shrink is required to place more transistors on the same area of silicon, increases manufacturing costs and decreases production yields. In order to match consumer expectations, processor manufacturers have therefore had to resort to a variety of alternative techniques such as for example increasing clock frequencies and developing multi-core processor designs. Ultimately however the success of these design 'tricks' to match Moore's law is limited by

the ability to dissipate heat generated by high density, highly clocked transistors and the communication latencies introduced as the surface area of multi-core processors become larger. As a result, programmers must increasingly look towards parallel algorithms for numerically intensive code when more computing power is required.

While huge multi-core, multi-processor solutions might promise to solve the large computing problems facing astronomy over the next decade, the price and environmental footprint of excessive power consumption is an increasing concern and building a nuclear power station next to each new radio telescope really isn't a viable option! This situation therefore mandates the use of innovative hardware and algorithms for imaging radio interferometric data and these I have investigated and explored in the context of developing *WFIT*.

1.1 The structure of this thesis

The main theme of this thesis is the study and implementation of new techniques to astronomical data processing. This is presented as two studies: one looking at how the CLEAN algorithm, traditionally related to the improvement of images from synthesis imaging radio interferometers, can be applied to the study of time series data from the time-variable microquasar SS 433; and the other investigating a relatively new algorithm for wide-field imaging of interferometric data using graphics processors (GPUs).

In this thesis I present in Chapter 2 a review of the fundamental theory of synthesis imaging of data observed by radio interferometers. I describe in particular the problems and a number of solutions to imaging wide-fields which pose a challege for low-frequency observations.

Chapter 3 then presents a comparative study of the CLEAN deconvolution algorithm, traditionally used to improve images produced in Fourier synthesis imaging, for the anal-

ysis of the periodicities in the complex dynamical motions of SS 433 using observations taken over the last 26 years. In particular I wanted to investigate a claim in the recent literature that there was a phase shift in one of the key periodicities observed.

In Chapter 4 I introduce the concept of programming GPUs for numerical processing. During the process of writing this thesis programming GPUs has evolved from a black art to an increasingly mainstream aspect of advanced high performance computing (HPC) and this chapter introduces both aspects of this. I describe a simple investigation of convolution using the graphics pipeline I have developed and I introduce the CUDA programming system developed by NVIDIA specifically targeted at numerical programming on GPUs.

Chapter 5 introduces the *WFIT* code which I developed as a platform for integrating and testing and comparing GPU and CPU algorithms for imaging interferometric data. This framework was used to generate the images and analyse the algorithms presented in this thesis.

Chapter 6 describes my implementation of the so-called w-projection algorithm for the imaging of wide-field interferometric data on the GPU. In addition to an analysis of the implementation I discuss the challenges and pitfalls of developing this algorithm on the GPU. I present a critique of the important consideration in aspects of code design for implementation on the GPU and compare and contrast this with deployment on the CPU describing how the CPU can, for certain steps in the algorithmic edifice required to process wide-field astronomical data, play a helpful complementary role.

Chapter 7 then presents results and analysis of using the GPU implementation of w-projection integrated into *WFIT* to image a simulated data set, which was used for testing and verification purposes, and a low-frequency wide-field observation by the GMRT[1].

The thesis closes with a summary of what has been learned from these studies and rec-

[1] http://gmrt.ncra.tifr.res.in/

ommendations of the direction for future research in these highly relevant methodologies that will be crucial for the future of next generation astronomical science.

Chapter 2

Synthesis imaging in interferometry

In this chapter I address the majority of the background core science and mathematical theory involved in writing the interferometric imaging code, central to this thesis. With the theme of this project being strongly computing based, this chapter serves to explain the context of the study.

I begin this chapter with a summary of the main principles fundamental to imaging with a radio interferometer. This discussion is somewhat based on the the excellent description given by Synthesis Imaging in Radio Astronomy II (1999) and Thompson et al. (2004). This is followed by an introduction to a number of challenges that occur when imaging low frequency and wide-field observations and a description of the w-projection algorithm developed by Cornwell et al. (2005). The w-projection algorithm is a computationally efficient solution to imaging wide fields of view and is the algorithm which I have chose to investigate when writing imaging code for the GPU, described in Chapter 6. It should be noted that w-projection also forms the basis for a range of new algorithms, currently under development, which correct for direction dependent imaging effects. One of the most promising of these is the A-projection algorithm (Bhatnagar et al. 2008) which attempts to correct for primary beam effects of individual antenna elements during gridding.

2.1 The Fundamentals of Synthesis Imaging

Interferometric arrays form images of the sky indirectly, following a number of distinct steps:

- Measurement of the mutual spatial coherence function of the electric field response of a number of pairs of antennas, generating a set of complex visibilities.
- Flagging and calibration of the measured visibilities.
- Fourier Inversion of the sampled visibilities.
- Deconvolution to remove the effects of incomplete and uneven sampling.
- Self-calibration to correct for systematic phase and amplitude errors in the measured visibility.

All of these steps are vital in forming the final astronomical image and each present their own challenges both computationally and in preserving the underlying physics.

2.1.1 Measuring the Interferometer Response

A synthesis interferometer measures the complex visibility at pairs of antennas and as such, the visibility measurements from an interferometer can be described by a collection of two-element instruments as shown in Figure 2.1. The two antennas, separated by a baseline vector $\overline{b}$, point at a distant source of interest indicated by unit vector $\overline{s}$. The wavefront from the source reaches one antenna at a time τ_g later than the other corresponding to the geometrical delay between the antennas in the direction of the source.

$$\tau_g = b \cdot \frac{s}{c}\,, \tag{2.1}$$

where c is the speed of light.

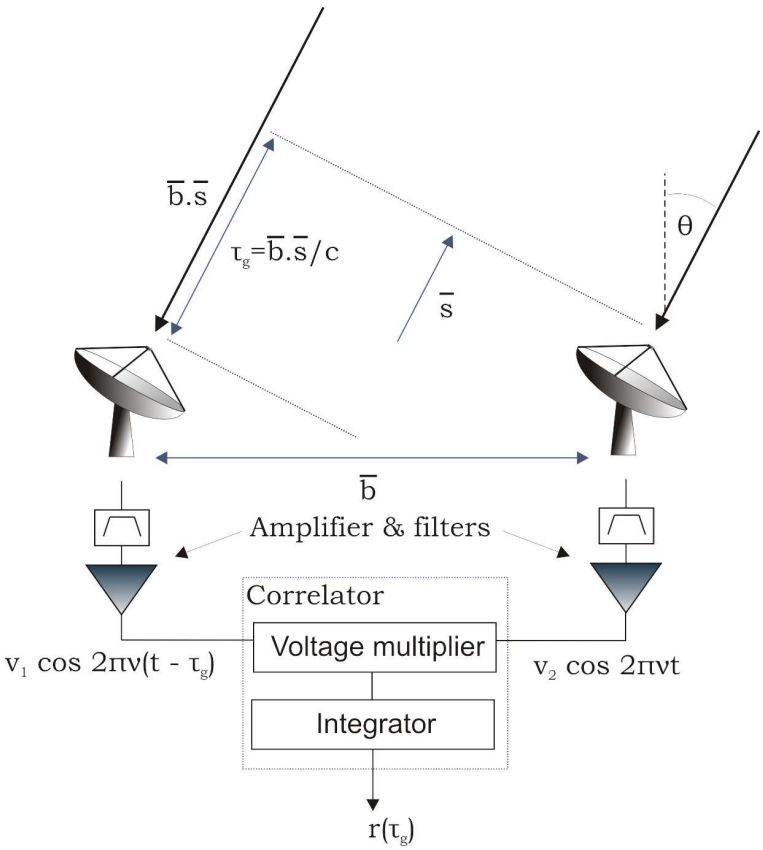


Figure 2.1: Schematic diagram of a two-element interferometer.

Signals from each antenna then pass though a signal processing block which includes amplifiers and filters to select the required frequency band and bandwidth. Following this, the signals are combined by voltage multiplication followed by time averaging in a correlator. The correlator takes input voltages $V_1(t)$ and $V_2(t)$ and generates an output

$$\langle V_1(t)V_2(t)\rangle \ . \tag{2.2}$$

We can represent the signals as quasi-monochromatic Fourier components of frequency ν, which have the form $V_1(t) = v_1 cos 2\pi\nu(t - \tau_g)$ leading to a correlator output:

$$r(\tau_g) = v_1 v_2 \ \cos(2\pi\nu\tau_g) \ . \tag{2.3}$$

The geometric time delay τ_g varies slowly with time as the Earth rotates, and resulting

oscillations of the cosine term in Equation 2.4 represent the motion of the source though the interferometer fringe pattern. The term $v_1 v_2$, represents the fringe amplitude and is proportional to the received power. It is useful to express the interferometer output as a function of the brightness of radio emission integrated over the sky. If $I(\mathbf{s})$ is the brightness or intensity in the direction of unit vector $\mathbf{s}$ at frequency ν and has the units of $\mathrm{Wm^{-2}Hz^{-1}sr^{-1}}$, then the signal power recieved in bandwidth $\Delta\nu$ from the source element $d\Omega$ is $A(\mathbf{s})\, I(\mathbf{s})\, \Delta\nu\, d\Omega$, where $A(\mathbf{s})$ is the effective collecting area in direction $\mathbf{s}$. $A(\mathbf{s})$ is associated with the cross power beam pattern of the antenna elements that make up the baseline being correlated. The resulting output from the correlator is proportional to the received power and to the cosine fringe term. Therefore the correlator output for the signal from solid angle $d\Omega$ in terms of the baseline and source position vectors is:

$$r = \Delta\nu \int_S A(\mathbf{s})\, I(\mathbf{s})\, \cos\, \frac{2\pi\nu \mathbf{b}\cdot\mathbf{s}}{c}\, d\Omega\,. \tag{2.4}$$

The integral is taken over the entire surface S of the celestial sphere and we assume that the bandwidth $\Delta\nu$ is sufficiently small that variation of the collecting area (A) and brightness (I) with frequency (ν) can be ignored.

In order to produce an interferometric image, the *phase tracking centre* or *phase reference position*, on which the field of view is centred must be specified. This is shown in Figure 2.2 by the vector s_0. The direction to the source of interest can then expressed in terms of the phase tracking centre vector s_0, as $s = s_0 + \sigma$ and therefore equation 2.4 can then be written as:

$$\begin{aligned} r &= \Delta\nu \cos\left(\frac{2\pi\nu\, \mathbf{b}\cdot\mathbf{s_0}}{c}\right) \int_S A(\sigma)\, I(\sigma) \cos\frac{2\pi\nu \mathbf{b}\cdot\sigma}{c}\, d\Omega \\ &- \Delta\nu \sin\left(\frac{2\pi\nu\, \mathbf{b}\cdot\mathbf{s_0}}{c}\right) \int_S A(\sigma)\, I(\sigma) \sin\frac{2\pi\nu \mathbf{b}\cdot\sigma}{c}\, d\Omega\,. \end{aligned} \tag{2.5}$$

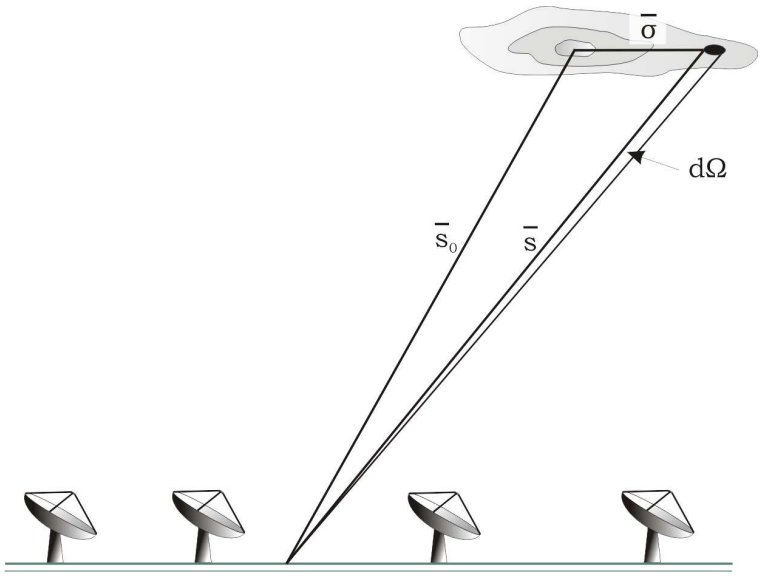


Figure 2.2: Position vectors relating to a patch of radio brightness $I(s)$, represented by contours in the figure.

Introducing the *visibility*, which is a measure of the spatial coherence of the source distribution, measured though the electric field, with dimensions of spectral power flux density ($\mathrm{W\,m^{-2}\,Hz^{-1}}$), and defined as

$$V \equiv |V|\, e^{i\phi_{\mathrm{v}}} = \int_S A'(\sigma)\, I(\sigma)\, e^{-2\pi i \nu \mathbf{b}\cdot\sigma/c}\, d\Omega\,, \tag{2.6}$$

where $A' \equiv A(\sigma)/A_0$ is the normalised antenna response. Since when forming an image antennas track the source, the system therefore responds to the modified brightness distribution $A'(\sigma)I(\sigma)$. Combining equations 2.6 and 2.5 gives:

$$r = A_0 \Delta\nu\, |V| \cos\left(\frac{2\pi\nu\mathbf{b}\cdot\mathbf{s_0}}{c} - \phi_{\mathrm{v}}\right). \tag{2.7}$$

Therefore visibility measurements can be obtained from an interferometer measurement of the phase and amplitude of the fringe pattern represented by the cosine term of the correlated response of equation 2.7. If measured for a sufficiently large range of $\nu\mathbf{b}\cdot\sigma/c$ (the baseline component in the direction of the source measured in wavelengths), the

visibilities, though inversion of equation 2.6 will then give the brightness distribution and therefore an image of the sky.

2.1.2 Coordinate system for imaging

The definition of a coordinate system is vital to defining a practical method to convert visibilities into a measure of the sky brightness distribution. Figure 2.3 shows the basic coordinate system used. The baseline vector has components (u, v, w) where the w points in the direction of interest or phase tracking centre. u, v, and w are measured in wavelengths at the centre frequency of the RF signal band, and in directions towards the East, the North, and the phase tracking centre, respectively. Positions on the sky are defined on a tangent plane specified in units of l and m, which are direction cosines specified with respect to the u and v axes. Distances in l and m are proportional to the sine of the angles measured from the origin according to an orthographic projection. In these coordinates, the equation relating the brightness distribution to the visibility (Equation 2.6) becomes:

$$V(u,v,w) = \int_{\infty}^{\infty}\int_{\infty}^{\infty} A'(l,m)I(l,m)\, e^{-2\pi i[ul+vm+w(\sqrt{1-l^2-m^2}-1)]}\, \frac{dldm}{\sqrt{1-l^2-m^2}}\,, \qquad (2.8)$$

where the integrand is taken to be zero for $l^2+m^2 \geq 1$ and $A'(l,m)\, I(l,m)$ is the modified brightness distribution.

2.1.3 Imaging by Fourier inversion

With a radio synthesis interferometer measuring a set of complex visibilities, $V(u, v, w)$, the sky brightness distribution or spectral intensity distribution, $I(l, m)$, can be obtained by inversion of equation 2.8. It is important to note that this equation only valid under the assumption of a phase tracking interferometer observing narrow-band, spatially incoherent, far-field radiation and that a synthesised image on the l, m plane represents a

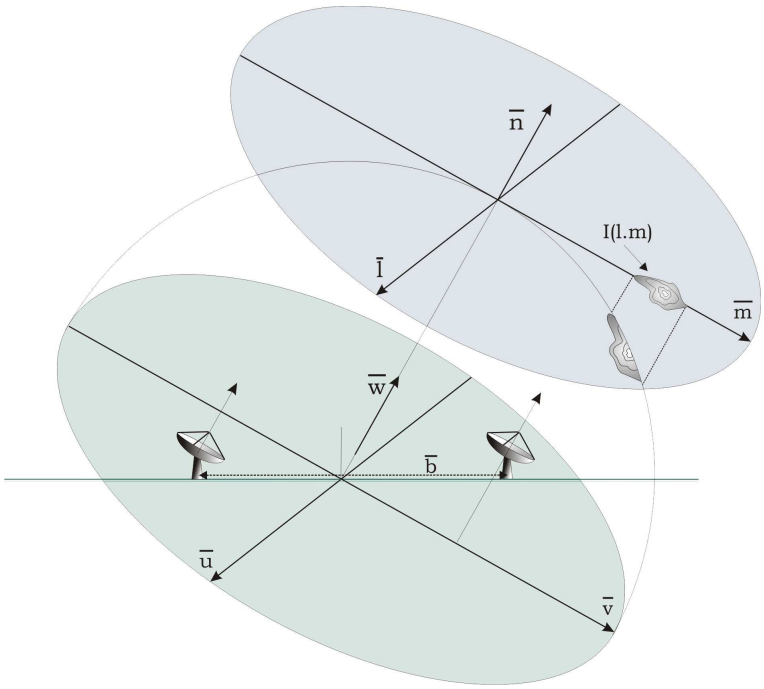


Figure 2.3: The (u, v, w) and (l, m, n) right-handed coordinate systems used to express the interferometer baselines and source brightness distribution respectively.

projection of the celestial sphere onto a tangent plane at the l, m origin (see Figure 2.3).

To simplify the inversion of equation 2.8, from which the brightness distribution is derived, it is useful to reduce this equation to a two-dimensional Fourier Transform relation. This is possible under two sets of conditions: (i) when the baselines are coplanar and/or (ii) when the field of view is sufficiently small.

Baselines are coplanar for one-dimensional east-west arrays (e.g. The Westerbork Synthesis Radio Telescope, WSRT) when as the earth rotates, the components of the baseline vector parallel to the earth's rotation axis are zero. When this is the case, the w-axis can be chosen to lie in the direction of the celestial pole, so that $w = 0$, and equation 2.8 reverts to a two-dimensional Fourier transform.

In an array such as the Very Large Array (VLA), the baseline vectors however do not remain coplanar in (u, v, w) space. Fortunately equation 2.8 also reduces to a two-dimensional FT if the field of view is small and therefore the $|l|$ and $|m|$ dimensions are

small enough that it is possible to write

$$(\sqrt{1-l^2-m^2}-1)\,w \approx -\frac{1}{2}\,(l^2+m^2)\,w \approx 0\,. \tag{2.9}$$

When this condition is satisfied it is clear from inspection, that equation 2.8 again reverts to the form of a 2-dimensional Fourier Transform.

$$V(u,v) = \int\int I(l,m)\,e^{-2\pi i(ul+vm)}\,dldm\,. \tag{2.10}$$

Both of these conditions can be shown to reduce to the requirement that the position dependent phase shift is much less than unity (i.e. $2\pi w\left(\sqrt{1-l^2-m^2}-1\right) \ll 1$). The value of this term scales roughly with the maximum telescope baseline B, the individual antenna diameter D and λ the observing wavelength as:

$$\frac{B\lambda}{D^2}\,, \tag{2.11}$$

so that for a small diameter antenna or low frequencies this term becomes significant.

2.1.3.1 Direct Fourier inversion

In the conditions of a two-dimensional visibility to sky brightness relation, the most straightforward method of obtaining the intensity distribution from measured visibility data is by direct Fourier inversion of equation 2.10. With the measured visibility expressed as:

$$V_{\text{measured}}(u,v) = S(u,v)\,w(u,v)\,V(u,v)\,, \tag{2.12}$$

where $S(u,v)$ is the transfer function or spatial sensitivity function and $w(u,v)$ represents any applied weighting. The Fourier transform of equation 2.12 is the measured intensity

distribution, which is

$$I_{\text{measured}}(l,m) = I(l,m) \ast\ast b_0(l,m)\,, \tag{2.13}$$

here the double asterisk indicates two-dimensional convolution and b_0 is the synthesised beam, the Fourier transform of the weighted transfer function.

With the visibility measured at an ensemble of n_d pairs of points symmetric about the (u,v) origin, the direct transform of the data is represented by

$$\sum_{i=1}^{n_d} w_i \left[V(u_i,v_i)\, e^{j2\pi(u_i l + v_i m)} + V(-u_i,-v_i)\, e^{-j2\pi(u_i l + v_i m)}\right]\,, \tag{2.14}$$

where the weighting factor w_i is introduced to control the shape of the synthesised beam and with the visibility at $(-u_i,-v_i)$ the complex conjugate of the visibility at (u_i,v_i) (i.e. Hermitian), the derived intensity is real. When taking this Fourier transform, the resultant intensity distribution is usually computed at points on a regular grid, with uniform increments in l and m, as this serves to make subsequent processing simpler. While inversion by direct transform may at first seem simple and therefore desirable, the number of operations required to evaluate the sky brightness on an $N \times N$ grid from M values of the complex visibility is $4MN^2$. In practice this will therefore make imaging by direct transform unfeasible for anything but very small images and alternatives such as the fast Fourier transform have to be used.

2.1.3.2 The sampling function and weighting

Equation 2.12 introduced the sampling function and visibility weighting. Both of these play a critical role in controlling the shape of the synthesised beam and therefore need to be considered carefully.

The sampling function $S(u,v)$ describes the distribution of measured visibility points

and can be expressed as a two-dimensional Dirac delta function distribution

$$S(u,v) = \sum_{k=1}^{n_d} \delta(u-u_k, v-v_k), \tag{2.15}$$

such that the sampled visibilities are given by

$$V^s(u,v) \equiv \sum_{k=1}^{n_d} \delta(u-u_k, v-v_k) V(u_k, v_k)\ . \tag{2.16}$$

Using this formulation, the brightness distribution, the Fourier transform of the measured visibilities, can be written as

$$I^D = \mathfrak{F}(V^s) = \mathfrak{F}(SV)\ , \tag{2.17}$$

and by the convolution theorem

$$I^D = \mathfrak{F}(S) * \mathfrak{F}(V)\ . \tag{2.18}$$

Using this, it can therefore be seen that for a point source of unit strength centered at position (l_0, m_0), the point source response of an array or the *synthesised beam* shape is given by

$$B(u,v) = \mathfrak{F}(S)\ . \tag{2.19}$$

According to equation 2.18, the sky brightness distribution given by the direct transform of equation 2.14 is that of the actual brightness distribution convolved with the synthesised beam.

As the sampling distribution of visibilities describes the shape of the synthesised beam and also as a direct baring on the signal-to-noise ratio in the raw brightness map, weighting of the samples can have a beneficial influence on the final image. As such, weighting

coefficients are commonly assigned to visibility points corresponding according number of distinct categories (see Briggs (1995)); data reliability (derived from integration time, system temperature and bandwidth etc.), tapering by a smooth function and density weighting to compensate for clumping of data in the (u, v) plane.

In general, to obtain the best signal-to-noise in the summation of measurements that contain Gaussian noise, the individual data values should be weighted by the inverse of their variance. Thus for the best signal to noise ratio, the weights should be inversely proportional to the variances, i.e.

$$w_i = \omega_i = \frac{1}{\sigma_i^2} \,. \tag{2.20}$$

If the data are obtained with a uniform array of antennas and receivers, and the averaging of time is the same for all data, the variances should all be the same and the maximum signal to noise is obtained by including all measurements with the same weight. This is known as *natural* weighting. For most arrays however, natural weighting results in a poor synthesised beam shape with wide skirts which result from shorter spacings being over emphasised. Thus the usual approach is to include a weighting factor that is inversely proportional to the sample density in the (u, v) plane. i.e.

$$w_i = \frac{\omega_i}{W_k} \,, \tag{2.21}$$

where W_k is a local density factor, whose amplitude is calculated by gridding the inverse variance of samples onto a grid with a pixel size based on the inverse of the image field of view. With this approach, known as *uniform* weighting, a Gaussian taper usually also has to be added to control the near in side-lobes of the resultant synthesised beam, which would otherwise pose serious problems to subsequent image processing. A third weighting scheme known as *robust* weighting was introduced by Briggs (1995) which goes

someway to avoid the problems of the other two approaches. Robust weighting combines the strategy of local sample density compensation with a knowledge of the signal-to-noise ratio of individual points to suppress the contribution of noisy samples. The *robust* weight for a sample i is given by

$$w_i = \frac{\omega i}{1 + W_k S}\,, \tag{2.22}$$

where ω_i is the inverse variance of the sample signal defined in equation 2.20, W_k is a density weighting factor given by the sum of the weights of neighboring samples as defined in *uniform* weighting, and S is a function of the robust parameter, R, defined as

$$S = \frac{(5 \cdot 10^R)^2}{\frac{\sum_k W_k^2}{\sum_i \omega_i}}\,. \tag{2.23}$$

The value of R is then chosen to give the desired balance between resolution and sensitivity. If R is large and positive, the weighting is close to *natural* giving maximum sensitivity, and if R is large and negative S is very small and the resulting weights are virtually *uniform*.

2.1.3.3 The Discrete Fourier transform and gridding visibilities

While inversion of the visibilities to form a map of sky brightness can be achieved by direct transform (as described by equation 2.14), this method is prohibitively time consuming for anything other than small maps. Fortunately, as this inversion takes the form of a Fourier transform relation, the efficiently of the fast Fourier transform (Cooley & Tukey (1965)) can be used to full advantage. Use of the fast Fourier transform (FFT) however, introduces two additional complications; interferometer geometries result in observed visibility points that do not lie on a regular Cartesian grid required for the FFT and gridding can lead to aliasing of parts of the image outside the synthesised field.

While there are many ways that interpolation of visibilities onto a grid can be per-

formed, convolution and re-sampling in the u, v plane is usually the preferred method as it leads to an image with predictable distortions. Gridding by convolution can achieved by evaluating a convolution at each grid point (u_g, v_g) according to:

$$\sum_{k=1}^{M} G(u_g - u_k,\, u_g - u_k)\, V^W(u_k, v_k)\,, \tag{2.24}$$

where $V^W(u, v)$ are the weighted, sampled visibilities and $G(u, v)$ is a convolution or smoothing function. The gridded visibilities are then described by the following expression:

$${}^2\mathrm{III}\left(\frac{u}{\Delta u}, \frac{v}{\Delta v}\right)\left[G(u, v) \ast\ast V^W(u, v)\right]\,. \tag{2.25}$$

The re-sampling is represented by the two-dimensional shah function ^{2}III defined as:

$${}^2\mathrm{III}\left(\frac{u}{\Delta u}, \frac{v}{\Delta v}\right) = \Delta u \Delta v \sum_{i=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} {}^2\delta(u - i\Delta u,\, v - k\Delta v)\,, \tag{2.26}$$

where ${}^2\delta$ is the two-dimensional delta function and Δu and Δv define the cell size or separation between grid points.

With the FFT inversion generating one period of the brightness distribution, imaging a region $N_l \Delta\theta_l$ radians in l, by $N_m \Delta\theta_m$ radians in m, grid spacings must be chosen such that for an image of N_l by N_m,

$$\begin{aligned} N_l \Delta u &= 1/\Delta\theta_l\,, \\ N_m \Delta v &= 1/\Delta\theta_m\,, \end{aligned} \tag{2.27}$$

where $\Delta\theta l$ and $\Delta\theta m$ are pixel separations in radians and as a result re-sampling makes the brightness map a periodic function of l and m by $1/\Delta u$ and $1/\Delta v$ respectively.

Another effect that requires careful attention is aliasing whereby under-sampling and

the truncation of sampling at the boundaries of the u, v plane, cause sky brightness outside the primary field of view to be folded back into it. While aliasing is best avoided by making the image large enough so that there are no sources of interest near the edges of the image and avoiding under-sampling completely, careful choice of a convolutional gridding function whose Fourier transform drops off very rapidly outside the edge of the image is in practice required. While this condition suggests that convolution functions should not be tightly confined within a small area of the u, v plane, the requirement for reasonable computational time dictates the opposite. In practice therefore a function giving the best compromise of these conditions has to be sought. Typical choices of the convolution function are (see Greisen (1979)):

- Pillbox function.
- Truncated exponential.
- Truncated sinc function.
- Exponential multiplied by a truncated sinc.
- Truncated spheroidal function.

with spheroidal functions usually being favoured due to a good balance of alias rejection across the image.

Finally, it should be noted that the effect of the gridding convolution is to multiply the sky brightness by a function $g(l, m)$, the Fourier transform of the convolution function $G(u, v)$. This effect can be removed by point-wise division by g to produce a *grid-corrected* sky brightness map.

Whereas the computational cost of inversion by direct transform required $O(4MN^2)$ operations this method of gridding followed by use of the fast Fourier transform requires

$O(M)$ operations for gridding followed by $O(N^2 \log_2 N)$ operations for the FFT, and therefore clearly saves a huge amount of time for large numbers of visibilities.

2.1.4 Refining the image

After Fourier inversion of equation 2.10 a brightness distribution, known as a 'dirty' image is formed. This distribution shows defects imposed by limited sampling of the u, v plane, as well as errors left behind from initial calibration of the visibilities. In the process of generating a map of the radio sky, this 'dirty' image is then often refined through the processes of deconvolution and self calibration.

2.1.4.1 Deconvolution

Deconvolution is a process which attempts to correct for poor u, v plane sampling and the resulting effect of spurious side-lobes on the brightness distribution. The intensity distribution $I^D(l, m)$ obtained in synthesis imaging can be expressed as the true intensity $I(l, m)$ convolved with the synthesised beam, or Fourier transform of the weighted sampling function $B(l, m)$:

$$I^D(l, m) = I(l, m) ** B(l, m) \, . \tag{2.28}$$

With knowledge of both $I^D(l, m)$ and $B(l, m)$ deconvolution attempts to solve for $I(l, m)$. For the case of measurements of visibility with a synthesis interferometer, an analytic solution for the true intensity distribution is impossible as the transfer function contains areas of unmeasured and therefore zero values. As a result other than weighting the measured visibilities as described in section 2.1.3.2 any process that improves the derived intensity map must involve placing non-zero visibilities in unmeasured areas and therefore result in a non-unique solution. While this is somewhat undesirable, the zero values in

the measured visibilities used to generate the dirty map, $I^D(l,m)$, are also somewhat arbitrary and a procedure by which visibility values at unmeasured points take values consistent with the most likely intensity distribution, while minimising the addition of arbitrary brightness structures is desirable. Various algorithms exist to this aim, with the principal ones being CLEAN and the Maximum Entropy method (MEM).

The CLEAN algorithm (Högbom (1974)) is a non-linear iterative procedure whereby scaled versions of the point spread function (dirty beam) are removed from the positions of peaks in the dirty image. Various implementations of this algorithm exist operating in both the image (l,m) and Fourier (u,v) domain with the following common steps:

1. Compute the measured intensity distribution or 'dirty map' and point spread function or 'dirty beam' from the visibilities and the weighted sampling function. The spacing of points should be chosen such that at least three points span the width of the synthesised beam.

2. Find the highest intensity point on the dirty map and from this subtract a scaled point response map centred on at this position. The scaling factor used in the subtraction is known as the loop gain and has a value in the range $0.0 \geq gain \geq 1.0$, typically 0.1. The position and amplitude of the component removed should be recorded by adding a delta-function component to a model that will become the cleaned map.

3. Repeat step 2 iteratively until all significant structure has been removed from the residual dirty image and the remaining residuals consist mainly of noise. This can be assessed by comparing the highest peak to the rms level of the remaining residuals or determining when the rms level fails to decrease after subsequent iterations.

4. Convolve the recorded delta functions in the cleaned model with a clean-beam response chosen such that its width equals the half-amplitude width of the synthesised

'dirty' beam.

5. Add the final residuals remaining from step 2 to generate the clean map.

The standard implementation of CLEAN, as described here, assumes that the brightness distribution can be modelled correctly by a collection of point sources and therefore can be weaker at deconvolving extended emission. Despite this short falling, groups of delta-functions can be used to successfully model extended sources with a variation on CLEAN known as multi-scale CLEAN (Cornwell (2008)) as been demonstrated to be particularly successful (e.g. Rich et al. (2008)). It is important to note however that while CLEAN has proven successful and is straightforward to implement, full analysis of the response of the algorithm is difficult with the response of the algorithm not producing unique solutions.

The Maximum Entropy method (MEM) algorithm, unlike CLEAN, is not procedural but selects an image that fits the data to within the noise level, while constraining the result to maximise the 'entropy' function, a measure of the image quality. Entropy is in imaging a quantity, that when maximised, produces a positive image with a compressed range in pixels. In maximisation of the 'entropy' the constraint is made that the resulting intensity model is consistent with the measured visibility data though a χ^2 statistic. In contrast to CLEAN, MEM provides a mechanism to analyse the effect on the resultant noise though the χ^2 statistic. Not assuming the model sky map to be a collection of point sources it also in general performs better on smooth, extended emission but can have difficulty in resolving point sources. As described by Cornwell & Evans (1985), the computational cost of MEM is generally found to be comparable to CLEAN. The exception to this occurs when the image being deconvolved is filled with extended or contains a small number of point sources. Extended emission requires a large number of CLEAN components to model and therefore performance would favour MEM however

in the regime of a few point sources CLEAN is found to be faster by up to an order of magnitude.

2.1.4.2 Self-Calibration

While traditional methods of calibration of the visibility amplitudes can be performed to a few percent, phase errors may be much larger. Self-calibration uses a series of consistency arguments to eliminate factors such as noise and spatial and temporal fluctuations in the earth's atmosphere. Broadly speaking self-calibration uses the fact that many of the systematic errors in the visibilities may be attributed to individual array elements and these are left as free parameters to be derived along with the sky intensity distribution. The procedure is to use a least-squares method to minimise the modulus of the difference between the observed visibilities and the visibilities derived from a model of the brightness distribution. The procedure takes the following steps:

1. Make an initial map of the brightness distribution.

2. Compute a set of antenna gains which calibrate the observed visibilities to those predicted by the model.

3. Use the gains to calibrate the observed visibilities and compute an updated map.

4. Use CLEAN to select a set of components to provide positivity and confinement of the image.

5. Test for convergence and return to step 2 as required.

One can restrict the self-calibration to an improvement of phase alone or to phase and amplitude. If properly used, this method leads to a great improvement in interferometer images, especially those of compact intense sources. If this method is used on objects

with low signal-to-noise ratios, it may give very wrong results e.g. concentrating random noise into spurious parts of the interferometer image.

In practice, in order to form the final image not only are successive stages of self-calibration and deconvolution required, but visual inspection with a good degree of experience is also critical.

2.2 The challenges of working at low frequencies

While challenging, the calculation of the intensity distribution of radio emission from a set of measured visibilities in the regime of two-dimensional Fourier inversion is relatively straightforward from both a conceptual and computational perspective. As explained in the following section, when working with low frequencies or wide fields of view many of the approximations used to derive the relatively simple imaging equations break down and the problem of evaluating sky the brightness distribution becomes a complex and computationally expensive task. The principal challenges of working at low frequency are described in the following sections.

2.2.1 Bandwidth smearing

Bandwidth smearing or chromatic aberration, not unique to radio astronomy, is a familiar problem to all observers. The general imaging equation for synthesis arrays (equation 2.8) assumes that the radiation is monochromatic. If, however, as is always the case for a real instrument, radiation is spread over a finite bandwidth, aberration will occur with the sources in the image being radially smeared, with the smearing not appearing as a position-independent convolution since the smearing function is a function of the (l, m) coordinate.

The effect of bandwidth smearing is proportional to the fractional bandwidth $\frac{\Delta\nu}{\nu}$

and separation from the phase tracking centre $\sqrt{l^2+m^2}$ and can be described by the bandwidth smearing parameter:

$$\beta = \frac{\Delta v}{v_0} \cdot \frac{\theta}{\theta_{\mathrm{B}}}\,, \tag{2.29}$$

where θ is the distance from the phase centre and θ_{B} is the half amplitude width of the synthesised beam.

This effect is particularly a problem for low frequencies where with a large field of view many sources lie at considerable distances from the phase tracking centre and where new instruments feature huge fractional bandwidths, rising to as much as 100% of the central frequency! Traditionally bandwidth smearing has also been less of a problem where affected sources are not directly of interest but only imaged to remove the effect of side lobes in a region closer to the phase tracking centre, however with the need to image ever larger high fidelity fields this cannot be assumed. In order to generate a undistorted image one must therefore either to use a single sufficiently narrow band, make use of narrow band synthesis by channelising the observed bandwidth prior to imaging or use some form of analytical deconvolution taking into account of the variation in smearing function with position in the image plane. While observing with a single band narrow enough to avoid the problem entirely the sensitivity loss makes this approach somewhat unattractive, especially when designing new instruments. Narrow band synthesis involves dividing the measured band up into a number of sub-bands which are sufficiently narrow for bandwidth smearing to be negligible and imaged separately. While this approach is attractive the computational cost of imaging each band separately is expensive.

2.2.2 Time-average smearing

Similar to bandwidth smearing, described above, the effect of time-averaging is that the the source variability in a sampled visibility is not constant and as a result smearing in the

image plane occurs. As a result of time-integration of data points (performed for practical reasons) each visibility point assigned to a position (u, v) by the correlator corresponds to a locus of points from the time range $|\delta t| \leq \tau_a/2$ where τ_a is the averaging period. This time range corresponds to a rotation in (u, v) due to the earth's angular velocity ω_e though an angle $\pm\,\omega_e\tau_a/2$ and since the Fourier transform commutes with rotation, the resulting brightness would be rotated though the same angle. The effect of this distortion is therefore equivalent to an azimuthal convolution whose width increases with radius $\sqrt{l^2 + m^2}$ and is therefore a particular problem for large fields of view and long baselines. The simplest solution to this problem is to choose an averaging time in the correlator such that the effect is minimal although this puts significant computational demands with both increased correlator output bandwidth and larger number of visibility points to invert.

2.2.3 Radio frequency interference

Low frequency radio signals can be contaminated by considerable amounts of Radio Frequency Interference (RFI) making as much as 15% of the signal unusable. Whereas at higher frequencies RFI originates mainly from external sources, such as communication networks, at low frequencies the internal electronics of the detector can also contribute significantly. Figure 2.4 shows the RFI induced by the 100 kHz oscillators used for intermediate frequency (IF) conversion at the VLA. In this case, RFI is characterised by a comb of narrow interference spikes which are extremely persistent with time and if not removed lead to significant degradation of the image. While the amount and exact characteristics of the RFI will depend heavily on the instrument, the selected frequency band and the environmental conditions at the time of observation a number of common strategies can be used for its removal. Historically the most common method of RFI removal is a simple data excision procedure where data points whose amplitude either varies significantly in time or frequency from neighbouring points or exceed some pre-determined

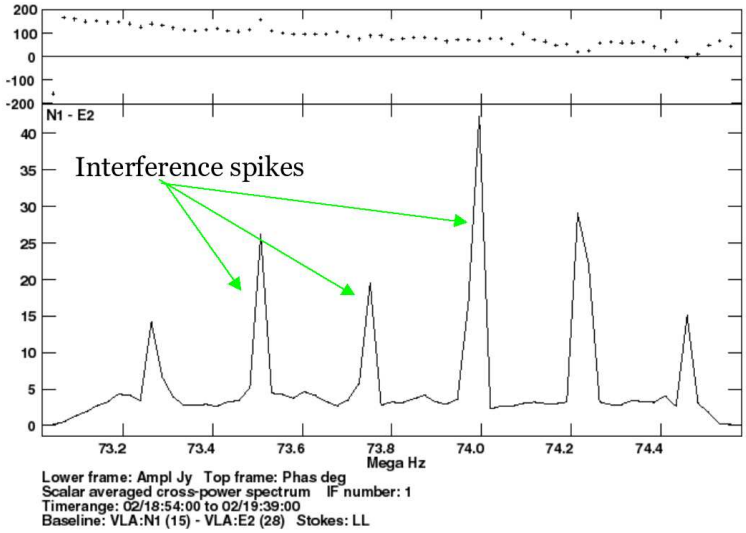


Figure 2.4: Low frequency RFI spikes at 74MHz from the VLA

level are flagged for removal in subsequent processing. Such a technique has a relatively low computation cost compared to imaging as it scales linearly with the number of data points. A number of more sophisticated RFI removal strategies have been suggested in the literature making use of various statistical methods to identify, model and remove interfering signals. These often however come at high computational cost and trade data loss against raising of the overall noise level of the data and therefore have to be carefully matched to the overall data processing strategy dictated by the science goal of the observation.

2.2.4 Anisoplanatism

In the assumptions used in calibration of visibilities it is commonly assumed that the complex calibration gains do not vary with position on the sky. This assumption is incorrect however for sufficiently large fields of view and especially at low frequencies where variations in the ionosphere, typically on the order of a degree, play a dominant role in leading to distortions across the brightness map. The solutions to these problem are still an area of much research and they require far more complicated calibration

algorithms than traditionally used. One solution is to split the image into small facets each of which have phase and gain solutions that can be considered constant. These facets are then calibrated and imaged separately prior to being mosaiced into the final brightness maps. A alternative and perhaps more elegant solution might be to model anisoplanatism as a direction dependent image plane phase and gain screen and solve for this during imaging and calibration however developing efficient algorithms for this approach requires a deep understanding of the instrumental response of the telescope and good knowledge of a global sky model and solving a potentially very large minimisation problem could be extremely computationally demanding.

2.2.5 Noncoplanar baselines

The requirement of a small field of view is fundamental to the approximation that the inversion of visibility measurements to a brightness map takes the form of a two-dimensional Fourier transform. This approximation requires that the w-term in the equation

$$V(u,v,w) = \int_{\infty}^{\infty}\int_{\infty}^{\infty} \frac{I(l,m)}{\sqrt{1-l^2-m^2}}\, e^{-2\pi i[ul+vm+w(\sqrt{1-l^2-m^2}-1)]}\, dldm\,, \tag{2.30}$$

may be ignored such that recovering the sky brightness $I(l,m)$ can be performed by simple inversion of visibilities $V(u,v)$. For this to be the case however, all measurements of visibility must either lie on a plane such that the w coordinate of the visibility can be expressed as $w = \alpha u + \beta v$, or that the field of view is limited to a small enough angular region such that $\sqrt{1-l^2-m^2} - 1 \approx 0$. The first condition is met for East-West aligned arrays or for any two-dimensional array for a snapshot observation.

When observing at low frequencies, modern interferometers cannot very often exploit these conditions as the field of view scales as λ/D (where λ is the central frequency and D the diameter of an antenna element) and for reasons of the desire to fill the (u,v) plane

as quickly as possible East-West arrays are not favored.

To illustrate the problem, table 2.1 lists the approximate FOV for some current instruments. As you can see all instruments fall far outside the criteria of a small angle approximation and with the next generation of low-frequency instruments this is unlikely to decrease with large numbers of small antenna-favoured over large dishes.

Instrument	frequency	FOV
GMRT	240 MHz	1.9°
EVLA	74 MHz	11.2°
LOFAR (low-band)	60 MHz	4.6°
LOFAR (high-band)	180 MHz	4.0°
ASKAP	700 MHz	2.4°
Meerkat	500 MHz	3.7°

Table 2.1: The FOV of several current generation low-frequency telescopes.

In addition to the effect of the w-term in equation 2.30, when the visibility sampling $S(u, v, w)$, is taken into account, the size structures sampled by a given pair of antennas vary across the field of view.

To estimate the consequence of neglecting the w term, consider the effect of a point source at (l, m) observed with a single interferometer. The phase error incurred is

$$error \approx \frac{\pi}{2} w \left(l^2 + m^2\right) = \pi w \theta^2 \,. \tag{2.31}$$

If w is a linear function of u and/or v, as in the case of a coplanar array, the linearly increasing phase error across the (u, v) plane will appear as a position error in the image. The apparent position shift is a function of zenith angle and azimuth of the source, therefore the source will appear to move during the observation. For noncoplanar arrays the effect is complex.

To better understand the relation between three-dimensional and two-dimensional space, consider shap-shot observations as illustrated in figure 2.5. Taking a single snapshot

of a source on the celestial sphere at position $A^{celestial}$ (figure 2.5, *top*) an instantaneous two-dimensional transform will project the source onto the local tangent at A^s instead of the true location at A^{true}. As the array geometry and local zenith moves around the celestial sphere (with earth rotation), the projection onto the tangent plane occurs at different angles (figure 2.5, *bottom*). Since the observation can be thought of as a sum of snap-shots, the resulting image on the tangent plane due to a long integration is the summed projection of the true brightness distribution on the unit sphere onto the tangent plane with a set of rotating array beams, whose ratio is fixed to locations of each object on the unit sphere. As a result, the image of any real object on the tangent plane will rotate in time, with the sum effect being a distorted image whose centroid is generally in the wrong position.

2.2.5.1 Some solutions

With the noncoplanar baseline effect causing such a fundamental problem to the computation of the sky brightness map for large fields a number of algorithms have been developed over the years.

- **Fourier Sum**. Equation 2.30 can be numerically integrated. While this method provides an accurate solution to the inversion, it is extremely computationally expensive.

- **Component models**. The brightness distribution can be described by a collection of discrete component models, the Fourier transform of which is known analytically. This can be used with good effect when combined with other less exact transformation methods.

- **Fourier Sum of bright pixels**. In a similar approach to component models a full Fourier sum is performed for the bright pixels with a more approximate transform

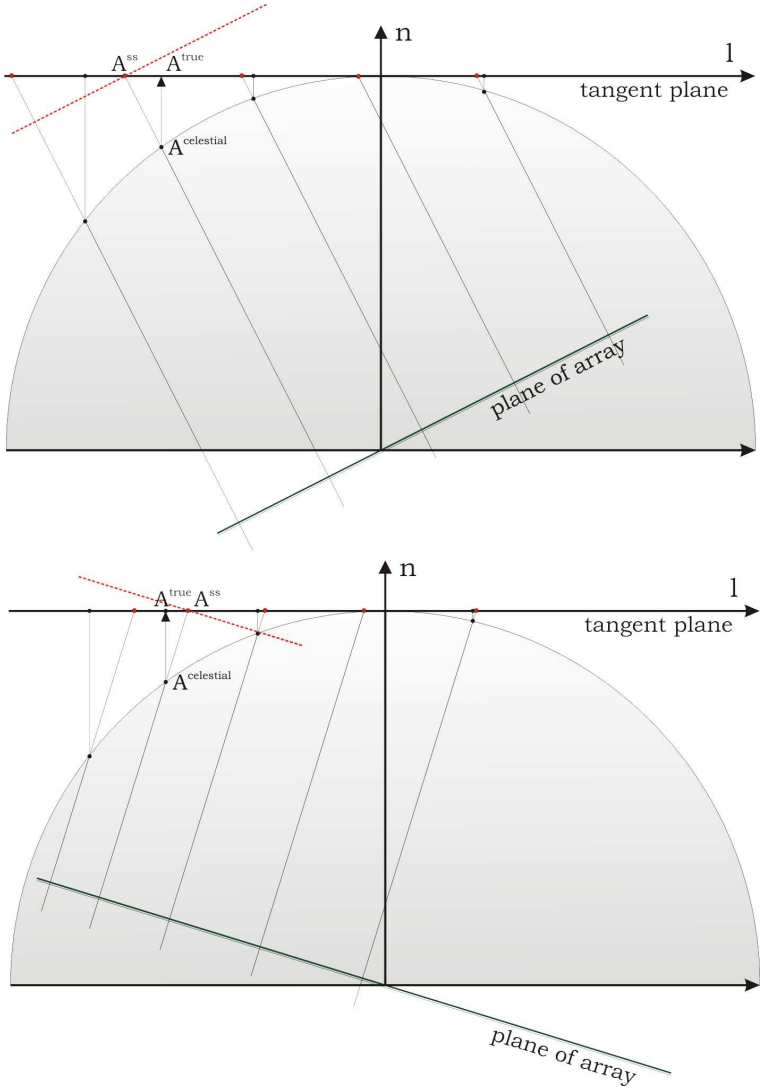


Figure 2.5: Representation of snap-shot images with the projection from the celestial sphere to the tangent plane. The *top* and *bottom* diagrams represent snap-shots at different times. At a later time the array geometry has changed due to earth rotation, so the projection is at a different angle. The apparent positions of the objects which are not located at the tangent point have changed with respect to an earlier observation.

for the rest of the image.

- **Warped Snapshots**. Since, as already noted, a two-dimensional array is instantaneously planar, a standard two-dimensional transform can be performed on each snapshot. Therefore a rotational re-gridding of image coordinates for each snapshot followed by image stacking can be used. While this again results in a highly accurate solution and allows the use of an efficient two-dimensional transform, the high preci-

sion image plane coordinate transform for each snapshot is highly computationally expensive.

- **Image-plane facets**. This approach involves refactoring the field of view into a number of facets, each of which are small enough for small angle approximations to apply and therefore imaged separately using a two-dimensional transform. This algorithm was developed by Cornwell & Perley (1992) and is the most common approach in the majority of current software. It is interesting to note that as the facet size approaches a single pixel this algorithm reverts to the Fourier sum approach.

- **Fourier-plane facets**. This approach is similar to image-plane facets but involves projecting the Fourier plane data onto a single tangent plane during imaging avoiding the need to deal with a large number of facet images.

- **3D transforms**. If we ignore the motivation to use a two-dimensional Fourier transform and instead solve for the brightness distribution embedded in a three-dimensional space with axes (l, m, n) this can be then solved with a three-dimensional fast Fourier transform. The main problem with this method is that for a large field of view the interior of the brightness cube is empty of any true emission and therefore the method is rather inefficient.

2.3 The w-projection algorithm

As identified in the previous section, several significant challenges exist when producing brightness maps from low-frequency, wide field data. Central to the problem is that of the non-coplanar baseline effect. If the computational cost of this could be reduced dramatically the improvements in the ability to image and survey at low frequencies would be gained as well as freeing up computational resources for correction of other effects such

as narrow band synthesis for the case of bandwidth smearing. To this aim and recognising that the main cause of the noncoplanar effect results form the w-term of equation 2.8, Cornwell et al. (2005) proposed the w-projection algorithm which I will introduce in this section.

The w-projection algorithm, as the name suggests, works by projecting the w part of (u, v, w) coordinates out of the problem, allowing a two-dimensional Fourier transform between visibilities and brightness distribution, thus avoiding having to deal with the problems of three-dimensional imaging. Projecting visibilities from a arbitrary w plane to $w = 0$, was introduced by Frater & Docherty (1980) and this concept was extended by Cornwell et al. (2005) to a solution which solves the re-projection of an entire set of three-dimensional visibilities to the two-dimensional set at $(u, v, w = 0)$.

To understand this, equation 2.30 is rewritten as convolution between the sky brightness and the Fourier transform of an image plane term parameterised by w.

$$V(u,v,w) = \int\int \frac{I(l,m)}{\sqrt{1-l^2-m^2}}\, G(l,m,w)\, e^{-2\pi i[ul+vm]}\, dldm\,, \tag{2.32}$$

where

$$G(l,m,w) = e^{-2\pi i[w(\sqrt{1-l^2-m^2}-1)]}\,. \tag{2.33}$$

Applying the Fourier convolution theorem this can be written as

$$V(u,v,w) = \widetilde{G}(u,v,w) * V(u,v,w=0)\,, \tag{2.34}$$

where $\widetilde{G}(u, v, w)$ is the Fourier transform of $G(u, v, w)$.

The form of $\widetilde{G}(u, v, w)$ can be understood by using a small angle approximation giving:

$$G(l,m,w) = e^{\pi i\left[w(l^2+m^2)\right]}, \tag{2.35}$$

$$\widetilde{G}(l,m,w) = \frac{i}{w} e^{-\pi i\left[\frac{(u^2+v^2)}{w}\right]}. \tag{2.36}$$

Therefore with a relatively simple set of functions, the three-dimensional visibility measurements from an interferometer can be related to a two-dimensional set.

To understand the effect further, consider figure 2.6 showing the radiation pattern propagating from the vertical direction onto a pair of antennas. If the time series of electric field measurements where to be correlated at the points A and B, then the correlation would be a two-dimensional Fourier transform. Since however, the correlation is actually between the two antennas at A and B′ this relation no longer holds. On propagating from B to B′ the electric field diffracts and the field on the plane at AB will therefore be a diffracted version of what is measured on the AB′ baseline. It is interesting to note that the effect of this diffraction is clearly decreases as the antenna diameter increases. This configuration can be used to analyse the physics of equation 2.32. If we suppose that point A is the origin of the (x, y, z) space, B is at $(x = x_a, y = y_a, z = 0)$ and B′ is at $(x = x_a, y = y_a, z = w\lambda)$. The correlated signal between the antenna AB′ is

$$\langle \Psi(0,0,0)\Psi^*(x_a, y_a, w\lambda)\rangle \ , \tag{2.37}$$

and to evaluate this we must evaluate the electric field on the two planes. Since the region around the antennas is source free, the electric field on the plane AB may be propagated to the plane A′B′ using diffraction theory. If this distance is such that the antennas are in each others' near-field we have to use Fresnel diffraction theory.

$$\Psi(x,y,z=w\lambda) = \frac{e^{-w\pi i w}}{i\lambda^2 w} \int \Psi(x',y',z'=0)\, e^{\frac{\pi i}{\lambda^2 w}((x-x')^2+(y-y')^2)}\, dx'dy' \ . \tag{2.38}$$

Note that this equation, re-expressed in terms of (u, v) is equivalent to equation 2.32 and therefore the physical cause of the convolution relation can be attributed to Fresnel

diffraction of the electric field propagating from one antenna to another. The size of the diffraction pattern in wavelengths is therefore given by the Fresnel zone diameter, $r_F \approx \lambda\sqrt{w}$. It is clear therefore that a interferometer only measures a single component of the

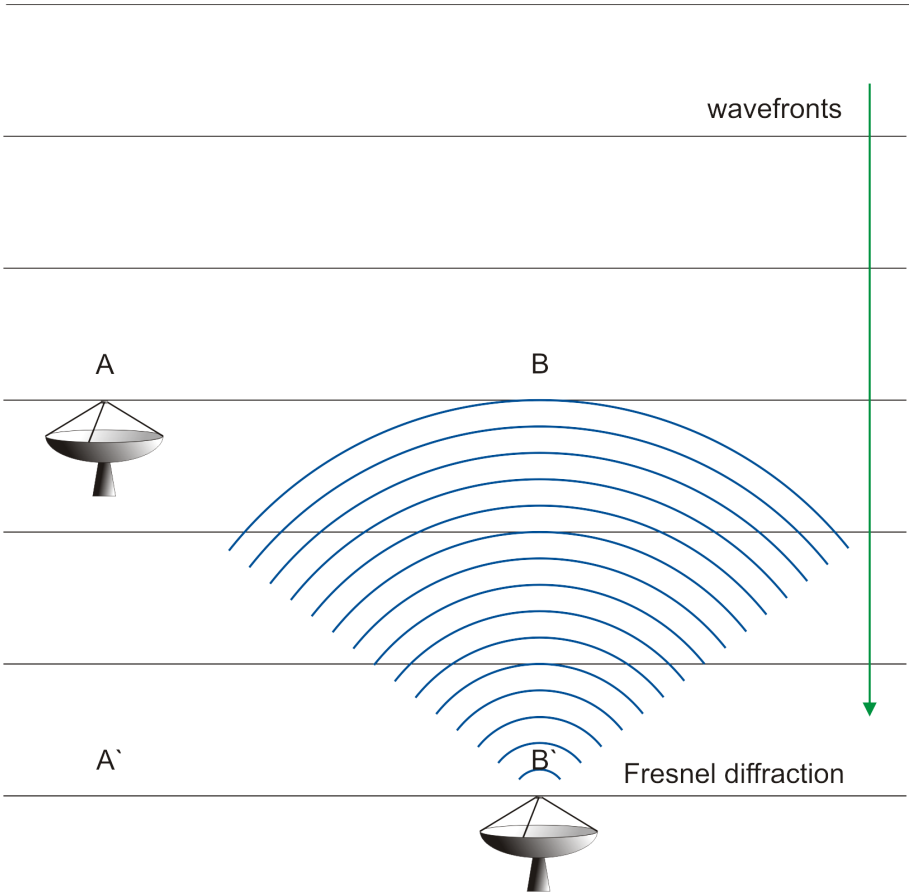


Figure 2.6: Two antennas sampling an electric field propagating in the vertical direction. The radiation received at B′ follows Fresnel diffraction in propagating from the AB plane.

sky brightness if w=0. In principle, it is therefore possible to measure a set of components at a fixed (u, v) but different w and recover information on the Fourier components within r_F/λ of the (u, v) sampling point.

The practical implementation of the w-projection algorithm therefore proceeds by projecting each (u, v, w) point onto the $(u, v, w = 0)$ plane using the w-projection function followed by a two-dimensional transform of these gridded visibilities. Depending on the level of complexity in the synthesised beam, the dirty map is then a reasonable estimate of the sky brightness distribution.

It should be noted that w-projection can be extended to correct for other image plane phase or gain screens during gridding. This is the strategy behind the recent A-projection algorithm which makes use of the Fourier plane response of the complex cross power beam pattern to correct for primary beam effects. While this requires a priori accurate modeling of individual element patterns it is a highly computationally efficient algorithm for dealing with the direction dependent image plane effects introduced by primary beams. In general it should be possible to correct for any well defined image plane gain and phase screen by combining its Fourier plane response with the w-projection gridding kernel. In practice however this requires an accurate a priori model of the effect being corrected which limits the number of effects that can likely be included. Corrections for atmospheric effects for example are unlikely to be easily incorporated due to the difficulty to maintain a highly accurate real time atmospheric model.

The computation involved in w-projection is gridding by means of a set of w-dependent convolution kernels. This will therefore scale with both the number of visibility data points and the width of the convolution functions used in gridding. To compare the computational performance of w-projection and faceting Cornwell et al. (2003) present the following scaling relations which give the computational cost per visibility sample:

$$t_{w-\text{project}} = 2\,(N^2_{w-\text{project}} + N^2_{\text{GCF}}) \cdot t_{\text{single}}\,. \tag{2.39}$$

$$t_{\text{facets}} = N^2_{\text{facets}} \cdot N^2_{\text{GCF}} \cdot t_{\text{single}}\,. \tag{2.40}$$

N_{GCF} is the width or support size of the normal gridding convolution function, t_{single} is the time taken to grid a single sample, N_{facets} is the number of facets on one axis and $N_{w-\text{project}}$ is the typical size of the w dependent convolution function. As both N_{facets} and $N_{w-\text{project}}$ are proportional to $B\lambda/D^2$, and for large fields-of-view will be approximately equal, the ratio of the computational time is very roughly equal to N^2_{GCF}. For a typical

gridding function this is between 25 and 49 and therefore one might expect a similar performance gain in using w-projection approach to wide-field imaging.

This is therefore clearly a promising solution to wide-field synthesis imaging and in Chapter 6, I present a study into its implementation on graphics processing units (GPU).

Chapter 3

A 1-D case study: periodicities in SS433

This chapter focuses on an investigation into a seemingly periodic signal observed in the microquasar SS 433, from spectroscopic data collected over 26 years of observations from 1978 to 2004. While this is the largest available data set for SS 433, the data series suffers from extremely irregular time sampling as a result of being collected over many years by different observers limited by the conditions of favourable telescope time, yet periodic signals do emerge. As a result of this highly uneven sampling, to properly investigate and characterise these periodicities conventional spectral analysis techniques are largely unsuitable and alternative solutions were required.

For this study I have investigated a number of spectral analysis techniques, eventually adopting a version of the CLEAN deconvolution algorithm for one-dimensional data first proposed by Roberts et al. (1987), a simple folding technique and the Lomb-Scargle periodogram method (Lomb 1976).

3.1 Description of SS 433

SS 433 is a microquasar, located in our Galaxy, 5.5 kpc from Earth (Blundell & Bowler 2004). Approximately every day, it ejects pairs of bollides of hydrogen, in opposite directions, at speeds about a quarter of the speed of light. Over successive days, the direction of the axis along which this jet plasma is launched precesses, tracing out a cone with semi-angle 20 degrees with a seemingly steady periodicity of about 162 days (e.g. Eikenberry et al. 2001). Superimposed on this precession is a nutation, or nodding, of this axis which gives rise to further periodicities of 5.8 and 6.3 days. These three periodicities are very clearly measured by spectroscopy (e.g. Margon 1984, Blundell & Bowler 2005). At the heart of SS433 there are two stars in orbit around one another: one a "normal star" whose gas is sucked away by its companion, which is a black hole of 16 solar masses (Blundell et al. 2008). The periodicity with which these two stars orbit one another is 13.08 days and this has been widely measured by timing eclipses via photometry by a number of authors (e.g. Kemp et al. 1986). Besides these two famous periodicities, others have been rumoured (e.g. Collins & Scher 2002) though not confirmed.

The behaviour of SS 433 in ejecting oppositely directed jets is very analogous to the behaviour of powerful radio quasars observable in the distant universe. Because SS 433 is nearby and bright, detailed investigation of its nucleus could be very helpful in understanding the phenomenon of jet launch and collimation.

One clue about the richness of the SS 433's behaviour came when Blundell & Bowler (2005) demonstrated that the speed at which the jet material is launched shows oscillations having a periodicity exactly the same as the orbital period – 13.08 days – as well as variations that were purely aperiodic, i.e. stochastic.

In order to establish whether there is any evidence for a change in this jet speed periodicity of 13.08 days, so remarkably similar to the orbital period, the data sets described

in the following section have been investigated and split up in time to investigate the persistence and stability of this periodicity in the jet launch speed.

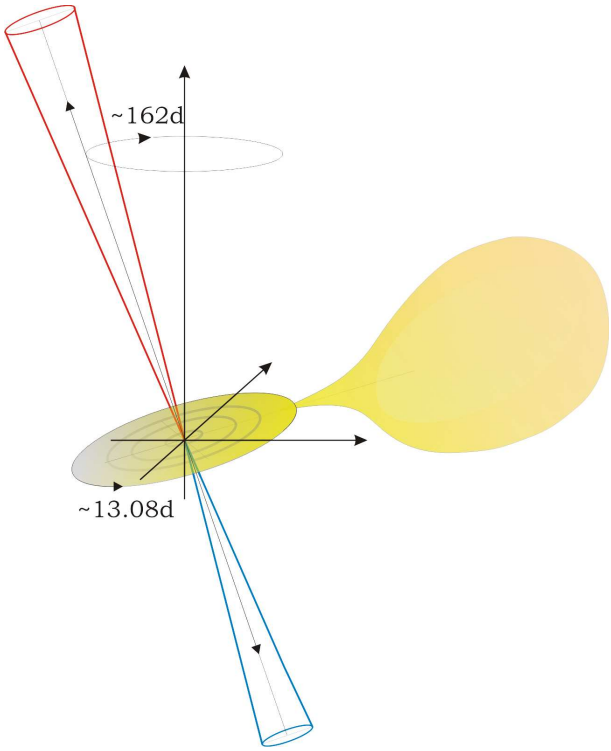


Figure 3.1: Schematic representation of SS 433, showing the geometry of the precessing jets.

3.2 Description of datasets

The source of data for this investigation came from archival compilations of spectroscopic data by two workers: Steve Eikenberry and George Collins III. These datasets represent all redshifts that were published from various observers using different spectrographs around the world that Eikenberry and Collins could obtain. See discussion in Blundell & Bowler (2005). These spectroscopic datasets, which aren't independent from one another, record the blueshift of the jet pointing towards Earth and the redshift of the jet pointing away from Earth measured instantaneously from the displacements of the Balmer H-α emission line, that is observed to move periodically (162 days) in wavelength as the orientation of

the jet axis precesses.

The redshifts themselves encode the rest wavelength of the observed emission line (Balmer H-α in this case, whose rest wavelength is 6563 Å) and, because of the Doppler effect, the line-of-sight velocity with which the radiating plasma is moving, which depends on the launch speed and direction of motion with respect to Earth. The redshifts are described in terms of this motion as follows:

If the precessing jet axis of SS 433 traces out a cone of semi-angle θ about a line which is oriented at an angle i to our line-of-sight with jet velocity β in units of c ($\gamma = (1-\beta^2)^{-1/2}$), the redshifts measured from the west jet (z_+) and the east jet (z_-) are given, if the jets are symmetric, by:

$$z_\pm = -1 + \gamma[1 \pm \beta \sin\theta \sin i \cos\phi \pm \beta \cos\theta \cos i] \,, \tag{3.1}$$

where ϕ is the phase of the precession cycle. Addition of z_+ and z_- in equation 3.1 gives an expression relating the observed (simultaneously measured) redshifts to the jet speed independently of any angular variation. Re-arrangement gives

$$\beta = \left[1 - \left[1 + \frac{z_+ + z_-}{2}\right]^{-2}\right]^{1/2} . \tag{3.2}$$

Subtraction of the expressions for z_+ and z_- gives the angular properties a of the orientation of the jet axis, with the speed divided out using equation 3.2:

$$a = \frac{z_+ - z_-}{2\beta\gamma} = \sin\theta \sin i \cos\phi + \cos\theta \cos i \,, \tag{3.3}$$

Thus, the two observables z_+ and z_- have been converted into two more interesting observables, the speed and the angle and it is the speed data β derived in equation 3.2 which is used for the analysis in this chapter.

3.3 Spectral analysis of unevenly sampled data

This section introduces some of the main issues of spectral analysis of irregularly sampled data sets and presents a couple of solutions which are used in the subsequent investigation of SS 433 speed data.

Spectral analysis of a time-varying signal typically involves Fourier inversion of a finite set of N discrete time samples t_r within a time range T resulting in a set of frequency components from which a frequency spectrum is constructed. In the general case, for discretely sampled data, the frequency spectrum will be distorted by the limited frequency resolution which results from the finite span of the data and also spurious spectral responses which are an artifact of incompleteness of the data sampling. It is the second effect which is particularly relevant to this investigation and which in part is solved by the implementation of the so-called CLEAN algorithm described later.

3.3.1 Simple Fourier Analysis

In order to better understand the problem of uneven time sampling, I first look at the theory behind simple spectral analysis.

If a function $f(t)$ known for all times t is both continuous and square-integrable, it can be reconstructed from a frequency spectrum $F(\nu)$, described by following equation:

$$f(t) = \int_{-\infty}^{\infty} F(\nu) e^{2\pi i \nu t} d\nu \,. \tag{3.4}$$

The spectrum itself is given by the Fourier transform relation

$$F(\nu) = FT[f(t)] = \int_{-\infty}^{\infty} f(t) e^{-2\pi i \nu t} dt \,. \tag{3.5}$$

In practice, data points are often sampled discretely at a set of N arbitrary times t_r, and

the data set can therefore be represented as

$$f_r = f(t_r), \quad r = 1, ..., N \, . \tag{3.6}$$

It is important to note that this discrete sampling of the data can be represented by sampling the continuous signal with a sampling or 'window' function $s(t)$, defined as

$$s(t) = \frac{1}{N} \sum_{r=1}^{N} \delta(t - t_r) \, , \tag{3.7}$$

(where $\delta(t)$ is the Dirac delta function).

With this in mind, the sampled signal can be written as $f_s(t) = f(t)s(t)$ and the Fourier transform of the sampled signal is therefore

$$F_s(\nu) = FT[f_s(t)s(t)] \, . \tag{3.8}$$

Making use of the Fourier convolution theorem this becomes

$$F_s(\nu) = F(\nu) \otimes S(\nu) = \int_{-\infty}^{\infty} F(\nu')S(\nu - \nu')d\nu' \, , \tag{3.9}$$

where $S(\nu)$ is the spectral window function, the Fourier transform of the sampling function

$$S(\nu) = \int_{-\infty}^{\infty} s(t)e^{-2\pi i\nu t}dt = \frac{1}{N} \sum_{r=1}^{N} e^{-2\pi i\nu t_r} \tag{3.10}$$

and $F_s(\nu)$ is known as the dirty spectrum. Note that the dirty spectrum as defined here is exactly analogous to the dirty image (or dirty map) well known in radio imaging and described in section 2.1. The dirty spectrum represents the true spectral information

contaminated by a convolution of the spectral window function $S(\nu)$,

$$F_s(\nu) = \int_{-\infty}^{\infty} f_s(t) e^{-2\pi i \nu t} dt = \frac{1}{N} \sum_{r=1}^{N} f_r e^{-2\pi i \nu t_r} \,. \tag{3.11}$$

The direct consequence of this is therefore that the sampling function determines the contamination of the spectral information and it is this that we need to look at in more detail if we are to recover the true spectral information in the time series, the ultimate aim of any form of spectral analysis.

3.3.1.1 The effect of discrete sampling

There are two distinct categories of sampling functions, $s(t)$, common to all discretely sampled time series and these are worth further consideration; the case of evenly spaced sampling and the effect of the finite sampling span. Firstly in the case of evenly spaced sampling with constant time interval Δt between samples, the sampling function can be represented as

$$s(t) = \sum_{r=-\infty}^{\infty} \delta(t - r\Delta t) \quad r = -\infty, \ldots, \infty \tag{3.12}$$

and in this case the spectral window function becomes

$$S(\nu) = \frac{1}{\Delta t} \sum_{r=-\infty}^{\infty} \delta(\nu - \frac{r}{\Delta t}) \,. \tag{3.13}$$

It can be seen therefore that periodic sampling in the time domain with interval Δt causes multiple peaks in the frequency domain with interval $1/\Delta t$. If we introduce the Nyquist frequency, $\nu_N = 1/2\Delta t$, the 'dirty spectrum' can be written as

$$F_s(\nu) = \frac{1}{\Delta t} \left[F(\nu) + \sum_{r=1}^{\infty} [F(\nu - 2r\nu_N) + F(\nu + 2r\nu_N)] \right] . \tag{3.14}$$

If the function $F(\nu)$ is zero for $|\nu| \geqslant \nu_N$, $f_s(t)$ is fully recoverable from $F_s(\nu)$. This is the well known Nyquist-Shannon sampling theorem, fundamental to information theory. If the condition is satisfied, the spectrum up to ν_N is not contaminated.

Secondly, the sampling function representing a finite data span is a box-car function which can be described mathematically as

$$s(t) = \begin{cases} 1 & \text{for } t_0 \leqslant t \leqslant t_N \\ 0 & \text{otherwise .} \end{cases} \tag{3.15}$$

(having finite data length $T = t_N - t_0$), the spectral window function in this case is of course a sinc function

$$S(\nu) = \frac{\sin(\pi\nu T)}{\pi\nu}\, e^{-\pi i\nu(t_1+t_N)} . \tag{3.16}$$

This effect causes smearing in the spectrum (known as spectral leakage). It is the width of this smearing which gives the frequency resolution, and is controlled by a function of the duration T ($\delta\nu \approx 1/T$) of the data set.

It can be seen therefore, even in these two very simple examples of discrete sampling that complications to the spectrum occur which may ultimately lead to difficulties in interpreting the raw dirty spectra. For the case of the time series available in this study the spectra are concealed further and to a far larger degree through the response of heavily uneven data sampling which gives an extremely complex spectral window function. This effect can be better understood looking at equation 3.11 for the dirty spectrum and noting that that if data points are missing from f_r, the resulting $F_s(\nu)$ is equivalent to the dirty spectrum of an interferometric data set, in which all the missing data are identically zero. This implicit assumption is not clearly not desirable and will easily lead to misinterpretation of spectra if care isn't taken in the analysis.

Fortunately, we can to a large extent remove the effect of undesirable sampling by

eliminating the known spectral window function from the dirty spectrum. A straightforward deconvolution in the frequency domain, however, is not possible due to the (mostly zero) nature of the sampling function. This problem can be circumvented by estimating the (complex) amplitude of a cosinusoidal component of the spectrum, removing its influence on the dirty spectrum including its side-lobes and iterating over this procedure.

This approach is an algorithm called CLEAN and has the advantage over analysis of the time series by simple one-dimensional Fourier transform in that it performs a non-linear deconvolution of the Fourier transformed spectra to remove the effect of side-lobes resulting from uneven time sampling.

3.3.2 The CLEAN algorithm

To remove the effect of uneven sampling Roberts et al. (1987) proposed an algorithm adapted from the two-dimensional CLEAN aperture synthesis technique used commonly in radio astronomy for image reconstruction (Högbom (1974)) from a sparsely sampled aperture. The CLEAN algorithm performs a nonlinear deconvolution in the frequency domain and in doing so provides a simple way to remove artifacts in a spectrum which result from non-uniform (or incomplete sampling).

For a single harmonic component with real amplitude A, frequency $\hat{\nu}$ and phase ϕ represented by

$$f(t) = A\,\cos(2\pi\hat{\nu}t + \Phi)\,, \tag{3.17}$$

the transformation into the frequency domain gives the spectrum

$$F(\nu) = a\,\delta(\nu - \hat{\nu}) + \delta^*(\nu - \hat{\nu})\,, \tag{3.18}$$

where

$$a = \frac{A}{2}\,e^{+i\Phi}\,. \tag{3.19}$$

If $f(t)$ is however sampled at N discrete times

$$f_s(t) = f(t)s(t) = \frac{1}{N}\sum_{r=1}^{N} f(t)\delta(t - t_r)\,, \tag{3.20}$$

and from using convolution theorem

$$F_s(\nu) = F(\nu) \otimes S(\nu)\,. \tag{3.21}$$

For the discrete time series, the dirty spectrum equation 3.11 becomes

$$F_s(\nu) = aS(\nu - \hat{\nu}) + a^* S(\nu + \hat{\nu})\,, \tag{3.22}$$

and if $S(0) = 1$, there is a peak in the frequency $\hat{\nu}$ given by

$$F(\hat{\nu}) = a + a^* S(2\hat{\nu})\,. \tag{3.23}$$

Writing F^* similarly and inserting into a^* it is possible to determine the amplitude of a peak if the frequency $\hat{\nu}$ is known as

$$a(\hat{\nu}) = \frac{F_s(\hat{\nu}) - F_s^*(\hat{\nu})S(2\hat{\nu})}{1 - \|S(2\hat{\nu})\|}\,. \tag{3.24}$$

The CLEAN algorithm uses equation 3.24 in order to find the complex amplitude of a cosinusoidal and remove its contribution to the spectrum F_s including all sidelobes using equation 3.22.

The iterative process by which this is achieved proceeds by selecting the largest peak in the dirty spectrum and removing its contribution (the response expected of a point source at this location), leaving a residual spectrum from which both the cosinusoidal component and spurious features due to sampling have been removed. After a number of

iterations we are left with a set of cosinusoidal components and a residual map of nothing but noise. The set of components found are the CLEAN components that are a model of the true spectrum $F(\nu)$.

As the CLEAN components are delta functions, to preserve the spectral resolution of the data and obtain a final spectrum without spurious super-resolved frequencies, the set of clean components is convolved with a Gaussian whose width is equal to that of the main response in the spectrum of the sampling function.

The iteration scheme given by Roberts et al. (1987) and described in figure 3.2 is:

1. Compute the dirty spectrum $F_s(\nu)$.

2. Start the iteration with the initial residual spectrum $R^0 \equiv F_s$.

3. On the i'th iteration find the maximum frequency $\nu_{\rm peak}$ in the previous residual spectrum R^{i-1} and calculate its complex amplitude $a(\nu_{\rm peak})$ using equation 3.24.

4. Use equation 3.22 to calculate the contribution of $a(\nu_{\rm peak})$ to the dirty spectrum and form the residual spectrum R^i by subtracting a fraction g $(0 < g < 1)$ of the result from R^{i-1}:

$$R^i = R^{i-1} - g[(a^i S(\nu - \nu_{\rm peak}) + (a^i)^\dagger S(\nu + \nu_{\rm peak}))] \,. \tag{3.25}$$

   Store the subtracted fraction ga^i to a clean component array at locations $\nu_{\rm peak}$ and $-\nu_{\rm peak}$. Continue the iteration until convergence criteria are reached.

5. After the iteration, convolve the clean component array with a Gaussian function to obtain a reasonable frequency resolution. Finally add the residual spectrum of the last iteration.

Roberts et al. (1987) suggested a number of stopping conditions. One straightforward method is the pre-definition of a threshold value for R^i so that the iteration stops, if R^i falls below that value, with anything of lower amplitude attributed to noise. In practice definition of the noise level by visual inspection is critical but can strongly depend on the spectral window function. Another termination method might be to base the cutoff criterion on the misfit to the data which can be computed only in the time domain. In most cases however it is convenient to simply iterate until a maximum number of iteration steps is reached. This number should be chosen to be large enough to ensure that the iteration does not stop before all the influence of signal is removed and furthermore it is interesting to note that as shown by Baisch & Bokelmann (1999) the result of CLEAN in one-dimension is highly insensitive to over-iteration. The number of iterations required will be heavily dependent on the CLEAN loop gain and number of components in the spectrum, however for a typical spectrum from the SS 433 data set and a CLEAN gain of 0.1, around 200 iterations were typical.

3.3.2.1 Significance level calculation for CLEAN spectra

One of the shortcomings of the CLEAN algorithm (over other techniques such as the Lomb-Scargle periodogram, Lomb (1976)) is the lack of a simple method with which to determine the significance of the frequency peaks in a given spectrum. Heslop & Dekkers (2002) however addressed this issue with a discussion of a Monte Carlo method of data stripping whereby a handle on the statistical significance of peaks in the CLEAN spectra can be assessed.

With the CLEAN algorithm excelling in the analysis of unevenly sampled data sets, a data stripping procedure is naturally suited to a Monte Carlo technique for assessing the statistical calculation of significance level. By stripping a sample of N_r random points from the data set of length N_{data} this generates $N_{\text{data}}!/[N_r!(N_{\text{data}} - N_r)!]$ different data

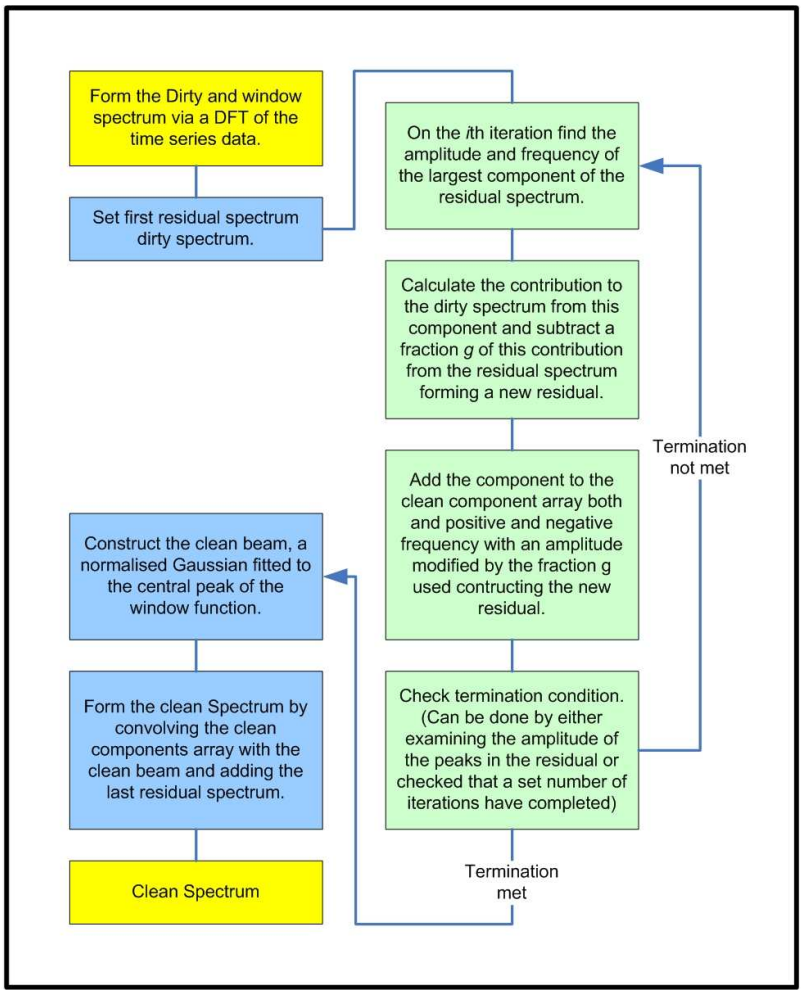


Figure 3.2: Schematic illustration of the CLEAN algorithm.

combinations, each of which can be processed with the CLEAN algorithm and analysed for comparison to see which periods appear to be robust.

This procedure will introduce artificial gaps into the data set, the size of which is controlled by the relative magnitudes of N_r and N_{data}, with the mean lengths of the artificial gaps approximating to N_{data}/N_r. The data stripping process is repeated for a number of loops I, with a CLEAN frequency spectrum being produced from the newly selected N_r data points each cycle. This results in I different spectra for which confidence intervals and significance limits can be calculated.

The 95% confidence limits for the frequency value allows a confidence interval to be

displayed about the mean of the frequency spectrum (using $[\overline{x}-1.96\sigma/\sqrt{I}, \overline{x}+1.96\sigma/\sqrt{I}]$). Sorting the power levels across all frequencies from all spectra shows a significance limit, α, to be determined for the mean spectrum. For example, if the Monte Carlo CLEAN procedure outputs 1000 spectra composed of 250 points each (giving a total of 250,000 modulus values) determining the point below which 95% of the sorted modulus array occurred would provide a 95% confidence value.

While this technique presents an interesting method for assessing the significance of CLEAN spectral components, the technique itself is extremely challenging for large datasets and errors in the spectra are dominated by effects other than the effect of the sampling.

3.3.3 Time Series Folding

The folding-averaging algorithm is an old method, widely used for determining periodic components of time series even before the computer age. The basic principle is to rebuild a new short time series by cutting the original time series into shorter sections of the length equal to a set period and then stacking and averaging them in these sections. In this stacking and averaging process, the amplitude of any signal with period equal to the specified fold period remains the same, but signals of different periods are averaged out and the random error is reduced. The amplitude and phase of the signal with a period equal to the fold period can then be estimated by fitting to a sinusoid of the folding period. By searching for the maximum of the amplitude through varying the fold period, the periods of signals which may be present in the time series can be determined, although this is potentially a very time consuming process if the period is not known a priori.

Folding, due to its simplicity, provides an excellent tool for examining and complementing the analysis performed using the CLEAN algorithm and in this investigation I use a combination of the two methods for my analysis.

3.3.4 Lomb-Scargle Method

The Lomb periodogram is a method of spectral analysis of uneven time series that, like CLEAN and folding, does not require evenly sampled data points or any form of interpolation.

This method evaluates the data, and spectral components only at times t_i that were actually measured.

For a time series $t_r = f(t_r),\ r = 1, 2, \ldots, N-1$ the normalised periodogram (spectral power density as a function of angular frequency, $\omega \equiv 2\pi\,\nu$) is defined as:

$$P(\omega) = \frac{1}{2\sigma^2}\left\{\frac{\left[\sum_r (t_r - \langle t\rangle)\ \cos\ \omega(t_r - \tau)\right]^2}{\sum_r \cos^2\ \omega(t_r - \tau)} + \frac{\left[\sum_r (t_r - \langle t\rangle)\ \sin\ \omega(t_r - \tau)\right]^2}{\sum_r \sin^2\ \omega(t_r - \tau)}\right\} \tag{3.26}$$

where τ is defined by the relation

$$\tan(2\omega\tau) = \frac{\sum_r \sin\ 2\omega t_r}{\sum_r \cos\ 2\omega t_r} \tag{3.27}$$

and $\langle t\rangle$ and σ^2 are the mean and variance of the time series given by the usual formulas

$$\langle t\rangle \equiv \frac{1}{N}\sum_{r=0}^{N-1} t_r \qquad \sigma^2 \equiv \frac{1}{N-1}\sum_{r=0}^{N-1}(t_r - \langle t\rangle)^2\ . \tag{3.28}$$

With the normalised power estimate $P(\omega)$ defined in this manner if the time series $f(t_r)$ is purely noise, then the power in $P(\omega)$ follows an exponential probability distribution. This exponential distribution can then be used as a convenient estimate of the probability that a given peak is a true signal or whether it is the result of randomly distributed noise.

3.4 Spectral Analysis Code

For this investigation I have developed a spectral analysis code which implements the CLEAN algorithm and allows investigation of the various parameters which influence the

result. I have also coded a simple version of the Lomb method for unevenly sampled data and a simple folding code. This section briefly describes the code developed.

3.4.1 The *SpectClean* code: Spectral analysis with CLEAN

My *SpectClean* code is implemented in C++ using the Qt[1] widget library for a basic graphical user interface. The code follows a modular class design with objects to handle data, spectra, the CLEAN algorithm and plotting. I designed the code to provide a flexible, portable and efficient version of the CLEAN algorithm which is fully adaptable to any future development. Where possible numerically intensive parts of the code have been parallelised using openMP[2], this being particularly important when evaluating large spectra by means of a direct Fourier sum. I chose to develop a simple a GUI for *SpectClean*, shown in Figure 3.3, in order to be able to investigate the various parameters of the CLEAN algorithm and to permit me to drive the CLEAN algorithm to its full potential on this challenging data set.

3.4.2 Folding code

In order to provide comparison to the spectral analysis by Fourier transform and CLEAN, simple folding was also performed. Folding involves wrapping the time series about a chosen frequency and averaging the wrapped data into a number of bins before fitting to a sinusoid of the folding frequency. This is a very simplistic method of frequency analysis but due to its extreme simplicity provides an excellent tool for examining further the results of analysis with CLEAN.

In order to be able to test this folding approach, I adapted a folding script written by Professor Katherine Blundell and wrote a simple script in Matlab[1] to fit a four point

[1]http://qt.nokia.com/
[2]http://www.openmp.org
[1]http://www.mathworks.com/

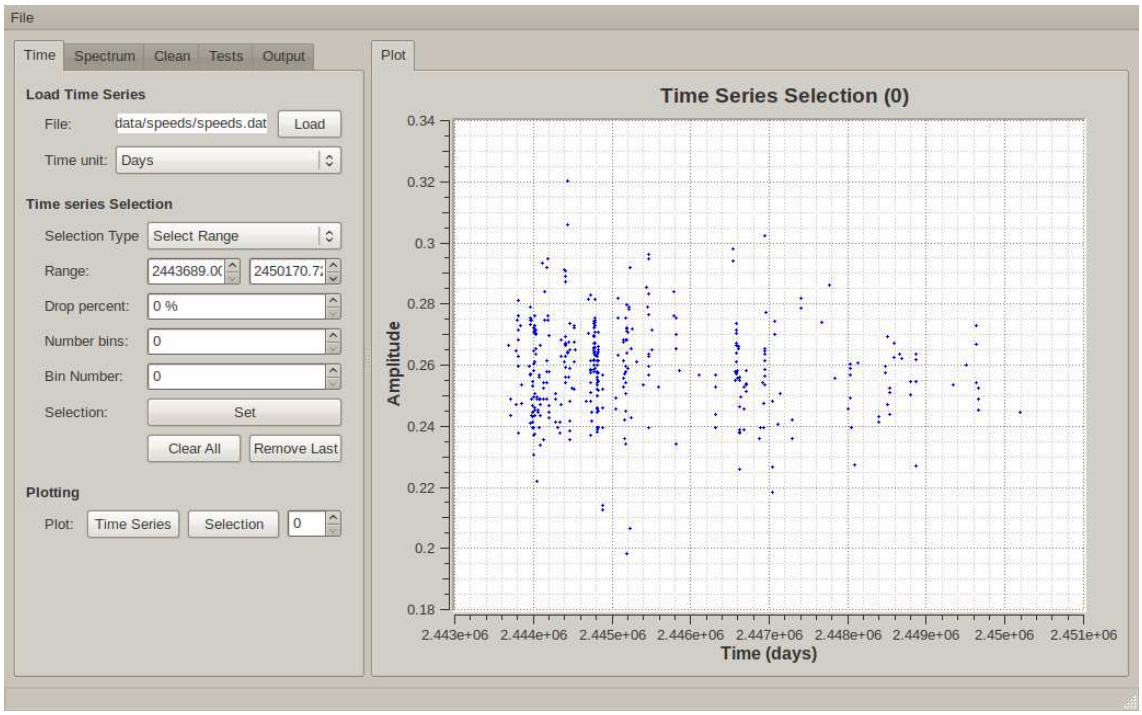


Figure 3.3: Screenshot of *SpectClean*, my spectral analysis code developed for this study.

sinusoid to the results.

3.4.3 Lomb periodogram code

In order to be able to test the Lomb-Scargle method of spectral analysis, described in Section 3.3.4, I coded a basic C console application in order to generate Lomb periodogram. This code was used to generate the Lomb periodogram spectra results presented in this chapter.

3.4.4 Binning the archive data

To establish if the speed data, described in section 3.2, showed any evidence of a change in the signal associated with the orbital period over the span of the archival dataset I wrote a script to divide this time series into sub-bins. In order to select sub-bins as free from bias as possible I choose to generate sub-bins matching 3 different criteria shown in figure 3.4 and described below.

A. Dividing up the time series evenly by a specified length of time.

B. Stepping though the time series and generating chunks of a specified length of time.

C. Selecting areas of the time series which are well sampled by performing a simple analysis of the spacings between time samples. This algorithm works by finding initial guess bins by looking at the average spacing between adjacent time points and picking bin edges where adjacent points are separated further than a specified minimum average spacing.

As this is a sparsely sampled historical time series of data, I have chosen to bin the data by each method with various parameters generating a range of bin lengths over the data series. This is aimed at generating bins with the best chance of reducing the possibility that features in the CLEAN spectra for a given time segment are solely due to the recording method, as each of these binning techniques should pick out data bins with slightly different criteria.

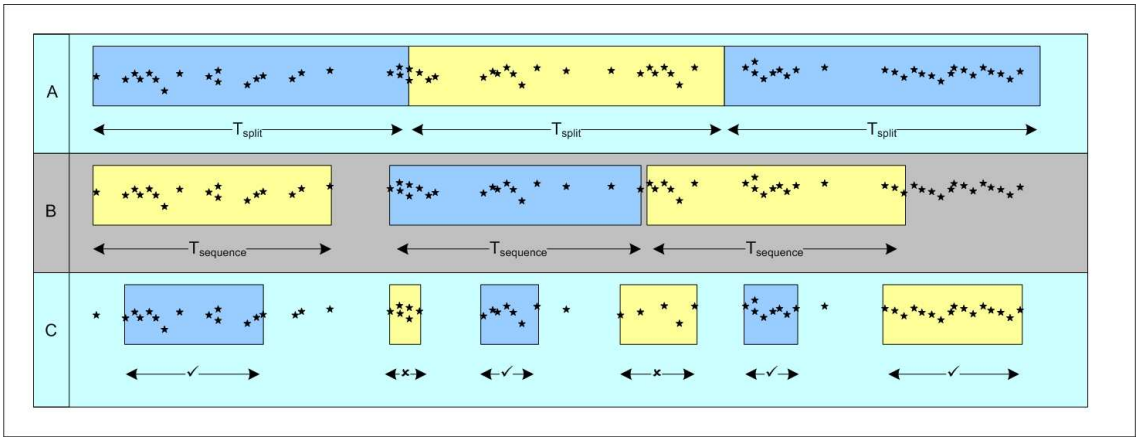


Figure 3.4: Binning methods. A, split binning of the time series into equal time segments. B, splitting the data into equal length series by stepping through the data sequentially. C, binning based on the criteria of spacings of data points, and the bin time length and number of points in a bin (is is carried out by first finding all the possible bins based on clumping then checking them based on data length and number of points, the tick and cross symbols shown on diagram C illustrate this final checking).

3.5 Code Validation with simulated data

In order to validate and test the code written for this analysis I generated a number of simple synthetic time series which were designed to test the behaviour of both CLEAN and time series folding methods under different sampling regimes. The time series were generated according to

$$f(t_r) = A_0 + \sum_i A_i \cos(2\pi\nu_i t_r + \phi_i) \, . \tag{3.29}$$

with the four spectral components shown in the table below.

i	A_i	ν_i	$\phi_i(/\pi)$
1	1.0	0.0	0.0
2	1.0	23.7	0.5
3	0.7	42.5	-0.25
4	0.4	53.0	0.5

Table 3.1: Spectral components of the synthetic time series data set.

The following sections present a number of synthetic time series and the analysis of these with both the CLEAN and folding codes. In all cases the analysis is able to effectively recover the parameters of the original time series.

3.5.1 Evenly Sampled data

Firstly, I generated an evenly sampled time series by evaluating the spectral components listed in Table 3.1 with 201 points in the range 0.0 to 1.0 and shown in Figure 3.5.

3.5.1.1 CLEAN spectra

Analysis with the CLEAN code gave the spectra shown in Figure 3.6. While this is a simple test it shows that CLEAN does an extremely good job of removing the side-lobes from the components in the dirty spectrum which result from the finite length of the time

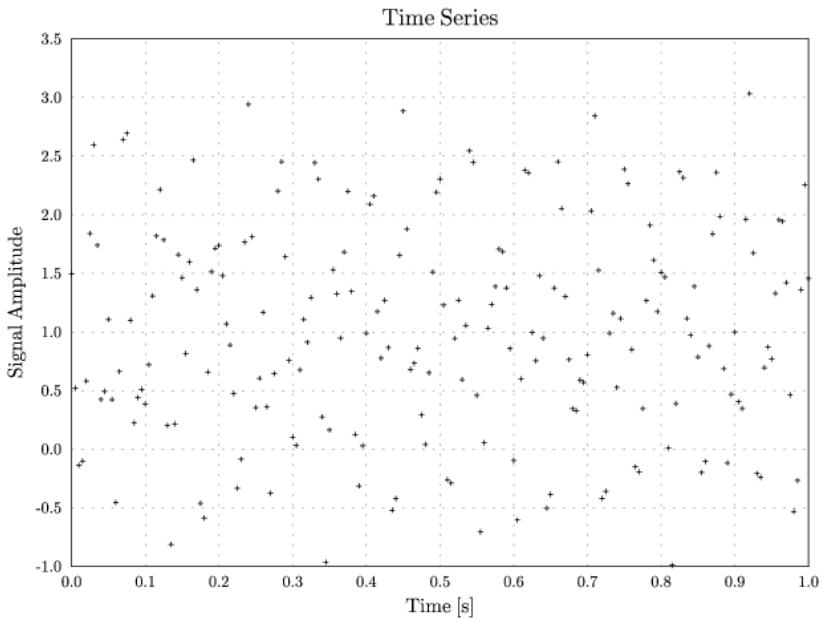
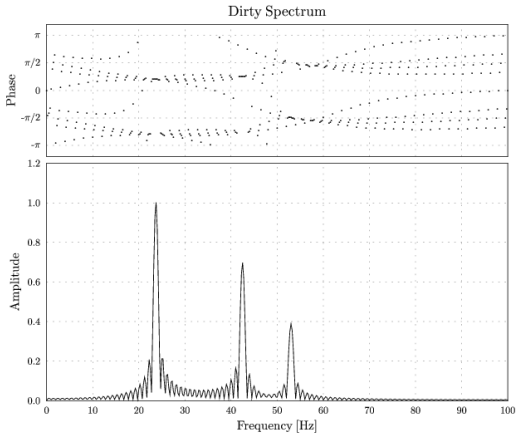


Figure 3.5: Time series consisting of 201 evenly spaced samples.

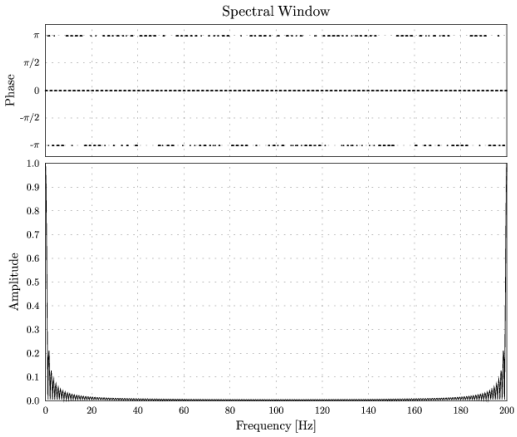
series. For this simple case the amplitudes measured from the CLEAN spectra precisely match the parameters of the simulated data. The CLEAN code also successfully returns the correct phases, although it should be noted that the phase plotted is that of the mean time so one must first subtract a quantity $2\pi\nu\langle t_r\rangle$, to recover the phase component relative to the origin. For example, with this time series, for the peak at $\nu = 23.7$ Hz and mean time of $\langle t_r\rangle = 0.5$ s and measured phase of $\phi = 0.2\pi$, the phase evaluated at $t = 0$ has the value $\phi = 0.5\pi$.

3.5.1.2 Results from folding

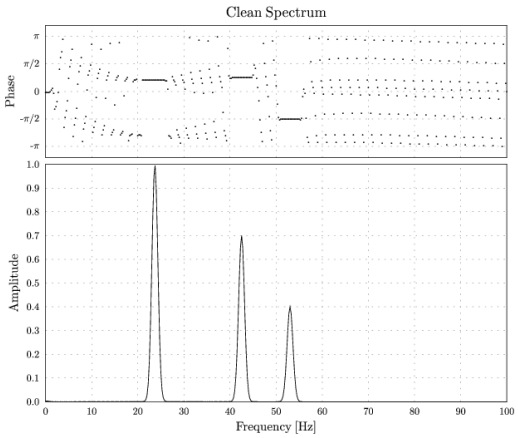
The results of folding the time series with the periods corresponding to the peaks $\nu =$ 23.7, 42.5 and 53.0 Hz is shown in Figure 3.7. Note that in every case the phase, amplitude and DC offset of component one from Table 3.1 is recovered to well within error as might be expected for such a simple test.



(a) Dirty spectrum



(b) Window spectrum



(c) Clean spectrum

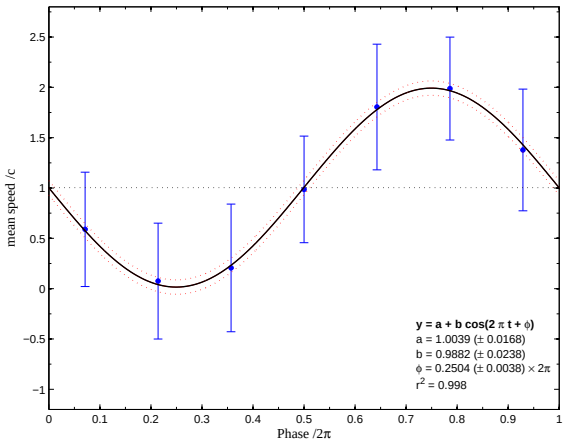
Figure 3.6: Spectral analysis of a synthetic dataset consisting of 201 equally spaced time samples, shown in Figure 3.5, using the CLEAN algorithm. The CLEAN spectrum in (c) is generated using 100 clean iterations and a clean gain of 0.5.

3.5.2 Random Drops

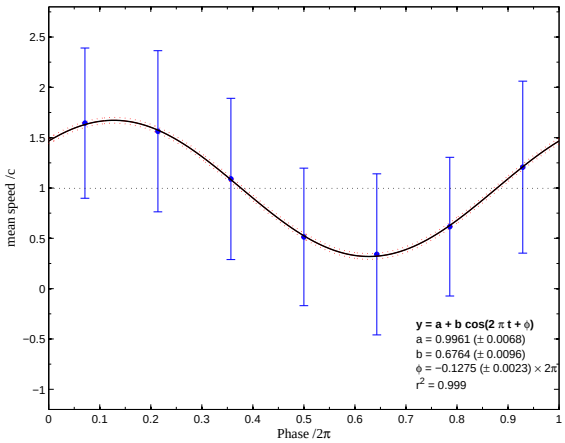
The "random" drops time series were generated by taking the 201 evenly spaced samples generated for the previous test and selecting 80% of the points to be removed leaving a time series of 40 points (see Figure 3.8).

3.5.2.1 CLEAN spectra

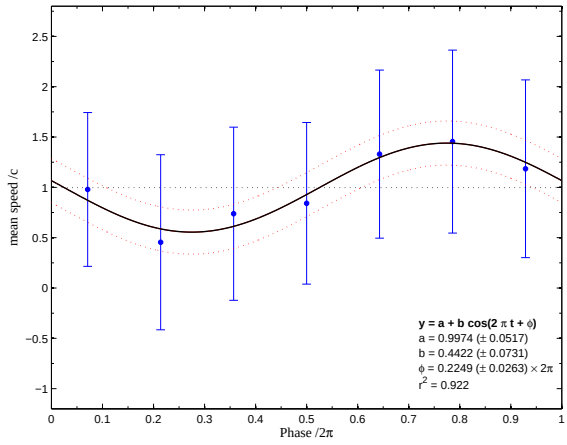
With 80% of the data points removed, spectral analysis is far more challenging. This is clearly evident from the dirty spectrum in Figure 3.9 where the three main spectral



(a) $\nu = 23.7$ Hz



(b) $\nu = 42.5$ Hz



(c) $\nu = 53$ Hz

Figure 3.7: Synthetic time series dataset folded into 7 bins.

components are clearly visible, but there is far more confusion from other peaks in the spectrum. Despite this, CLEAN again performs well clearly identifying the main spectrum components correctly.

3.5.2.2 Results from folding

Folding the time series was also successful (see Figure 3.10) although the errors in the fitted sinusoidal curve are correspondingly far higher than for the better sampled time series.

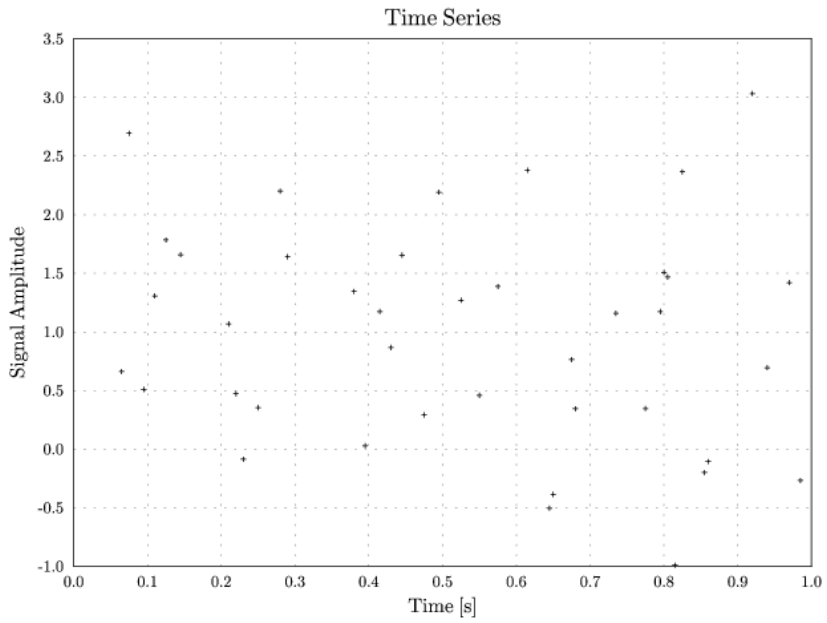


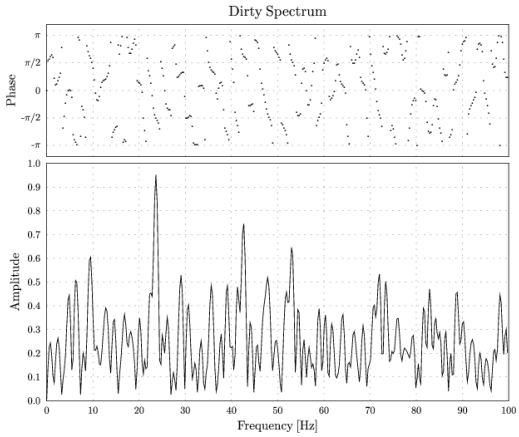
Figure 3.8: 201 evenly spaced time samples, with 80% random drops.

3.5.3 Sampled data with a Duty Cycle

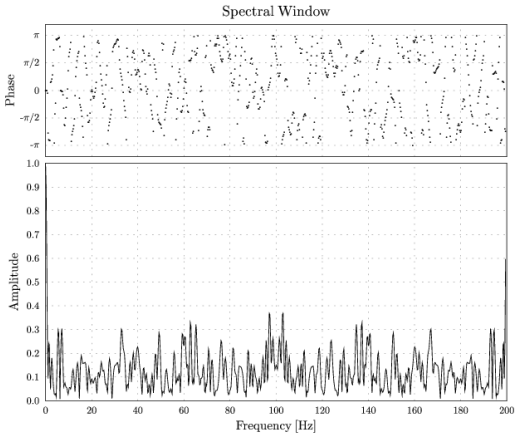
Finally a test time series was generated with a 20% duty cycle which resulted in 42 points sampled in the range 0.0 to 1.0 (see Figure 3.11).

3.5.3.1 CLEAN spectra

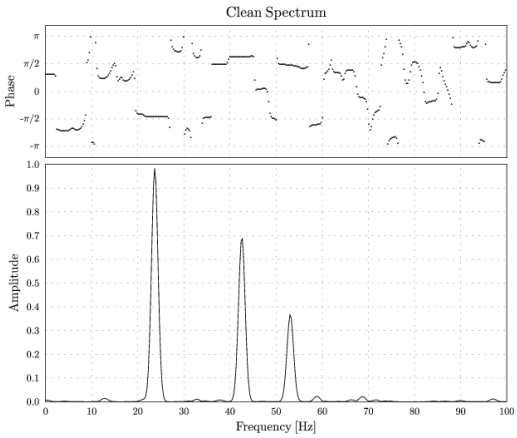
Clearly this type of sampling results in spectra which are extremely difficult to interpret due to dominant periodicities resulting from the regular underlying sample period introduced by the rectangular wave nature of the duty cycle, which is clearly shown in the window spectrum of Figure 3.12. This test provides a great example of where conventional Fourier analysis of the time series would have failed, but CLEAN recovers the spectral components.



(a) Dirty spectrum



(b) Window spectrum

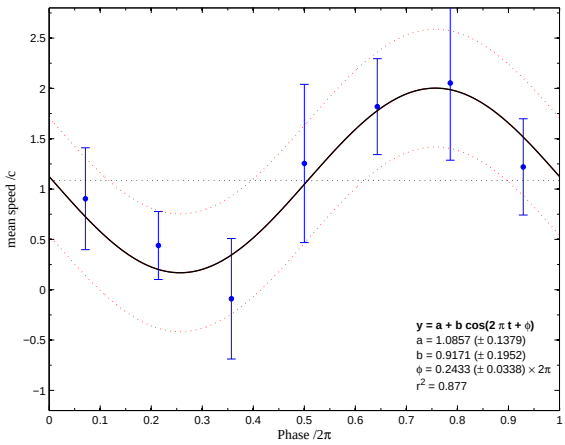


(c) Clean spectrum

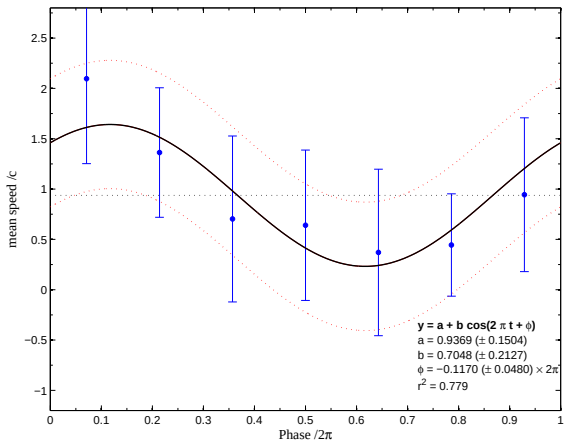
Figure 3.9: Spectral analysis of the same synthetic dataset of 201 equally spaced times but with 80% random drops (see Figure 3.8), using the CLEAN algorithm. The CLEAN spectrum in (c) is generated using 100 clean iterations and a clean gain of 0.5.

3.5.3.2 Results from folding

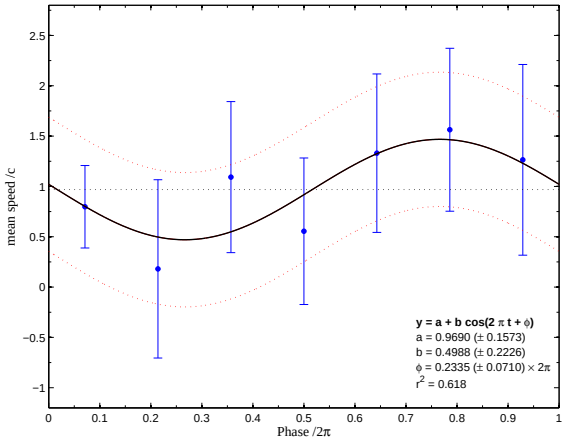
Folding the duty cycle time series also demonstrates the success of this method as can be seen in Figure 3.13.



(a) $\nu = 23.7$ Hz



(b) $\nu = 42.5$ Hz



(c) $\nu = 53$ Hz

Figure 3.10: Folded time series of the same synthetic dataset having 201 equally spaced times but with 80% drops.

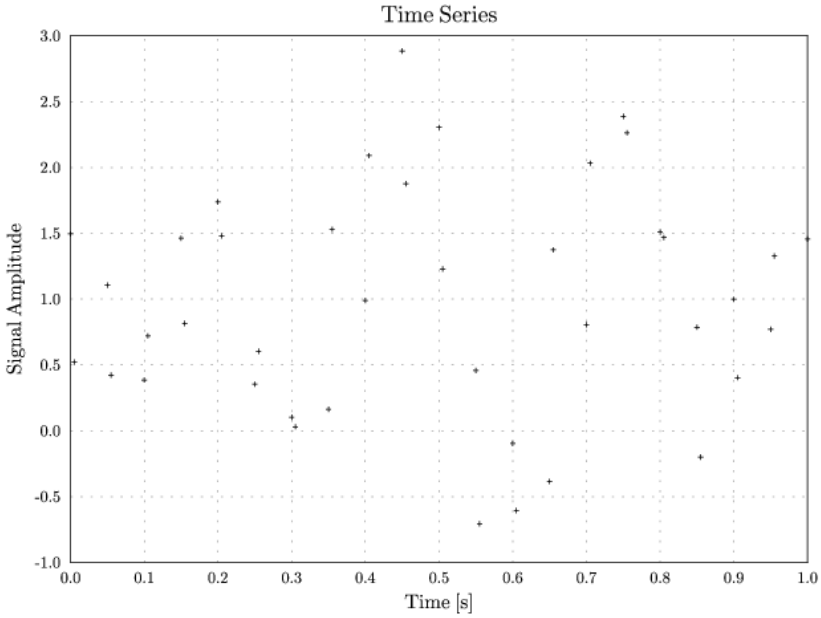
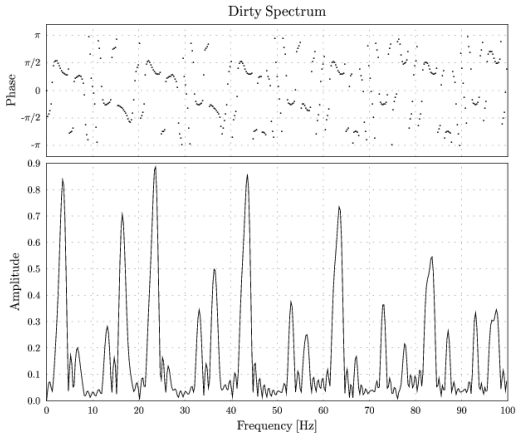
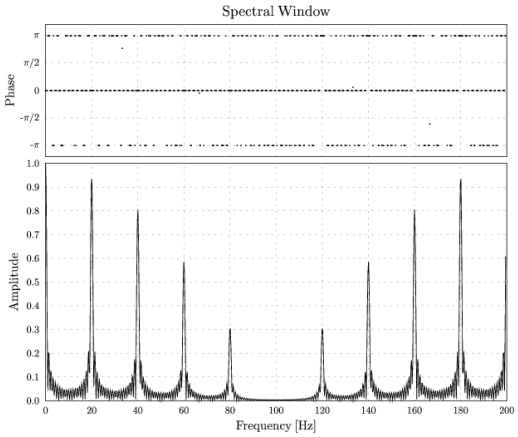


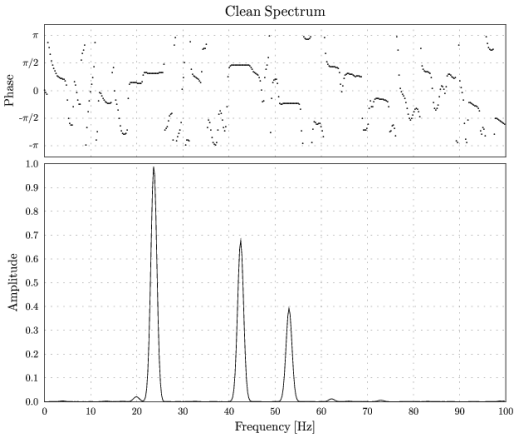
Figure 3.11: Time series generated with a 20% duty cycle.



(a) Dirty spectrum

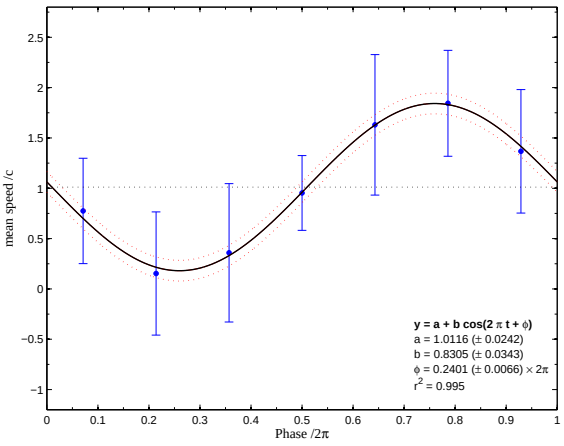


(b) Window spectrum

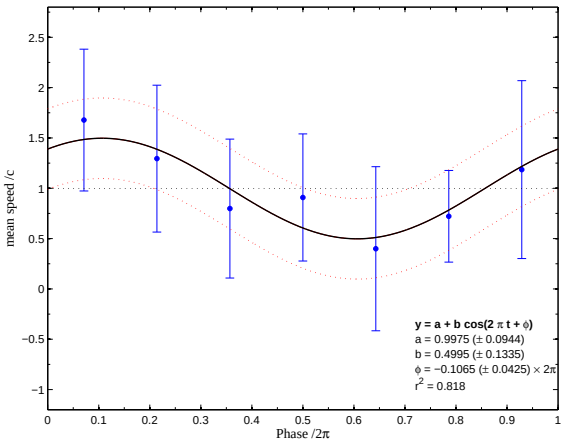


(c) Clean spectrum

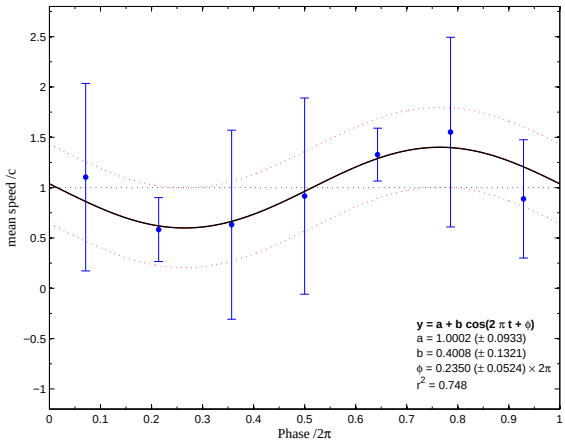
Figure 3.12: Spectral analysis of the 20% duty cycle time series (Figure 3.11), using the CLEAN algorithm. The CLEAN spectrum in (c) is generated using 100 clean iterations and a clean gain of 0.5.



(a) $\nu = 23.7$ Hz



(b) $\nu = 42.5$ Hz



(c) $\nu = 53$ Hz

Figure 3.13: Folded time series of 20% duty cycle.

3.6 Analysing the SS 433 data sets

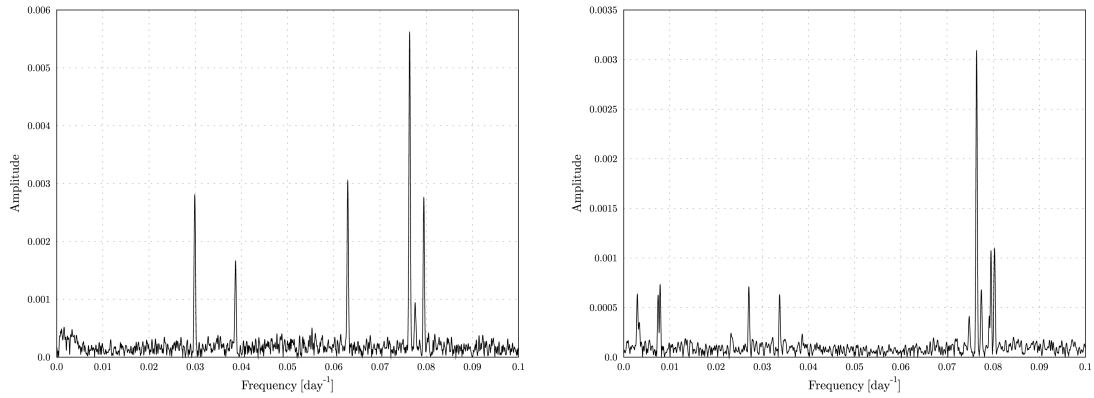
The aim of this investigation is focused on establishing if there is any further evidence for a change in the phase of the 13.08 day signal found in the jet speed data by Blundell et al. (2007) can be found by looking at the archival data set in more detail. In this analysis, by folding the archival data set and a data set from 2004 show evidence of a phase shift of a quarter of a cycle in the peak of the jet speed at 13.08 days was observed.

To investigate this further I first re-analysed the archive data from dates between 1978 to 1996 and the 2004 data using the new analysis written for this investigation and then looked at sub-sections of this. Using both CLEAN and folding techniques the aim was then to establish if there is any evidence that the phase or indeed frequency of the spectral component of the jet speed associated with the orbital period changed steadily and systematically over time or whether any change was more consistent with a stochastic wander.

3.6.1 Experiments with CLEAN parameters

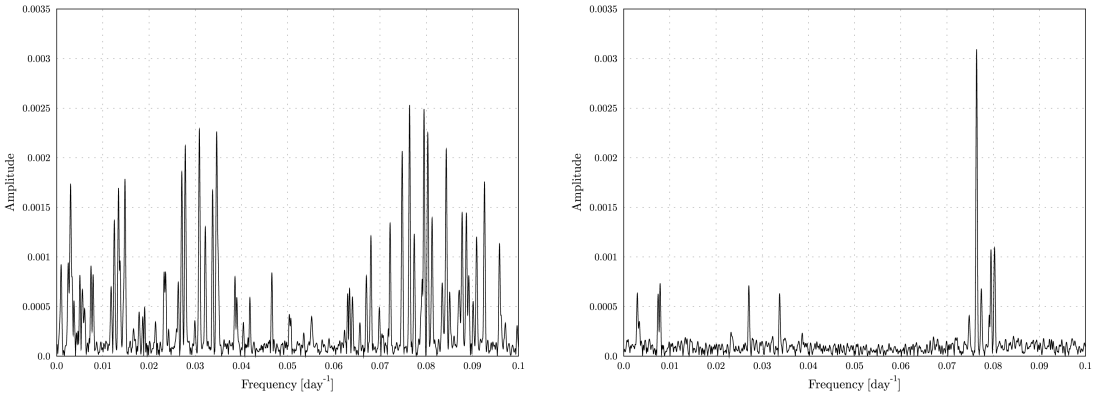
During the analysis of the SS 433 spectra it was found that for particularly complicated dirty spectra, the CLEAN spectra are extremely sensitive to the choice of CLEAN parameters: the clean gain and number of iterations, the degree of over-sampling (sub-Nyquist sampling) of the spectra, and the maximum frequency to which the spectrum is generated. It should be noted that for unevenly sampled data series standard Shannon-Nyquist sampling theory applies only loosely. The effect of each of these parameters is illustrated in Figures 3.14, 3.15 and 3.16. It was found that choosing a maximum frequency that was too low was disastrous due it having the effect of leaving un-cleaned side-lobes of large spectral components outside the frequency range in a way similar to that of bright sources outside the field of view in radio imaging. The choice of over-sample was found

to be less critical as long as the spectrum contained enough points such that peaks in the spectrum were identified sufficiently accurately. The choice of CLEAN gain proved problematic because, for the purposes of stability of the solution, it is highly advisable to choose a small gain for the cases of poor time series data but choosing a higher gain can allow useful positive bias to the cleaning of the highest peak in the spectrum in a way very loosely similar to that of CLEAN boxes in the deconvolution of two-dimensional radio images.



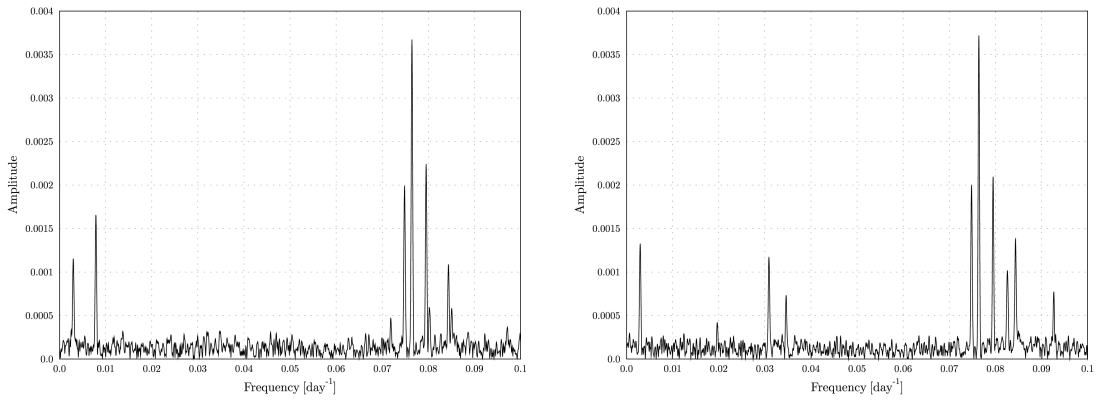
(a) gain = 0.8 (b) gain = 0.1

Figure 3.14: The effect of CLEAN gain.



(a) 0.6 days^{-1} (b) 1.2 days^{-1}

Figure 3.15: The effect of maximum frequency.



(a) oversample = 4 (b) oversample = 25

Figure 3.16: The effect over-sampling the spectrum.

3.6.2 A strategy for finding spectral components

After looking at both CLEAN and folding as methods for finding spectral components in the time series data it can be seen that each pose a number of advantages and disadvantages. CLEAN is sensitive to the choice of input parameters and identifying the peaks correctly from which to subtract the window spectrum.

These problems can to a certain extent be avoided by oversampling the spectrum and selecting a small clean gain. Having said this CLEAN provides a fast method of identifying peaks with reasonable accuracy where folding is rather more of a hit and miss technique as scanning over a large number of fold frequencies is computationally expensive. Experience has shown that folding also suffers from sensitivity to the number of bins. CLEAN also suffers from a lack of statistical basis with no direct method of evaluating the significance of spectral components found and in addition the suggested methods for gaining statistics on CLEAN pose an unfeasible computational problem for large data sets.

For this investigation I have therefore adopted to first analyse peaks with CLEAN and then fold the candidate peaks to provide a means of verification. Producing sub-bins from the archive data was carried out with the binning script described in Section 3.4.4.

3.7 Analysis of the archival data series

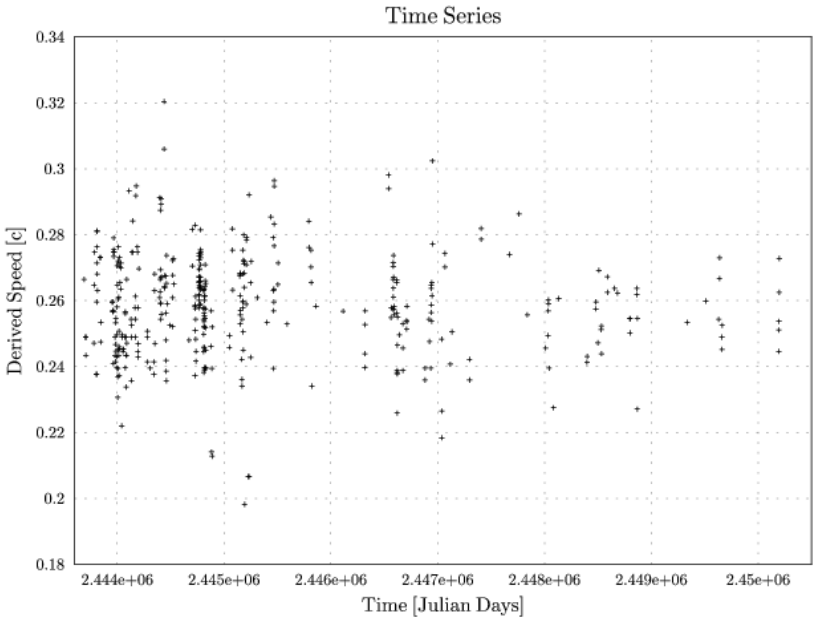


Figure 3.17: Archival time series speed data (1978 to 1996).

3.7.1 CLEAN spectra

After extensive testing of the various parameters, CLEAN spectra have been been generated with a 5.2 day^{-1} maximum frequency, an over-sample of 25 and cleaned to an average residual level of approximately 2% of the original peak amplitude using a gain of 0.1. A high over-sample was selected to minimise the errors when subtracting the highly structured spectral window.

Results of the CLEAN spectra are shown in the Figures 3.18 and 3.19 and a table 3.2 lists the principal spectral components. It should be noted that the phase displayed in the plots of spectra is the phase at the mean time whereas the phase displayed in the table is relative to the optical ephemeris using the convention of Goranskii et al. (1998), namely Julian date 2450023.62.

Other than the 13.0860 (± 0.0346) day peak which is very close to the orbital period its not clear that any periodicites listed in table 3.2 can be attributed to known parameters of SS 433 (the 12.58 day peak has been shown by Blundell & Bowler (2005) to be consistent with being a beat with the Earth's orbital period of 365.256366 days). In an attempt to see if any of the peaks from the CLEAN spectra could be further validated, a Monte-Carlo-like method was adopted where 20% of the points were dropped at random for a number of runs to build up a set of spectra with different sampling. The results of this were far from conclusive but are shown in Figure 3.20.

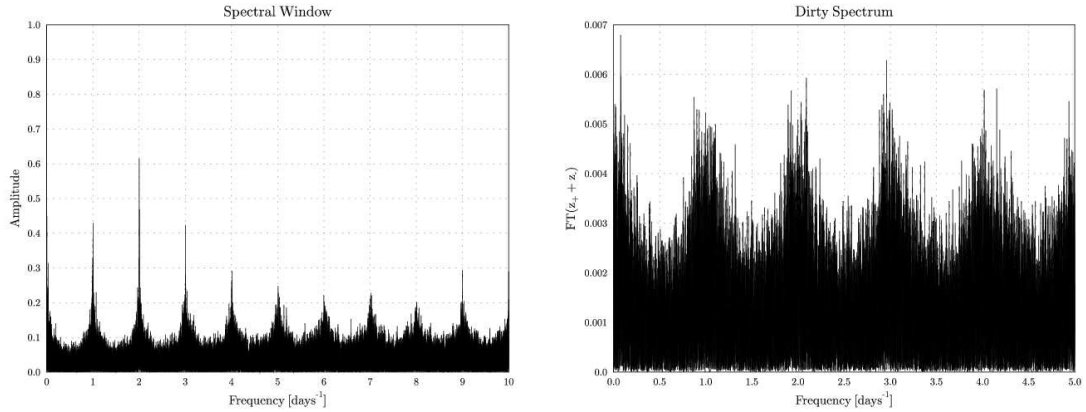
It is interesting to observe that while the peak at 13.086 days is very close to the orbital period, established from eclipses observed with optical photometry to be 13.08211 (± 0.00008) days (see Goranskii et al. 1998) the CLEAN spectra peak at a slightly different position.

i	A_i	$\nu_i(\pm 0.0002)$	Period	$\phi_i^{<t>}/\pi$	ϕ_i/π
1	0.000514	0.00297	336.700	–	–
2	0.000910	0.00790	126.662	–	–
3	0.000779	0.02710	36.903	–	–
4	0.000751	0.03377	29.610	–	–
5	0.003106	0.07642	13.086	-0.53	-0.67
6	0.000708	0.07745	12.912	–	–
7	0.001084	0.07953	12.575	–	–
8	0.001076	0.08030	12.453	–	–
9	0.000776	0.11973	8.352	–	–
10	0.001048	0.18123	5.518	–	–
11	0.001930	0.87273	1.146	–	–

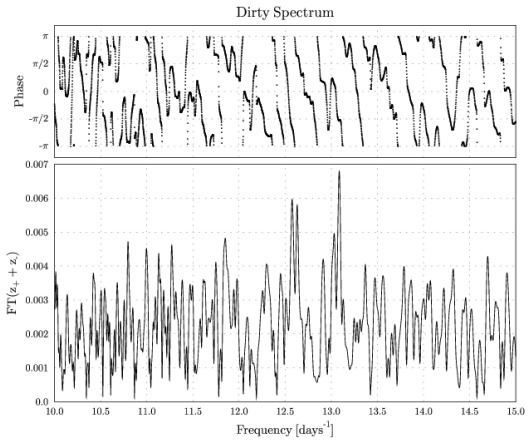
Table 3.2: Spectral components identified from the archival speed data.

3.7.2 Folded data

Unlike spectroscopic analysis with CLEAN, folding provides no direct method of searching for periodicities in a time series however by carrying out a range of folds over different periods and looking for where the amplitude of the fitted curve reaches a maximum, a



(a) (b)



(c)

Figure 3.18: Spectra of the archival speeds data.

crude search can be performed. Figure 3.21 illustrates this. It can be seen that both the amplitude and phase at the peak in this folded search match well with the CLEAN spectra in Figure 3.19. For the subsequent investigation, I therefore chose to fold both over the established orbital period of 13.08 days as well as over the frequency of the maximum peak in the clean spectrum most similar to with the orbital period, the results of which are shown in Figure 3.22. It should be noted that the fold search clearly demonstrates the sensitivity of the phase calculation on the position of the peak amplitude or fold period.

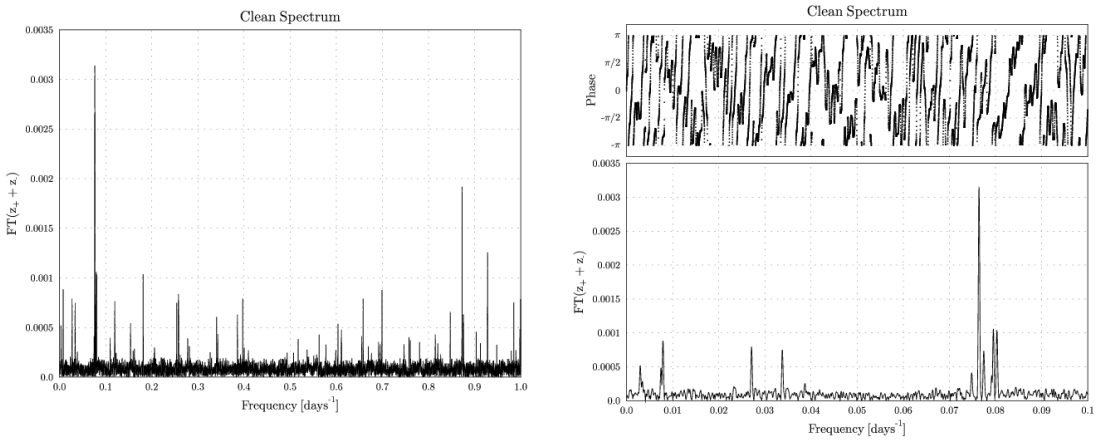


Figure 3.19: CLEAN spectra of the archival speed data.

3.7.3 Lomb periodograms

The Lomb periodograms for the archive data are shown in figure 3.23. In line with the CLEAN and folded results, the peak in the Lomb periodogram at 13.0845 ($\pm$ 0.03) days establishes the presence of the orbital period in the jet speed data. The only other peak in the spectrum above the 95% confidence level corresponds to a period of 12.58 days and as described in Section 3.7.1, this can be attributed to a beam with the Earth's orbital period.

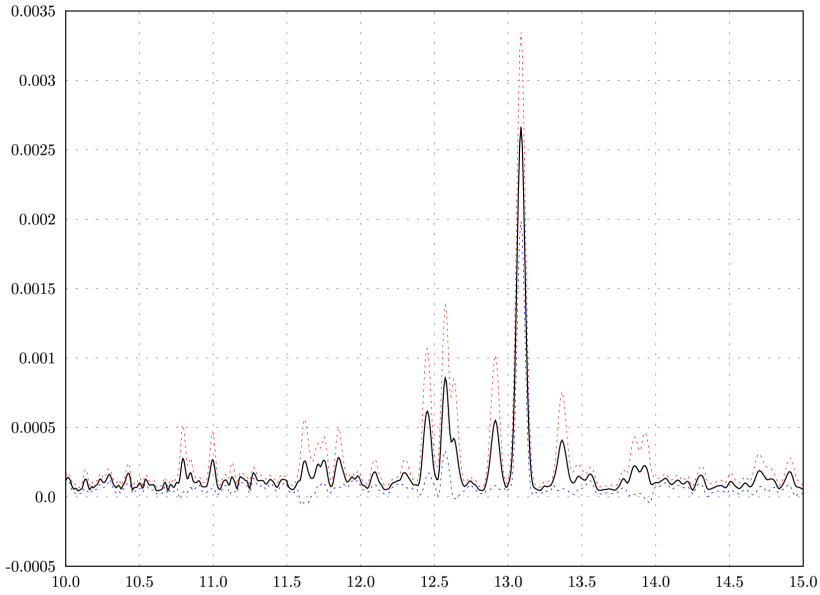
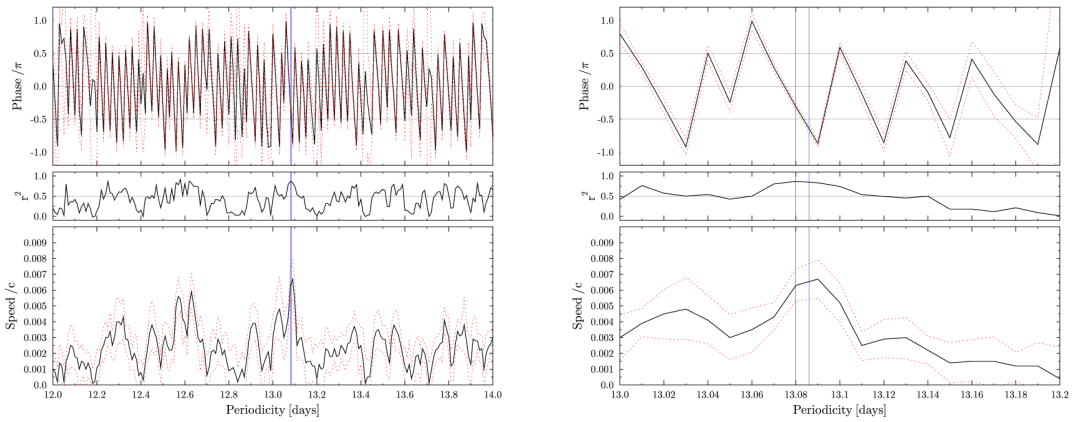
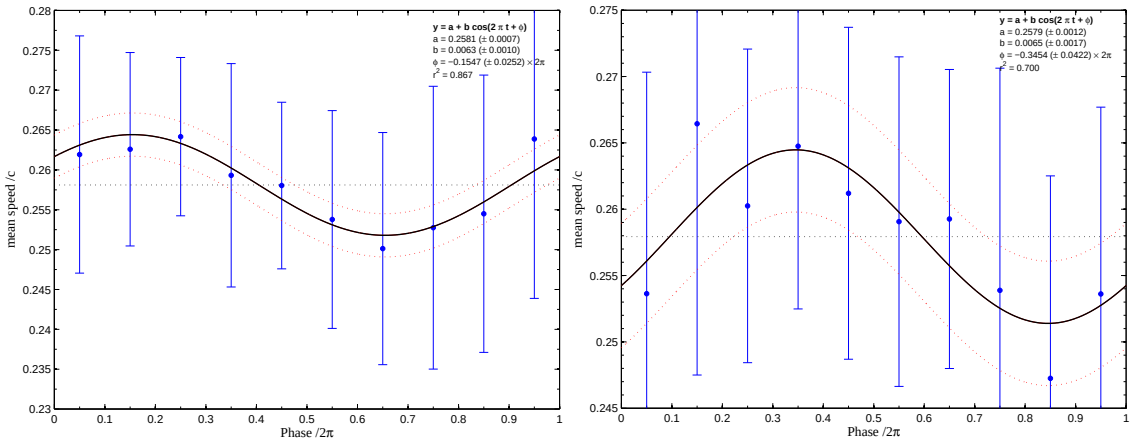


Figure 3.20: Mean spectrum (black line) resulting from 50 CLEAN spectra of the archival speed data with 20% random drops. The error on the mean (mean $\pm$ 3 σ) is shown by the red and blue dotted lines.



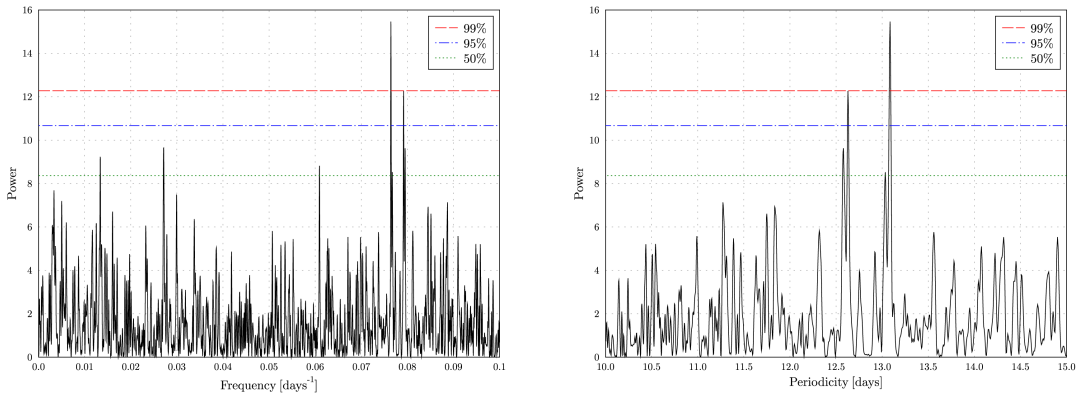
(a) (b)

Figure 3.21: Folded archival speed data (10 bins).



(a) 13.08 days (b) 13.086 days

Figure 3.22: Folded archival speed data.



(a) (b)

Figure 3.23: Lomb periodogram of the archival speeds data.

3.8 Analysis of the 2004 data

Further to the archive jet speed data, Blundell et al. (2007) present a series of data taken over 28 days in 2004 from which it was suggested that the phase of the 13.08 day peak had shifted. This data shown in Figure 3.24 is re-analysed here.

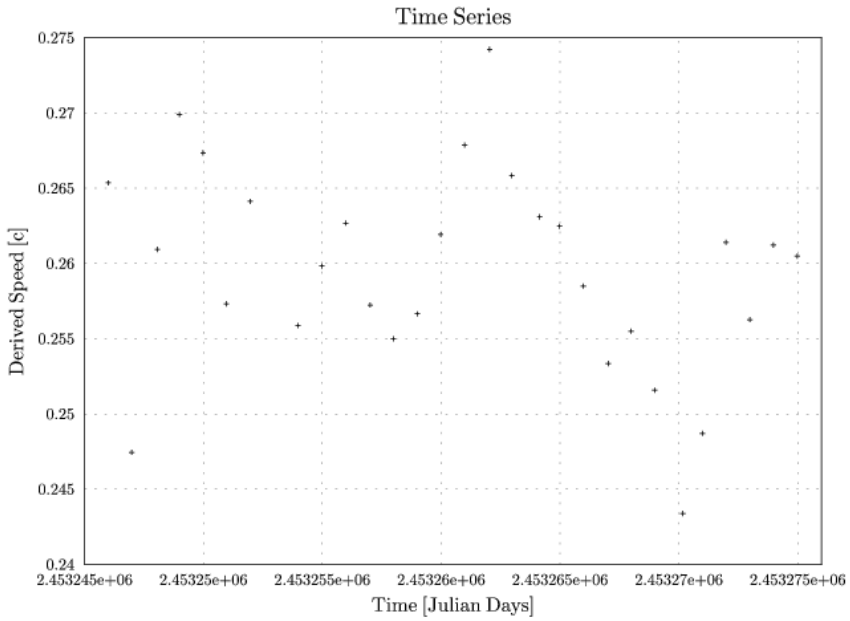


Figure 3.24: The 2004 time series speed data.

3.8.1 CLEAN spectra

Results of processing the data with the CLEAN code are shown in Figure 3.25 with the main spectral components identified in Table 3.3. Again there is evidence of a peak coinciding with the 13.08 day orbital period however the maximum amplitude of the peak occurs at 13.348 ($\pm$ 2.52) days. Folding the data with both about the orbital period (figure 3.26) shows the phase shift seen by Blundell et al. (2007) however folding at the peak in the clean spectrum and the results of a search though fold periods (figure 3.27) shows how sensitive this phase is to misidentification of the peak periodicity.

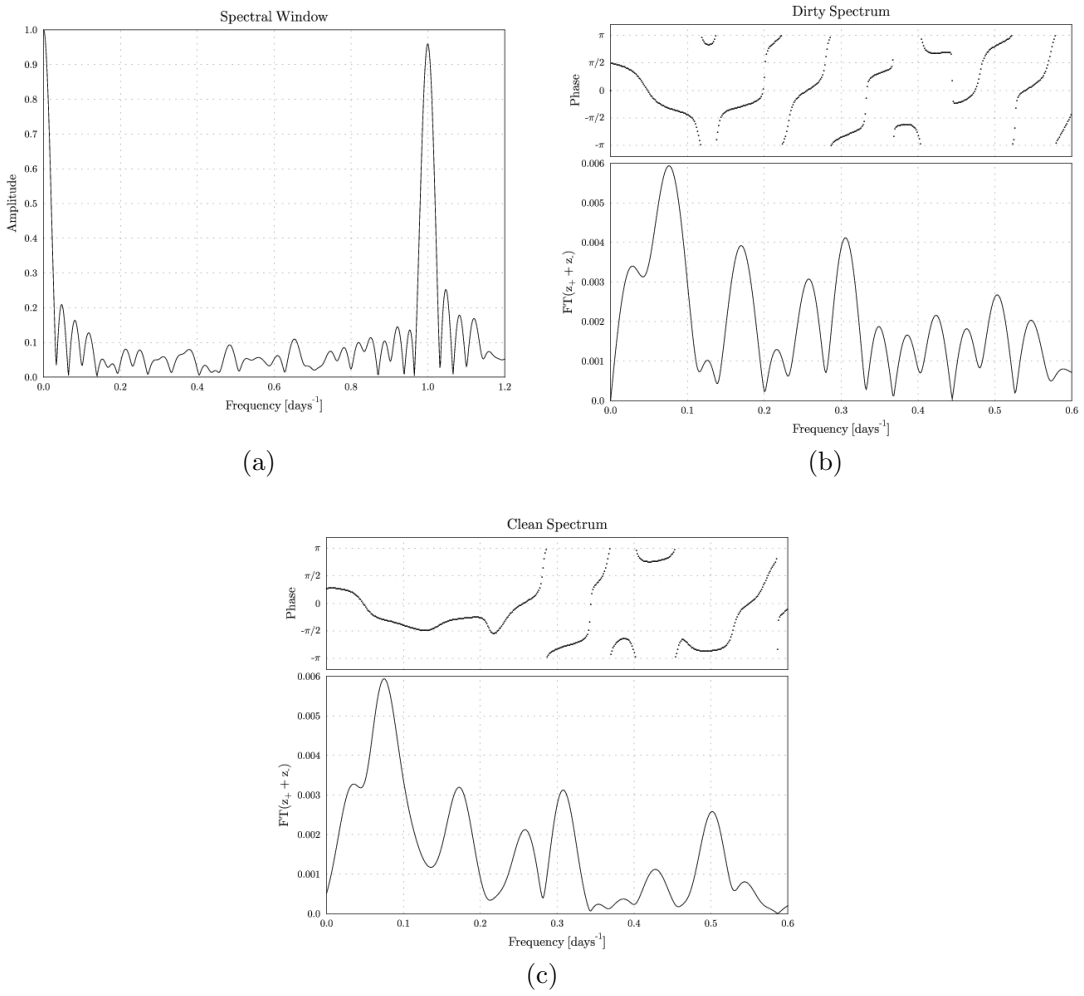


Figure 3.25: Spectra of the 2004 time series speed data.

i	A_i	$\nu_i(\pm 0.018)$	Period	$\phi_i^{<t>}/\pi$	ϕ^i/π
1	0.00328	0.0333	30.030	–	–
2	0.00594	0.0749	13.348	-0.30	0.66
3	0.00320	0.1733	5.770	–	–
4	0.00210	0.2567	3.896	–	–
5	0.00313	0.3079	3.248	–	–
6	0.00110	0.4274	2.340	–	–
7	0.00256	0.5010	1.996	–	–

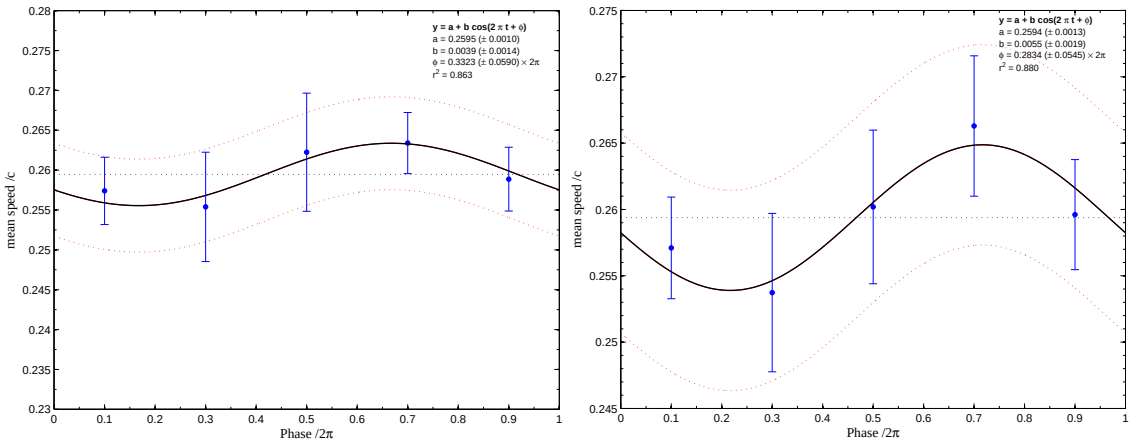
Table 3.3: Spectral components of the 2004 speeds data set with $\langle t \rangle$ = 2445391.061117 days. Goranskii et al. (1998) optical ephemeris = 2450023.62 ± orbital period days.

3.8.2 Folded data

Plots showing the 2004 speed data folded at 13.08 and 13.348 days are shown in Figure 3.26. The CLEAN and folding algorithms, in contrast with Lomb (see Figure 3.28), reveal

subtle hints of a periodicity consistent with the longer duration monitoring archival data spread over many years. In the brief time range encompassed by the '2004' data, namely two months, it is remarkable that the CLEAN can reveal subtle hits especially when one considers that the microquasar was exhibiting a major outbust/flare event towards the end of this time rather than a continuous duration of its normal behavior (Blundell et al. 2011).

Figure 3.27 shows a plot of the amplitude response of a four-point sine fit to a large number of folded time series, folded in the range 5 to 20 days. This demonstrates that while there is some evidence for the a peak in the spectrum at around 13.08 days, the signal recovered by folding is very weak and has a lot of uncertainty in its position.

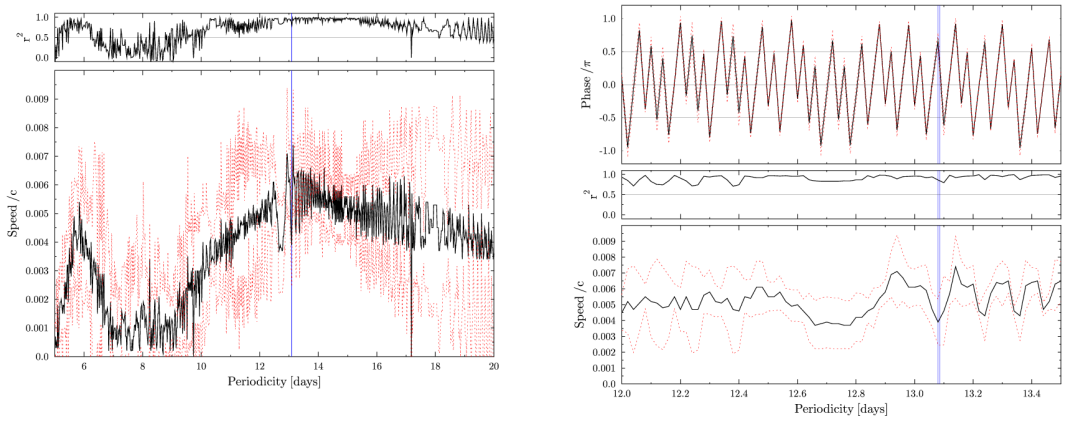


(a) 13.08 days (b) 13.348days

Figure 3.26: Folded 2004 speeds data. Red dotted line indicates the 2 σ level around the fit and the error bars indicate the standard deviation of data points from the time series making up the folded points.

3.8.3 Lomb periodogram

Lomb periodograms for the 2004 speed data is shown in figure 3.28. This figure confirms the absence of any hits of a signal discerned by the the Lomb method that is consistent with periodic 13.08 day signal which emerges from the much longer duration archival



(a) (b)

Figure 3.27: Search though fold periods for the 2004 speeds data.

dataset. This demonstrates that much longer duration time-series data are needed to properly establish the nature of the varying jet speed.

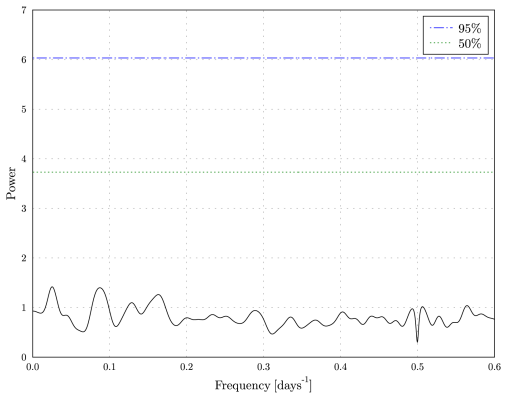


Figure 3.28: Lomb periodogram for the 2004 speed data.

3.9 Analysis of the split data

While the analysis of both the complete archive data set, and the data taken in 2004 shown here and by Blundell et al. (2007) show a degree of correlation of the jet speed's sinusoidal variation with the orbital period and evidence that the phase of this correlation has changed with time. In order to establish the validity of this in further detail a number

of sub-sections of the archive data were analysed. The data were split into 12 month bins, 24 month bins and other bins chosen around the best sampled data. These are displayed in Figures 3.30 through 3.37 with bins containing at least 25 points selected for analysis.

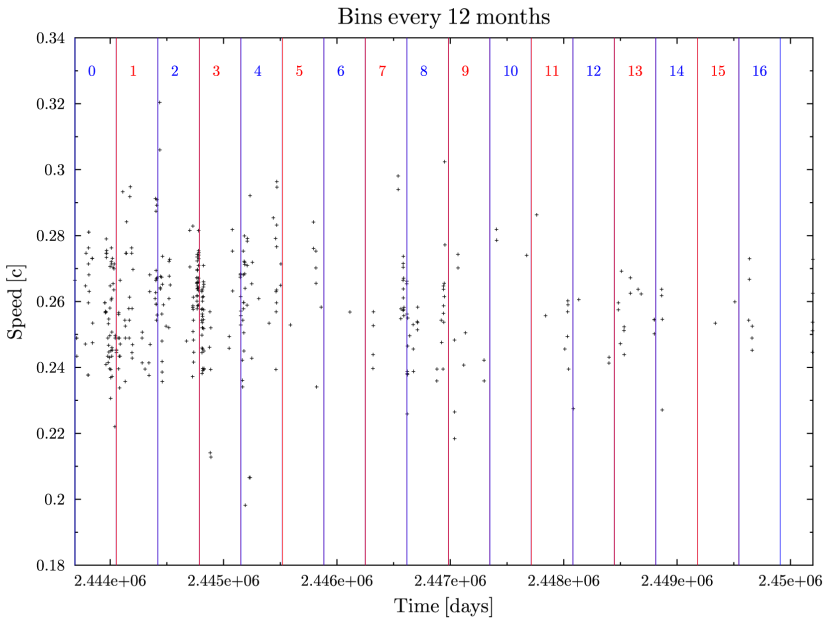


Figure 3.29: Archival speed data split into bins of 12 months.

3.10 Conclusions

While previous investigations (Blundell & Bowler (2005) and Blundell et al. (2007)) have shown the relative jet speed data shows a clear signal correlating closely to the 13.08 day orbital period, this investigation has shown that while this period is evident in the average speed over *long* periods of observation, the jet speed sinusoidal variation does not correlate with orbital period in a detailed way on shorter time scales. While the evidence of a 13.08 day peak in the archive data indicates a long term acknowledgement of the orbital period by the jet speed variation in this system, close inspection of this signal does not match precisely with the well determined orbital period of 13.082 days indicating that even this

correlation may be rather loose.

This study therefore concludes that with the available data there is no evidence of steady and systematic behaviour of the periodicity changing in the jet speed variation in a detailed way with orbital period. Having said this it should be noted that the data of this nature is extremely hard to analyse and the observations that will be carried out by the Global Jet Watch project (www.GlobalJetWatch.net) should provide a far more conclusive answer because the time sampling will be much finer as well as being much more sustained.

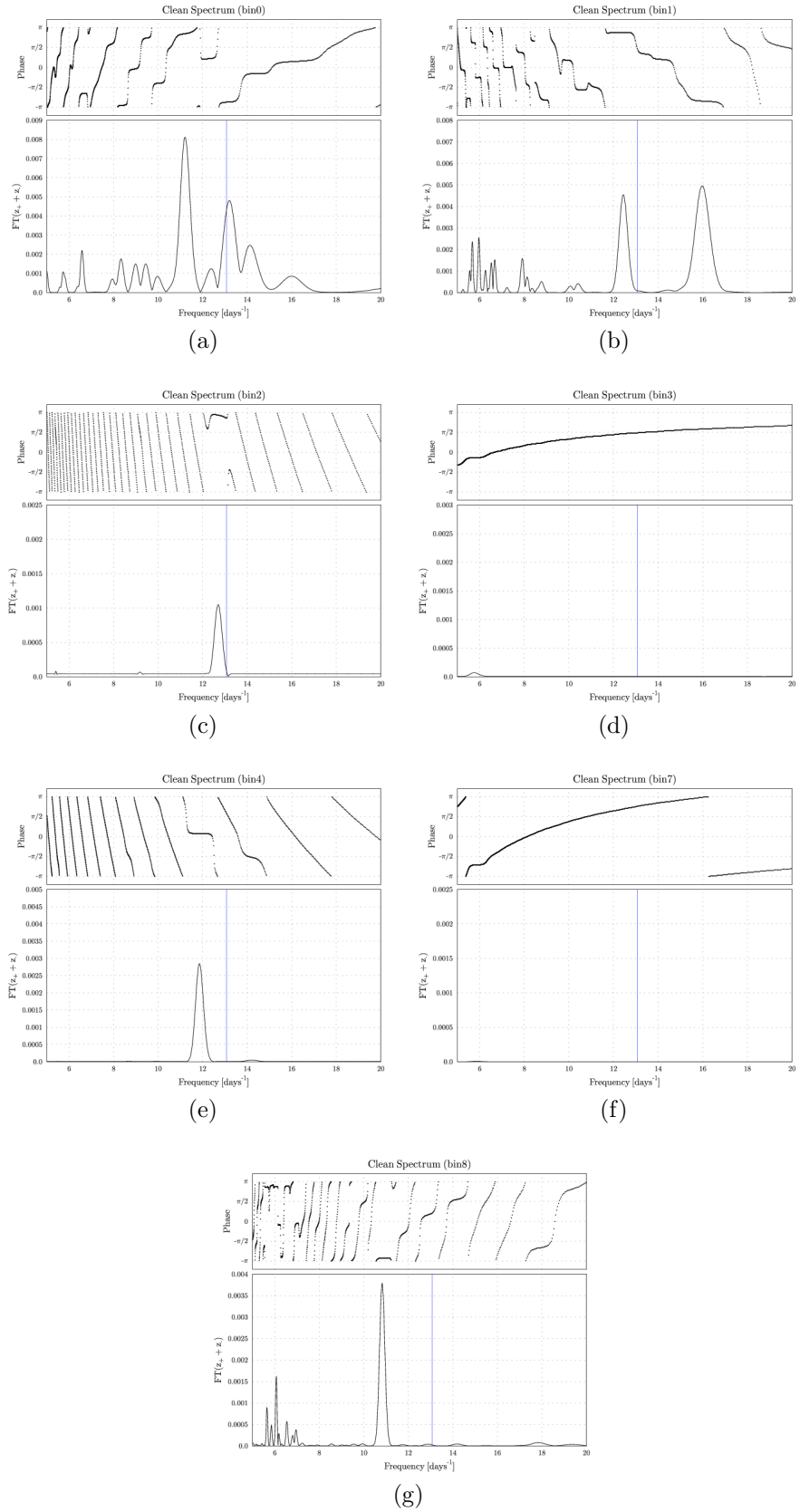


Figure 3.30: Spectra of archival speed data binned every 12 months.

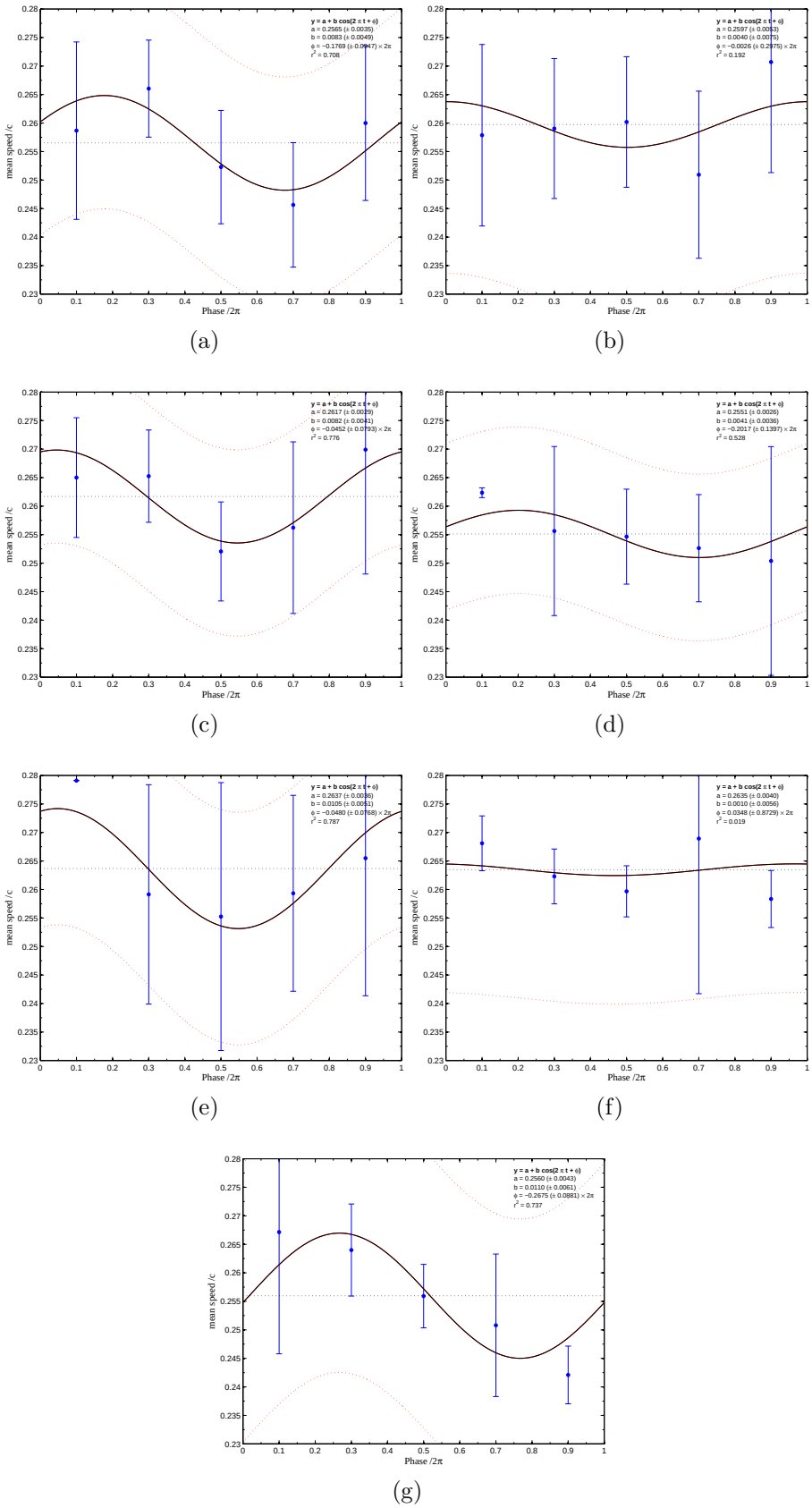


Figure 3.31: Archival speed data in 12 month bins folded with a period of 13.08 days.

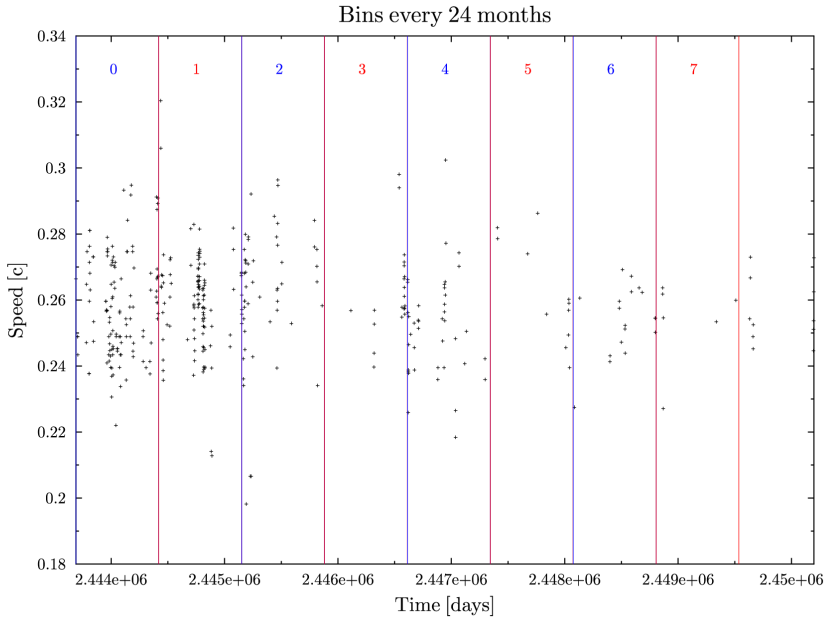


Figure 3.32: Archival speed data split into bins of 24 months.

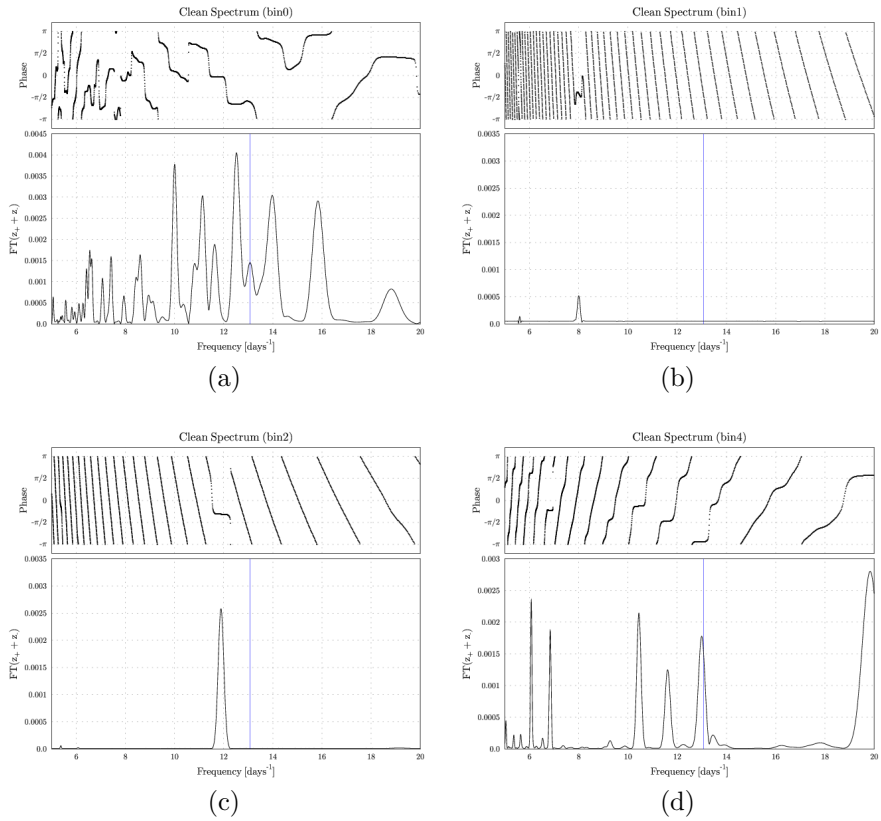


Figure 3.33: Spectra of archival speed data binned every 24 months.

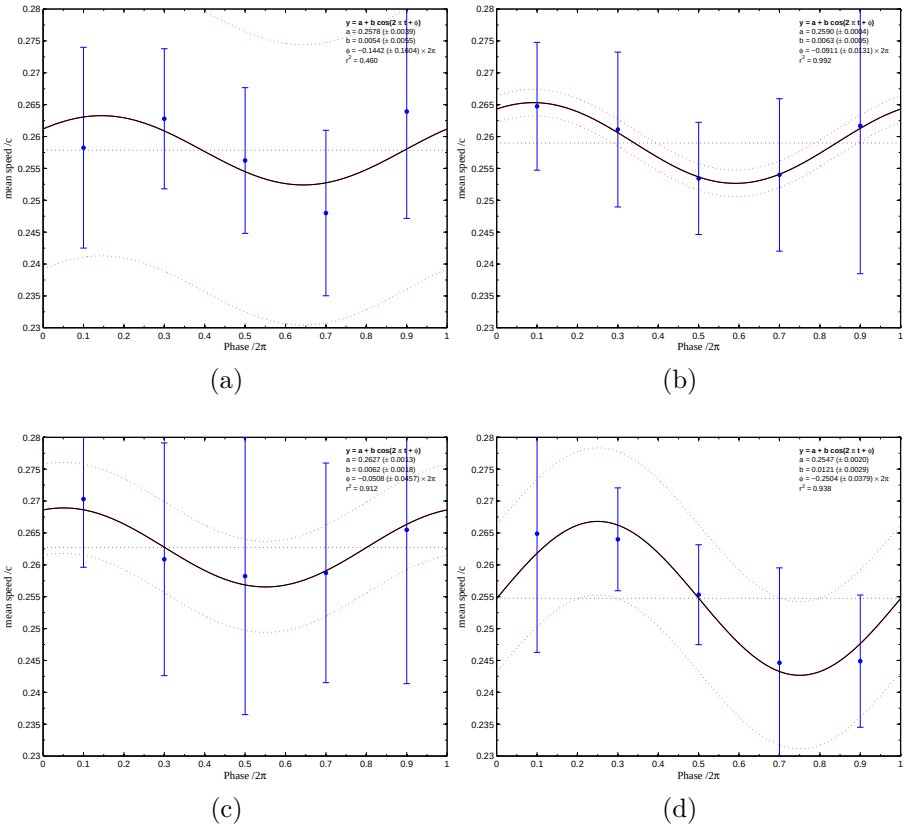


Figure 3.34: Archival speed data in 24 month bins folded with a period of 13.08 days.

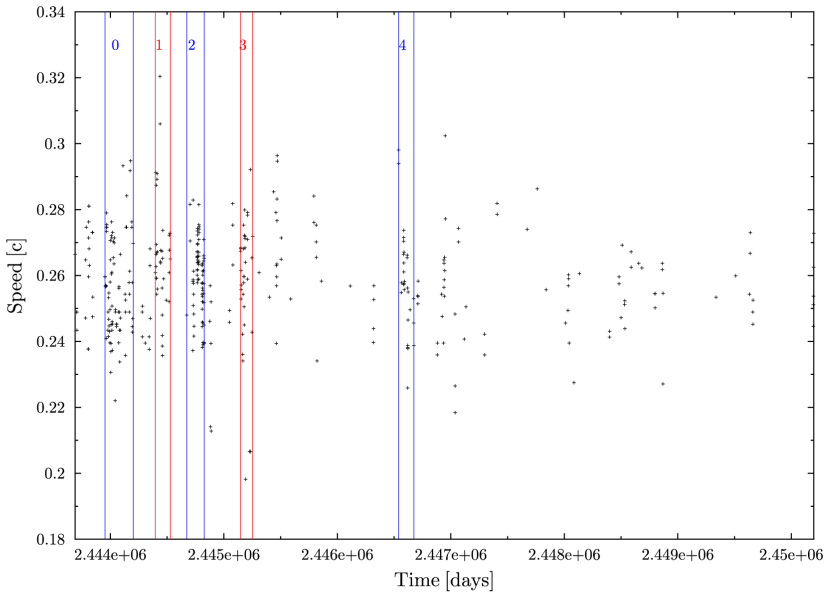


Figure 3.35: Archival speed data split into bins selected around areas having dense sampling.

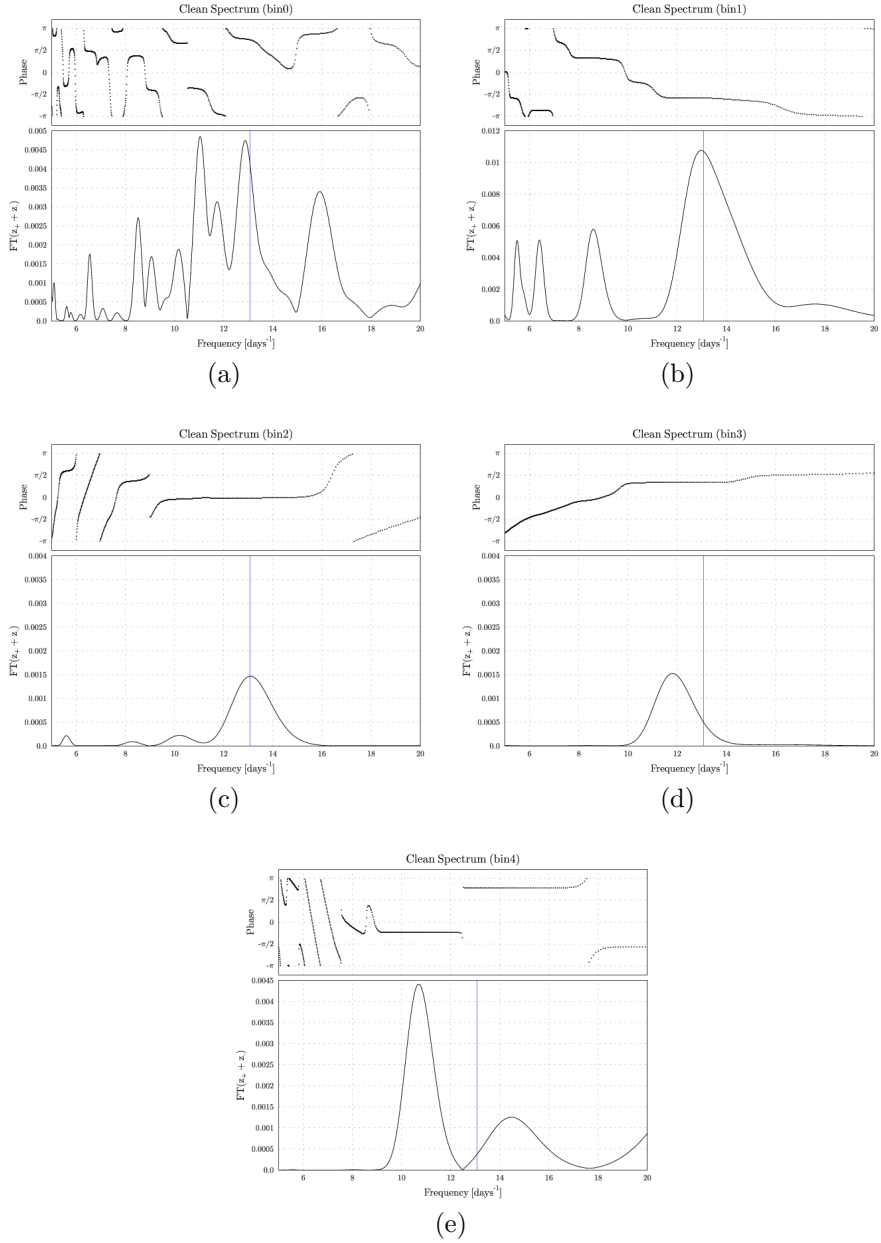


Figure 3.36: Spectra of data in bins selected around areas having dense sampling.

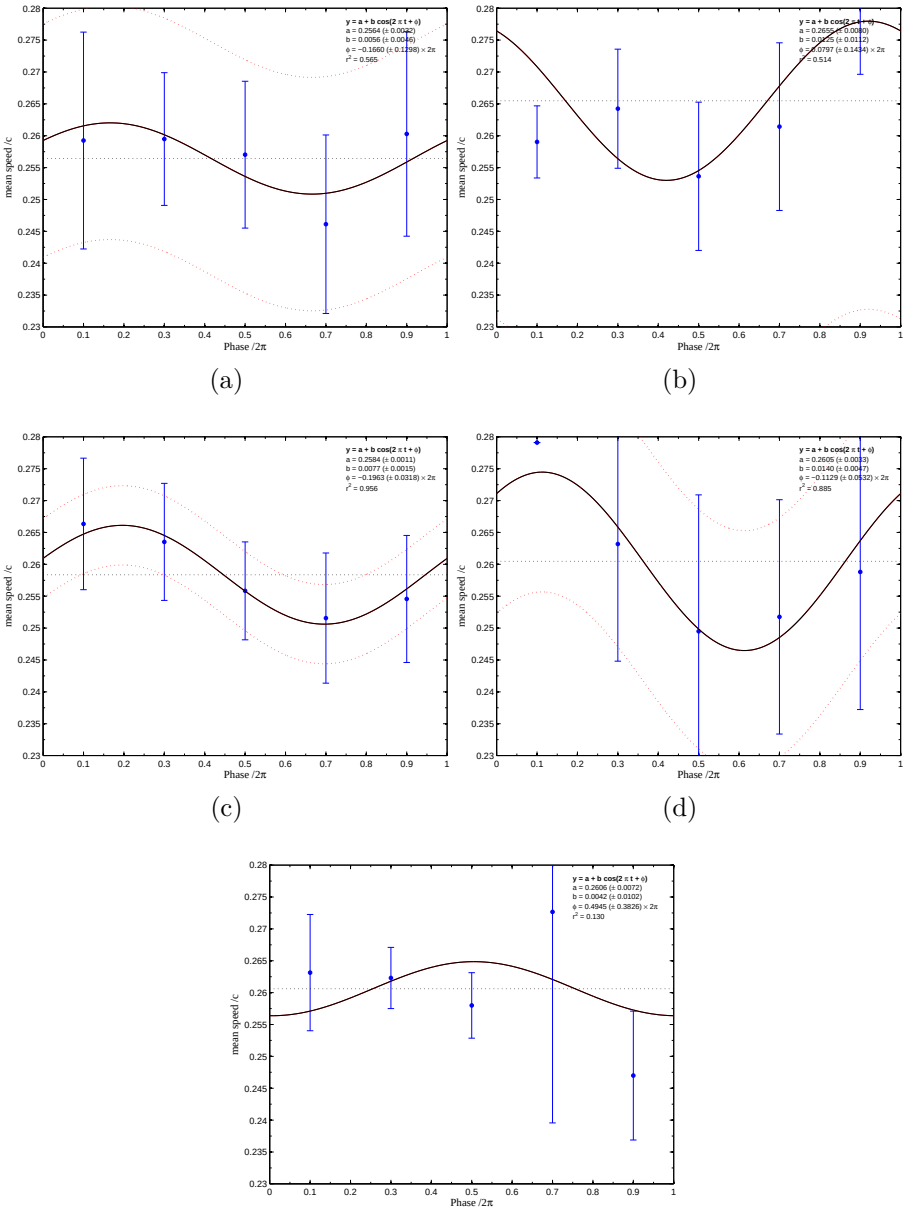


Figure 3.37: Bins of densely sampled data folded with a period of 13.08 days.

Chapter 4

GPUs for numerical programming

In this chapter I provide an overview of programming on GPUs for numerical code. During the process of working on this thesis, using GPUs for numeric programming has developed from a a black art of manipulating the programmable graphics pipeline to an increasingly mainstream aspect of advanced high performance computing.

(a) (b)

Figure 4.1: Photographs of (a) a modern CPU, and (b) a modern GPU. While on the surface these two processors have around 2 billion transistors and cover roughly the same area of silicon, their architecture and programming models are very different.

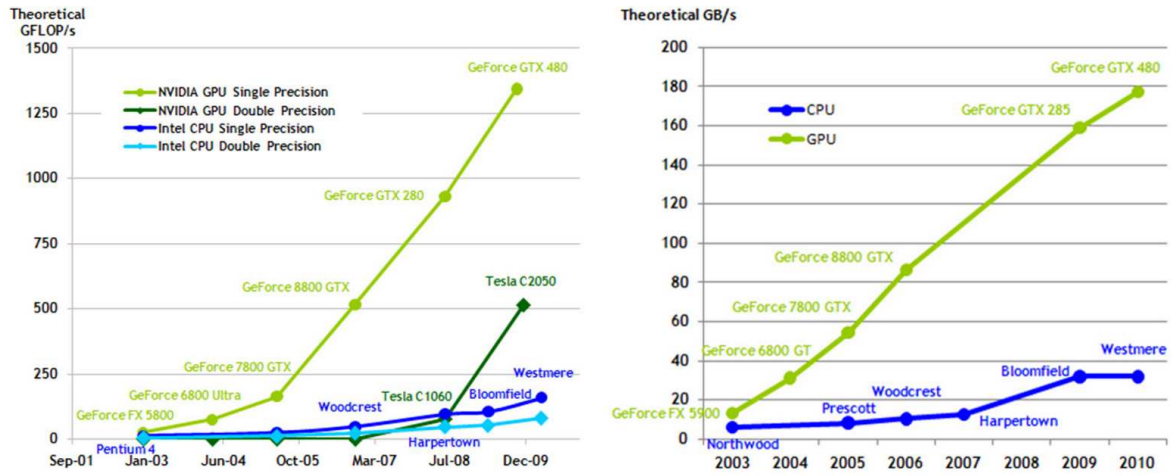
4.1 Introduction

Over the last six years, general-purpose computation on graphics processing units has evolved from a rather unknown research area to a technique with increasing prospect and

potential in real-world high performance computing applications.

Historically, graphics processors contained a fixed function rendering pipeline but over time this has evolved to incorporate greater and greater programming capabilities to the state today where the GPU can simply be thought of as a highly multi-threaded co-processor.

Computer graphics is a computationally demanding task which, driven largely by the electronic entertainment industry has led to the development of processors with both floating point performance and memory bandwidth which greatly exceed that of the CPU as shown in Figure 4.2. The performance gap between CPUs and GPUs, can be largely



(a) (b)

Figure 4.2: Floating point operations (a) and bandwidth comparison (b) between the CPU and GPU (NVIDIA Corporation 2010).

explained by the approach that GPUs take in their attempt to solve the problem of bandwidth starvation, common to modern high performance processors. Figure 4.3 illustrates much of this difference. CPUs attempt to maintain a high level of activity in their numerical processing units by dedicating a large proportion of their transistor budget to fast memory cache, super-scalar architecture and branch prediction. In contrast, GPUs dedicate almost all of their transistor budget to numerical processing units. While this

might initially seem to worsen the problem, by executing multiple concurrent threads, each of which access different elements of the data, the bottleneck of to having to feed a small number of extremely fast numerical processing units is all but avoided. This processing model, where large numbers of threads each act on different sections of the data, is known as single instruction multiple data, or SIMD, architecture (Flynn 1972). Another important benefit of the GPUs thread model is that with very large numbers of threads queued for execution, it is possible to swap out threads waiting for memory access in favour of those that are ready to proceed and this has the effect of hiding much of the the memory access penalty.

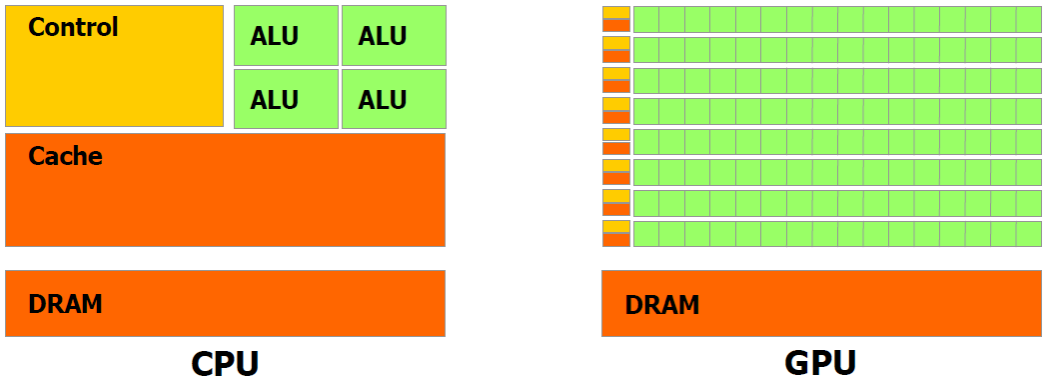


Figure 4.3: A Comparison of CPU and GPU architectures, NVIDIA Corporation (2010). CPU architecture dedicates a large amount of its transistor budget to control logic and cache whereas the GPU is constructed mainly from a large number of arithmetic logic units (ALUs).

Realising the huge potential of GPUs has been exploding in popularity over the last couple of years and today more and more large HPC systems using GPUs are being deployed. One of the reasons for this is the FLOPS per watt that GPUs offer. A typical GPU can provide just over one TFLOPS of peak performance for around 300 Watts, whereas a modern multi-core CPU can provide only a tenth of the compute capability for around 100 Watts resulting in the running cost per FLOP and carbon emissions of a GPU solution being extremely attractive. This is clearly demonstrated by the *green500*

(www.green500.org) list of supercomputers where five out of the top ten largest supercomputers, ranked by FLOPS per Watt, are now GPU based systems.

While GPU architectures clearly offer huge advantages for numerical processing, the downside is that applications and algorithms have to be ported to make use of them. To gain full advantage of the potential of GPUs, algorithms must be massively parallel with high arithmetic intensity and when writing code for GPUs careful consideration has to be made to achieve this.

The following sections give an overview of how GPUs are used to program numerical code first describing how this was done prior to the end of 2006 when graphics processors were comprised of a programmable but heterogeneous fixed graphics pipeline, and a description of how GPUs are programmed today with NVIDIA's CUDA language which was developed as part of the move in hardware to a homogeneous unified shader architecture.

4.2 GPGPU before CUDA

To satisfy the hunger for real-time, realistic animations driven by increasing demands in the games industry, graphics cards continue to gain ever faster processing capability.

While this processing power was been highly desirable to many areas of computing, the first generations of graphics processors developed with programmable SIMD extensions were far from trivial for use in non-graphics processing applications.

Fortunately one of the dominant drivers in the development of the latest graphical effects has been the ability for programmers to create their own custom instructions, and therefore the degree of programmability of the graphics pipeline has increased with time, to the point where the field of research known as General-Purpose computation on Graphics Processing Units (GPGPU) evolved (e.g. Harris 2002, Thompson 2002).

Despite different GPU manufacturers having variations in the exact specification of

their architecture, one of the large factors in making GPGPU possible was the development of a common programming interface supported by the various graphics card drivers. It is perhaps interesting to observe that today this situation is more favourable because CUDA, arguably the predominant system for programming GPUs, is vendor specific.

While a set of common standards for programming graphics cards existed, namely the OpenGL Shading Language (GLSL), the High Level Shading Language (HLSL), and C for Graphics (Cg), the difficulty of using these to program for GPGPU was considerable as these application programming interfaces (API) and languages were designed specifically for graphics applications with many core language constructs and programming models unsuitable for more general algorithms. Some early attempts were made at developing a general purpose API for programming GPUs for GPGPU such as BrookGPU (Buck et al. 2004), but these were plagued with performance penalties compared with using the native graphics languages due to insufficiently exploiting the GPUs features and exposing necessary GPU optimisations to the developer.

The following sections give an overview of the process of developing code for the first classes of GPGPU-capable GPUs. While this programming model was superseded at the end of 2006 by the release of the unified shader architecture, and dedicated GPGPU programming languages such as CUDA, described in section 4.3, in developing the GPU imaging code for this thesis I spent a considerable amount of time experimenting with this programming model and learnt many lessons which could be transferred to programming CUDA-capable GPUs to which the current version of my imaging code (described in later chapters of this thesis) is targeted.

4.2.1 GPU pipeline architecture

In order to understand how graphics processors were developed from a fixed instruction set architecture, designed strictly for polygon rendering, to a programmable processor

capable of huge numbers of floating point instructions per second, it is useful to look at some aspects of their architecture. Figure 4.4 shows the architecture of a typical com-

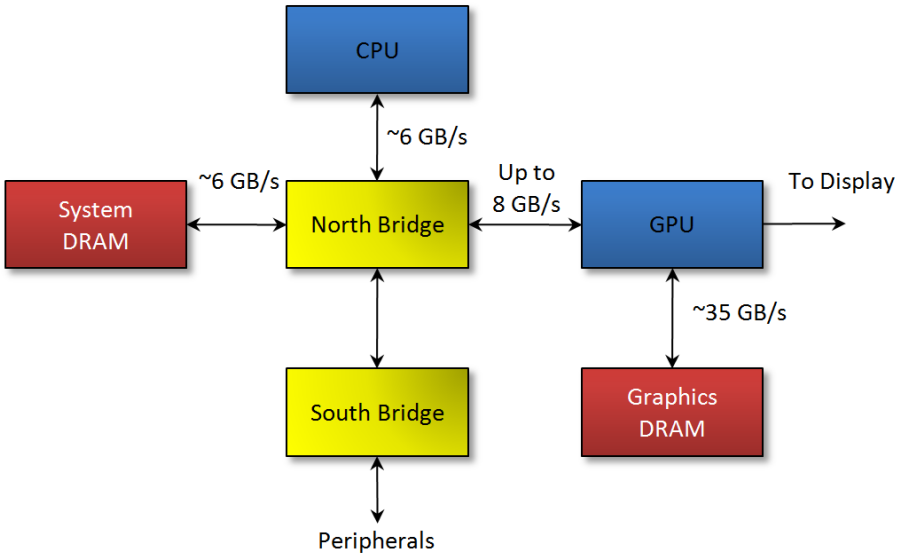


Figure 4.4: The system architecture of a typical desktop PC from ∼2006. Bandwidth values are given for an NVIDIA GeForce 6 series GPU, connected via Peripheral Component Interconnect Express (PCI-e) version 1.0, and double data rate synchronous dynamic random-access memory (DRAM).

puter system from around 2006, when mainstream GPUs were first becoming GPGPU capable. The bandwidth figures are shown to illustrate that there is significant variation in connection speed between different elements of the system, something that has to be taken carefully into account when moving data between the host system memory and the GPU. A generalised architecture for a typical GPU prior to the development of a unified shader architecture, described in section 4.3, is shown in Figure 4.5. It can be seen that the GPU consists of two main processing units called vertex shaders and fragment (or texture) shaders based on the task to which each is assigned. The vertex processors perform co-ordinate transforms on sets of points in 3-dimensional space which define the geometric shapes being rendered. The resultant 3-dimensional vector geometry is then converted into grids of pixels by the rasteriser before being passed to the fragment shaders. Fragment processors then operate on these pixel grids to perform complex per-pixel effects,

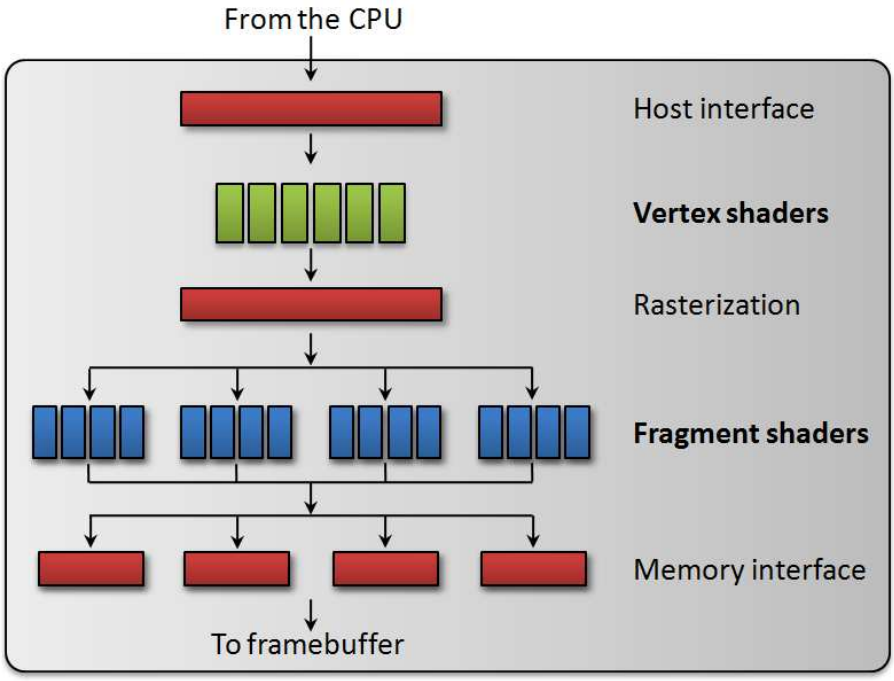


Figure 4.5: Schematic representation of the GPU pipeline architecture, similar to that of the NVIDIA GeForce 6 Series. Each stage of the pipeline has dedicated hardware for each operation.

such as manipulation of the colour attributes of textures applied to the scene. The huge processing performance of this architecture stems from exploitation of large amounts of parallelism in shader processors, which operate under the SIMD paradigm. Successive generations of GPUs have therefore grown in performance by adding larger numbers of these parallel processing units. GPUs of this type therefore adhere to a streaming architecture where data passes down the pipeline from one stage of processors to the next.

Running a typical GPU program can be split into a number of stages of this pipeline:

1. Commands, textures, and vertex data required for the GPU program are sent from the host CPU. This is accomplished by a command stream written by the host CPU which initializes and modifies the state of the GPU, sending the references to vertex and texture data to be used and associated rendering commands.

2. The vertex processors then allow for a set of commands to be applied to each set of

vertex data.

3. Vertices are then grouped into primitives (points, line or triangles) and culled and/or clipped to remove primitives not in the defined viewing space.

4. A process called rasterisation then calculates and generates the candidate screen pixels (also known as fragments) that are covered by each vertex primitive. A depth selection operation (known as a z-cull) which discards pixels that hidden by objects in front of them is also performed at this stage.

5. The texture or fragment processors next apply a shader program to each fragment (or pixel) independently. Like the vertex processors, fragment processors exploit parallelisation through having a number pipelines running concurrently. Processors operate on blocks of four pixels called quads, in a SIMD fashion capable of working on groups of hundreds of pixels at a time.

6. Processed fragments are finally written to a target pixel buffer consisting of either the frame buffer, that drives the display device, or an off-screen render target. GPU memory consists of multiple independent partitions which allow the memory subsystem to operate efficiently regardless of whether small or large blocks of data are written. GPUs also typically use a wide (e.g. 256 bit) memory interface giving very high bandwidths compared to that of the host, CPU accessible, system memory.

4.2.2 Programming Shaders

The development of programmable shader processors led to a number of high level shader programming languages used to program them. Among the most high profile of these are: the OpenGL Shading Language (GLSL), created by the OpenGL architecture review board (ARB), which was introduced as an extension to OpenGL 1.4; High Level

Shader Language (HLSL), created by Microsoft for use with their DirectX API; and C for Graphics or Cg, which was developed by NVIDIA which has a compiler which outputs both DirectX and OpenGL shader programs. These high level languages gave the possibility of programming GPU shaders for GPGPU, but with their focus on graphics the implementation of many general purpose algorithms proved extremely challenging. When using these fragment shaders however most GPGPU code is written to execute on the fragment shaders, whereby careful setup of the rendering pipeline, fragment processors can be configured to work on textures in a way analogous to working on 2-dimensional arrays in the host system memory.

4.2.3 Code example: simple convolution

In order to describe how fragment shaders are used to perform calculations on the GPU, this section illustrates some code, which I developed, to perform a simple 2-dimensional convolution. The code makes use of the OpenGL framework with fragment shaders written in Cg. I decided to develop this simple convolution code because convolution is a key algorithm used in imaging, and while relatively simple, the GPU implementation makes use of more than one input data array and uses a succession of two Cg fragment shaders in order to first evaluate a convolution kernel on the GPU followed by the convolution itself. The code takes care to keep the results of the first rendering stage in GPU memory which is important due to the considerable performance hit of transferring data to and from the GPU on the older architectures for which this code was written.

The code therefore takes a 2-dimensional array of data and generates a small 2-dimensional convolution kernel to perform the convolution operation shown in figure 4.6, and described by:

$$y[m,n] = x[m,n] * h[m,n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i,j] \cdot h[m-i, n-i], \tag{4.1}$$

where $x[m, n]$ is the input data, and $h[m, n]$ is the convolution kernel. As the GPU renders individual pixels in the output array in parallel, convolution is performed as effectively a gather operation per output pixel.

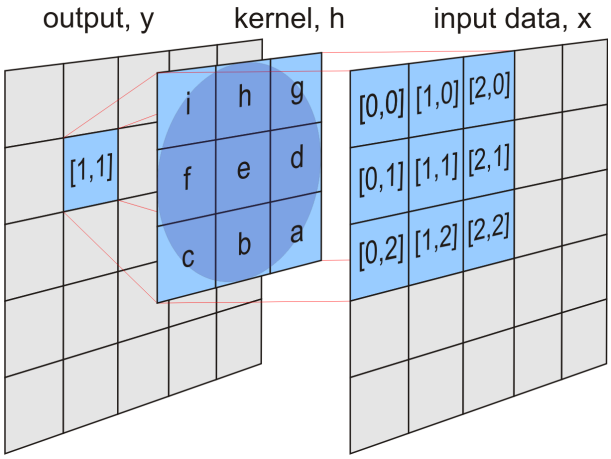


Figure 4.6: 2-dimensional convolution shown as a gather operation.

The Cg shader programs, which perform the calculation of both the convolution kernel and the convolution itself, are instantiated by a series of OpenGL commands which set up an environment for off-screen rendering and carefully map the textures to be rendered such that there is a one to one mapping between texture elements, known as texels, and pixels as required for numerical computations.

4.2.3.1 The program

The program follows a number of steps described below; the small source code snippets show relevant commands where appropriate:

1. **Generate or load data array in the CPU system memory.** For this I generate an empty 2-dimensional array and place values at a number of positions chosen at random.

2. **Initialise OpenGL to create a context to access graphics hardware.** For this I have used the OpenGL Utility Toolkit (GLUT), which while providing functions to handle such things as mouse click events and menus, is used in this code just to create an OpenGL context that is independent of the window system running on the computer (see listing 4.1).

Listing 4.1: Creating a OpenGL context

```
glutInit(&argc, argv);
glutCreateWindow("GPGPU convolution");
```

3. **Initialise the OpenGL extensions** required for general floating point computations using the OpenGL Extension Wrangle Library (GLEW) (see listing 4.2).

Listing 4.2: Initialise GLEW to obtain function pointers.

```
GLenum status = glewInit();
// Check if function entry points have been defined.
if (status != GLEW_OK || !GLEW_ARB_fragment_program)
    exit(ERROR_GLEW);
```

4. **Prepare the GPU rendering pipeline for offscreen rendering** of IEEE 32-bit floating point arrays. The traditional end point of every rendering operation is a frame buffer, a special chunk of graphics card memory from which the images that appear on a computer display are read. While typical displays use 32-bits of color depth (16.7 million colours), this is split into four separate 8-bit channels (red, green, blue, alpha) and the colour values are clamped in the range 0 to 255. Therefore in order to use 32-bit precision values directly, without having to resort custom arithmetic to convert the four 8-bit channels to single 32-bit value, an OpenGL extension, called `EXT_framebuffer_object`, is used. This provides an offscreen render target with the desired bit depth and avoids value clamping often associated

with colour buffers. See listing 4.3.

Listing 4.3: Prepare for offscreen rendering.

```
GLuint fboId;
// Create FBO (offscreen framebuffer).
glGenFramebuffersEXT(1, &fboId);
// Bind the offscreen buffer.
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboId);
```

5. **Setup 1:1 mapping between texture elements and pixels.** Data which are stored as arrays on the CPU memory are stored as textures on the GPU. In order to control which data elements are accessed in texture memory it is necessary to construct a special projection that maps from the 3-dimensional graphics environment to the 2-dimensional screen space and gives a 1:1 mapping between pixels, which are rendered to by the shader programme, and texture elements (known as texels), which are used to access input data. This is achieved by creating an orthogonal projection and a carefully selecting the viewport being rendered to be based on the size of the textures allocated as show in in listing 4.4.

Listing 4.4: Setup 3D space for 1:1 mapping of texels to pixels.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, size, 0.0, size);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0, 0, size, size);
```

6. **Create textures for both data and results.** The `framebuffer_object` extension conveniently allows direct rendering to a texture. Textures on the GPU are however read or write only so three textures have to be used in this convolution code; one for the data, one for the kernel and one for the result of the convolution. OpenGL allows a number of options for creating two dimensional textures, but the most commonly used for GPGPU, and the one used in this code, is

GL_TEXTURE_RECTANGLE_ARB. Unlike more commonly used texture targets used for graphics purposes, this provides unnormalised texture coordinates and arbitrary, non-power of two, dimensions up to a size limited by the graphics hardware. Textures can be configured to store either four 32-bit or single 32-bit values per texel. While storing a single value might seem the most obvious choice - which maps more closely to CPU arrays because the GPU allows for simultaneous computation on four-tupels of data. Therefore if a texture holding 4 values per texel can be used effectively in the algorithm, a greater computational throughput can be achieved. While in this example there is no need to work with texture data storing four points per texel I have chosen to use this storage format as it allows easy expansion of the code and the performance decrease is negligible. See listing 4.5.

Listing 4.5: Create textures to hold data and results.

```
const GLenum target = GL_TEXTURE_RECTANGLE_ARB;
const GLenum internalFormat = GL_FLOAT_RGBA32_NV;

// Create a new texture name.
GLuint textureId;
glGenTextures(1, &texureId);

// Bind the texture to a texture target.
glBindTexture(target, textureId);

// Turn off filtering and set correct wrap mode.
glTexParameteri(target, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(target, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(target, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(target, GL_TEXTURE_WRAP_T, GL_CLAMP);

// Tell renderer to replace current colour with the texel
// colours.
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

// Allocate the graphics memory.
glTexImage2D(target, 0, internalFormat, size, size, 0, GL_RGBA, GL_FLOAT,
    0);
```

7. **Copy input data to textures.** To transfer data to the GPU it is necessary to bind the texture to a texture target and schedule the data for transfer with an OpenGL call. The result of the texture transfer is that the CPU array is mapped row wise

to the texture. See listing 4.6.

Listing 4.6: Copy data to the GPU.

```
const GLenum target = GL_TEXTURE_RECTANGLE_ARB;
glBindTexture(target, textureId);
glTexSubImage2D(target, 0, 0 , 0, size, size,
GL_RGBA,GL_FLOAT,(GLvoid*)h_data);
```

8. **Setting up the Cg runtime.** In order to use a fragment shader it is necessary to setup the Cg runtime. This is achieved by first creating a Cg context as an entry point for the runtime, then obtaining a fragment profile and setting up a programme container for the shader programme that needs to be run. This procedure is standard to graphics processing code and is well documented in the various Cg manuals but a simplified example is shown in listing 4.7.

Listing 4.7: Setup the shader program runtime.

```
// Create a CG context.
CGcontext context = cgCreateContext();

// Obtain a shader for the specified class.
CGprofile profile = cgGLGetLatestProfile(CG_GL_FRAGMENT);

// Create the fragment program from code in the specified
// source file.
CGprogram program = cgCreateProgramFromFile(context, CG_SOURCE,
   sourceFile,
profile, entryFunctionName, 0);

// Prepare the program for binding.
cgGLLoadProgram(program);
```

9. **Set textures as render targets.** Assuming the frame buffer object (FBO) is already bound as the render target, and the texture has already been allocated (as described earlier), in order to render directly to a texture it needs to be attached to the FBO at one of the colour buffer attachment points using standard OpenGL calls. The support for attachment of various texture types to the FBO is somewhat hardware specific so this step has to be carefully checked for errors. Listing 4.8

shows an example of this.

Listing 4.8: Specify a texture as the render target.

```
const GLenum target = GL_TEXTURE_RECTANGLE_ARB;

// Attach target texture to first FBO attachment point.
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
  resultTextureId, 0);

 // Check for errors.
 GLenum status = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
 if (status != GL_FRAMEBUFFER_COMPLETE_EXT)
    exit(status);

// Set the texture attachemnt as render target.
glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);
```

10. **Drawing the scene to run the shader.** Computation on the GPU is finally triggered by rendering a suitable geometry. This is extremely simple after all the care and attention needed to setup textures and render targets. Given the projection and viewport is set up to create a 1:1 mapping of texels to pixels all that is required is rendering a filled quad that covers the whole viewport, as shown in listing 4.9. The four corner vertices and their associated texture coordinate attributes are specified, and in doing so the rasteriser will generate a fragment for each pixel covered by the quad. These fragments are then passed automatically to the fragment shader and as a result fragment program execution is triggered for each output position in the data array we are processing.

11. Finally the result is **transferred back from GPU memory** using the `glReadPixels()` command. This reads back the whole texture we just rendered and as this remains attached to the framebuffer object, it be can be copied directly back to the host system memory as shown in listing 4.10.

Listing 4.9: Draw the scene to perform processing on the GPU.

```
// Set up Cg parameter handles by name. (this very much
// depends on the specifics of the fragment program)
CGparameter d_size = cgGetNamedParameter(program, "d_cSize");
cgSetParameter1i(d_cSize, cSize);
CGparameter d_cFunc = cgGetNamedParameter(program, "d_cFunc");
cgGLEnableTextureParameter(d_cFunc);
.. etc ..

// Enables the Cg profile making the necessary OpenGL calls.
cgGLEnableProfile(profile);

// Bind the fragment program to the current state.
cgGLBindProgram(program);

// Make quad filled to cover every pixel / texel.
glPolygonMode(GL_FRONT, GL_FILL);

// Render a quad.
glBegin(GL_QUADS);
{
    glTexCoord2f(0.0, 0.0);
    glVertex2f(0.0, 0.0);
    glTexCoord2f(dataSize, 0.0);
    glVertex2f(dataSize, 0.0);
    glTexCoord2f(dataSize, dataSize);
    glVertex2f(dataSize, dataSize);
    glTexCoord2f(0.0, dataSize);
    glVertex2f(0.0, dataSize);
}
glEnd();
```

Listing 4.10: Copy results of computation back to the CPU memory.

```
// Read the rendered texture to the CPU array.
glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
glReadPixels(0, 0, size, size, GL_RGBA, GL_FLOAT, (float*)h_result);
```

4.2.3.2 Fragment shaders

The convolution code for this example, makes use of two Cg fragment programs: one for generating the convolution kernel and one for performing the convolution itself. As a result steps 8 and 10 of the list above are repeated for each of these programs with the appropriate textures attached. The Cg fragment sharers used in this code are relatively simple and are shown in listings 4.11 and 4.12. Listing 4.11 is the Cg fragment program used to evaluate the convolution kernel, which for the purposes of this simple example, is a 2-dimensional Gaussian and is parameterised only by its size, in pixels, and variance as

Listing 4.11: Cg shader used to evalue the convolution kernel

```
void convFunc(
        in float2 coord : TEXCOORD0,
        uniform int d_csize,
        uniform float4 d_variance,
        out float4 output : COLOR)
{
    const float x = coord.x - 0.5f - floor((float)d_csize / 2.0f);
    const float y = coord.y - 0.5f - floor((float)d_csize / 2.0f);
    const float x2 = x * x;
    const float y2 = y * y;
    output.r = exp(-(x2 + y2) / (2.0f * d_variance.r));
    output.g = exp(-(x2 + y2) / (2.0f * d_variance.g));
    output.b = exp(-(x2 + y2) / (2.0f * d_variance.b));
    output.a = exp(-(x2 + y2) / (2.0f * d_variance.a));
}
```

expressed by

$$f(x) = e^{-\frac{(x^2+y^2)}{2\sigma^2}}, \tag{4.2}$$

where σ^2 is the variance. As already explained Cg programs are executed by the GPU for each fragment of the render target. As I have carefully set this up to map directly to values in the function array I want to evaluate the function which can be though of as being executed for each element of the convolution kernel. As such the texture coordinates which appear as the first argument of this function and are provided by automatically by the runtime can be used directly to evaluate x and y coordinates of the function being evaluated. As shader programs work most efficiently with four-tuplet data (red, green, blue, and alpha color channels), expressed in this listing with the `float4` type specifier, I have chosen to generate different widths of kernel in each of these channels. This feature can be used in practice to store pairs of complex valued kernels which are required for synthesis imaging code.

Listing 4.12 performs a convolution between an input data array and the convolution kernel evaluated in the previous Cg shader code. This function evaluates the result of the convolution for output pixels of the result texture being rendered. The input data, convolution function and data being convolved are provided in the form of the textures

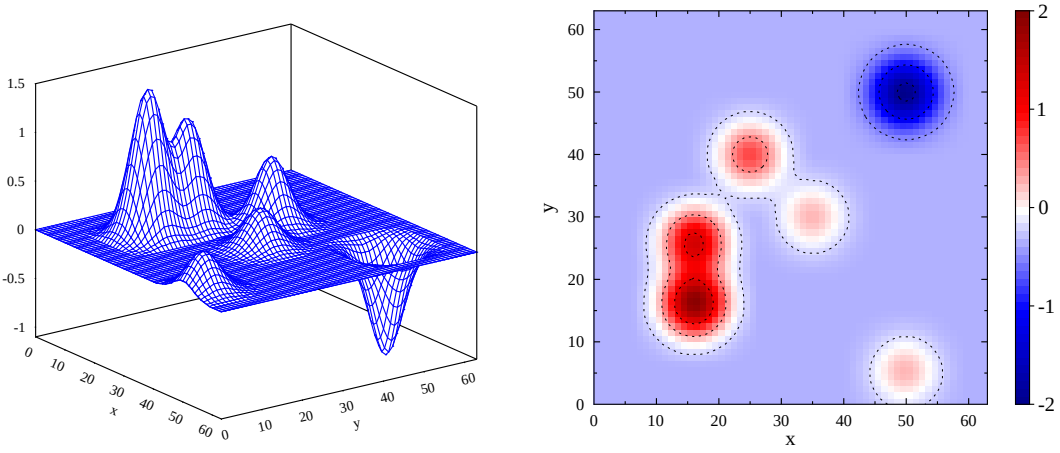
Listing 4.12: Cg shader used to perform convolution.

```
void convolve(
        in float2 coord : TEXCOORD0,
        uniform int d_cSize,
        uniform samplerRECT d_cFunc : TEXUNIT0,
        uniform int d_size,
        uniform samplerRECT d_data : TEXUNIT1,
        out float4 output : COLOR)
{
    // Coordinate of the output pixel.
    const float x = coord.x - 0.5f;
    const float y = coord.y - 0.5f;

    // Initialise the output value.
    output = float4(0.0f, 0.0f, 0.0f, 0.0f);

    // Centre pixel value of the convolution kernel.
    const int cCentre = floor((float)d_cSize / 2.0f);

    // Evaluate the convolution.
    for (int iy = 0; iy < d_cSize; ++iy)
    {
        for (int ix = 0; ix < d_cSize; ++ix)
        {
            const int datax = x - ix + cCentre;
            const int datay = y - iy + cCentre;
            const float data = texRECT(d_data, float2(datax, datay)).r;
            const float4 conv = texRECT(d_cFunc, float2(ix, iy));
            output.r += data * conv.r;
            output.g += data * conv.g;
            output.b += data * conv.b;
            output.a += data * conv.a;
        }
    }
}
```

`d_cFunc` and `d_data`. As with the previous Cg program this convolution function evaluates four convolutions for each texel simultaneously, making use of the four independent colour channels of the texture render target. In this code this is simply evaluating the convolution of the same data with the kernels of different width however in general could be used to evaluate four independent convolutions at the same time, which could for example be used in gridding four planes of uv data at once in the w-projection algorithm, described in section 2.3.

4.2.3.3 Running the code

Figure 4.7 shows the result of running this code with data corresponding to the six points listed in table 4.1. As can be clearly seen in this figure, the code executes successfully. While I designed this code as a simple experiment to illustrate the steps required for programming Cg shaders for GPGPU, rather than for generating any results which can be compared with CPU imaging code, I believe it clearly both shows the capability which was offered by early GPGPU as well as some of the complexity required to program such a simple algorithm.



(a) (b)

Figure 4.7: The result of convolution on the GPU.

i	x	y	z
1	16	16	1.5
2	16	25	1.1
3	25	40	0.7
5	35	30	0.4
4	50	5	0.4
6	50	50	-1.1

Table 4.1: Table of input values for the convolution.

4.2.4 Towards a unified shader architecture

While a graphics pipeline with the heterogeneous shader architecture described in figure 4.5 offered a huge improvement over the fixed function pipeline of the first 3-dimensional capable GPUs, there was still an ever growing demand for greater flexibility in the graphics pipeline. A fixed number of vertex processors, for example, was severely limiting for applications with scenes requiring complex geometry with high numbers of vertices and vertex transforms. As direct result of this, the two major GPU vendors at the time, NVIDIA and ATI, both released (NVIDIA at the end of 2006 and ATI in mid 2007) a new architecture with more general unified shader architecture processors, that could be repurposed as needed. It is this technology which is found in GPUs today. This architecture change pretty much immediately revolutionised the programming of GPGPU, making many of the old GPGPU coding methods out of date and inefficient.

This unified shader architecture combined the fragment and vertex shaders into a single, flexible processing core capable of both vertex and fragment shader operations, and more importantly for GPGPU turned GPUs into in effect an array of simple yet fast general purpose processors. Also, since the data flow no longer followed a fixed sequential pipeline many of the potential bottlenecks found in GPGPU algorithms were removed.

The remainder of this chapter describes NVIDIA's implementation of the unified shader architecture, and a programming system they released in early 2007 to exploit

its potential for GPGPU, called CUDA.

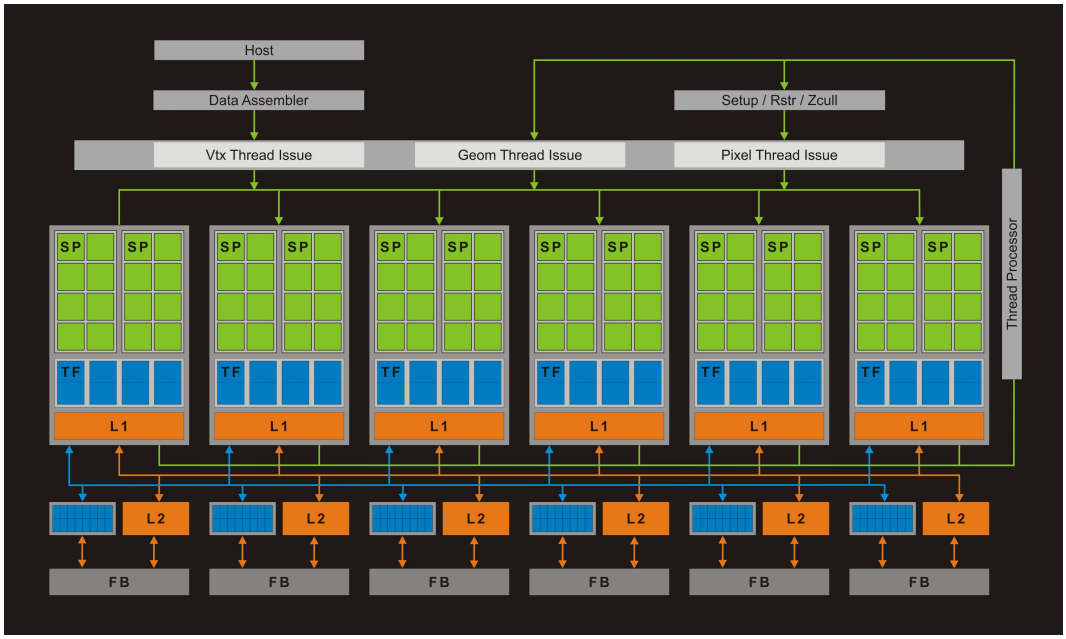


Figure 4.8: Architecture of the NVIDIA G80 processor; first generation unified shader architecture (NVIDIA 2006). Processing units are comprised of a number of stream processors (SPs) with associated Texture Filtering (TF), Texture Addressing and cache units (L1/L2).

4.3 CUDA

This section introduces the Compute Unified Device Architecture (CUDA) programming system which I have used extensively for the wide-field imaging code described in chapters 5 and 6. CUDA, released by NVIDIA in 2007, was arguably the first fully featured GPGPU programming system, giving high level access to the GPU without focusing on graphical applications. CUDA consists of both a hardware and software model and this section introduces each of these aspects and describes how their understanding is used to write efficient GPU code. CUDA is currently at version 3.2 and much of the information in this section is based largely on that provided by the CUDA programming manual and CUDA C best practices guide (NVIDIA Corporation 2010).

4.3.1 CUDA Hardware model

CUDA makes use of the unified shader architecture which was first introduced at the end of 2006 on NVIDIA's G80 processor and replaced the discrete shader architecture of previous generations of GPU, described in section 4.2. GPUs adhering to this architecture are an extreme form of superscalar design implemented as an array of multi-threaded streaming multiprocessors (SMs). Multiprocessors are designed to execute hundreds of concurrent threads using an architecture known as SIMT (Single-Instruction, Multiple-thread) which employs hardware multi-threading. As a result huge numbers of threads execute one common set of instructions, to achieve large amounts of parallelisation. Each multiproceessor is constructed from a number (typically eight) of very simple single instruction multiple-data (SIMD) arithmetic logic units, known as stream processors, such that both processing and associated memory can be deployed in a scalable fashion as shown in figure 4.9. The processing and memory architecture follow a hierarchical layout where each of the stream processors has a number of 32-bit registers, each multiprocessor

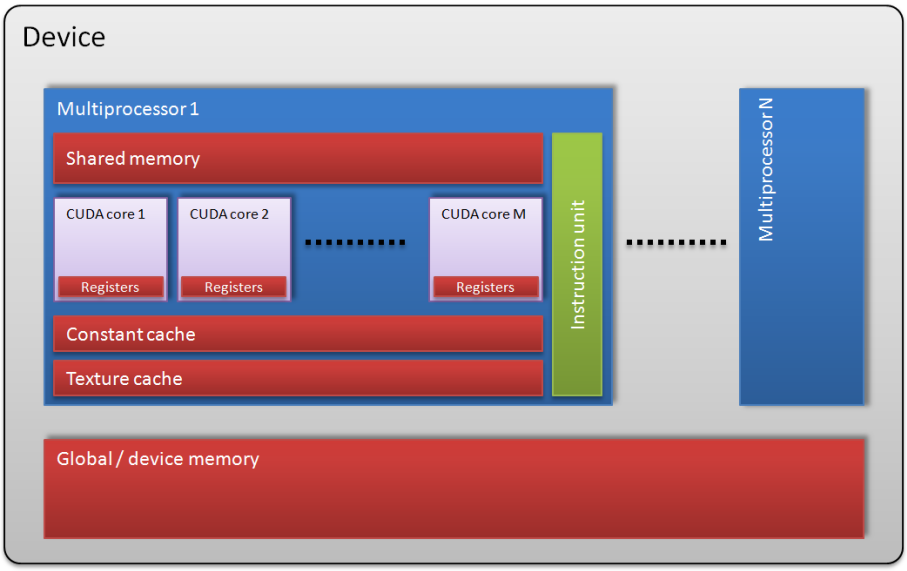


Figure 4.9: Schematic showing the architecture of the CUDA hardware model. Stream processors are grouped together into multiprocessors. Processors have access to a hierarchy of memory spaces.

has access to an amount of shared memory (accessible to the stream processors which belong to it), and there is global access to device memory at all levels. In addition, as shown in the figure 4.9 there is an amount of memory known as texture cache and constant cache which is available at the multiprocessor level.

A interesting feature of this model, is that while the numbers of multiprocessors (and stream processors per multiprocessor) can vary between different models or generations of GPU, the high degree of regularity allows code to be written which is both portable and scales well to the architecture. This is shown in figure 4.10 where two different GPUs, one with two multiprocessor cores and one with four, can both automatically be parallelised with predicable scaling.

4.3.2 CUDA Software model

The software model of CUDA effectively treats the GPU as a data-parallel coprocessor to the CPU. In the CUDA context, the GPU is therefore known as the device and the CPU is known as the host. Functions executed on the device are called kernels, and Kernels are

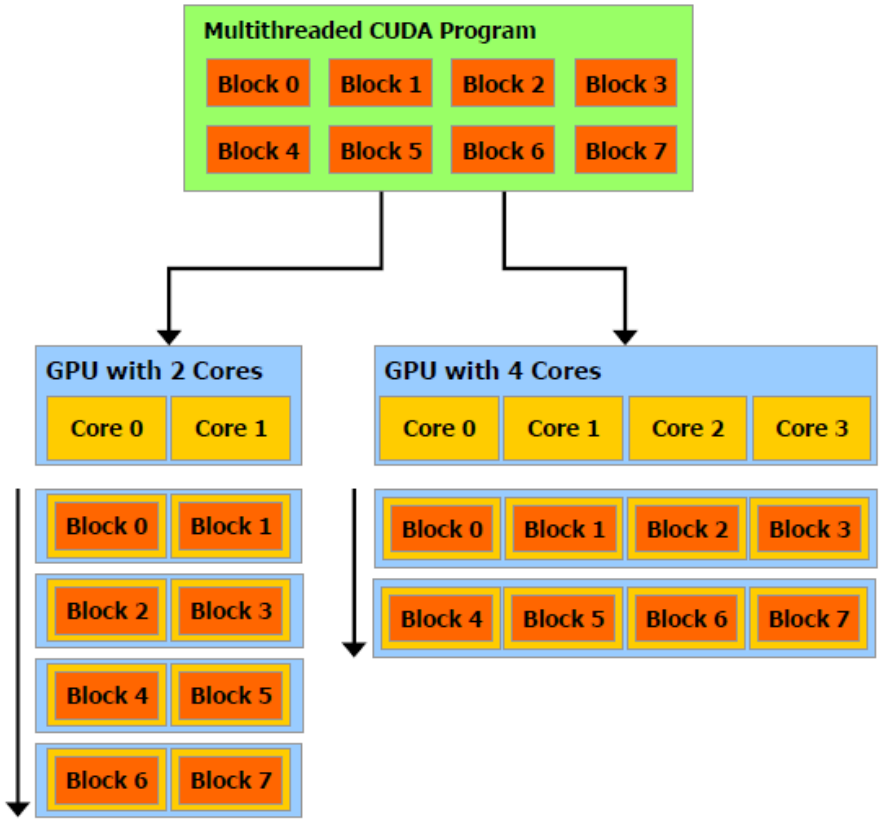


Figure 4.10: A multi-threaded program is partitioned into blocks of threads that execute independently from each other, so that GPUs with more cores will automatically execute the program in less time than a GPU with fewer cores (NVIDIA Corporation (2010)).

executed in the single program multiple thread-model, where a user-configured number of threads execute the same program at the same time.

Threads executing the kernel are batched together in chunks (of ≤ 512 threads) called thread blocks and threads within a block can cooperate to a certain degree by using the memory shared by all threads in the block and by synchronisation with a barrier-like construct called `__synchthreads()`.

The different thread blocks are organised in a grid. A grid can consist of up to 2^{32} blocks, resulting in a total of up to 2^{41} threads! Unlike threads in a block, different blocks in the grid execute completely independently and can't be synchronised. If synchronisation is required by all threads, the work has to be split into separate kernels, since kernels are not executed in parallel. Within a kernel program, the large numbers of threads are

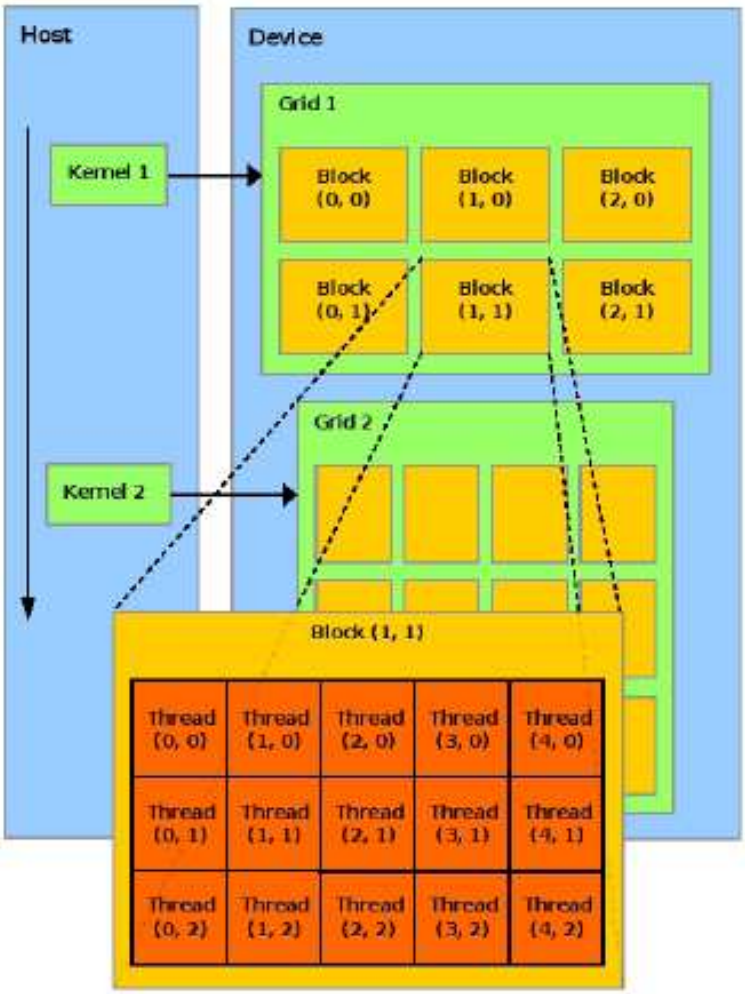


Figure 4.11: The CUDA execution model. A number of threads are grouped into blocks, which themselves are grouped into a grid. Calls to successive GPU functions, called kernels are executed in serial (NVIDIA Corporation (2010)).

handled by manipulating their indices. Threads in a block have an index addressed in 1-, 2-, or 3- dimensions and thread blocks in the grid are addressed using a 1-, or 2- dimensional index. Indices in any dimension have a maximum size set by the hardware so that when a kernel requires more than 2^{16} thread blocks, 2-dimensional block indices have to be used. For example, if an algorithm performs processing on a 8192 pixel by 8192 pixel image by dividing the image into sub-image blocks of 16 by 16 pixels, this would require a total of 2^{18} blocks and therefore single dimensional block indexing cannot be used. In this scheme each thread has a unique identifier available to the kernel program according to its block and local thread index. The choice of grid and block dimensions is made at

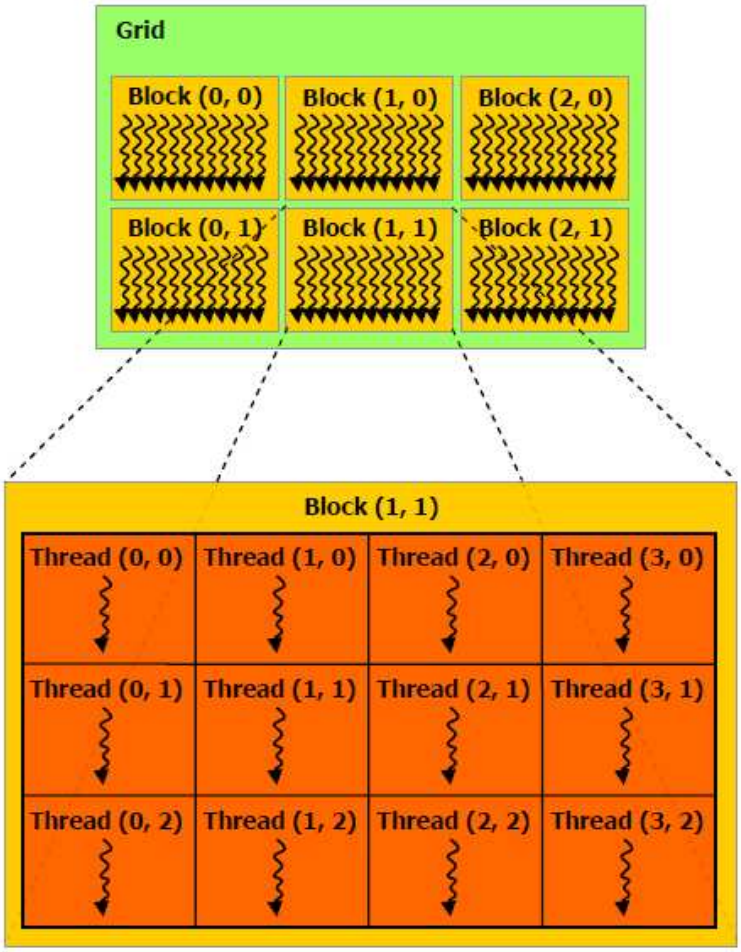


Figure 4.12: Schematic showing the organisation of threads in the CUDA programming model. Threads are arranged into blocks and blocks of threads are arranged on a grid (NVIDIA Corporation (2010)).

run-time, usually in a way to simplify the mapping of data onto threads, and does not have to be the same for successive kernel executions. These concepts are illustrated in figures 4.11 and 4.12.

A grid of thread blocks is executed on the device by scheduling thread blocks onto multiprocessors. Each thread block is assigned to one multiprocessor, and will execute there until the kernel function returns. Multiple blocks of the grid can be mapped onto the same multiprocessor and in this case resources such as registers and shared memory have to be shared between the thread blocks. This limits the number of blocks that can be mapped onto the same multiprocessor and by limiting the concurrent execution

occupancy of threads can have a bearing on performance.

Thread blocks are further split automatically by the device into SIMT groups called warps for the purposes of scheduling execution. While threads in a block are executed logically in parallel, threads in a warp are executed physically in parallel. Each warp contains the same number of threads, called the warp size and this is defined by the hardware. Warps are executed by scheduling threads on the stream processors within a multiprocessor. On the G200 GPU (NVIDIA Corporation 2008), which I used during this thesis, the warp size is 32 and there are 8 streaming processors per multiprocessor so a warp requires at least 4 cycles to execute its instructions.

The memory spaces that can be accessed by threads is linked strictly to the hardware memory hierarchy described in the previous section. As shown in figure 4.13, each thread has its own local memory, threads within a block can access shared memory, and all threads in the grid have access to device global memory. While per-thread local and per-thread block shared memories are only accessible for the lifetime of a kernel function call, global memory is persistent between multiple kernels. In addition, threads have access to read only, constant and texture memories. The address space of shared and global memory is linked directly to the hardware but local thread memory is implemented by using both registers and device memory automatically allocated by the compiler. As the amount of registers is limited and there is a huge performance penalty in using device memory, care has to be taken in the amount of local memory used by any given thread.

Another important feature of CUDA programming is that kernel function calls, unlike their CPU counterparts, are non-blocking. This means that functions running on the GPU will not be synchronised with instructions executing on the CPU unless explicitly specified by the calling code. Synchronisation is done implicitly however, whenever data is read from or written to global device memory and therefore device memory can only be accessed if no kernel is active.

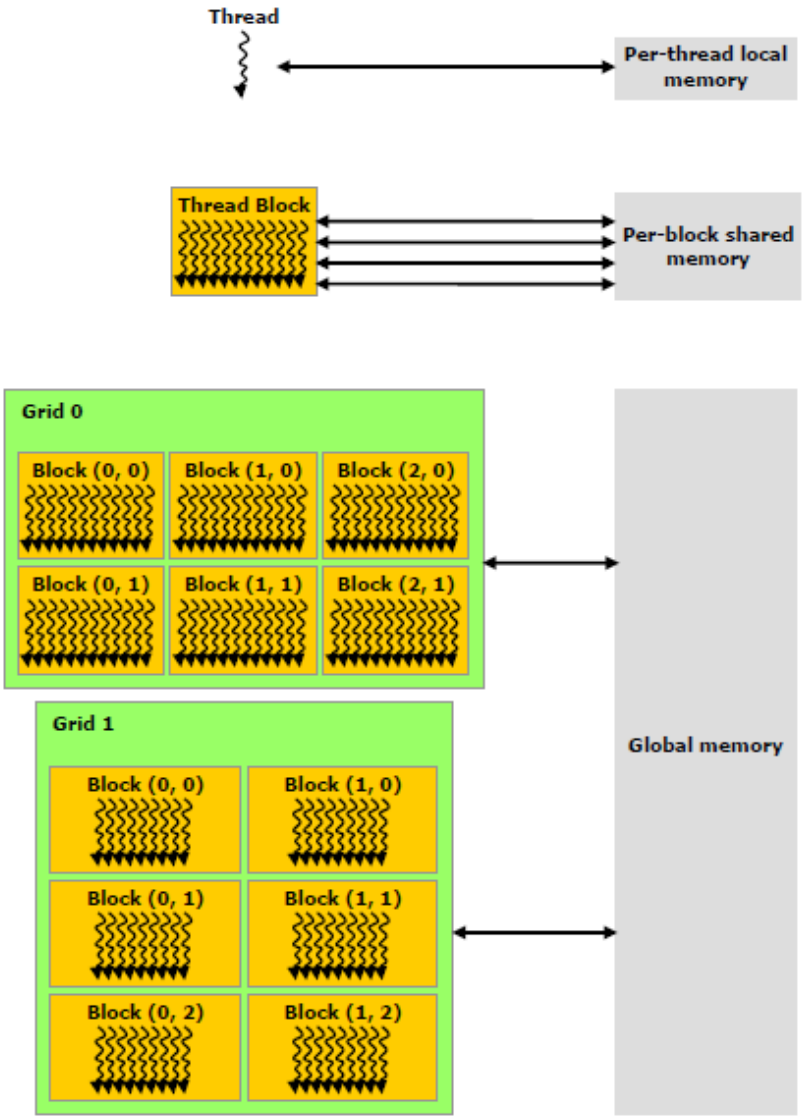


Figure 4.13: Schematic showing the memory hierarchy accessible by CUDA threads. Each thread has its own local memory, thread blocks have access to shared memory, and all threads have access to persistent global memory (NVIDIA Corporation (2010)).

4.3.3 Programming with CUDA

This section gives a brief summary of programming with CUDA. A far more detailed description of the language constructs and application programming interface can be found in the CUDA C programming guide and CUDA reference manual, NVIDIA Corporation (2010).

The CUDA API is based on C, with a minimal set of language extensions and a run-time library. The CUDA run-time library consists of a CPU component, which is

responsible for controlling access to device memory and for instantiating the execution of kernels on the GPU, and a device component which schedules threads on the GPU. The runtime is started automatically by the first CUDA (runtime) function called from the host code and once it has been initialised the CUDA context is locked to this calling thread.

Differences from standard C include a number of special type qualifiers.

- The `__global__` qualifier defines a function that is callable by the host code and will be executed on the device i.e. kernel functions.
- The `__device__` qualifier defines a function that is executed on the device and only callable from the device. These functions are inlined into the calling code by the compiler. This qualifier is also used to specify variables residing in device memory. If no additional type qualifier is specified, variables are stored in global memory and have a lifetime equal to that of the application.
- The `__shared__` qualifier defines a variable belonging to per-thread block shared memory. These variables are accessible only by threads within the block from which they are initialised, and the memory is inaccessible once the thread block terminates.

In addition to the extended qualifiers, CUDA introduces a special syntax for launching kernel functions. In order to launch a kernel function it is necessary to specify size of the grid of thread blocks blocks and the number of threads per block with which the kernel should be run. This information is encapsulated in a special calling syntax and passed to the runtime system. The code shown in Listing 4.13, for example, launches a kernel executing on 500 thread blocks, with each block containing 256 threads.

Device code for writing kernel functions also extends C with a number of special built-in variables.

- `dim3 threadIdx`, the thread index within the local thread block.

Listing 4.13: The syntax for calling a CUDA kernel.

```
dim3 num_thread_blocks(500, 1, 1);
dim3 num_threads_per_block(256, 1, 1);
KernelFunction <<< num_thread_blocks, num_threads_per_block >>> (...);
```

- `dim3 blockIdx`, the block index within the grid.
- `dim3 blockDim`, the dimensions of a thread block.
- `dim3 gridDim`, the dimensions of the thread block grid.

where `dim3` is a structure representing a 3-dimensional index which is stored in the fields `x`, `y` and `z`. These variables, defined when the kernel is launched, are used in kernel functions to dereference the thread index. This index is then typically used to control which data elements should be accessed and what processing should be done.

Finally CUDA device code also contains a number of intrinsic functions which expose specific GPU functionality. Perhaps the most commonly used of these is `__syncthreads()` which provides a barrier synchronisation of all threads in the thread block and is used extensively when a number of threads in a thread block cooperate via shared memory. Another important class of intrinsic functions are fast math calls (e.g. `__sincosf()` and `__expf()`) which are faster but less accurate versions of their standard counterparts.

4.3.3.1 Writing simple CUDA functions

Writing a very simple CUDA function typically involves the steps listed below:

- **Setting up memory on the device.** Allocating linear device memory in CUDA takes a form very similar to the classical C approach. The function `cudaMalloc(void** device_ptr, size_t count)` is called from the host code to allocate `count` bytes of memory. The address of the memory allocated returned in the variable `*device_ptr` which is a pointer to the 32-bit address

space on the device and as such dereferencing the pointer on the host is undefined. Once allocated, memory can be initialised by the function `cudaMemset(void* device_ptr, int value, size_t count)`. In case of failure both of these functions return error codes.

- **Copying data and input values to the device memory.** Transferring data between the host and device is usually performed by calling `cudaMemcpy(void* dest, const void* source, size_t count, enum cudaMemcpyKind kind)`. This function copies `count` bytes of data between the host and device memory at the addresses specified by the pointers `dest` and `source`. The direction of the copy is defined by the value of the enumerator `kind`, and to copy data to the device the value `cudaMemcpyHostToDevice` is specified. Unlike many other CUDA calls, the `cudaMemcpy()` function blocks the calling thread and if a kernel is currently executing on the device it will wait until the kernel has finished before starting the transfer to avoid undesirable effects.

- **Launching a kernel to process data on the device.** In order to perform processing on the GPU, a kernel function must be launched. Kernel functions are defined by the `__global__` type qualifier and are called from the host code with the special calling syntax introduced in the previous section. Kernel code is similar to a C function however as the same code is executed by each thread, parallelism is achieved by using the thread index, obtained from the built-in index variables, to allocate unique work to each thread. Listing 4.14 shows an extremely simple example kernel which squares elements of the array in device memory passed to it. While in this example a very simple 1-dimensional thread index is used to assign one element of the array to each thread far more complicated access patterns can be used. For very complex operations it is also often advisable to split the algorithm

into a number of kernel calls which can be executed one after another.

Listing 4.14: Example kernel which take the square of all element of the supplied array.

```
__global__ void square_array(unsigned num_values, float * array)
{
  // Obtain the thread index.
  int tid = blockIdx.x * blockDim.x + threadIdx.x;

  // If the thread index corresponds to a value in the array,
  // take the square of the value.
  if (tid < num_values)
      array[tid] *= array[tid];
}
```

- **Copying the results of processing back to the host memory.** Once all processing on the GPU is complete device memory is copied back to the host using `cudaMemcpy()` but with the value of the direction enumerator set to `cudaMemcpyDeviceToHost`. The host memory to which the data is coppied must be allocated prior to this call.

- **Cleaning up** Finally device memory must be freed to release it for subsequent use and to prevent memory leaks. This is achieved by the `cudaFree(void * device_ptr)` function.

Listing 4.15 is an example of some host code which calls the kernel in Listing 4.14. In this simple example an array of length `num_values` stored in host memory at the address `h_array` is processed in parallel by blocks of threads of size `block_size`.

4.3.3.2 Compiling CUDA code

CUDA code is compiled with a special compiler, called *nvcc* (NVIDIA Corporation 2010). The *nvcc* compiler uses the special type qualifiers placed in the source code to distinguish between instructions that need to be compiled for each target architecture. This splitting of the compilation of device and host codes is achieved by making use of the native C or C++ compiler, to generate CPU assembly instructions, and the NVIDIA compiler to pro-

Listing 4.15: Example host code for calling the kernel defined in Listing 4.14

```
// Allocate device memory to hold the array.
float * d_array = 0;
size_t num_bytes = num_values * sizeof(float);
cudaMalloc((void**)&d_array, num_bytes);

// Copy the array from the host to device.
cudaMemcpy(d_array, h_array, num_bytes, cudaMemcpyHostToDevice);

// Execute the kernel.
int num_blocks = num_values / block_size;
square_array <<< num_blocks, block_size >>> (num_values, d_array);

// Copy the array from the device back to the host.
cudaMemcpy(h_array, d_array, num_bytes, cudaMemcpyDeviceToHost);

// Free device and host memory.
cudaFree(d_array);
```

duce parallel thread execution (PTX) assembly code suitable for execution on the GPU. The *nvcc* compiler also automatically handles linking of both CPU and GPU libraries with CUDA compiled objects. On Linux this *nvcc* is therefore an extension of the native system GNU compilers (gcc and g++) capable of compiling source code which contains both device and host instructions.

The basic process of writing CUDA code is therefore very similar to that of writing C code on the CPU. Function prototypes (which store the function signature, and are used by the compiler for compile-time consistency checking) should be placed in header files and the CUDA source code is placed in files with a '.cu' extension, compiled with *nvcc*.

4.3.3.3 Performance considerations

Performance on the GPU is largely governed by a careful balance between processor operations and memory accesses. A number of common instructions and their respective costs in terms of processor clock cycles are listed in table 4.2. Looking at this table, it is immediately striking that: synchronisation of all threads in a thread block has almost the same cost as an addition; accessing shared memory or registers comes at almost no cost; and reading from device memory costs an order of magnitude greater than any other

Instruction	Cost (clock cycles per warp)
FADD, FMULT, FMAD, IADD	4
bitwise operations, compare, min, max	4
reciprocal, reciprocal square root	16
access to registers	0
access to shared memory	≥ 4
reading from device memory	400-600
synchronising all threads in a block	4 + waiting time to read barrier.

Table 4.2: Instruction costs for the G200 architecture (NVIDIA Corporation 2008).

instruction.

While the time taken to access a location in global memory is very slow, it has an extremely high potential bandwidth (as shown in in Table 4.3). In order to make use of this to its full potential, memory access made from groups of 16 threads in a half-warp must be coalesced (i.e. fused together). If this is the case the hardware can then fetch (or store) the data in the fewest number of transactions. Data fetches to global memory occur as a single 32 byte, 64 byte or 128 byte aligned transaction, and if the memory cannot be coalesced, then separate memory transactions have to be issued for each thread in the half-warp which is a very costly operation! The rules required to satisfy coalesced transfers depend on the CUDA device architecture, however for the 1.3 compute architecture, which I used when writing this thesis, transfers of 32-bit floating point numbers are coalesced when any accesses made by the half-warp fits into a segment of 32-, 64- or 128-bytes. Meeting this criteria is therefore extremely important when writing algorithms on the GPU.

Reading data from global memory should however in all cases be minimised as much as possible. One method of doing this is by constructing algorithms in such a way that multiple threads in a thread block share a common set of memory addresses. When this is the case it is possible to make use of the fast shared memory as a software managed cache.

As seen in Table 4.2, shared memory is an extremely fast memory space, however in order to achieve optimal access times some care has to be taken. Shared memory is organised into 16, 32 bit wide, banks with successive 32 bit words belonging to different banks. Maximum performance is only achieved when groups of 16 threads (known as a half-warp) each access different banks of the memory. If multiple threads in the group attempt to access locations in the same bank, serialisation occurs and performance is therefore reduced. This is illustrated in Figure 4.14 where (a) and (b) have no bank conflicts and (c) and (d) have two and eight way conflicts respectively. The only exception is when a number of threads access exactly the same memory address in a single fetch operation, and in this case a broadcast occurs.

The hardware thread scheduler will attempt to hide as much of the time taken for memory transfers as possible, by switching out threads that are waiting for memory in favour of those ready to proceed. While this is of great help when writing GPU code, the costs of even well optimised memory transfers can not be completely hidden in most cases. The programmer can however help influence the scheduler in this regard by ensuring that there are enough thread blocks assigned to each multiprocessor. This is achieved by arranging that the number of resources (registers and shared memory) used by each thread block doesn't exceed more than half the total resources available to a multiprocessor, and by writing algorithms in such a way that they generate a large number of thread blocks. A measure of the success in hiding memory access by swapping out threads is commonly referred to as the occupancy.

As explained in Section 4.3, threads are executed on the hardware as single instruction multiple thread (SIMT) groups called warps and all threads in a warp are required to execute the same instruction. This is problematic when considering any type of control flow instructions (if, switch, for, do, while), since it requires uniform control paths across all threads in a warp. CUDA avoids this problem by serialising any divergent execution

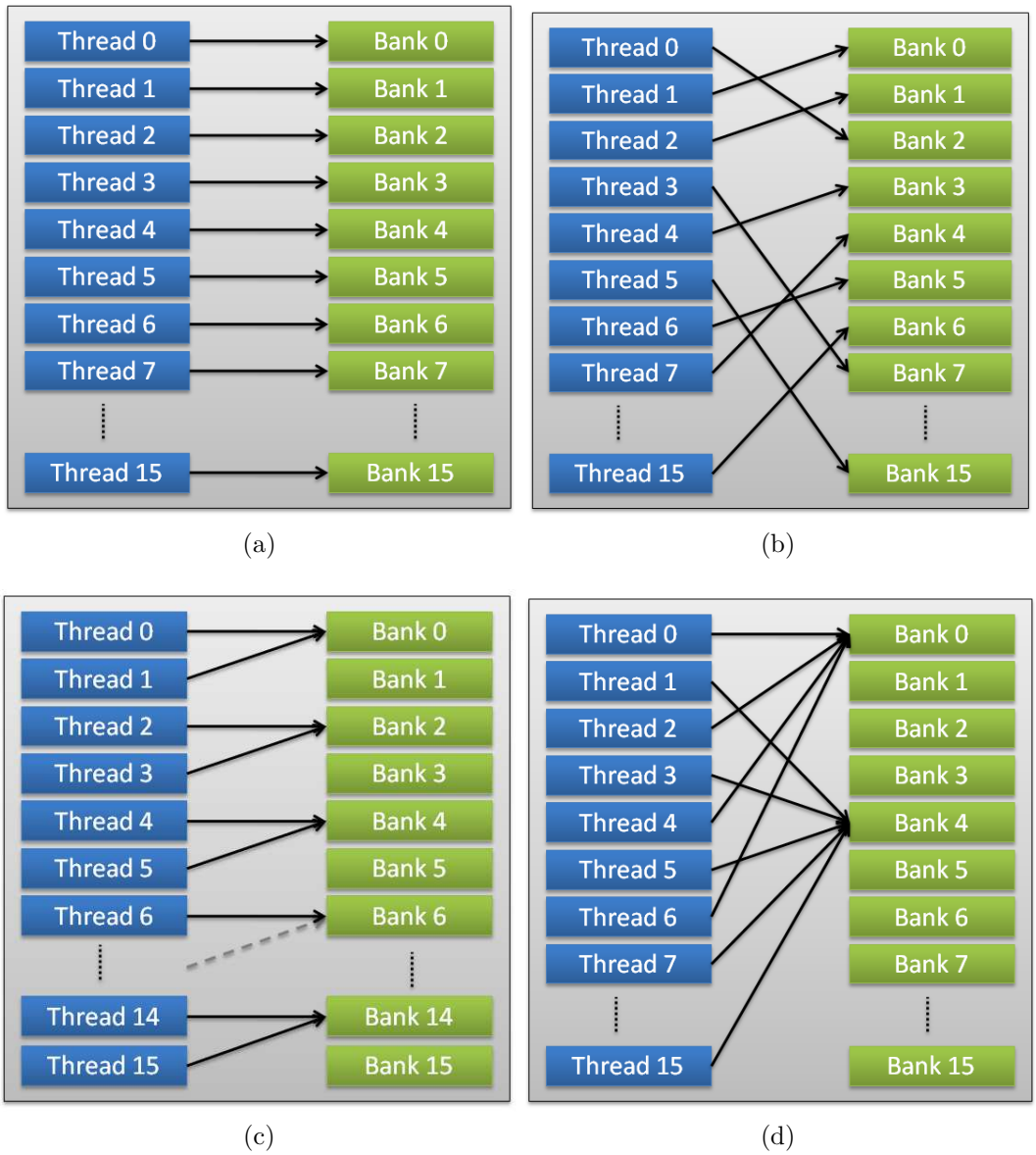


Figure 4.14: Illustration of CUDA shared memory bank-conflicts. Conflicts occur when multiple threads in the same half-warp (group of 16 threads) attempt to access memory from the same memory banks. (a) & (b) No bank conflicts (c) 2-way bank conflict (d) 8-way bank conflict.

paths within the warp. The different execution paths are therefore run one after another thus increasing the number of instructions done by the warp potentially greatly reducing performance. The code serialisation is performed by the hardware in an unspecified way, which while giving the correct result, means that performance is almost impossible to predict under such a regime. When a warp does not diverge, only the control flow instruction itself is executed and there is no performance loss.

4.3.4 Measuring performance of GPU code

Key to optimising code is measuring its performance. This has to be done accurately and take into account of the various issues that can affect performance, where for example on the GPU memory bandwidth plays a key role.

4.3.4.1 Timing

CUDA calls can be timed by either CPU or GPU timers. Any sufficiently high resolution CPU timer can be used to measure the elapsed time of a CUDA function call. The code which I have developed for this thesis described in Chapter 5 uses this approach by making use of the timer provided by the Qt library. This is a high-resolution (millisecond accuracy) timer with a simple interface lightweight interface.

Care has to be taking however when using CPU timers to measure many of the CUDA API calls as they are asynchronous (including kernel functions) and return immediately to their calling thread. Therefore in order to accurately measure the elapsed time it is necessary to synchronise the CPU thread with the GPU by calling `cudaThreadSynchronize()` before starting and stopping the timer to block the CPU thread until previous CUDA calls are complete.

Listing 4.16: Timing code using CUDA events.

```
cudaEvent_t start, stop;
float time;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, 0);
kernel<<<grid, threads>>> (...)
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

The CUDA event API also provides an interface to record events with timestamps

and use these to calculate timing. An example of this is given in Listing 4.16 where `cudaEventRecord()` is used to place the start and stop events into the specified stream so that the device will record timestamps for these events. These events can then be queried by the `cudaEventElapsedTime()` function which returns a value in milliseconds with a resolution of around 0.5 microseconds. Measuring timings in this way also has the distinct advantage that it is using the GPU clock which is is independent of the operating system.

4.3.4.2 Bandwidth

The bandwidth of data transfers is a very important factor in bottlenecking performance, especially in GPU code. As already noted, optimisation of GPU code should always be made in the context of memory access which serves to reduce the data bandwidth. It is therefore often useful to calculate theoretical and effective bandwidth of GPU code.

Theoretical bandwidth is calculated from the hardware specification. For example the NVIDIA GTX 275 used for most of the development of code for this thesis uses double data rate memory with a clock rate of 1.134GHz and a 448-bit wide memory interface. Using this the theoretical peak bandwidth is therefore:

$$(\,1134 \cdot 10^6 \cdot (448/8) \cdot 2\,) \,/\, 1024^3 = 118.3 \text{ GB/sec.} \tag{4.3}$$

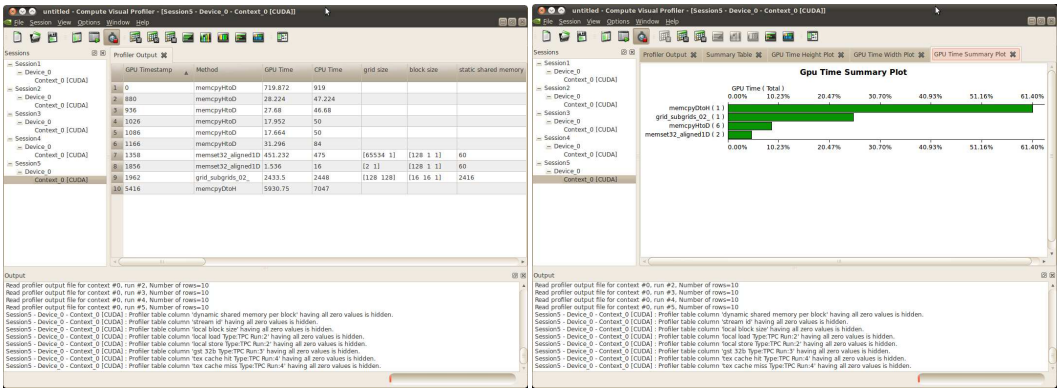
Effective bandwidth is calculated by timing specific parts of the code and then knowing how data is accessed by the program and using the following equation:

$$\text{Effective bandwidth} = \frac{[(B_r + B_w) \,/\, 1024^3]}{\text{time}}\,, \tag{4.4}$$

where B_r is the number of bytes read and B_w is the number of bytes written per kernel and time is in seconds such that the effective bandwidth is in units of GB/s.

4.3.4.3 The CUDA Compute Visual Profiler

Since mid-2010, the development version of CUDA also includes the Compute Visual Profiler (NVIDIA Corporation (2010)). The Compute Visual Profiler (shown in Figure 4.15) is a fantastic graphical user interface based profiling tool which can be used to measure performance and find potential opportunities for optimization. Running the profiler results in a set of metrics in the form of plots and tables which are obtained by recording the GPU hardware event counters during execution of the code.



(a) (b)

Figure 4.15: Screenshots of the NVIDIA Compute Visual Profiler.

Counter values obtained by the visual profiler are therefore not the same as those that can be obtained from inspecting the code and provide an additional tool towards understanding how to best optimise. Counters are obtained for events within a warp rather than at the individual thread level and are recoreded from one of the multiprocessors. As a result, profile data tends to be slightly inconsistent between kernel invocations unless all multiprocessors perform exactly the same amount of work in each execution. The profiler executes the GPU code a number of times successively as not all counters can be collected concurrently therefore choosing a execution grid that gives consistent results can also be important when profiling.

A small subset of the counters that the profiler reports and which have been used in

optimising the code in chapter 6 are summarised below:

- The number of branches taken by threads executing a kernel. The counter is incremented by one if at least one thread in the warp takes the branch.
- The number of divergent branches in a warp. This is incremented if at least one thread in the wrap follows a different execution path via a data dependent conditional branch.
- The number of instructions executed on the multiprocessor.
- The number of bank conflicts caused by two addresses of a memory request to shared or constant memory falling in the same memory bank resulting in the access having to be serialised.
- The number of memory load and store requests to global memory.
- The number of coalesced memory loads and stores to global memory.

In addition to these counters the profiler has the capability of calculating a number of derived statistics such as the global memory and instruction throughput of a kernel as well as a measure of the average number of warps that were active on a multiprocessor on a given cycle. Combined with knowledge of the CUDA hardware and software models these results are therefore extremely valuable in understanding how a particular GPU code might be optimised.

4.4 Other GPGPU programming languages

While the code I have developed for this thesis has been written exclusively using CUDA and a combination of OpenGL and Cg, in this section present I a brief summary of the other main GPGPU programming languages. It should be noted that I have no direct

experience of programming in any of these languages, and therefore the information I present is taken from respective literature.

- BrookGPU (Buck et al. (2004), Brook (2008)) is a compiler and runtime implementation of the Brook stream programming language developed at Stanford University. It is implemented as an extension to ANSI C which compiles to an OpenGL, DirectX or ATI Stream backend. Brook was first released in 2004 under a BSD licence and has had various developments and branches since then with AMD/ATI's brook+, an extension of BrookGPU to support integer and double precision on AMD GPUs being the most prominent.

- Sh (McCool et al. (2002), McCool et al. (2004), http://libsh.org) is a metaprogramming language, built on top of C++, providing an extremely high-level interface to programming the GPU. While Sh is no longer actively supported by its developers it has become commercialised as part of RapidMind's Multi-core development platform now owned by Intel. However, an archived version of Sh remains readily available, under the GNU Lesser public licence.

- The AMD Accelerated Parallel Processing (APP) framework (AMD), formerly known as ATIs stream computing SDK and Close To Metal (CTM), like CUDA provides set of hardware and software technologies for GPGPU programming but targeted at AMD GPUs. While earlier versions of this API were based on an optimised version of the brook GPU language, APP is now essentially AMDs implementation of the Open Computing Language (OpenCL).

- DirectCompute, which is part of DirectX 11 (Microsoft), is an API for GPGPU on Windows Vista and Windows 7. It shares a range of interfaces with both CUDA and OpenCL and exposes GPGPU features of DirectX 10 and DirecX 11 capable hardware.

- Open Computing Language (OpenCL (2009)) was initially developed around 2008 by Apple Inc. in collaboration with AMD, IBM, Intel and NVIDIA. OpenCL is a framework standard for writing applications that execute across heterogeneous platforms of GPUs, CPUs and other multi-threaded processors. OpenCL shares interfaces with both CUDA and DirectCompute and AMDs Stream SDK to give applications access to GPGPU. OpenGL is analogous to the industry standards of OpenGL, and OpenMP and is maintained by a non-profit consortium.

4.5 Why a GPU is no CPU

GPUs are designed to execute large numbers of data parallel operations. Programs on NVIDIA GPUs requiring tens of thousands of concurrent threads to achieve maximum performance. Unfortunately massive parallelism comes at a cost. The lack of branch prediction, tiny amounts of fast memory which can be used as cache, limited inter thread communication and the requirement that algorithms map to such a high degree of parallelism means that obtaining good performance is often difficult. In addition to this, GPUs impose a number of limitations to how kernels functions can be written. For example, function calls from device code are handled by the compiler by inlining which prevents the use of recursion and therefore many algorithms, such as reduction of large arrays, have to resort to multiple kernel invocations. GPUs are not only designed for high amounts of data parallel work but also optimised for high arithmetic intensity as device memory accesses are far more expensive than most numerical operations. This situation, which is very different from writing CPU code, requires algorithms written for the GPU to have:

- High arithmetic intensity, the ratio of numerical operations to memory accesses.
- High degree of parallelism so that many threads can be assigned with approximately equal amounts of work.

- Predictable memory access patterns so that complex memory hierarchy can be used as efficiently as possible.

- Minimal use of divergent branching as this results in serialising threads within a warp with different execution paths.

Another consideration when writing GPU code is that double precision floating point operations are still a relatively new feature. This is because for graphics applications single precision values (32 bits of information) are sufficient to generate 16.7 million colours, more than can be perceived by the human eye. Double precision was introduced with the GT200 series chip (CUDA architecture version 1.3, NVIDIA Corporation (2008)) and until the very latest 'Fermi' processors (CUDA architecture version 2.0, NVIDIA Corporation 2009) GPUs have been heavily optimised with for single precision operations. To avoid this severe performance penalty (which is as much as a factor of eight) when writing code targeted at a number of architectures, it is therefore highly necessary to use single precision values and pay special attention to the numerical effects that can result due to lower precision rounding.

4.5.1 A comparison of some typical hardware

Tables 4.3, 4.4 and 4.5 present a comparison of the GPU and CPU hardware which I have used during this thesis. In this section I expand on these differences from the point of view of memory bandwidth, floating point performance, and power consumption.

4.5.1.1 Memory performance

The high memory bandwidth for GPU device memory, shown in table 4.3, is achieved as a result of the GPU using multiple memory banks giving a wide memory interface. GPU memory is based on double date (DDR) rate memory which performs two, 64 bit data

Parameter	Value
Transistors	$1.4{\cdot}10^9$
CUDA cores	240
Processor clock	1404 MHz
Device memory (GDDR3)	896 MB
Memory clock	1134 MHz
Memory interface width	448-bit
Memory bandwidth	118.3 GB/s
Global memory latency	350 - 650 ns
TDP	219 watts

Table 4.3: Hardware specification of an NVIDIA 275 GTX GPU. Values are taken or calculated from the NVIDIA product specification (NVIDIA Corporation (2008)).

Parameter	Value
Cores	4
Processor clock	2.66GHz
Memory IO clock	667 MHz
Memory bandwidth	19.8 GB/s
Memory latency	10.5 ns
TDP	95 watts

Table 4.4: Hardware Specification of an Intel Core i5-750 using dual channel DDR3-10600 memory. Values are taken or calculated from Intel product specification (Intel (2010)).

word transfers per channel and therefore with an interface wide enough to support seven channels very large data rates are achieved. In comparison the CPU system memory, shown in table 4.4, has only two channels and therefore its data rate is far lower. It should be noted however that the latency of the GPU is high compared to the CPU due in part to a lack of hardware managed cache.

Parameter	GPU	CPU	Improvement
Maximum theoretical performance (GFLOPS)	43	674	x 15
GFLOPS per watt	0.448	3.08	x 6.9
GFLOPS per	0.3	3.4	x 11.3

Table 4.5: Performance comparisons of the CPU and GPU described by Tables 4.3 and 4.4.

4.5.1.2 Floating-point performance

From theoretical performance comparisons the GPU is capable of around 15 times the performance of the CPU for single precision code. It should be noted however that this comparison only holds for highly optimised codes on both architectures. As it is very unusual to find code that performs to these levels, the degree in speedup between the two architectures will depend hugely on the implementations of both algorithms. For example, CPU code not making full use of streaming SIMD extensions (SSE) for packed floating point operations typically performs at around 1 to 3 GFLOPS on the Intel i5 (compared to the theoretical value of ∼40 GFLOPS) and GPU code which fails to use sufficient memory optimisations and has poor arithmetic intensity would perform equally badly despite promising a theoretical maximum of over 600 GFLOPS. This sort of variation makes any measure of relative speedup hard to quantify precisely. In practice however anything from around 10 to 100 times speedup of GPU code over moderately well written CPU code can be expected.

4.5.1.3 Performance per watt

A key consideration of processing hardware is its power consumption. This is true for both green credentials and the simple fact that the more energy efficient a processor is the larger the numbers that can be deployed in while remaining inside of heat and power budgets.

For an Intel Core i5-750 running 4 cores at 2.66GHz, with optimised code making full use of all four cores and streaming SIMD extension (SSE) optimisations the theoretical maximum performance would be

$$4 \cdot 2.66\,\text{GHz} \cdot 4 = 42.6\,\text{GFLOPS},$$

where SSE instructions allow each of the four cores to perform four fused multiply-add (FMAD) operations per cycle. As the Intel i5 processor has a thermal design power (TDP) (maximum operating power) of 95 watts, under conditions of extreme optimisation, it is capable of 448 MFLOPS per Watt.

By comparison, the GPU used in this thesis, an NVIDIA 275 GTX which is built around the GT200b processor and can perform two fused multiply-add (FMAD) per cycle on each of its 240 stream processors. The theoretical maximum single precision performance of this processor is therefore

$$1404\,\text{GHz} \cdot 2 \cdot 240 = 674\,\text{GFLOPS}.$$

It should be noted that the double precision performance of this GPU is a factor of eight lower than this value as it has one double precision processor for every 8 single precision units. With a thermal design power of 219 watts, the GPU is therefore capable of an impressive 3.08 GFLOPS per watt.

While these comparisons have been derived using theoretical fully optimised performance values, I would fully expect well written code to behave with a very similar trend and with a power per watt ratio over six in favour of the GPU this further demonstrates the great potential of GPUs to large scale numerical computing.

Chapter 5

WFIT: Interferometric imaging code

This Chapter describes the interferometric imaging code which I have developed for this thesis, called the Wide-Field Imaging Testbed (*WFIT*).

5.1 Introduction

Early during my thesis, I spent some time investigating the possibility of parallelising wide-field imaging algorithms in existing radio astronomy software packages. This effort was primarily focused on the faceting algorithm, described in Chapter 2, implemented as part of the leading software package of the time, namely *AIPS* (Fomalont (1981), Greisen et al. (1990) & Greisen et al. (2007)). This algorithm was chosen as it is computationally demanding and, as identified by Golap et al. (2001), has a large potential for parallelism. The faceting algorithm makes images of a large number of separate sub-fields, each of which are generated independently of one another before being combined to form a map of the sky. This algorithm is therefore well suited to being parallelised, and with no communication between tasks assigned to create sub-field images it is ideal for implementation using a message passing framework such as MPI (Snir et al. (1996), Gebriel et al. (2004) & Gropp et al (1996)).

After spending some time learning MPI, I worked closely with Dr Eric Greisen, one of the leading developers of *AIPS*, looking at how *AIPS* might be deployed to run in parallel on a distributed memory cluster. In this investigation, I tried some basic experiments towards parallelising faceting in the *AIPS* IMAGR task, but ultimately came to the conclusion that despite being a fantastic package, widely used throughout the world, the core design of *AIPS* severely restricts integration of efficient parallel code within its tasks. While this is mainly a result of the way it handles dynamic memory, I came to the realisation that without rewriting potentially thousands of lines of core *AIPS* code (an extremely difficult, if not impossible task) introducing the parallelism I wanted would not be possible.

Given these restrictions, I also considered using *aips++* (Croes (1993) & Noordam (1994)), a software package and library which has now evolved into CASA (Jaeger (2008)). While *aips++* offered quite a lot of potential at the time its code base was still very much a work in progress, and being rather unwieldy to both build and modify, I decided that unfortunately this could also not provide the platform that I was looking for.

During my original investigation into parallelising the faceting algorithm, I had read about a relatively new algorithm called w-projection, which had been recently developed by Cornwell et al. (2005). This algorithm, described in Section 2.3, presented an extremely promising alternative to faceting for imaging wide-fields being both parallelisable and far more computationally efficient. While learning how to program using MPI, I had also came across a field of research using graphics hardware for numerical computing. This was an extremely new concept the time, and numerical code had to be written using the graphics API and programmable shader processors, which had evolved in order to meet the ever increasing demands of real-time graphical effects in video games. While MPI is a well established parallelism framework, it usually relies on launching processing tasks as scheduled jobs on large shared computing resources and this is not particularly well suited

to the high level of interactivity that interferometric imaging often requires. General purpose computing on graphics processors (GPGPU) on the other hand, by harnessing the huge computing capability already in many desktop workstations, allows applications far more interactivity.

As a result of these considerations, I came to the conclusion that the best way to proceed would be to develop my own software package. By doing this I could develop a framework for the rigorous testing of interferometric imaging algorithms which would be sufficiently flexible to support the integration of graphics processors, something I was keen to explore. Another distinct advantage in this approach is that I could tailor the software specifically to provide a simple interface for testing algorithms along with all their possible configurations and options.

In the following sections of this chapter, I describe the development of this software, which I called *WFIT*, and present a brief overview of its current capabilities. *WFIT* provided the platform in which my GPU based wide-field imaging code was integrated, the results of which are presented in Chapter 7.

In addition to developing the imaging algorithms which I describe in Chapter 6, by writing this code I learned a huge amount about application and software, ranging from understanding how to make best use of build systems and compiler optimisations to integrating numerical code with various graphical user interface APIs.

5.2 Design objectives and principles

WFIT has been designed as a flexible environment to explore a number of wide-field synthesis imaging algorithms on both the CPU and GPU. Ultimately, the final product was 16000 lines in total, plus a further 4000 lines of comment. It comprised seven modules, namely widgets, utility, math, data, CUDA, core and convFunctions. Each of these

modules is further split into a number of classes (varying between four to 20 classes) in order to make the code as modular as possible.

With a focus on development of new algorithms, flexibility was one of the major design goals when writing the code. This was realised by keeping the code framework, used to call algorithms, as simple and lightweight as possible as well as adopting an iterative design methodology with which the code could be tested, refined and refactored with ease. Writing the code in this way allowed it to be robust enough to adapt to any unforeseen changes arising during development. Keeping the design as lightweight and simple as possible is also of vital importance when writing numerical code, as care has to be taken to avoid potential overheads that an overly complicated framework can incur.

Arguably the most important element in developing flexible code is modularity, and this forms the core of the *WFIT* design. In development, optimisation and testing of various algorithms a high degree of modularity allows different versions to be evaluated either as part of a pipeline or individually and avoids excessive rewriting of code as the algorithm design progresses. As a major requirement of the code was to compare parallel algorithms on both the CPU and GPU and by writing these as discrete modules with a common interface, the calling code can swap between different implementations with ease. Modular design of the numerical code was accomplished by making use of common function interfaces, wherever possible, for sets of algorithms performing similar operations. The common interface is obtained by passing container classes (special data structures) rather than raw data types to modules and designing these container classes to encapsulate all of the required parameters. The code within each module then extracts the raw data types before operating on them. This was found to be particularly important for performance as compilers often ignore derived data types during code optimisation.

While the focus of the code is on algorithm development and testing, this was found to require a fairly large amount of support functions in order to provide input data and a

mechanism to analyse the results. Early on in *WFIT* development I realised that I could not ignore this aspect of the design and as such chose to create a number of classes of support functions which serve the algorithmic modules. These were all designed to be completely separate from the numerical code. This was chosen to give as much reuse of these components as possible as the algorithm modules were being developed.

As testing algorithms typically requires exploring a large number of different parameters, as well as visualising the results, part of the design of support modules included a simple modular GUI environment. Following the modular design principle, this GUI was constructed in a way such that various options, control and visualisation widgets could be plugged together as needed. I decided to write a visualisation module for the GUI as, while there are many existing packages capable of visualising astronomy data, being able to integrate visualisation closely with algorithm development proved tremendously helpful. Some examples of this GUI are shown in the Section 5.4, found at the end of this chapter.

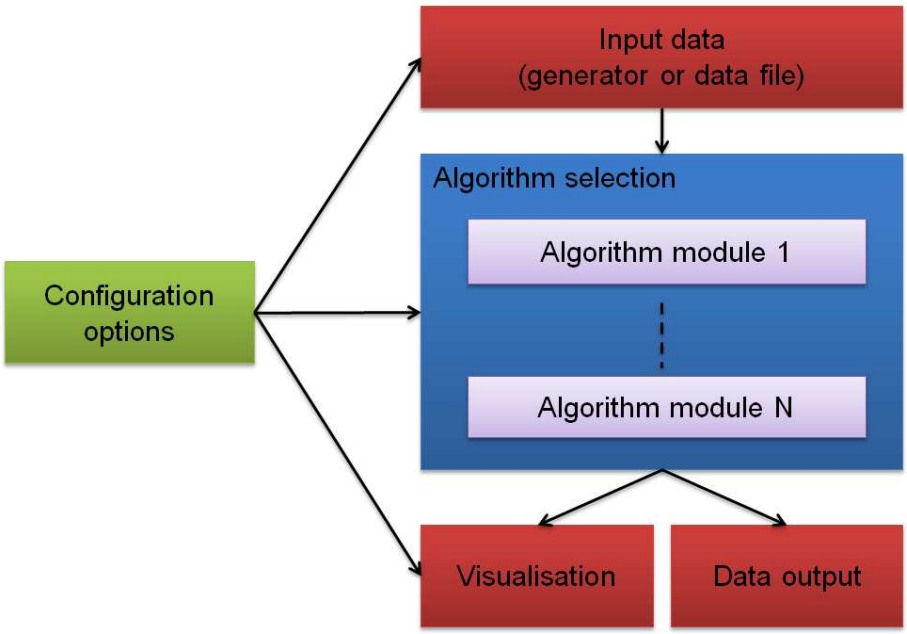


Figure 5.1: Workflow diagram showing the various modules in the *WFIT* design.

Figure 5.1 illustrates the design of a typical workflow connecting the various key *WFIT*

modules, each of which are sumarised below:

- **Configuration options**. This module is responsible for the providing the vast number of parameters that are often required for testing algorithms. The configuration option module is part of the support framework design and typically consists of a number of GUI widgets for selecting the various parameters.

- **Input data**. This module provides the data required by the algorithms being testing. The data input modules in the current *WFIT* design are part of the support framework and provide visibility data by either loading it from file (UVFITS in the case of visibility data sets) or via a number of generators. The data once loaded is encapsulated in a special container class allowing a common interface between input data modules and numerical code.

- **Numerical algorithm modules**. Algorithm modules in *WFIT* run on both the GPU and CPU and can be swapped in and out as require for testing. This is facilitated by a design whereby data is passed between modules inside container classes which encapsulate all of the raw data and parameters.

- **Visualisation**. When testing new algorithms it is important to be able to fully characterise their performance. The visualisation module, which is a support framework GUI class in *WFIT*, gives the ability to quickly inspect the input data and data products at various stages in the processing.

- **Data output** While visualisation provides a great tool for inspecting the results of algorithms during testing, the output module which is also designed as part of the *WFIT* support modules gives the possibility of recording results in the form of timing logs as well as saving the final wide-field images in FITS format so they can be analysed in a number of other astronomy software packages.

Input data	Loading UV data	Time range Channel range Polarisation Pointing
	Generate UV test data	
Imaging options	Select input UV data Transform FT type Select Dirty Image or PSF Image size in pixels Cellsize in choice in units Toggle UV weighting	
Convolution kernel	Choice of kernel	Pillbox Exponential Exponential times sinc W-projection kernel
	Kernel ordering option Toggle for grid correction Toggle to save kernels	
GPU processing options	Toggle kernel generation on GPU Toggle gridding on GPU Toggle FFT of visibility grid on GPU	
Plotting options	Select data to plot Set axis ranges, colour scale, etc	
Saving options	Save plots png/ps/pdf/FITS	

Table 5.1: Table of configuration options for WFIT. See also screeen grabs in subsequent pages.

A developer manual is currently available, though not yet a user manual.

5.3 Implementation

After careful consideration of the design objectives, I chose to develop *WFIT* in C++ as the number of readily available libraries and its native modular (object-oriented) coding style was well suited to this project. Another strong bearing on this decision was that many GPU libraries which I wanted to explore, namely OpenGL, Cg and more recently CUDA, are primarily C/C++ based.

While C++ is more often associated with application development than high performance numerical code due the difficulty optimising compilers can have with frequency use of derived data types and the overheads certain object oriented language constructs (e.g. virtual functions) impose on the runtime code, throughout the development of *WFIT* I have taken particular care in avoiding shortcomings. To this aim, for the high performance numerical parts of the *WFIT* code, I have restricted the use of C++ to its simpler C like constructs.

In addition to the GPU libraries OpenGL, Cg and NVIDIA CUDA, described in Chapter 4, *WFIT* makes use of the following libraries, selected to provide the funtionality described below and because of wide availability across a number of operating sytems as I was keen not to overly restrict the portability of the the code.

- **Qt4** for general utility functions (e.g. file I/O, timers, string manipulation, regular expressions etc.) and for its graphical user interface widgets.
- **Qwt5** for plotting widgets used to construct the visualisation module.
- **CFITSIO** for reading UV FITS visibility data files and for writing FITS images.
- **FFTW** for its highly optimised CPU FFT functions.

- **Boost** for handling command line arguments to *WFIT*.
- **OpenMP** in order to write threaded numerical algorithm modules on the CPU.
- **CppUnit** for its unit testing framework.

5.3.1 Input data module

The input data module, shown in Figure 5.1, performs the task of populating a container class encapsulating visibility data either observed by a telescope or generated for the purposes of testing. This task is therefore carried out by either generating simple test data though a number of generator functions, or by reading random group UV FITS (Greisen & Harten (1981)) files. The random groups FITS format is a standard for storing radio interferometry data used by the popular *AIPS* radio astronomy software package. While the FITS format is an excellent format for storing image and binary table data, unfortunately the UV FITS format uses a rather unconventional storage scheme designed with magnetic tapes in mind and while the CFITSIO library provides a number of utility functions for handling this format it is poorly documented and I initially had some problems reading this data type correctly.

Support for UV FITS was chosen for compatibility with AIPS which was the main relevant software package available at the time of writing. Support for other formats however could be easily added by writing a data adaption class to convert from the representation in the file of choice to the internal UV data format used by WFIT. The entire data set is currently loaded into system memory prior to processing. Therefore a restriction is at present placed on the size of the data set that can be processed which is determined by the system RAM. This limit could in princple be avoided by only loading sections of the data to system memory for processing as needed. However, careful consideration would be needed to avoid adversely affecting performance due to disc loads.

5.3.2 Numerical algorithm modules

In the current version of *WFIT* I have implemented algorithms focused on investigating the wide-field imaging algorithm w-projection on the GPU. Implementation of the algorithm, described in more detail in Chapter 6, involves a number of function evaluations, convolutional gridding, and imaging by Fourier inversion and as such during this thesis I have implemented modules to perform each of these tasks on the GPU and the CPU. In addition to wide-field imaging I implemented a version of the simpler 2-dimensional imaging approach similar to that in the *AIPS* IMAGR task for verification purposes and comparison with the w-projection imaging results. The GPU and CPU modules were written with a common interface by making use of container classes for visibility data, convolution functions, visibility grids and images. In order to provide as fair as possible a comparison of the performance between CPU and GPU versions of the code I have used OpenMP threading on the CPU to utilise all of the available cores wherever possible. It should be noted that while the current set of algorithms implemented in *WFIT* is limited to forming images from interferometric data the code could be easily extended to investigate other algorithms such as deconvolution, the next step in most radio imaging pipelines, as the framework is now in place.

5.3.3 The graphical user interface and visualisation

In addition to a basic command line interface written, using the Boost program options library, I chose to develop a graphical user interface for *WFIT* in order to provide some interactivity when investigating different algorithm parameters. This GUI is written as a collection of widgets for configuration options, algorithm control and visualisation written using Qt and Qwt. The various widgets are then plugged together, communicating via Qt's signal and slot mechanism in order to provide a flexible reconfigurable GUI environ-

ment. Some examples of these widgets are shown in Figures 5.2 though 5.7.

The visualisation of data from various stages of processing is handled by the *WFIT* the visualisation module widget. This wraps a number of Qwt plot canvas functions to provide a selection of scatter and image plots suitable for visualising the various types of data products used in wide-field imaging. As visualisation, especially when forming lots of large images, uses reasonably large amounts of system memory and processing resources, I have written this widget in such a way that it can be enabled and disabled at run time in order to avoid influencing the results when timing the performance of the various algorithms.

5.3.4 Output module

The output module for *WFIT*, shown in Figure 5.1, provides two main functions. First it can be used to record log messages issued from modules which include events such as timing results needed to assess performance and second it provides a number of functions for writing the results as either ascii files, or images in various formats such as png and FITS. Writing fits images, achieved by use of the CFITSIO library, is particularly useful as these can be loaded into a number of other astronomy imaging software for subsequent analysis.

5.4 Using WFIT

In this section I present a selection of screenshots of the *WFIT* GUI, illustrating the main steps required to generate and visualise an image formed from a visibility data set loaded from a UV FITS file.

The *WFIT* application consists of a control GUI with three main component widgets: a UV data options widget which selects and controls input visibility data; an Image

options widget for settings controlling image generation; and a Processing Control widget which toggles GPU implementations of the various modules.

The first stage of making an image is loading the visibility data. This is handled by the visibility data selection widget tab in the main *WFIT* control GUI, as illustrated in Figure 5.2. Once a UV FITS file is loaded into the application this widget then allows a subset of the visibility data set to be loaded based on the range of time samples, frequency channels, polarisations and/or pointing directions selected. The generator tab, shown on the left of Figure 5.2 below the open selection tab gives provides a alternative way to obtain visibility data for imaging where the visibility data set is populated with synthetic data from a number of specified generators designed for algorithm testing.

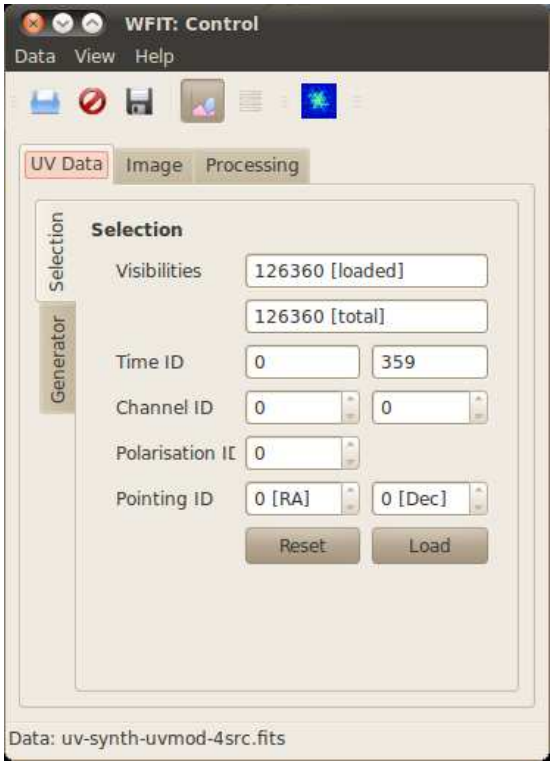
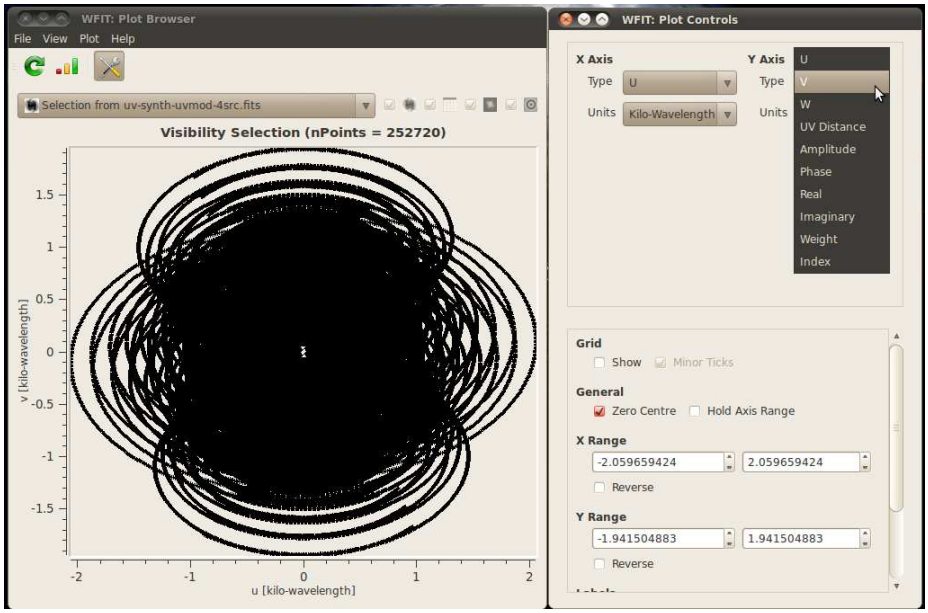


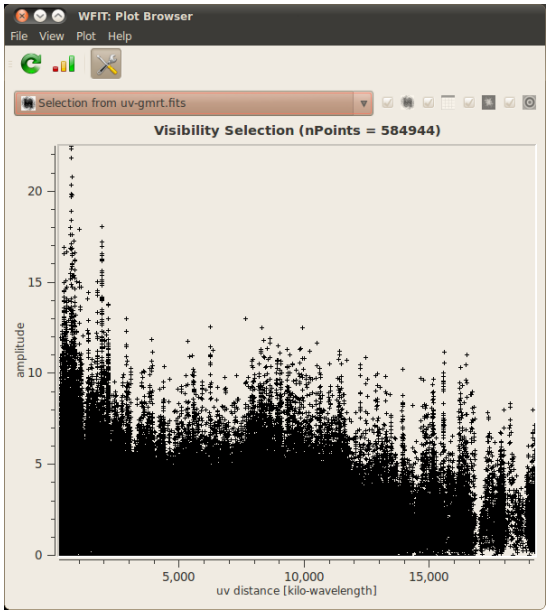
Figure 5.2: The *WFIT* UV data selection options widget. This widget allows a subset of the loaded data set to be selected for imaging.

Once loaded, visibility data sets can be displayed in the visualisation widget as shown in Figure 5.3. The visualisation widget gives a number of options for how visibility data can be displayed. Figure 5.3(a) show a plot of the baseline coordinates in the u, v plane,

and Figure 5.3(b) shows a plot of baseline length against the modulus of the visibility amplitude.



(a)



(b)

Figure 5.3: Visualisation of loaded visibility data using the *WFIT* visualisation GUI. The widget provides a number of different options for visualisation of visibility data with (a) showing a plot baseline coordinates in the u, v plane (b) showing the u, v distance against the modulus of the visibility amplitude

In order to make an image from the visibility data it is then necessary to select and configure a gridding convolution function and specify a number of image options such

as the image size and pixel size. The widgets for configuration of w-projection gridding kernels and the associated image options are shown in Figure 5.4.

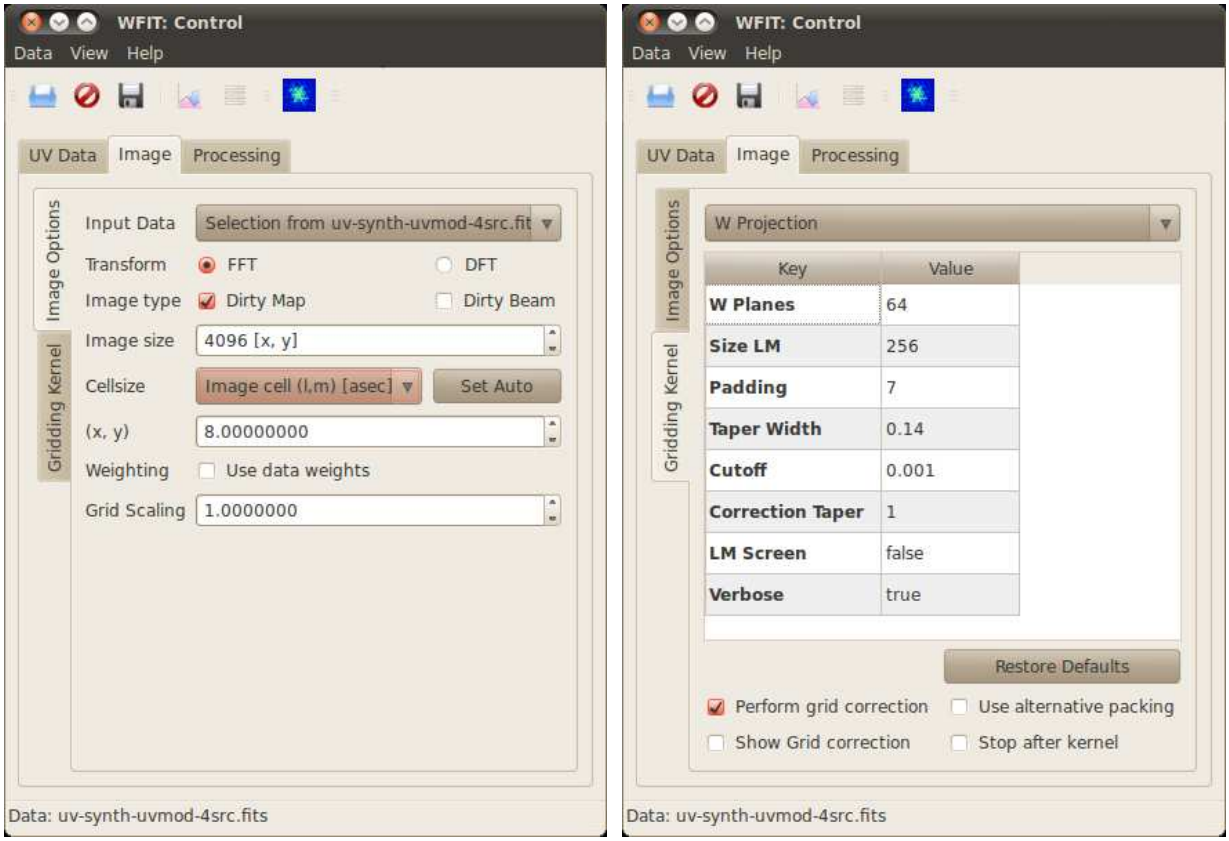


Figure 5.4: Widgets for (a) image options and (b) convolution function options. In this example the configuration table for w-projection kernels is loaded.

Once the image has been generated it can be visualised by the *WFIT* visualisation widget as shown in Figures 5.5, 5.6 and 5.7. This widget is designed to provide a quick and lightweight method of assessing the effectiveness of imaging from within the code.

By using the output module described in the previous section, *WFIT* is also capable of saving FITS images which can then be viewed or processed in other astronomy software packages. With its current compliment of algorithms *WFIT* produces raw wide-field 'dirty' images but does not do any further post processing beyond this point. While subsequent post processing by deconvolution and self calibration normally plays an important role in imaging and can be particularly effective in improving sparsely sampled

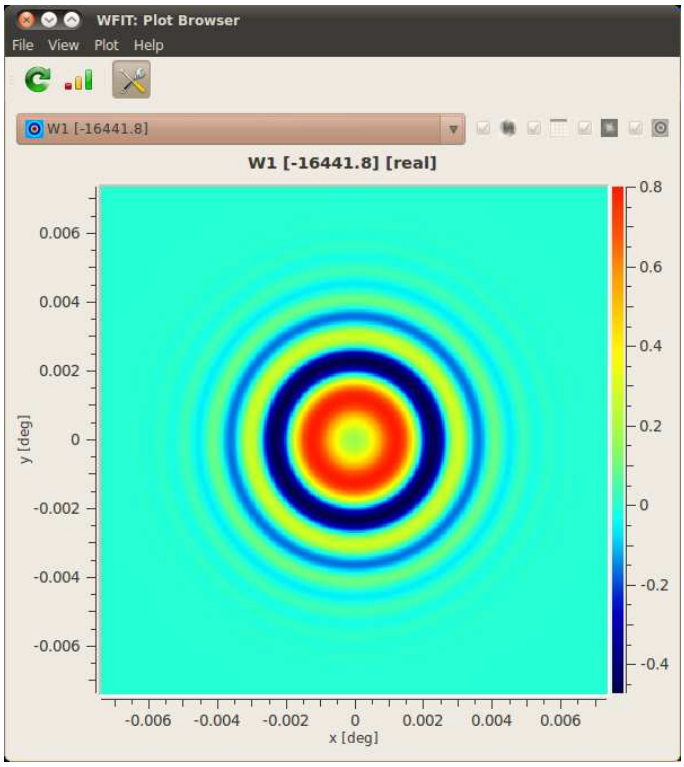


Figure 5.5: Visualisation of the real part of w-projection convolution kernel for a W plane centred at a w value of -16.8 kilo-wavelengths. A description of the generation of w-projection kernels by *WFIT* can be found in Section 6.3.

spectral data, as seen in Chapter 3, in this study I have not looked into this form of post processing for image data.

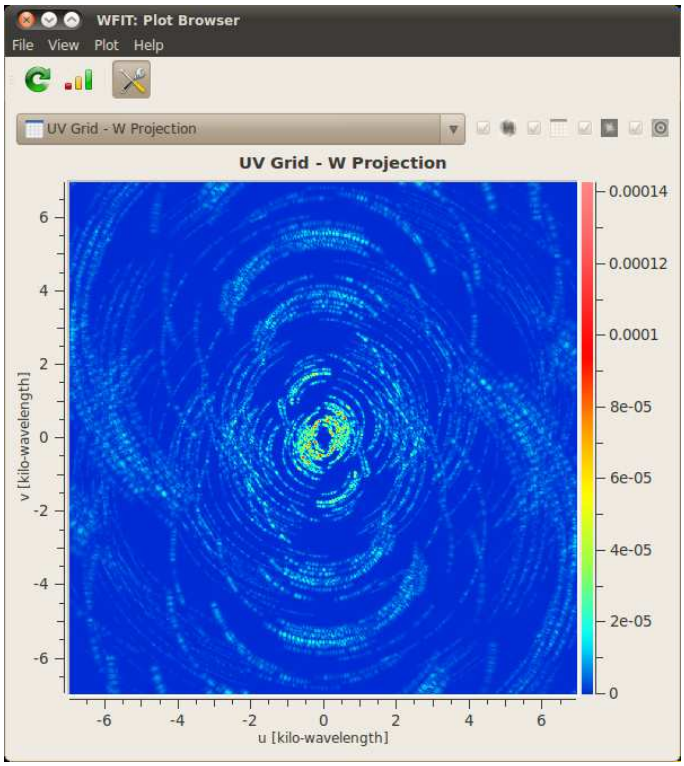


Figure 5.6: Visualisation of the real part of the visibilities, gridded using the w-projection kernels. The image is zoomed in on the central part of the grid.

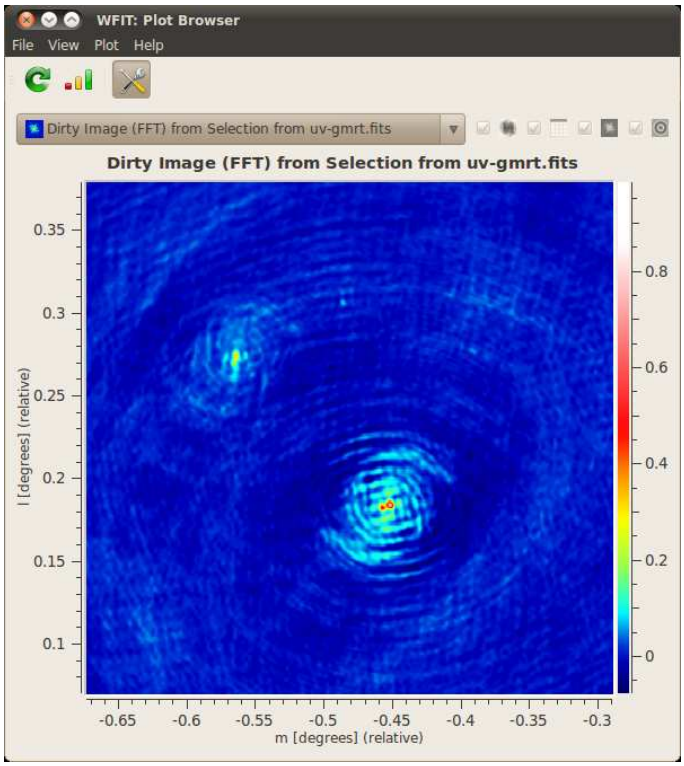


Figure 5.7: Visualisation of a small section of a dirty image generated generated using w-projection.

Chapter 6

Wide-field interferometric imaging using GPUs

This chapter describes the design and implementation of a number of algorithms that I have developed for the imaging of interferometric data using GPUs. While in this chapter I present these algorithms without direct reference to processing astronomical data, all of these methods have been extensively verified by imaging data from both simulations and observations at the GMRT using the code framework described in Chapter 5, and the results of this testing presented are in Chapter 7.

6.1 Introduction

The processing of interferometric data, introduced in Chapter 2, presents a number of potential algorithms that could easily be considered for implementation on the GPU. For this thesis I have chosen to focus on the investigation and implementation of wide-field imaging. This, along with deconvolution (investigated in a 1-dimensional context in Chapter 3), is one of the most challenging algorithms required to produce high fidelity, high dynamic range images in terms of both processing time and complexity of implementation on highly threaded architectures. As the next generation of radio telescopes are being

designed with large, instantaneous fields of view they will produce extreme data rates that will likely require (near to) real-time imaging, and the need for the development of high performance imaging algorithms of this sort has never been more relevant.

As described in Chapter 2, interferometers sample the Fourier space response of the antennas to the sky brightness distribution by measuring the correlated signals of a number of detectors. In order to form an image of the sky it is therefore necessary to perform an inverse Fourier transform of the recorded signals, as described by equation 2.8. Fortunately, two extremely well understood algorithms already exist for evaluating this inverse Fourier transform: the direct Fourier transform, or DFT, and the fast Fourier transform, or FFT. The DFT evaluates the Fourier integral by means of a weighted sum over the discretely sampled u, v plane signals where weighted coefficients in the sum correspond to a geometric phase term given by the dot product of the position vector of the pixel in the sky map being evaluated and the baseline vector of the signal. The FFT algorithm extends this principle by making use of symmetry relations between the complex phase weights which occur when the Fourier space samples lie on a regular grid and in doing so much reduces the number of operations required to evaluate the integral. While imaging using a DFT is attractive due to its simplicity, and with each pixel in the output brightness map being independent of one other when evaluating the sum giving a huge potential for trivial parallelism, in practice it is found that the computational efficiency of the FFT makes it the only viable option for producing large images. As observed in Chapter 2, for an image of N pixels in each dimension and M data points, the FFT scales as $\mathrm{O}(N^2 \log N + M\,C^2)$ (where C is the width of the gridding convolution function, typically 3 to 9 grid cells) compared to the DFT which scales as $\mathrm{O}(N^2\,M)$. Therefore when forming an image of 4096 pixels by 4096 pixels from 500,000 data points, which might be typical when imaging a field of view of a couple of degrees across using a using moderately sized interferometer, the number of operations for the DFT ($\sim$67 TFLOPS) compared to

the FFT (∼1 GFLOPS), makes the DFT approach completely impractical. Therefore for the following work I have chosen to focus exclusively an FFT approach to imaging.

One downside of the FFT however, is the requirement for data samples to be situated on a equispaced Cartesian grid. Constructing an interferometer which samples the Fourier plane on such a grid is extremely impractical (if not impossible) for the majority of observations which make use of earth rotation aperture synthesis to accumulate enough u, v plane samples to meet their required imaging sensitivity. Therefore in order to make use of the FFT, interpolating data samples onto a grid is almost always required. This interpolation is performed by an algorithm known as convolutional gridding, whereby data points are convolved with a special convolution kernel and the result is sampled all points which lie on the desired grid and where the convolution returns a non-zero result. This approach of gridding followed by an FFT is carried out successfully by all of the current major radio astronomy imaging packages using a single threaded CPU implementation and therefore provided the natural starting point of investigating the implementation of imaging on the GPU. The other key consideration which I wanted to investigate in its application to the GPU was that of being able to successfully image large fields of view. As described in Section 2.3, the imaging of wide fields invalidates the use of a simple 2-dimensional Fourier transform relation between data samples in u, v space and the tangent plane projection of the sky brightness distribution given by the 2-dimensional FFT. While this is far from a new problem, and has traditionally been solved with an algorithm known as Faceting (Cornwell & Perley 1992), whereby an image of the sky is made by combining a number of small sub-field images that adequately meet the required small angle approximation for this thesis I have chosen to implement and test a relatively new algorithm developed by Cornwell et al. (2005) known as w-projection. As described in Section 2.3, by use of a carefully selected w-dependent convolution kernel, this algorithm projects the measured u, v, w Fourier plane samples onto a grid corresponding

to the Fourier space of a 2-dimensional tangent plane image where the w component is identically zero. This algorithm has the huge advantage of being only a slight modification of conventional convolutional gridding and by placing samples on to a single grid prior to the FFT it is also computationally far more efficient than other wide-field imaging solutions. While writing the code for this thesis, the w-projection algorithm has been implemented also in CASA, the data reduction and imaging software written for the e-VLA and for the ALMA telescopes, however the implementation I present here is distinctly different in that it makes use of GPU acceleration whereby the CASA implementation is strictly designed for the CPU.

The following sections of this chapter describe the GPU algorithms I have implemented for wide-field imaging of interferometric data on the GPU. I begin with a discussion of my GPU gridding algorithm, then describe the generation of the special w-dependent convolution kernels required for the w-projection algorithm and finally make some comments on the use and performance of the CUDA FFT library for the inversion of the u, v plane grid to generate a image.

6.2 Gridding

As described in the previous section, in order to make use of the extremely efficient FFT algorithm, interferometric data samples must be interpolated onto a equispaced Cartesian grid, in a process known as known as gridding. Gridding is carried out using a convolution-interpolation scheme, where data points are convolved with a specially chosen convolution function and then evaluated at points on the desired sample grid. In this section I introduce the gridding algorithm in more detail and I describe an implementation of it which I have developed for the GPU.

6.2.1 The traditional approach to gridding on the CPU

Before considering the gridding algorithm on the GPU it is important to understand how the algorithm is usually implemented on the CPU. Gridding on the CPU, illustrated in Figure 6.1, can be described by the following steps:

1. Take a sample from the input data set.

2. Evaluate the multiplication of the data sample with a specially weighted constant matrix, known as the convolution kernel, at points corresponding to values on the output grid.

3. Accumulate the result of the kernel multiplication with the sample grid.

4. Repeat these steps until all the elements in the input data set have been processed.

The convolution function, used in step-2, is usually chosen to be a symmetric function which is zero for all but a small region of the output grid. This reduces the number of operations needed for gridding which scales as the number of data points times the number of pixels in the region of the non-zero convolution kernel. Choosing a convolution function that is non-zero for only a small region of values is equivalent to multiplying a smooth convolving function, C, by a pill-box function, B, of a finite radius in terms of pixels in the output grid. This finite radius is often known as the support radius. In order to avoid the computational cost of evaluating the convolution kernel multiple times, which in particular for the case of w-kernels described in Section 6.3 would be prohibitively expensive, it is usually pre-computed and stored in a look-up table. The look-up table is constructed with the function evaluated at a number of points for each grid cell determined by a quantity commonly called the over-sample. When the convolution kernel is used for any given sample, the value of the kernel, C is then taken to be that at the nearest of these pre-computed points. With a sufficiently high over-sample there is no

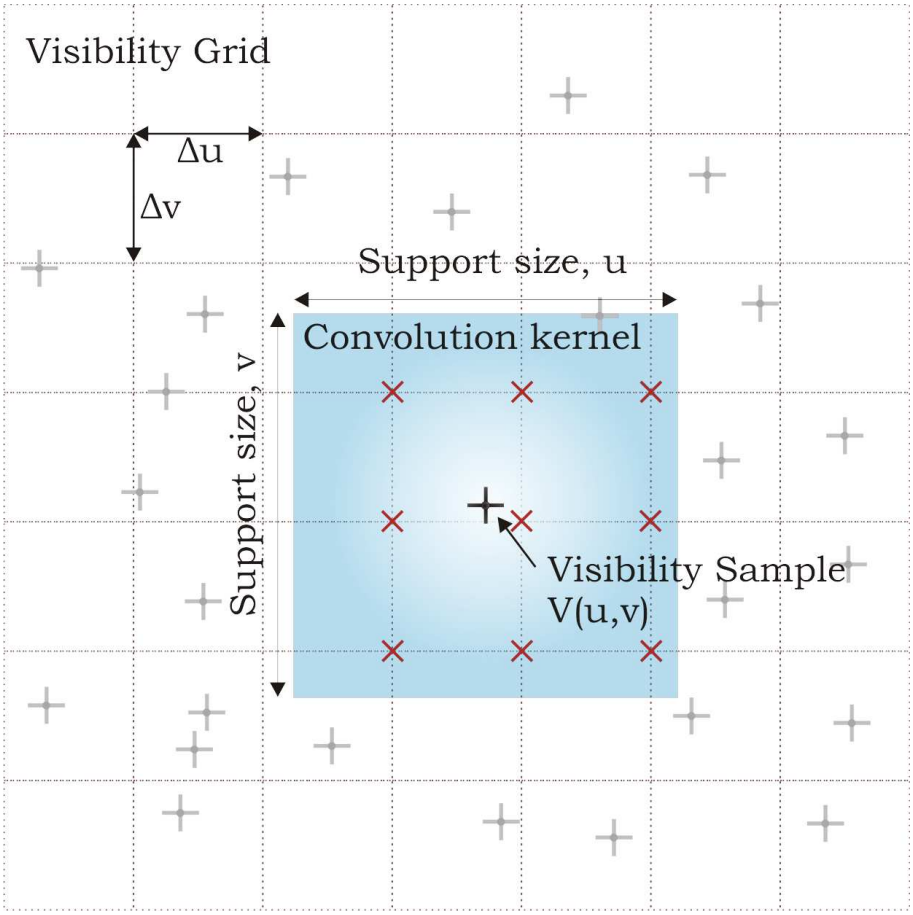


Figure 6.1: Gridding by convolution. The convolution of each visibility point, denoted by +, with a well defined convolution kernel, represented in blue, is evaluated at grid pixels within a region defined by the extent of the kernel.

need for interpolation of values in the look-up table, thus much reducing the cost of the main loop in gridding. The convolution kernel look-up table can therefore be expressed by the step function:

$$C_s = P * \left({}^2\mathrm{III} \cdot B \cdot C\right) , \tag{6.1}$$

where $*$ is the convolution operator, C is analytical form of the convolution function, B is the pill-box function which defines the extent of the kernel, P is a pill-box function that describes the separation of pixels in the look-up table, and ${}^2\mathrm{III}$ is a 2-dimensional bed-of-nails function representing the sampling of the C. The implication of using a look-up table, described by Equation 6.1, is that the map produced by the Fourier transform of the smoothed observations is the product of the desired map and the Fourier transform

of the sampled kernel, C_s. As such the resultant visibility grid can be expressed as:

$$V'' = {}^2\mathrm{III} \cdot (C_s * (V \cdot S)) \,, \tag{6.2}$$

where $V \cdot S$, is the continuous Fourier plane response V sampled by the telescope according to a sampling function $S = \sum_j W_j \, {}^2\delta \, (u - u_j, v - v_j)$, C_s is the convolution function described above, and ${}^2\mathrm{III} = \sum_n \sum_m {}^2\delta \, (u - m\Delta u, v - n\Delta v)$ which defines the coordinates of the output grid.

The image formed by the FFT of this visibility grid is then given by:

$$I'' = \overline{{}^2\mathrm{III}} \, * \, (\overline{C_s} \cdot (I \, * \, B)) \,, \tag{6.3}$$

where I'', $\overline{{}^2\mathrm{III}}$, $\overline{C_s}$, I, and B are the Fourier transforms of V'', ${}^2\mathrm{III}$, C_s, V and S, respectively. If we then ignore the effects of aliasing (i.e. the $\overline{{}^2\mathrm{III}} *$ term) this equation can be rearranged to give an expression for the image commonly known as the dirty map

$$\mathrm{I}^D = \frac{\mathrm{I}''}{\overline{C_s}} = \mathrm{I} * \mathrm{B} \,, \tag{6.4}$$

the ultimate product of gridded, FFT based imaging. Note that the division by $\overline{C_s}$, the Fourier transform of the convolution kernel look-up table, results in an image which is unaffected by the process of sampling data onto the regular u, v grid and as such this division is known as grid correction.

If we again consider the description of gridding given in this section, it can be seen that a gridding can be implemented on the CPU in a relatively straightforward and efficient manner. The pseudo code in Listing 6.1 describes a typical implementation of gridding on the CPU and is very similar to the CPU gridding code I have written to verify correctness and performance of gridding on the GPU. Parallelising this code for use with multi-core

CPU hardware can be easily achieved by splitting the data points between each of the cores and declaring the grid as a resource managed in shared memory. My implementation of the CPU code used for comparison of GPU performance does this making use of the OpenMP[1] shared memory parallel programming API.

Listing 6.1: Pseudo-code of gridding on the CPU.

```
for(int i = 0; i < num_data_points; ++i)
{
    int grid_x = find nearest grid pixel to the data point in x direction.
    int grid_y = find nearest grid pixel to the data point in y direction.

    for (int y = 0; y < width_of_kernel_in_grid_pixels; ++y)
    {
        for (int x = 0; x < width_of_kernel_in_grid_pixels; ++x)
        {
            grid[grid_y + y][grid_x + x] +=
                 kernel evaluated at (x, y) * data sample amplitude;
        }
    }
}
```

6.2.2 Introduction to gridding on the GPU

As described in Chapter 4, in order to write effective code for the GPU, the first goal of any algorithm is that it must make use of the huge number of parallel threads that the GPU provides. For gridding interferometric data, a large number of data points are interpolated onto a grid containing a large number of grid points, and as such there are two natural parallelisation methods by which a massive number of threads could be generated; parallelising over the data points or parallelising over the grid pixels. These two possible approaches are explored in the following sections.

While gridding on the GPU is new to radio astronomy, for medical imaging of MRI data, which shares a number of similarities to radio interferometry, some steps have already been made to accelerate this algorithm for the GPU as described in Gregerson (2008). When implementing my GPU gridding code I was unaware of these independent developments:

[1] http://www.openmp.org/

it is encouraging to note their approach shares some similarities to that which I have taken for my implementation of the algorithm and the differences in terms of handling data halo regions between the two algorithms give rise to some interesting possibilities for future development of the gridding algorithm as applied to radio interferometer data.

6.2.2.1 Gridding as a scatter operation

In Listing 6.1 which shows some pseudo-code for gridding on the CPU, each data sample taken in turn applies its contribution to the grid. By parallelising over the outer loop of this code one could generate the huge numbers of threads required for a GPU algorithm with each thread being assigned to one or more data points. Gridding in this way is essentially a scatter operation where the contribution of each data point is placed onto the grid after convolution with the gridding function. This method is maximally efficient from the point of view of both the number of operations and memory accesses, scaling as $\mathrm{O}(N\,C_{\mathrm{size}}^2)$ for N data points and a convolution kernel of width of C_{size}. The downside is however that potentially many visibility points can influence a single location of the output grid so memory access race conditions in writing the results can occur if not properly managed. With a managed shared memory system such as that of multi-core CPUs this manifests itself as a performance penalty requiring the use of synchronisation and atomic operations. On the GPU however where data sharing is limited to small blocks of threads running on the same multiprocessor, implementation of this approach is extremely difficult and potentially highly inefficient. In order to reduce the number of threads that are trying to write to any given pixel in the output grid one might consider assigning a number of data points, rather than one, to each thread and pre-sorting the data such that threads writing to the same location are allocated to the same thread block which has access to shared memory resources. While this approach might make this data-driven gridding model tractable, the reduction in the degree of possible parallelisation and

the requirement for careful assignment of which data points are assigned to which thread means the solution is both extremely complex and has limited scalability. Alternatively, while providing far less parallelism, one might also consider making use of the CUDA basic linear algebra library routines for parallelising the inner loop of Listing 6.1 using a GPU accelerated vector scalar product operation. This approach should be relatively simple to implement and as it only requires addressing one data point sample at a time, it avoids multiple threads writing concurrently to the same pixels in the grid. This approach however offers very poor scaling for large numbers of data points.

6.2.2.2 Gridding as a gather operation

In contrast to the approach described in the previous section, gridding can also be written as a gather operation according to the pseudo code in Listing 6.2 and illustrated in Figure 6.2. In this approach threads are assigned to one or more grid pixels. Each thread loops

Listing 6.2: Pseudo-code showing gridding as a gather operation.

```
for(int i = 0; i < num_grid_pixels; ++i)
{
    for (int k = 0; k < num_data_points; ++k)
    {
        distance = find distance of data point from grid pixel.
        if (distance < kernel radius)
        {
            grid[i] += data amplitude * kernel evaluated at distance of
               data
                point from the grid pixel.
        }
    }
}
```

though the available data and for data points which are close enough to the pixel for the convolution with the gridding kernel to return a non-zero result, the contribution of the data sample is added to the pixel being processed by the thread. By assigning threads to the grid pixels rather than to the data points, this method avoids all of the problems of multiple concurrent threads trying to write to the same memory address.

What is immediately clear however is that while the scatter-based gridding method in the previous section scales as $O(M\,C_{\text{size}}^2)$ (for M data points and a convolution kernel of width C_{size}), this approach scales as $O(N^2\,M)$ (for a grid of N by N pixels) which would result in a hugely increased number of operations and memory accesses for large grids. This is because most of the wasted operations and memory accesses in this approach can be attributed to having to examine data points which have no influence on grid pixels to which data is being gathered. This problem can be largely avoided by pre-arranging the data into groups which affect certain regions of the final grid. As described in Chapter 4 GPUs naturally split their processing threads into groups so having one block of GPU threads process data for one block of the output grid and its associated data presents a natural way to parallelise gridding on the GPU. Furthermore, dividing the data into blocks reduces the number of operations and memory accesses by a factor corresponding approximately to the number of blocks, making this approach far more feasible than a naive implementation would initially suggest.

Therefore with the clear advantage of avoiding problems of inter-thread synchronisation given by the scatter gridding approach, and with a solution to avoiding the huge number of memory accesses of the most basic implementation of the gather-based gridding, I therefore chose this grid pixel parallel approach to implement in my GPU gridding algorithm.

6.2.3 Putting data into sub-grids

As discussed in Section 6.2.2.2, a vital step in gridding data on the GPU, when threads are assigned to grid pixels, is grouping data into bins or sub-grid cells, corresponding to their region of influence on the final grid. This is required to reduce the number of memory accesses per thread that would otherwise have to be made when implementing a gather approach to gridding.

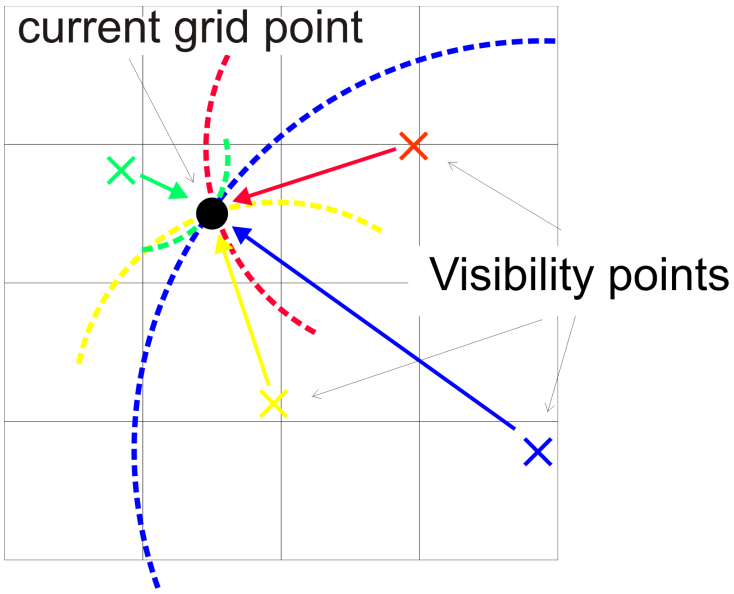


Figure 6.2: Cartoon showing gridding as a gather operation. The contribution of data points which have a non-zero result after convolution with the gridding kernel are added to the mid-point of each grid pixel.

As described in Section 6.2.3.2 my efforts at porting this algorithm to the GPU were ultimately rather unsuccessful and as a result for the final GPU imaging code this step was performed on the CPU. I however include a description of the approach I made with this algorithm for completeness and so that any future work on GPU gridding might avoid this approach!

6.2.3.1 Algorithm and implementation

Splitting the data into bins which affect a sub-section of the final grid is essentially a form of spatial binning. Spatial binning operations are well known to be notoriously difficult algorithms to parallelise as they either require a large degree of cooperation between threads or scale poorly in a similar fashion to the scatter- and gather-based approaches to gridding described in Sections 6.2.2.1 and 6.2.2.2.

In designing an algorithm for binning data on the GPU it is therefore necessary to overcome a number of challenges.

1. Global thread cooperation is not possible as the CUDA programming model limits

thread communication to small groups of threads known as a thread block.

2. The memory address space that can be shared between threads is small, local to the small blocks of threads and lacks automatic mechanisms for atomic operations.

3. The data being binned is typically too large to fit into the shared memory space.

4. With a lack of a priori knowledge of the number of data points that belong to each bin (this is dependant on the Fourier plane of the interferometer for the particular observation) creating efficient data structures to store the binned data is nearly impossible.

With these challenges in mind I attempted to implement a GPU algorithm which collects the data into bins corresponding to its influence on a sub-grid cell region. To do this I have written the algorithm as two stages described as follows:

1. Evaluate the influence of each data point on the grid divided into sub-grid regions by taking into account the radius of the gridding kernel. This is achieved by:

   - For each thread (each assigned to a single data point):

     (a) Load a data point with an index chosen based on the thread ID.

     (b) Scale the data point coordinates into the sub-grid coordinate space.

     (c) Round the scaled position including the radius of the kernel to evaluate the limits of the sub-grid cell space over which the data point has influence.

2. For each sub-grid cell, using the data sub-grid influence limits evaluated in step one, find and store data points which belong to each sub-grid cell by:

   - For each thread (each assigned to a sub-grid cell):

     (a) Loop though the data and evaluate if the sub-grid cell limits found in step-1 correspond to the ID of the cell being handled by the thread.

(b) If a data point is found to have influence on the cell, store it in the memory assigned to that sub-grid cell bin.

A two-stage approach was chosen to hugely reduce the number of floating point operations needed. Without the first stage, it is necessary to recalculate the influence of each data point on the sub-grid cells many times.

The first part of the algorithm, illustrated in Figure 6.3, is relatively trivial to implement on the GPU. One thread is assigned to each data point, the influence of which can be evaluated independently. This results in a large number of threads and a high degree of parallelism, well suited for computation on the GPU. With each thread loading values from global memory separated with a linear stride based on the thread ID, data loads are naturally coalesced by the GPU's memory system. The results of the evaluation of sub-grid cell limits are also written to global memory in a naturally coalesced fashion. Lastly, as this operation requires minimal use of thread local registers, memory access to global memory can be well hidden by a high occupancy of threads per multiprocessor which results in good use of the low cost thread switching that occurs on the GPU when threads are waiting on memory loads and other threads are available to be switched in for work. As such even a basic implementation of this part of the algorithm performs well on the GPU, evaluating over 400 million data points per second.

The second part of the algorithm, illustrated in Figure 6.4, requires far more consideration. With threads assigned to each sub-grid cell, each thread must iterate though all of the sub-grid cell data limits evaluated in the first stage of the computation collecting data which influence the cell in question. While avoiding the considerable difficulty of inter-thread communication on the GPU, this scheme still poses a number of challenges.

- The number of data points per sub-grid cell is unknown in advance and therefore it is extremely difficult to allocate an efficient data structure to store the results.

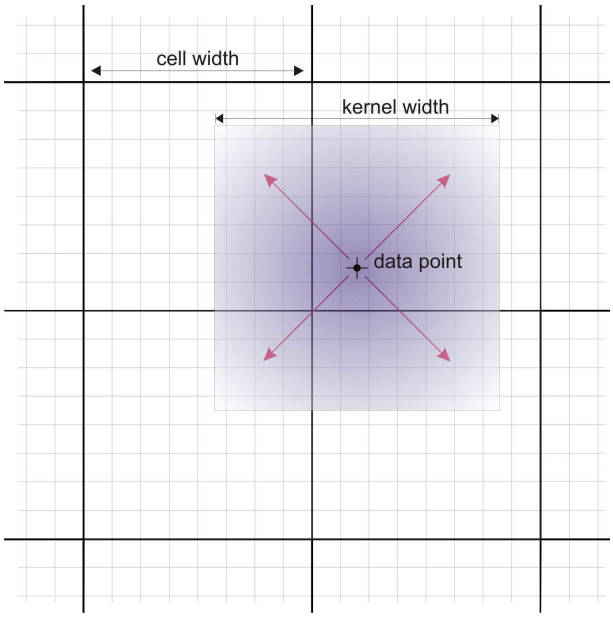


Figure 6.3: Cartoon showing the influence area of a data point on sub-grid cell space. The first part of the sub-grid binning algorithm evaluates the range of sub-grid cells affected by each data point.

- GPUs are well known to perform poorly with code containing a large amount of branching, and this algorithm is entirely based on a series of comparisons for each data sample.

- The algorithm is extremely memory intensive requiring $\mathrm{O}(N_{\mathrm{sub-grids}} \cdot N_{\mathrm{data}})$ memory accesses.

While branching is central to this algorithm and little can be done without a completely different approach, in order to greatly alleviate the problems of intensive memory access I implemented a software-managed cache using the shared memory available to thread blocks. By loading successive chunks of the data into the fast shared memory, shared by groups of threads in a thread block and using this cache to search for values to be added to each sub-grid bin much of the slow global memory access is avoided. The number of threads in a thread block and the size of cache used was also found to be a key factor in optimisation because if either of these were too large the per-thread block resource usage in terms of registers and shared memory resulted in a poor thread occupancy on the GPU

multiprocessors, and if the number of threads in a block was too small the benefit of the caching is reduced. Care also had to be taken in order to avoid bank conflicts in shared memory, where multiple concurrent accesses to data within the same bank are serialised, when storing the data sub-grid indices in cache.

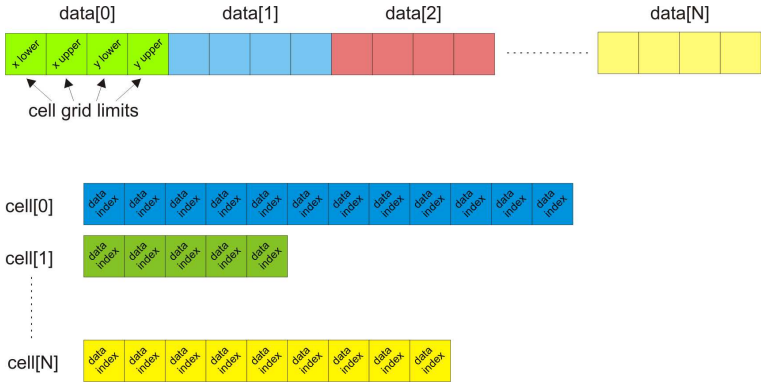


Figure 6.4: Cartoon representing the collection of data point indices found to belong to each sub-grid cell. Note that the number of data points per sub-grid cell is not known a priori so assigning efficient data structures on the GPU is extremely difficult.

6.2.3.2 Results and future work

For the first stage of the algorithm, even with a very straightforward implementation I am able to process over a million visibility points in less than one thousandth of a second on an NVIDIA GeForce 275 GTX GPU (the average time taken for 50 iterations processing 1 million data points was found to be 0.00056 seconds measured using GPU hardware counters). This is not particularly surprising considering the algorithm requires only 8 floating point operations and 6 global memory accesses per data sample and with a register count of 8 registers per thread, full occupancy and therefore a high level of hiding of most of the slow global memory access transfers though thread switching, is achieved.

The second part of the algorithm represented far more of a challenge and as described in the previous section a number of optimisation steps were taken to improve its performance. The simplest implementation of the algorithm where all memory accesses were to global memory, running on an NVIDIA GeForce 275 GTX, took 0.113 seconds to bin 100,000

data points into 4096 sub-grid bins with a kernel width of 9 grid pixels. Above this number of data points in certain sub-grids for a typical distribution of u, v plane samples which is typically more dense towards the centre of the distribution caused some cells to contain more data than would fit in the allocated GPU memory. The data caching optimisation taking care to avoid memory bank conflicts, described in the previous section, reduced this time to 0.032 seconds i.e. a factor of 3.2 performance gain over the simple implementation.

While this result shows that binning of data on the GPU taking this approach can be achieved reasonably quickly, it should be noted that the simple CPU implementation which I used in order to test the correctness of the binning operation took only 0.005 seconds on an Intel i5-750 for the same amount of data.

Further investigation and testing using the CUDA profiler, described in Chapter 4, highlighted the main performance loss to be a result of a very large number of branches that this algorithm relies on in the assessment of placing data samples in the sub-grid bins. This approach was taken to avoid memory race conditions of the standard CPU bucket sort algorithm. However, on the latest generation of GPU architectures (2.0 and above) there is now support for floating point atomic operations which would avoid this problem and might prove a interesting route for future developments. Alternatively as binning is effectively a sorting operation, incorporating some of the optimisations from developments of sorting data on the GPU might be investigated, although this sort of algorithm is far from suited for the GPU's data-parallel, single instruction multiple thread (SIMT) architecture.

As a result of the rather disappointing performance binning the data into sub-grids on the GPU for the main imaging code I therefore decided to perform this step on the CPU prior to transferring the data to the GPU for gridding.

6.2.4 GPU gridding: algorithm implementation

As introduced in Section 6.2.2.2, I chose to adopt a gather-based approach to gridding on the GPU. In doing so I assumed data has been pre-arranged into sub-grid bins as described previously. If the data is not pre-arranged into sub-grid bins, each thread must iterate over the entire data set resulting in an inordinate number of memory accesses and conditional evaluations both of which need to be avoided to obtain reasonable performance. I chose to assign each sub-grid region to a thread block for processing. This allows a number of optimisations using the fast thread block local shared memory and, by allocating one thread per pixel the huge total thread count gives a good level of parallelism.

The strategy by which I implemented gridding can therefore be summarised as follows:

1. Assign a thread to each position in the grid. More specifically, as threads on the GPU are executed in blocks which share a set of common memory resources, I chose to assign blocks of threads to the sub-grid regions for which the data has been binned.

2. Each thread then loops over the data samples within the sub-grid bin and evaluates the distance of the data point from the grid pixel.

3. If the distance is greater than the radius of the gridding kernel the data point is discarded as it makes no contribution to the value of the grid pixel.

4. For distances less than the kernel radius, the kernel amplitude is evaluated from the nearest value in the convolution function look-up table and multiplied by the amplitude of the data point. This product is then accumulated to the grid pixel amplitude. The size of the lookup table typically has to be chosen such that the interval between samples limits the effect of aliasing of the Fourier transform of the convolution kernel on the final image, as described by Greisen (1979). While this

means that the lookup table is usually too large to be loaded onto the GPU to make best use of caching strategies, typical GPU memories are large enough that no compromise has to be made to the size of the lookup table which would impact on image accuracy. The lookup tables used in w-projection typically range from a few kilobytes to a couple of hundred megabytes in the extreme cases each of which fit easily in the memory of a modern GPU.

When binning data into sub-grid bins, data points in a halo region corresponding radius of the gridding kernel have to be included. This increases the amount of data held in the bins by a factor proportional to the relative width of the convolution kernel and sub-grid. An increased amount of data has adverse effects on scaling performance of the algorithm as the extra data results in more evaluations in gridding. I therefore assigned sub-grids to be as large as possible under the constraints that the number of threads per block is limited on current hardware to 512 threads. In order to maximise thread execution occupancy the memory usage of thread blocks must be kept to less than half of the resources per GPU multiprocessor. This resulted in sub-grids of 16 by 16 pixels in size.

Listing 6.3 shows code for the simplest possible, unoptimised implementation of my GPU gridding kernel function. While simple, this code illustrates how the gather-based gridding algorithm is implemented on the GPU and highlights a number of optimisations that can be made. Looking at the source listing, it is immediately evident that the number of floating point operations is relatively small compared to memory accesses. This regime is well known to be sub-optimal for GPUs and therefore the biggest optimisations will come from reducing the number of slow, global memory transfers. Memory access to the global memory here occurs when accumulating to the grid pixels, loading data from the sub-grid bin and extracting the amplitude of the gridding function from its look-up table. In order to reduce the cost of memory writes in accumulating to grid pixels it is possible to cache the sub-grid in the thread-block local, fast shared memory until all of the sub-

grid data have been processed. In the version of the algorithm shown in listing 6.3, a group of threads in a thread block, each load the same data points directly from global memory. This can be reduced to a single load of blocks of data into a shared memory cache and then having each thread access data from the local cache. Finally the kernel look-up table poses more of a problem, as access to the gridding function look-up table is essentially random. Implementing a software managed cache is impractical and while the GPU provides a small amount of cached constant memory that would usually be ideal for this, it is limited to 64 kilobytes on current architecture which is unfortunately too small. Fortunately, caching of constant look-up tables is also required in graphics applications in texture rendering, so the GPU API provides a limited amount of hardware cache by storing kernels as textures. In the final implementation my gridding function makes use of all of these optimisations, the results of which are shown in the next section.

6.2.4.1 Results & Optimisations

As described in the previous section I implemented a number of optimisations to the code in Listing 6.3. The results of these, run on a NVIDIA GeForce 275 GTX, are summarised in Table 6.1. Successive rows in the table show the results of adding the additional optimisation described. For comparison the first column records the time taken for a gather-based gridding threaded C++ CPU code which I wrote using OpenMP for testing the correctness of the GPU results. The CPU code has been timed on a quad-core Intel i5-750 running at 2.66 GHz. The data being gridded was chosen to be in a spiral distribution which is fast and simple to generate yet shares some features characteristic to tracks of u, v plane sample points of an earth rotation synthesis interferometer. Figure 6.5 shows an image of the results of gridding such data on the GPU using a Gaussian kernel.

The results in Table 6.1 clearly demonstrate both the need for pre-sorting the data into sub-grid bins and the effectiveness of optimising to avoid global memory accesses.

Optimisation	Time taken (ms)
CPU gridding	294
No sub-grid pre-sorting, no optimisation	1127
Sub-grids no optimisation	62.3
Caching sub-grid pixels	35.8
Caching sub-grid data	18.1
Convolution function look-up table texture memory	17.4

Table 6.1: Optimisations of the GPU gridding kernel function. Times are shown for gridding 100,000 data points onto a 4096 by 4096 pixel grid using, a convolution function of radius 7 grid pixels, oversampled by 100 points per pixel in the look-up table.

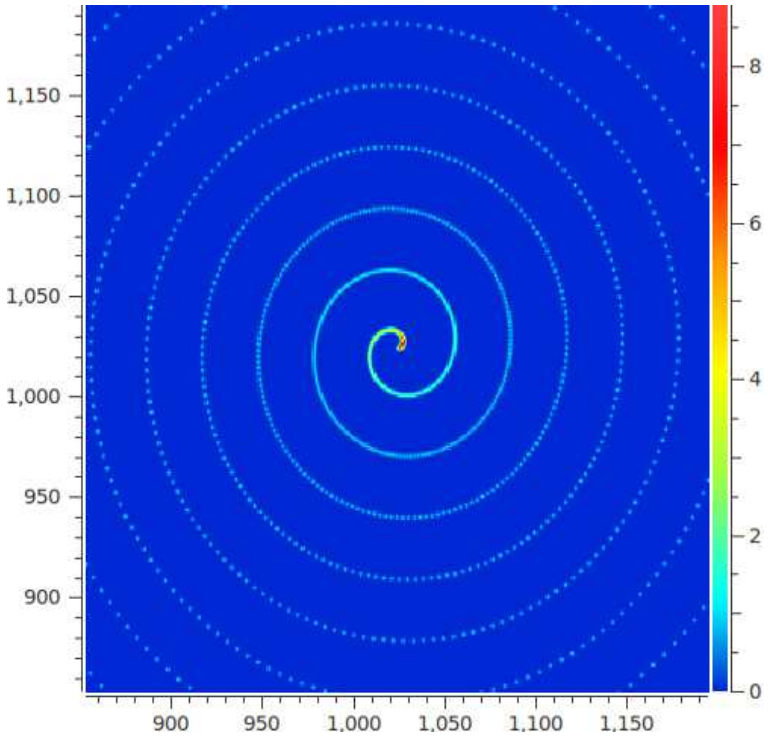


Figure 6.5: Image showing the result of gridding test data on the GPU. The image shows a zoomed portion of the central area of the grid generated from data in an Archimedean spiral gridded using a Gaussian convolution function.

Putting aside the case where no sub-grid bins were used and each thread is forced to loop over the entire data set, memory optimisations which reduce global memory access successfully improved the performance by a factor of ~ 3.5. For a grid size of 4096 by 4096 pixels, gridding data using a convolution function of radius 7, the GPU outperforms the CPU with a speedup of ~ 17. While this is a modest speedup compared to the theoretical processing power difference of the GPU and CPU being used to run the code, it should

be noted that this algorithm far too low arithmetic intensity for the GPU to demonstrate its full processing potential.

Another important consideration of any code is the scaling to different input parameters. Figures 6.6, 6.7, 6.8 and 6.9 compare scaling of the most optimised version of my GPU gridding code with the CPU implementation.

Figure 6.6 shows scaling with number of input data points. As expected for the gather-based algorithm, which scales as $O(N^2M)$ (for M data points and N grid pixels), both the CPU and GPU have approximately linear increase in processing time with increasing numbers of data points. Note however that the gradient for the GPU is flatter for small numbers of data points which can most likely be attributed to the cost of instantiating a large number of threads which have little to do other than memory accesses. In fact for very small numbers of data points, where the grid is extremely sparse I found the GPU to perform slower than my CPU implementation.

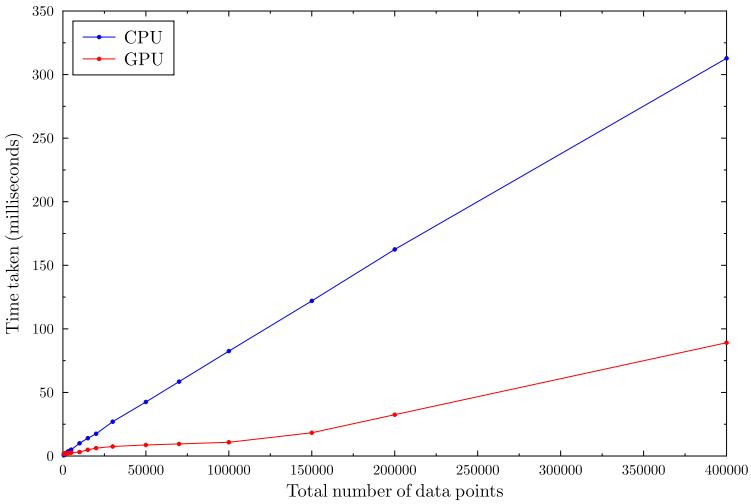


Figure 6.6: Gridding scaling time verus total number of data points. For a grid size of 2048 by 2048 pixels and a convolution function of radius 5 grid pixels (oversampled by 100 points per pixel in the lookup table).

Figure 6.7 shows the scaling as the width of the convolution function is increased for 50,000 initial data points arranged in an Archimedean spiral and a grid of 2048 by 2048 pixels. This has two effects. By increasing the number of data points per bin,

corresponding to a larger data halo on each sub-grid, the number of evaluations that have to be made per grid pixel increases. This is shown by the black curve in the figure. Also, the time taken to access values in the convolution look-up table increases as the number of entries, in proportion to the square of the width of the gridding function. As shown by the scaling on the CPU (blue curve) this algorithm is particularly susceptible to poor scaling with convolution function radius. The GPU however does not show this effect as it is masked by the huge degree of parallelism given by this implementation. It should be noted that this is rather beneficial to gridding with w-projection convolution functions as their radius increases with w, and for large fields of view it is not uncommon to require wide kernels.

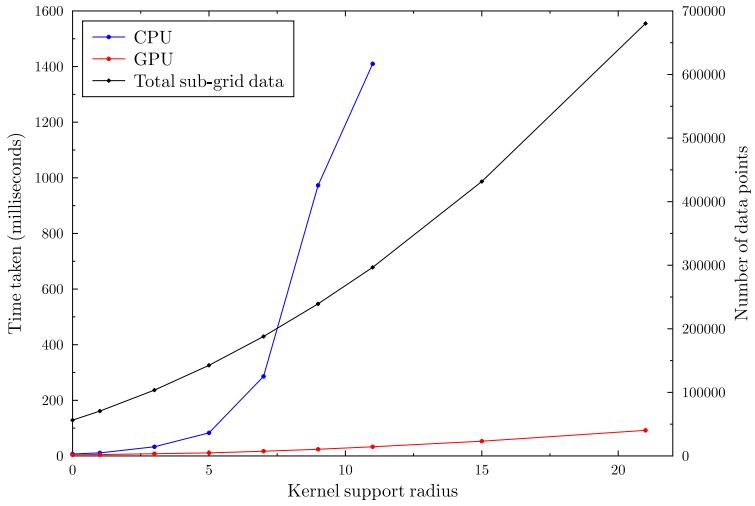


Figure 6.7: Gridding scaling with convolution function support radius. Timings are for 50,000 data points and a grid size of 2048 by 2048 pixels. Scaling results for the CPU beyond a radius of 11 are omitted due to the extremely poor scaling of this implementation.

Figures 6.8 and 6.9 shows the scaling with grid size for both the GPU and CPU. While the number of grid pixels increases as the square of the grid size, as the data has be pre-sorted in sub-grids, the number of data points that have to be evaluated per grid pixel decreases accordingly. This is clearly seen in the figures as the processing time does not increase as the square of the grid size but at a much lower rate that can be attributed to extra thread initialisations, and empty loop evaluations (for the CPU case) required

for larger grids.

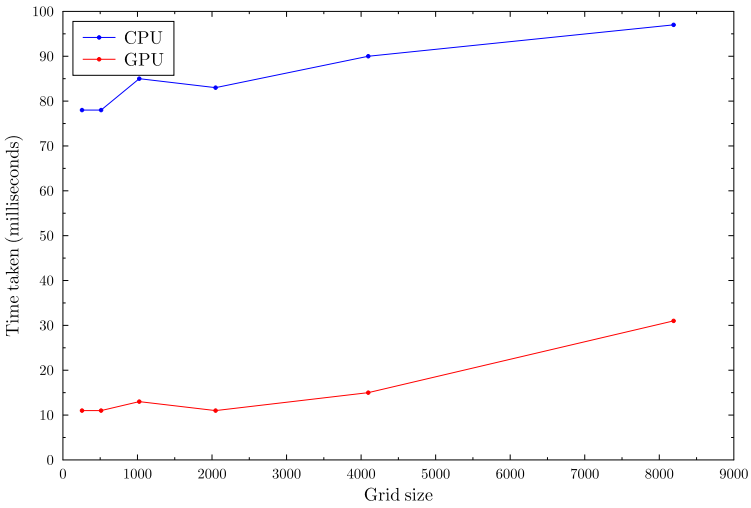


Figure 6.8: Grid scaling with grid size. Timing are for 50,000 data points and a convolution function of support radius 5 grid pixels.

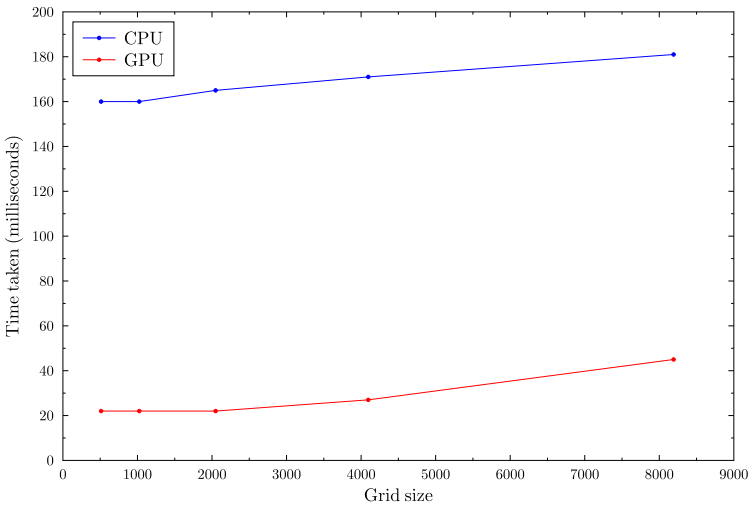


Figure 6.9: Grid scaling with grid size. Timing are for 100,000 data points and a convolution function of support radius 5 grid pixels.

6.2.4.2 Future Work

The results of the gather-based GPU gridding implementation in Section 6.2.4.1 raise a number of interesting directions for future work gridding data on the GPU.

While I managed to achieve a reasonable performance gain, $\sim$3.5, in optimising the gridding kernel function it less than what might be expected given the theoretical band-

width difference of global and shared memory. This can most likely be explained by the combination of poor arithmetic intensity of the gridding kernel. Typically, many threads do very little other than a conditional evaluation of the distance of a number of data points from their grid location. Therefore, in order to avoid evaluating more distances than needed, without decreasing the sub-grid size, it might advantageous to sort the sub-grid bins in coordinate space prior to gridding. Sorting, like putting data into sub-grid bins is difficult to optimise for the GPU however for small numbers of data points in a sub-grid bin a reasonably optimised implementation may be possible.

With the latest generation of GPU architecture (which was not unfortunately not available while writing gridding code) there is now support for floating point atomic operations. This offers a very interesting possibility to gridding. By using a scatter-based algorithm within each sub-grid, made possible using atomic operations in shared memory, conditional evaluation of distances of data points from the grid can be avoided entirely. In addition, scatter-based gridding would allow more optimised memory access patterns when using the gridding function look-up table.

Another area that might be investigated, is poor load balancing between threads as a result of an uneven distribution of data on the grid. In order to alleviate this problem one might consider investigating the use of variable sized sub-grids to match the density of data in the grid. While this poses a number of difficulties in thread assignments, given the sparse nature of the spacial Fourier plane coverage of many telescopes this is well worth future investigation.

6.3 Generation of w-projection convolution kernels

Gridding data on the GPU described in the previous sections of this Chapter could be used to interpolate any type of 2-dimensional data onto a grid. While imaging interferometric

radio telescope data, gridding is a considerable proportion of the compute time so there is an immediate clear advantage over the traditional CPU approach. Further advantage can be made by carefully considering the convolution function used to interpolate data onto the grid.

So far the gridding method I have described makes use of a convolution function that is passed to the gridding function as a pre-computed look-up table. While for narrow-field imaging it is sufficient that this function is a small, real, symmetric distribution chosen to suppress aliasing in the image plane. If however this function is chosen such that it acts to negate the w-dependent exponential term in the 3-dimensional interferometer response, shown in Equation 2.8, gridding the data by convolution also allows the possibility of high dynamic range imaging of wide-field, non-coplanar Fourier plane data samples. This is the w-projection algorithm (Cornwell et al. (2005)) described in Chapter 2.

Unlike the simple convolution functions that are used in imaging narrow fields of view, convolution functions used in w-dependent data need to be evaluated for each observation corresponding to the range of values of the w coordinate in the data. As a result kernel evaluation becomes a non trivial step in imaging data and worth considering for evaluation on the GPU.

6.3.1 Algorithm and implementation

As described in Section 2.3, the w-projection kernel is defined in terms of an image plane phase term,

$$G(l, m, w) = e^{-2\pi i[w(\sqrt{1-l^2-m^2}-1)]}\,, \tag{6.5}$$

where l and m are direction cosines along the tangent plane of the observation phase centre. As this function does not have an analytical u, v plane description, in order to evaluate the convolution kernel used for gridding the following steps are required:

1. Evaluate the image plane function, $G(l, m, w)$ on a grid defined such that the range of l and m results in the desired u, v plane sampling of the function ($\Delta u = 1/N\Delta l$, $\Delta v = 1/N\Delta m$, for N pixels in the image plane function).

2. Multiply the image evaluated in step-1 by a radial taper function to control aliasing.

3. Fourier transform the resultant image plane function to obtain the desired Fourier plane w-dependent gridding convolution function.

4. Normalise this u, v plane function and reshape at a suitable cutoff level to limit its size.

Step-1 and -2 is essentially a simple function evaluation which can be implemented on the GPU by assigning one thread to each pixel. As pixels in the function can be evaluated independently, this gives a massive amount of trivial parallelism. As GPUs have a set of special intrinsic functions for evaluations of sines and cosines evaluating large numbers of trigonometric functions as is required for this functions, it is extremely fast on the GPU. Listing 6.4 shows a CUDA kernel used to evaluate this function. When evaluating the image plane function it is necessary to ensure that it is wide enough in l, m space such that the u, v plane response, after taking a Fourier transform in step-3, has pixels separated a number of times closer together than the pixel size of the grid. This ensures there is no need for interpolation of the look-up table in the gridding step. As generating such a large image plane function is wasteful with a function that tapers to zero in practice I choose to evaluate a small image plane function, e.g. 256 pixels by 256 pixels chosen to cover an l, m space corresponding to the final field of view of the image and then zero pad this by a factor corresponding to the desired over-sample in the u, v look-up table. Figure 6.10 shows a typical padded l, m plane function evaluated from steps 1 and 2 of the algorithm.

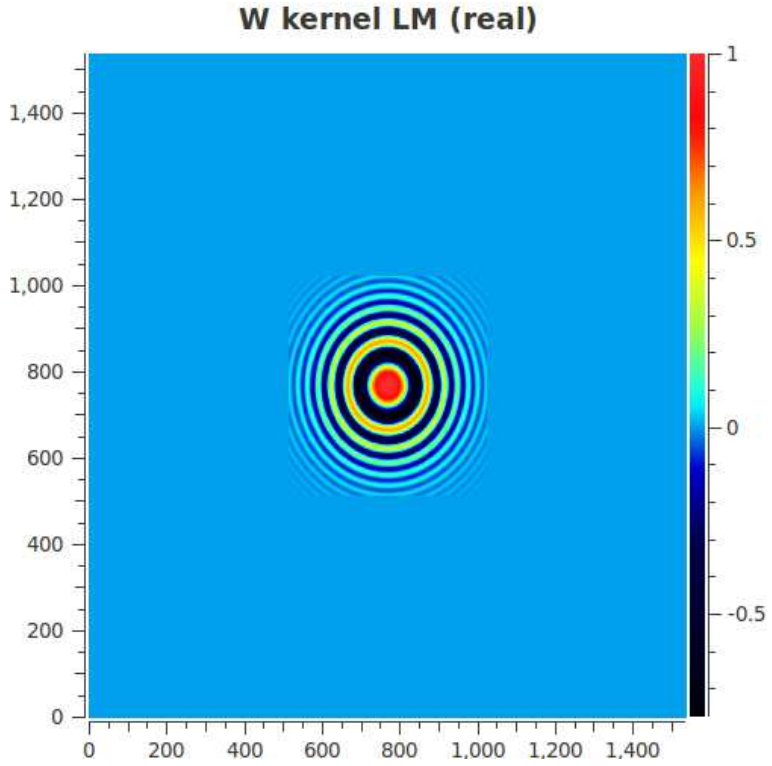


Figure 6.10: .
Example image plane w-projection convolution function. This function is Fourier transformed to give the u, v w-projection gridding kernel.

Step-3 involves taking the complex to complex fast Fourier transform of the image plane function generated in the previous steps. I have used the CUDA FFT library for this, which is a highly optimised FFT library for the GPU. A more detailed discussion of the performance of using this comparing it to the performance of FFTW a highly optimised CPU FFT library can be found in the next section. Figure 6.11 shows the results of taking the Fourier transform of the image plane kernel in Figure 6.10 on the GPU.

The w-dependent convolution obtained from step-3 slowly tapers towards zero towards the edge of the function. The closer the value of w is to zero, the more tapered the u, v plane w function becomes, to the point where at w=0 it is simply the Fourier transform of the image plane taper function used to suppress aliasing and therefore a few grid pixels in width. As a result in order to avoid having to compute the convolution of data

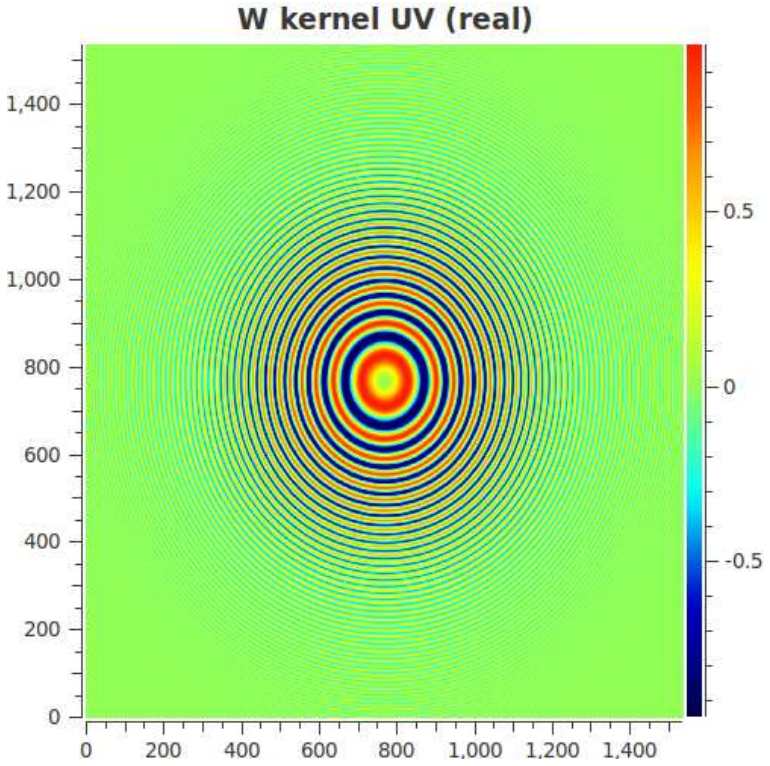


Figure 6.11: .
Example Fourier plane w-projection convolution function. This function is formed by taking the Fourier trasnform of Figure 6.10.

points with a mostly zero valued function the u, v plane kernel is normalised and resized according to the point where the absolute value of the function drops below a specified taper level. If this value is chosen to be too small the convolution kernels generated are large and grid processing becomes expensive and if the value is too large gridding functions become overly truncated at the edges leading to spurious image plane artifacts. While it is hard to quantify the optimal level for this cutoff, I found a value of 10^{-4} gave a reasonable compromise between the two regimes. As the w dependent gridding kernels are symmetric, normalisation is simply achieved by dividing by the absolute value of the complex amplitude at the centre of the evaluated function. This division is kept on the GPU by using a simple kernel function which uses one thread per pixel to multiply by the inverse of the normalisation amplitude. Finding the cutoff amplitude on the GPU is rather more difficult. After experimenting with a number of techniques of finding this

value on the GPU, I found it was far more efficient to copy a line of pixels from the evaluated function corresponding to a line along the positive u axis. By iterating along these pixels and looking for the pixel where the amplitude drops below the specified cutoff level this operation is easy and fast on the CPU whereas on the GPU it would require relatively complicated inter thread communication. Once the cutoff pixel is found the rest of the u, v function is discarded and the remainder is passed onto the gridding algorithm.

6.3.2 Results

Forming the w gridding convolution functions follows a relatively straightforward series of steps. In this section I present a comparison of the performance on the GPU in comparison to threaded CPU code which I have written in using OpenMP. All results are timed on a NVIDIA GeForce 275 GTX GPU and an Intel i5-750 2.66 GHz CPU.

The first stage of the algorithm prepares an image plane gridding function. Table 6.2 shows the relative performance of this stage for two slightly different approaches on the GPU and a reference CPU implementation. The function has been evaluated with 512 pixels by 512 pixels and a factor of 5 in zero-padding width. While evaluation of the function is extremely fast in all cases, typically a large number of w gridding functions have to be generated to grid the data for each data set so that any optimisations are amplified. One major optimisation in the GPU implementation comes from the way zero-padding of the function is implemented. This can either be achieved by padding generating the l, m plane function and then using a number of memory copy operations to pad with a zero valued array, or alternatively the function can be evaluated directly in place in the padded zero initialised memory space. The latter of these two is found to be faster by a factor of ~ 3.6 as the memory copy operations, even in fast GPU DRAM, is slower than the cost incurred in a slightly sub-optimal memory access pattern. Another effect on potential optimisation which I investigated for this evaluation was the effect

of using intrinsic functions on the GPU for evaluating the trigonometric functions and the exponential term in the Gaussian taper. This class of functions, seen in Listing 6.4 prefixed by a double underscore, are executed on the GPU's special function unit and trade some numerical precision for improved evaluation performance. I found a small amount of performance gain, $\sim 10\%$ by using these functions in the GPU kernel function and could find no detrimental impact of the lower numerical precision. As a result I have used these functions in my GPU code but the performance gain in their use is marginal for this case compared to other steps in the algorithm. Both of the GPU implementations are, as expected, faster than the CPU as this function evaluation is the sort of trivial parallel algorithm which GPUs excel at.

Implementation	Time taken (ms)
CPU	30.7
GPU padding by memcpy	4.0
GPU generation in place	1.1

Table 6.2: Table showing the time taken to generate the image plane w-projection convolution function. The function is evaluated for 512 by 512 pixels and zero padded by a factor of 5 times its original width.

After evaluation of the image plane function this is converted to the u, v plane by a fast Fourier transform and then resized according to truncating the resulting function for values smaller than a specified cutoff. The cost of this step is dominated by the Fast Fourier transform which I implemented on the GPU using the CUDA FFT library and on the CPU using the FFTW API. Table 6.3 shows the time taken to generate the final u, v plane gridding convolution function on the CPU and GPU.

Implementation	Time taken (ms)
CPU	191.0
GPU	9.4

Table 6.3: Table showing the time taken to generate the a u, v plane w-projection gridding convolution function.

This result therefore shows that generating the w-projection gridding convolution functions is very well suited to execution on the GPU. This is ideal considering the added benefit of avoiding the extra memory transfer to the GPU that generating the gridding function directly in GPU memory provides. Typically, forming an image from 240 MHz GMRT data requires gridding the visibility data split into approximately 64 independent w-planes each with their own gridding kernel. Therefore gridding convolution function generation on the GPU alone saves over 10 seconds in generating a 'dirty' image.

6.4 Image formation: The FFT on the GPU

The final step in recovering a map of the sky brightness distribution is to apply a fast Fourier transform of the gridded visibility. As the sky brightness map is by definition real, the visibility grid will be Hermitian complex so that a complex to real fast Fourier transform algorithm can be used. Since the CUDA FFT library (NVIDIA Corporation 2010) can be used for this purpose and is highly optimised for my imaging code I have chosen to use this rather than developing my own GPU Fourier transform.

While the API for the CUDA FFT library is reasonably simple, a number of considerations have to be made when constructing the dirty image from the visibility grid. Firstly, in order to make use of the real to complex transform to make an image of the sky of N by N pixels a complex visibility grid of $N/2+1$ by N pixels must be supplied to the function. This has implications for gridding as only half the grid needs be generated reducing the gridding time and memory to store the grid by a factor of two. Secondly, the FFT algorithm returns an image with the zero frequency component, or phase centre at the edges of the image and therefore some rearrangement has to be made to compensate for this, and this is commonly known as an FFT shift operation. This can be achieved by a rather expensive memory copy reordering or by applying a phase shift to the grid

and image pixels. As memory operations are extremely expensive in comparison to independent per pixel multiplication operations, this step was performed on the GPU using a simple kernel which multiples each pixel, i, j, by $-1^{(i+j)}$. The results of the difference in execution time for these two approaches are shown in Figure 6.12 where it should be noted that if the memory copy approach is taken for large images this then approaches the time taken for the FFT algorithm itself.

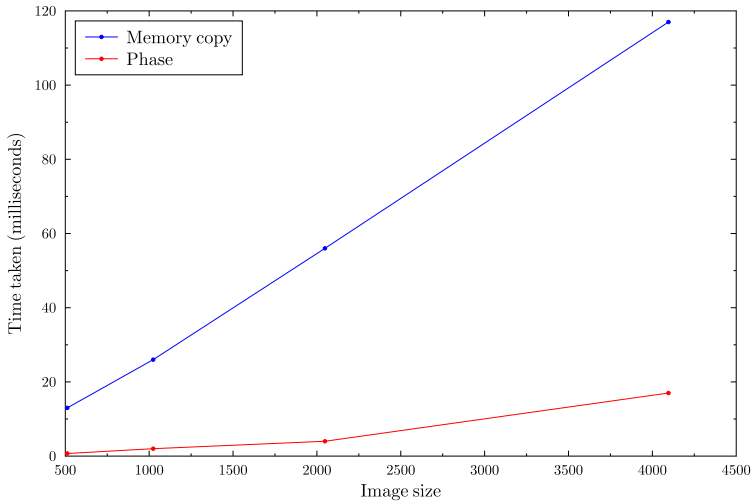


Figure 6.12: Comparison of the performance memory copy rearangement and phase shifting in order to centre the image produced by the FFT.

For the imaging code which I have developed for this thesis, described in Chapter 5, I have implemented versions of the FFT of the visibility grid by both using the CUDA FFT library and for comparison and verification purposes the FFTW library (Frigo & Johnson 2010), one of the most respected high performance CPU-based FFT libraries. Figure 6.13 shows a performance comparison of these two libraries measured from my code. For fair comparison both methods include the execution time of the re-centering of the FFT and for the case of the GPU, transferring the data to and from the GPU memory.

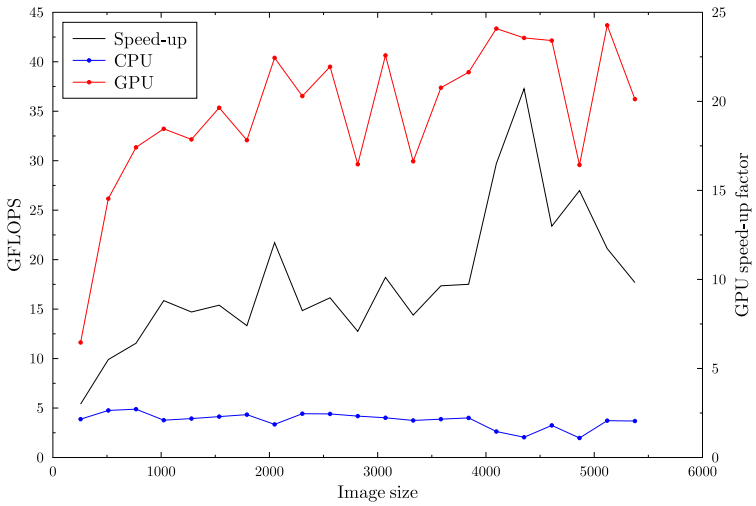


Figure 6.13: Comparison of the performance of the CUDA FFT library and FFTW. Note that the value of GFLOPS given is evaluated using the execution time which includes re-centering of the FFT by multiplying each pixel by -1^{i+j} and for the case of CUDA FFT transferring data to and from the GPU.

Listing 6.3: A simple CUDA kernel for gather-based gridding on the GPU.

```
__global__ void grid_subgrids(unsigned nCells, unsigned * sub_grid_count,
    unsigned * sub_grid_start, float * sub_grid_u, float * sub_grid_v,
    float2 * sub_grid_amp, unsigned cOversample, unsigned cSupport,
    unsigned cSize, unsigned cCentre, float2 * convFunc, float pixelSizeUV,
    unsigned gSize, unsigned gCentre, float2 * grid)
{
    // Sub-grid cell index for the thread block.
    int iCell = blockIdx.y * nCells + blockIdx.x;

    // Number of data points in the sub-grid cell bin.
    int nData = sub_grid_count[iCell];

    // No data? Nothing to do!
    if (nData == 0) return;

    // Global (grid) pixel index for the current thread.
    int iuGrid = blockIdx.x * blockDim.x + threadIdx.x;
    int ivGrid = blockIdx.y * blockDim.y + threadIdx.y;

    // Check pixel index is inside the grid!
    if (iuGrid >= gSize || ivGrid >= gSize) return;

    // Pointers to the sub-grid data bin.
    int iStart = sub_grid_start[iCell];
    float * u = sub_grid_u + iStart;
    float * v = sub_grid_v + iStart;
    float2 * amp = sub_grid_amp + iStart;

    // Grid index for the current thread.
    int gIdx = (ivGrid * gSize) + iuGrid;

    // Convolution function radius in grid pixel units.
    float cRadius = (float)(2 * cSupport + 1) / 2.0f;

    // Grid space pixel id for the current thread.
    int uGrid = iuGrid - (int)gCentre;
    int vGrid = ivGrid - (int)gCentre;

    // Initialise grid pixel to zero.
    grid[gIdx] = make_float2(0.0f, 0.0f);

    // Loop over data in the cache accumulating it onto the grid.
    for (int i = 0; i < nData; ++i)
    {
        // Distance of the data point from the grid pixel being processed.
        float uOffset = (float)uGrid - u[i];
        float vOffset = (float)vGrid - v[i];

        // If data point is out of range of grid pixel continue.
        if (fabsf(uOffset) > cRadius || fabsf(vOffset) > cRadius)
            continue;

        // Find index into convolution function look-up table.
        float uDelta = uOffset * (float)cOversample;
        float vDelta = vOffset * (float)cOversample;
        int iuKernel = (int) round(uDelta) + cCentre;
        int ivKernel = (int) round(vDelta) + cCentre;
        int cIdx = ivKernel * cSize + iuKernel;

        // Accumulate to grid pixel.
        float2 cAmp = convFunc[cIdx];
        grid[gIdx].x += cAmp.x * amp[i].x - cAmp.y * amp[i].y;
        grid[gIdx].y += cAmp.y * amp[i].x + cAmp.x * amp[i].y;
    }
}
```

Listing 6.4: GPU kernel for evaluation of the image plane w-convolution kernel.

```
__global__ void generate_image_plane_w_func(float2 * func, unsigned size,
    float w, float pixelSize, float sigma2)
{
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int iy = blockIdx.y * blockDim.y + threadIdx.y;

    float iCentre = size / 2.0f;
    float x = (float)ix - iCentre;
    float y = (float)iy - iCentre;

    // Gaussian taper.
    const float x2 = x * x;
    const float y2 = y * y;
    const float taper = __expf(-(x2 + y2) / (2.0f * sigma2));

    // w-convolution function.
    const float l = x * pixelSize;
    const float m = y * pixelSize;
    const float r2 = (l * l) + (m * m);
    const float phase = -2.0 * M_PI * w * (sqrtf(1.0f - r2) - 1);

    float im, re;
    __sincosf(phase, &im, &re);

    func[iy * size + ix].x = re * taper;
    func[iy * size + ix].y = im * taper;
}
```

Chapter 7

Demonstration of *WFIT* on simulated and real data

This chapter explores the results of using the GPU imaging code *WFIT*, described in chapter 5, which I developed for this thesis. I analyse the effectiveness of imaging both synthetic data and low frequency observations from the GMRT.

7.1 Introduction

While the main goal of this thesis has been to investigate and demonstrate the suitability and advantages of writing GPU code for a number of aspects of synthesis imaging, developing code in isolation of the data for which it is targeted is never a good idea! For this reason I first present the results of imaging a synthetic data set, which I have used to test and validate the code by comparing my results with images generated using the long-standing and well tested AIPS package. Following this I present some results of imaging wide field data observed with the GMRT. The GMRT data set was chosen as its field of view is sufficiently large that there is a clear need for 3-dimensional imaging corrections which the w-projection algorithm provides.

7.2 Simulated data

Validation and testing is a vital component in any rigorous code development process. Towards this aim I made images using a synthetic model data set, the results of which are presented in this section. This data set serves two purposes: firstly it contains a known source distribution so the result of imaging can easily be verified, and secondly it is free from noise, radio frequency interference, and other corruptions which might otherwise make validation difficult or at least ambiguous.

In order to test the correctness of all aspects of my imaging code, I have used the synthetic data set in a number of ways. By imaging with a traditional 2-dimensional algorithm and comparing the results to *AIPS*, this provides a direct method of verifying my code against a well established and tested imaging package as well as illustrating the need for 3-dimensional correction of wide field images. I have also compared the results of the w-projection implementation in my code with images made using a traditional (yet more computationally expensive) facet-based 3-dimensional correction algorithm, implemented in AIPS, to assess the quality of images generated by these two methods.

In order to minimise any effects of bandwidth smearing (resulting from breaking the narrow band approximation of the Fourier inversion) which can be confused with 3-dimensional imaging errors, for each of the tests I have chosen to image as narrow a frequency range of the data set as possible. As the simulated observation contained enough visibility data points in a single channel of the observed frequency band to produce an image sensitive enough to see all of the sources present in the model I selected the central frequency channel of the band for all of the images presented here.

I also chose to ignore any weighting or tapering of the visibility data. While a degree of manipulation of the PSF by tapering the visibility amplitudes according to their local density or position can be beneficial in producing high quality images I wanted to keep

these possible enhancements separate from 3D imaging effects and as a result natural weighting of UV data points (Briggs (1995)), where no modifications were performed for local density, was used throughout.

7.2.1 Description of the data

The synthetic data was generated by Dr Ian Heywood using the *AIPS* simulation tasks UVCON and UVMOD. The simulation was based on the VLA D compact configuration observing at a Declination of 40.0 degrees (i.e. near the zenith at the VLA) for twelve hours at a frequency of 600 MHz. In order to avoid limiting the field of view to a fraction of a degree, which would otherwise be the case for 25 m antennas observing at 600 MHz, primary beam effects been ignored in favour of an isotropic antenna response. Four point sources were added to the field using UVMOD at a range of positions that was both simple to interpret and provided enough separation from the phase centre to ensure the need for 3-dimensional imaging corrections. No noise was added to the model as this would only serve to confuse the imaging, and no calibration was required as the simulated data contains no phase or amplitude errors. The source positions used to generate the model visibility set are given in Table 7.1.

Source ID	Relative R.A. (deg)	Relative Dec. (deg)	Intensity (Jy)
0	0.0	0.0	1.0
1	0.0	2.0	1.0
2	0.0	3.0	1.0
3	3.0	0.0	1.0

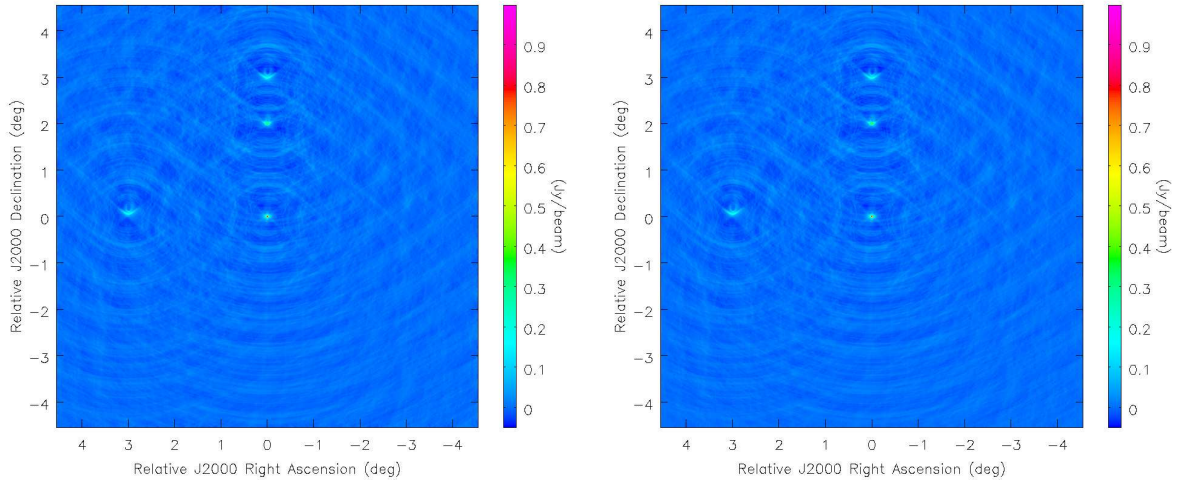
Table 7.1: Source positions in the sky model used to create the synthetic data set. Positions in the table are specified in units relative to the phase tracking centre.

7.2.2 Results

7.2.2.1 2-dimensional imaging: comparison to *AIPS*

Figure 7.1 compares the results of dirty images produced by *AIPS* and my imaging code, *WFIT*. Both images were made using a Gaussian times Sinc convolution kernel ignoring the w visibility coordinate and as such contain 3-dimensional correction errors. As expected the images are in good agreement. The difference in power of $\sim 1\,\%$ towards the edges of the field can be most likely attributed to a combination of effects arising from the rounding method used in gridding, which causes a slight variation in side-lobe positions of the point spread function, and the instability of single precision floating numerics when correcting for the image taper. At the edge of the field, the image plane taper correction applied is large ($\sim 10^5$) and therefore differences in rounding from using single precision numerics of order 10^{-7}, will manifest themselves as an amplitude error of around 10^{-2}. While this effect is not a particular concern for the analysis of data presented in this thesis, it should noted that it will ultimately limit the dynamic range that can be achieved as a function of radius in the final deconvolved images. It is however be possible to mitigate this amplitude error by careful choice of a convolution function with a more favourable image plane taper or by the use of double precision numerics. Overall, this test shows that when using standard 2-dimensional imaging gridding kernels my *WFIT* code performs in good agreement to both the results expected from the given model sky and also to the output of the well tested software package *AIPS*.

What is immediately clear looking at the dirty images in Figure 7.1 (a) or (b) is the smearing of the sources away from the field centre. This is a direct result of applying no 3-dimensional imaging correction when making the image. This effect can also be clearly seen to worsen for sources further away from the phase centre.



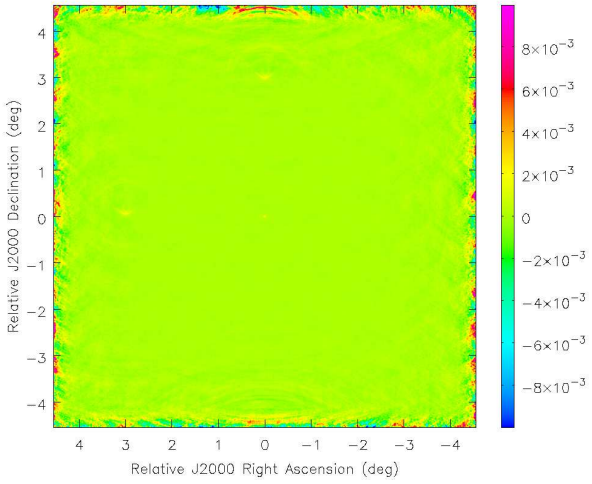


Figure 7.1: Comparison of standard imaging (without 3-d correction) between (a) *AIPS*, and (b) *WFIT*. (c) Shows the difference between the plots, plot (a) - plot(b).

7.2.2.2 3-dimensional imaging: w-projection and facets.

Figure 7.2 shows the results of imaging the same simulated data using the w-projection gridding algorithm from *WFIT* and the faceting algorithm in *AIPS*. The asymmetry in the field around the phase centre is because I have chosen to zoom in on the area of the image plane containing the sources. Figures 7.2 (a) and (b) clearly demonstrate that both algorithms are successful in correcting for the smearing of sources away from the phase tracking centre that occur with the 2-dimensional imaging technique (Figure 7.1). In both

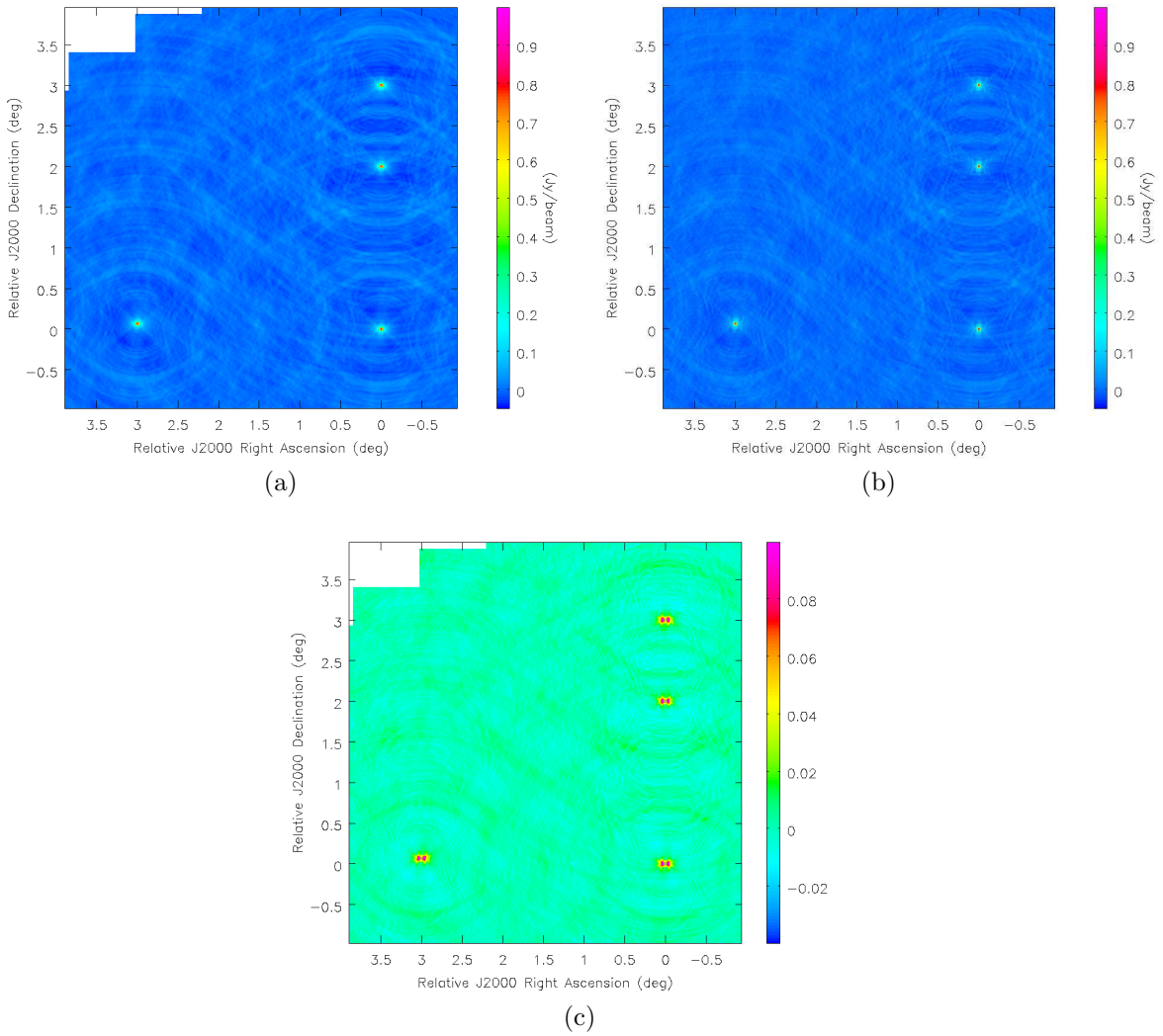
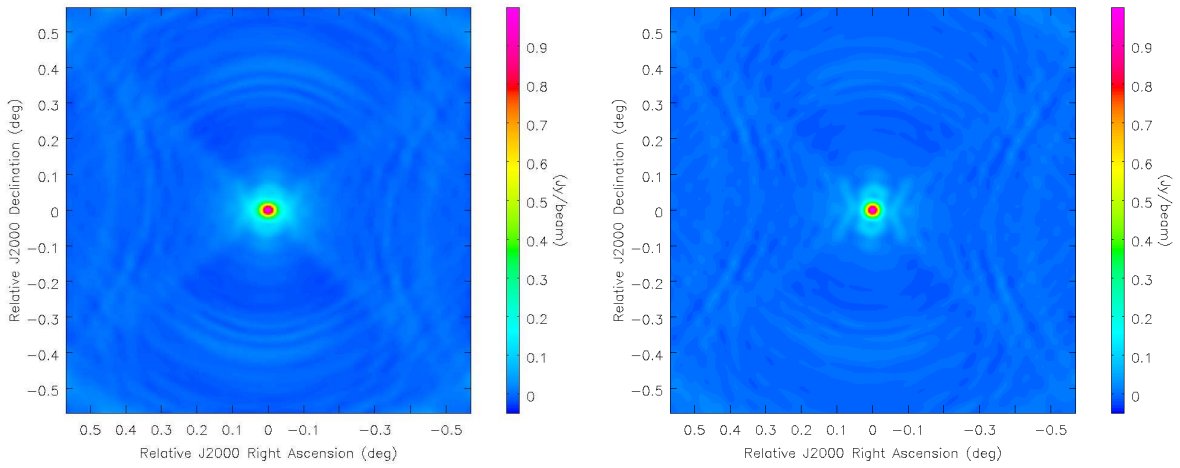


Figure 7.2: Comparison of 3-dimensional imaging. (a) Faceting with *AIPS*, (b) *w*-projection using *WFIT*. (c) difference (a)-(b)

cases the peak amplitudes of the sources from the sky model, which were all 1.0 Jy, are recovered correctly with the the peaks in the dirty maps measured to be 1.0 ± 0.01 Jy.

The difference of the images, seen in Figure 7.2 (c), has a double peaked pattern around each source. This can be explained by the difference in shape of the inner portion of the point spread function, shown in Figure 7.3, which is consistent with using different gridding kernels. The zero at the center of the double peak shows that the sources have been recovered in the same positions with the same amplitude in both images. It should be noted that while the results presented here do not perform any post processing of the



(a) (b)

Figure 7.3: Point spread functions for figure 7.2 (a) and (b). (a) PSF of centre facet. (b) PSF at the field centre for the w-projection image.

dirty images, a clear and symmetric point spread function pattern around each of the sources in the dirty image gives a good indication that deconvolution using the CLEAN algorithm is likely to be extremely successful.

Obtaining a metric to assess the effectiveness of the 3-dimensional correction in a meaningful way without deconvolution of the resultant images is tricky. However, given knowledge of the sky model used to generate the simulated data, I decided to compare the results using an effective Strehl ratio. For optical telescopes, the Strehl ratio is used to establish how close a system is to perfect, i.e. diffraction limited imaging and is a key measure of the improvement of imaging capability of a telescope's use of adaptive optics. The metric is defined as the ratio of a peak amplitude in the observed image to that of the theoretical maximum of the peak. Table 7.2 presents this measure for each of the four sources imaged from the simulated data set. While this method provides a reasonable estimate of the improvement given by the 3-dimensional imaging correction, a degree of care has to be taken in its interpretation as for some cases, such as for last point in the table, the measurement of the peak amplitude of a source in an uncorrected image is extremely difficult and potentially biased as a result of the smearing effect. Without further

Distance from	Peak amplitude measured from:		
phase centre (deg)	2-d imaging (Jy)	w-projection (Jy)	faceting (Jy)
2.0	0.468	0.988	0.988
3.0	0.206	1.001	0.986
3.0	0.188 (0.206)	0.995	0.987

Table 7.2: Peak amplitudes of sources away from the phase tracking centre for each of the imaging methods. The amplitude of the last source, taken from an image without 3-dimensional corrections, is 0.188 at the source position but 0.206 at peak as a result of asymmetrical smearing. Sources are listed in the same order as they appear in Table 7.1.

image processing though deconvolution, it is very hard to generate any other useful metrics that are not overly influenced by the effect of side-lobes of the point spread function. However, these results clearly establish that the w-projection algorithm, implemented on the GPU in *WFIT*, accurately reproduces an image of a known sky model and that the w-projection algorithm performs comparably to faceting in the sense of computational accuracy for 3-dimensional corrections. A potential pitfall of the faceting approach is the small edge errors introduced when stitching facets back together to construct the final mosaic image. To avoid this problem the size (and therefore number) of facets was chosen to be such that the RMS error introduced by joining facets was always equivalent or below that introduced by tabulating the w projection grid kernels at discreet values of w. An equally important consideration is the relative performance of the two methods which I now explore.

The images presented in this analysis so far in were constructed to be of size 4096 pixels by 4096 pixels. This was chosen in such a way that there were $\sim$5 pixels across the diameter of the main lobe of the synthesised beam and so that the field of view of the image was large enough to extend over all four sources in the model data. Imaging using the faceting approach was performed in *AIPS* using the SETFC, IMAGR and FLATN tasks in sequence. First SETFC was used to automatically specify the image plane facets needed cover the entire field given the range of the w coordinate in the data, which

resulted in 91 facets being allocated. The facets then imaged with the IMAGR task using a Gaussian times Sinc gridding kernel which has a radius of 3 grid pixels. This step resulted in 91 sub-images which are combined to form a single 4096 pixel by 4096 pixel dirty image using the task FLATN. Forming the dirty image in AIPS on an Intel i5-750 processor running at 2.66GHz took a total of 19.6 seconds with the majority of the time being taken in imaging the facets. It should be noted that while *AIPS* is well known to use highly optimised code developed over many years, it is a single threaded application and as such can only make use of one core of the CPU, so a speedup corresponding to the number of cores on the CPU might be expected if a multi-threaded implementation was available. The w-projection image was created using the GPU implementation in *WFIT*, described in Chapter 6. I chose to grid the visibility data-set which contained 252,720 samples in 64 separate planes separated evenly in $\sqrt{w}$ each with their own w dependent convolution function. 64 planes were chosen as this was found to generate an image devoid of 3-dimensional smearing effects and increasing the number of planes gave no appreciable improvement in the image. Sorting of the data set into bins according to each data point's w coordinate was done on the CPU by sorting the data in order of w using the native C++ quicksort algorithm which is known to be very efficient. This resulted in 64 data sets each containing between 100 and 10,000 visibility points which were then gridding by their respective w dependent convolution kernels. Kernels were generated on the GPU, as described in Section 6.3, and after truncating to a level corresponding to 10^{-4}, the convolution functions ranged in radii of between 5 and 23 grid pixels. Gridding of each plane was then performed in a serial fashion with the gridding implemented on the GPU as described in Section 6.2 and accumulating to a single 4096 pixel by 4096 complex UV space grid. Finally the grid was Fourier transformed to the image plane using a complex to real transform from the CUDA FFT library as described in Section 6.4, and multiplied by a Gaussian taper to correct for the image plane taper introduced by the kernel. The

total processing time for this was 0.61 seconds, including dividing the data set into w-planes, binning each plane into sub-grids required by the GPU algorithm and memory transfers between the GPU and CPU. The resultant speedup of my GPU w-projection algorithm over facet based imaging in *AIPS* was therefore found to be by a factor of ~ 32. Full details of the processing for each stage is shown in Table 7.3 below.

Processing	Time taken (seconds)
Dirty image using faceting in *AIPS*	19.6
Dirty image using w-projection in *WFIT*	0.61
Generating 64 w-Kernels	0.12
GPU gridding of the 64 planes	0.21
CUDA FFT of the 4096 by 4096 grid	0.07
Image plane taper correction	0.05

Table 7.3: Comparison of processing times for imaging the simulated data-set using *AIPS* and the w-projection algorithm in *WFIT*, implemented on the GPU.

7.3 GMRT data

7.3.1 Description of the dataset

This data-set consists of a 9 hour observation with the Giant Metrewave Radio Telescope (GMRT) at 240 MHz taken on the 25th October 2008; one channel of 62.5 kHz bandwidth was selected for the tests that are described in this section. Initial calibration, i.e. excision of radio-frequency interference and correction for amplitude gains via observations of 3C286 and correction for phase gains via observations of the nearby phase calibrator 1252+565, was performed in AIPS. Because the GMRT interferometer comprises antennas whose diameters are 45 metres, the primary beam at this frequency is a few degrees and as such 2-dimensional algorithms for imaging are insufficient to faithfully image all but the innermost region of the observation, closest to the pointing centre. Therefore, I now explore the effect that w-projection in *WFIT* and faceting in *AIPS* have on the imaged data far from the phase centre of the observation.

7.3.2 Results

As with the simulated data, I have imaged the GMRT dataset to produce dirty maps using the standard 2-dimensional method (by gridding with a Gaussian times Sinc kernel) and with 3-dimensional imaging corrections from the faceting algorithm in *AIPS* and the GPU based w-projection algorithm in *WFIT*.

The images were generated with 4096 pixels by 4096 pixels and a pixel size of 3.0 arcsec chosen so that the imaged field of view covers nearly the entirety of the primary beam from the 45 metre dishes. For both the faceted and 2-dimensional imaging a Gaussian times Sinc convolution kernel was selected with a radius of 3 grid cells, evaluated from a look-up table oversampled at 100 times per cell. The faceted image was generated using 212 image plane facets chosen by the SETFC task in *AIPS*. Generating the image using the faceting algorithm in the *AIPS* IMAGR task (which images each of the 212 facets as separate 512 by 512 pixel sub-fields by first phase rotating the visibility data for each facet and then applying a 2-dimensional gridding kernel) took 31.5 seconds on an Intel i5-750 2.66 GHz quad core processor. As *AIPS* is a single threaded application, and as faceting is a trivial parallel algorithm one might expect a ~ 4 times improvement if all the CPU threads could be utilised. Generating the image with no 3-dimensional correction by gridding first with a Gaussian-Sinc kernel and then taking the fast Fourier transform of the 4096 pixel by 4096 pixel grid took 0.382 seconds using the 2-dimensional imaging capability of my *WFIT* code using processing on the same CPU and using GPU accelerated gridding and FFT on an NVIDIA 275 GTX GPU, as described in Chapter 6. As for the synthetic data, the w-projection image was formed by gridding the visibility data separated into 64 planes separated evenly in $\sqrt{w}$ with their associated w-dependent kernels that ranged in radius from 5 to 33 grid pixels. This took 1.81 seconds using the GPU accelerations in both w-kernel generation, gridding and the FFT.

Unlike the relatively simple simulated data set however this is a rather complicated field with many sources of various intensity, and despite very careful calibration quite a lot of noise remains. A full view of the field is shown in Figure 7.5 but in order to demonstrate the effectiveness of the 3-dimensional correction provided by my implementation of w-projection I have chosen to show a zoom in on the three regions shown in the figure by numbered green boxes.

Looking at the expanded regions of the field seen in Figures 7.6, 7.7, and 7.8, there is clearly a distinct need for 3-dimensional imaging correction and both faceting and w-projection can each be seen to provide this to similar, good effect. Both 3-dimensional correction techniques reveal sources that are smeared out beyond all recognition in the image where 3-dimensional corrections are ignored. It should also be noted that after 3-dimensional image correction, the point sources appear to be convolved, as expected, with a regularly structured point spread function which would indicate deconvolution (the next step in the imaging processing) should be successful. This is particularly clear in Figure 7.8 (c) where a pattern matching very closely to the w-projection point spread function, shown in Figure 7.9, can be seen centred on the brightest source in the field. Also as seen in the simulated data facetting and w-projection generate slightly different point spread functions, as would be expected with using different convolution functions in gridding. The w-projection point spread function is more compact which is most likely a result of having a larger extent in the UV plane. This difference in point spread function is likely to also have a bearing on the slightly different apparent noise levels of the images, resulting from variation in the structure of the side-lobes of the respective point spread functions.

The histograms in Figure 7.4 show that both methods have noise correctly centred about zero which further demonstrates verification that the *WFIT* w-projection code is introducing no spurious noise artifacts. The standard deviation appears slightly lower for

w-projection (bottom plot in Figure 7.4) although without further processing it is difficult to draw any major conclusions from these subtle differences.

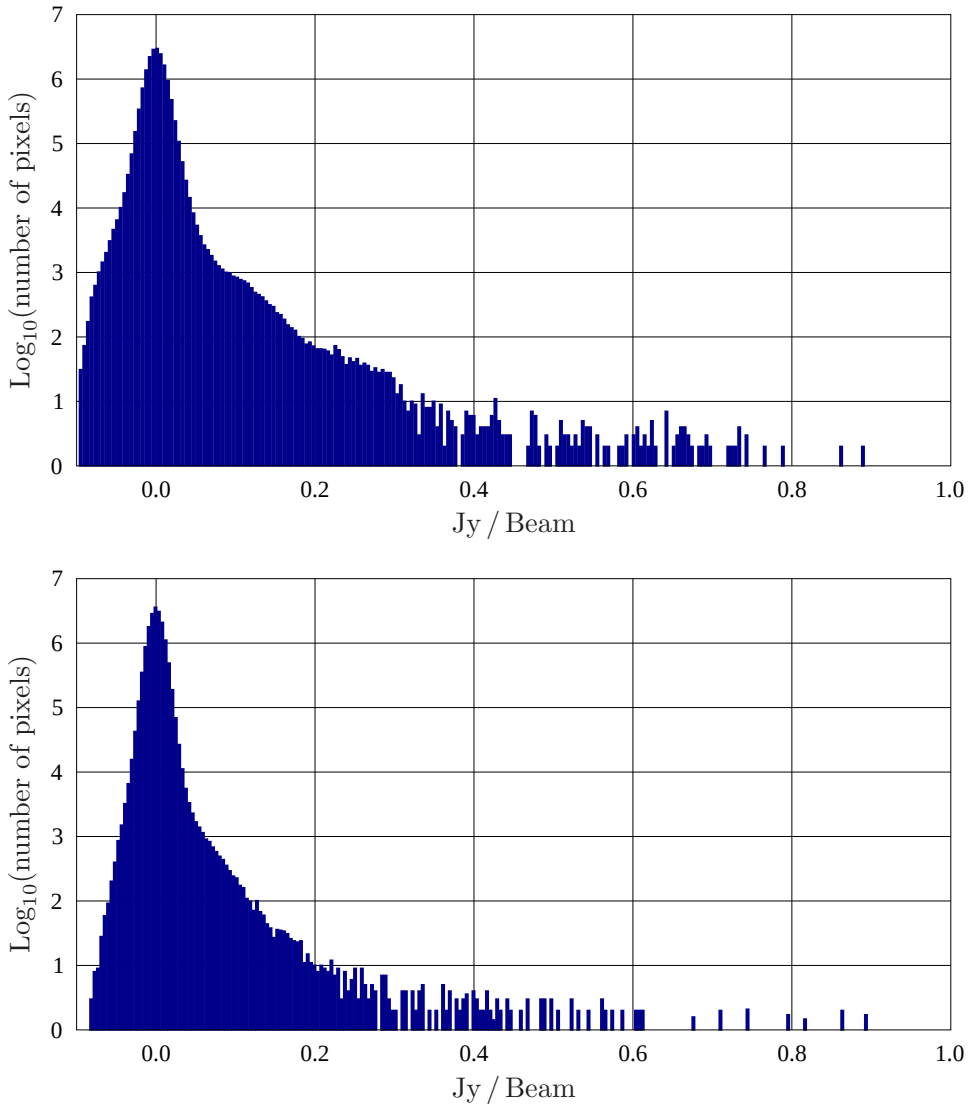


Figure 7.4: (top) Histogram of the pixel values from the facet image. (bottom) Histogram of the pixel values from the w-projection image.

As with the synthetic data, the w-projection algorithm successfully images this GMRT data in a way at least comparable to the faceting approach. With its clear improvement in processing time w-projection on the GPU, as explored in Chapter 6, is therefore very attractive.

In order to design a metric to assess the effectiveness of the 3-dimensional correction I

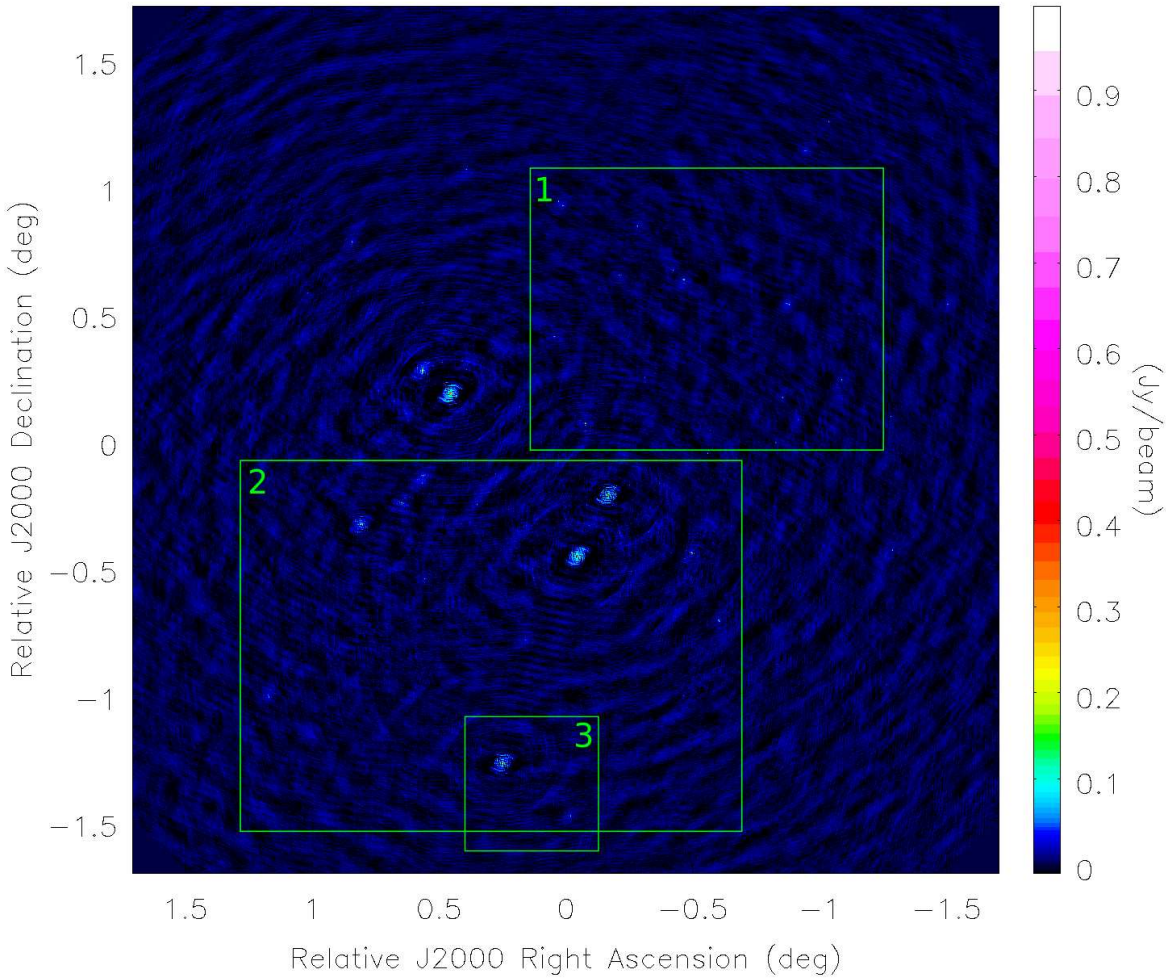
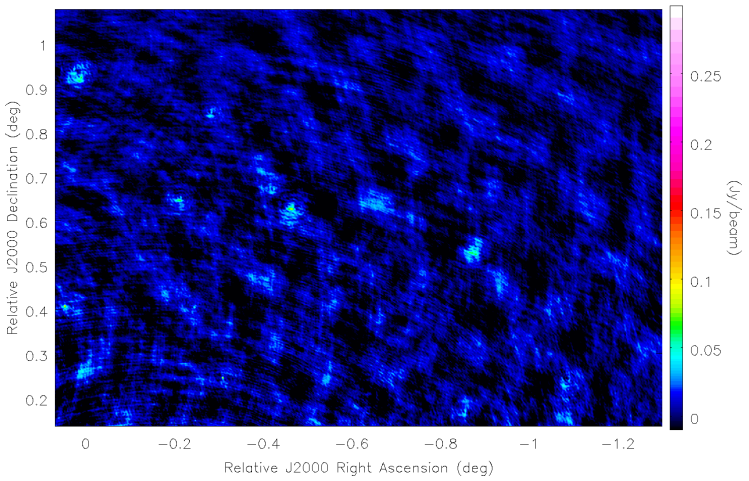


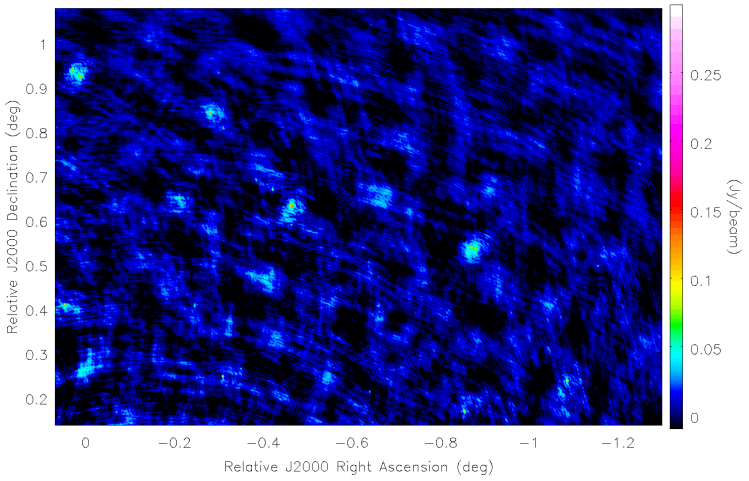
Figure 7.5: GMRT data set dirty image formed using the w-projection algorithm. At this zoom level, due to the size of the field, very little is visible. The boxes marked 1, 2, and 3 correspond to the zoomed areas in Figures 7.6, 7.7, and 7.8 respectively.

attempted a measure of a modified Strehl ratio in a similar way to that with the simulated data. In order to do this, I selected the 30 brightest sources visible in the field all of which had an amplitude of at least five standard deviations and measured the peak pixel amplitude of each source in all three dirty images. The sources selected are shown in Figure 7.10. Figure 7.11 shows the results of this and clearly demonstrates the effectiveness of the 3-d correction. In the top plot of Figure 7.11, the w-projection and faceting approaches are again seen to be approximately equivalent, with only slight variation in the ratio of source peak amplitudes and offset from the expected mean ratio of 1.0. The difference in the ratio from 1.0, which would be expected if the two approaches were identical, is likely

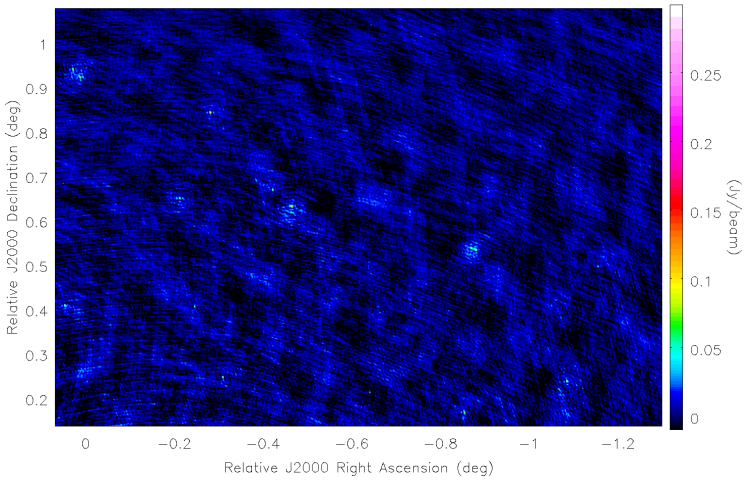
a result of differences in normalisations in gridding and differing contributions to the peak amplitudes as a result of interference of side-lobes of different point spread functions. The lower plot in Figure 7.11 shows how the image generated without 3-d corrections becomes effectively useless as we move away from the phase pointing centre and that far enough from the phase centre sources become so weak that they become buried in the noise, seen here by the seemingly random fluctuations in values at larger distances from the pointing centre.



(a)

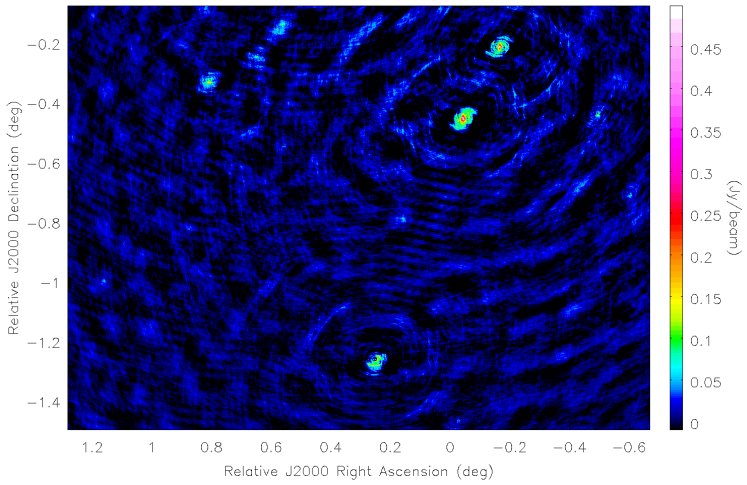


(b)

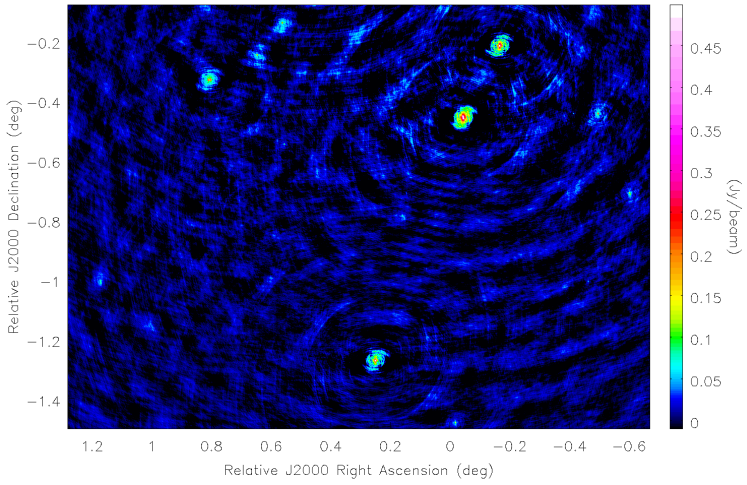


(c)

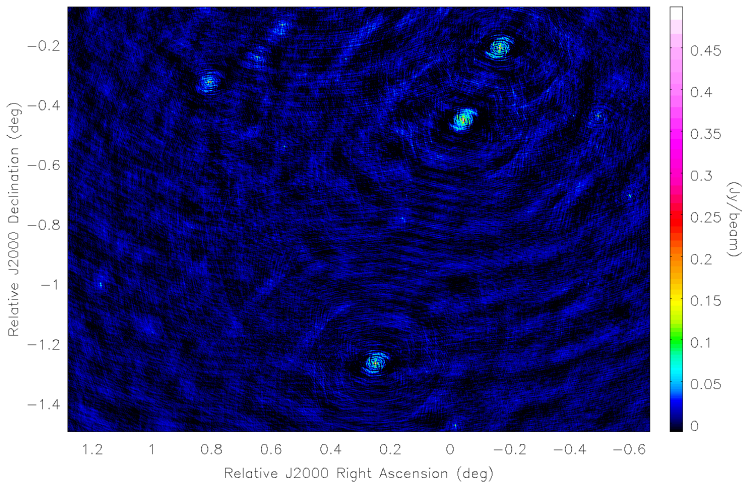
Figure 7.6: Expanded region 1 from Figure 7.5. (a) without 3-d corrections, (b) facets, (c) w-projection. Before 3-d correction (i.e. Figure 7.6 a) it is difficult to make out more than one source whereas after 3-d correction 14 sources are can be distinguished across the field. The brightest sources in this region have peaks around 0.1 Jy/beam which is about a tenth of the peak brightness in the primary beam being imaged.



(a)

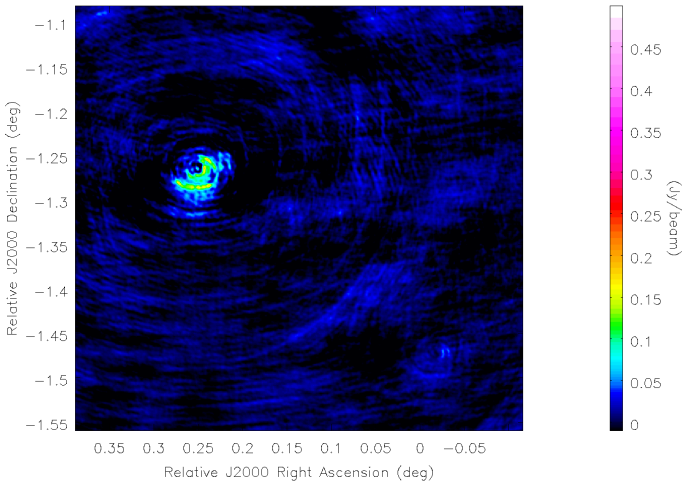


(b)

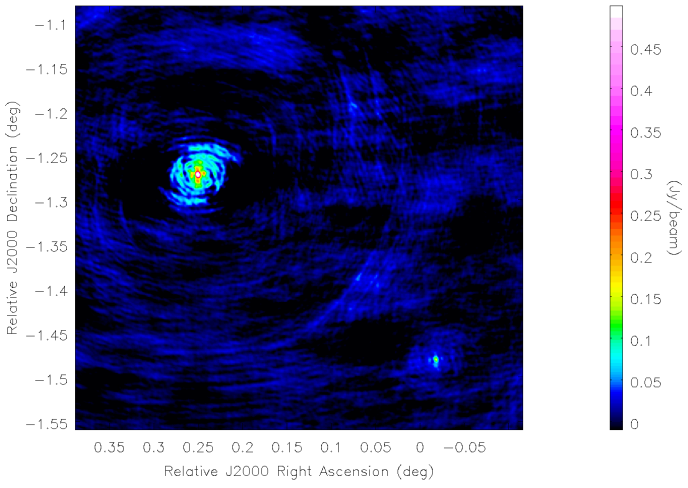


(c)

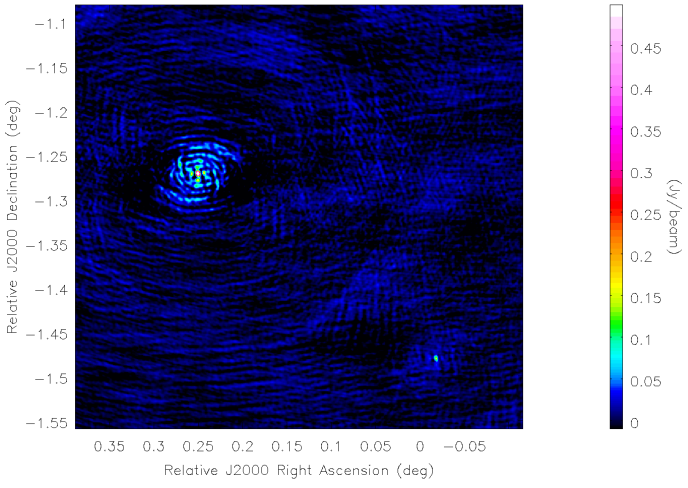
Figure 7.7: Expanded region 2 from Figure 7.5. (a) without 3-d corrections, (b) facets, (c) w-projection. This region contains a number of the brighter sources observed in the primary beam. After 3-d imaging corrections (either via faceting or w-projection) 15 sources can be distinguished in both figures (b) and (c) with peak pixel brightnesses ranging from 0.6 Jy/beam down to just above the noise floor at 0.05 Jy/beam.



(a)



(b)



(c)

Figure 7.8: Expanded region 3 from Figure 7.5. (a) without 3-d corrections, (b) facets, (c) w-projection. Two point sources are clearly visible in this region after 3-d correction. The brightest of these has a peak amplitude of 0.75 Jy/beam while the other has a peak amplitude of 0.147 Jy/beam.

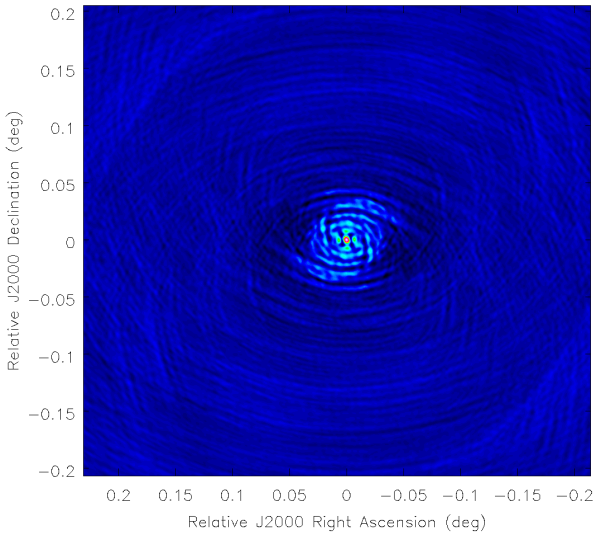


Figure 7.9: Point spread function at the phase tracking centre of the w-projection image.

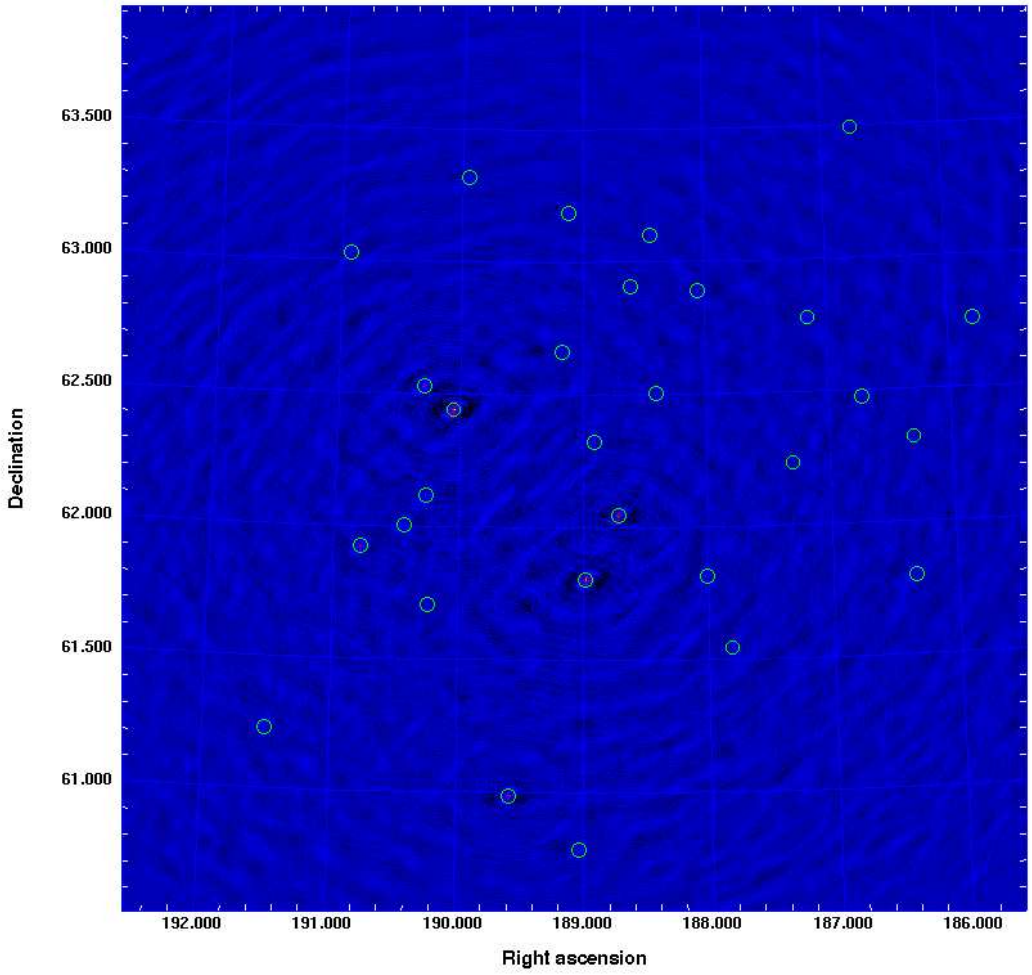


Figure 7.10: Positions of the 30 brightest sources used for the Strehl ratio comparison metric.

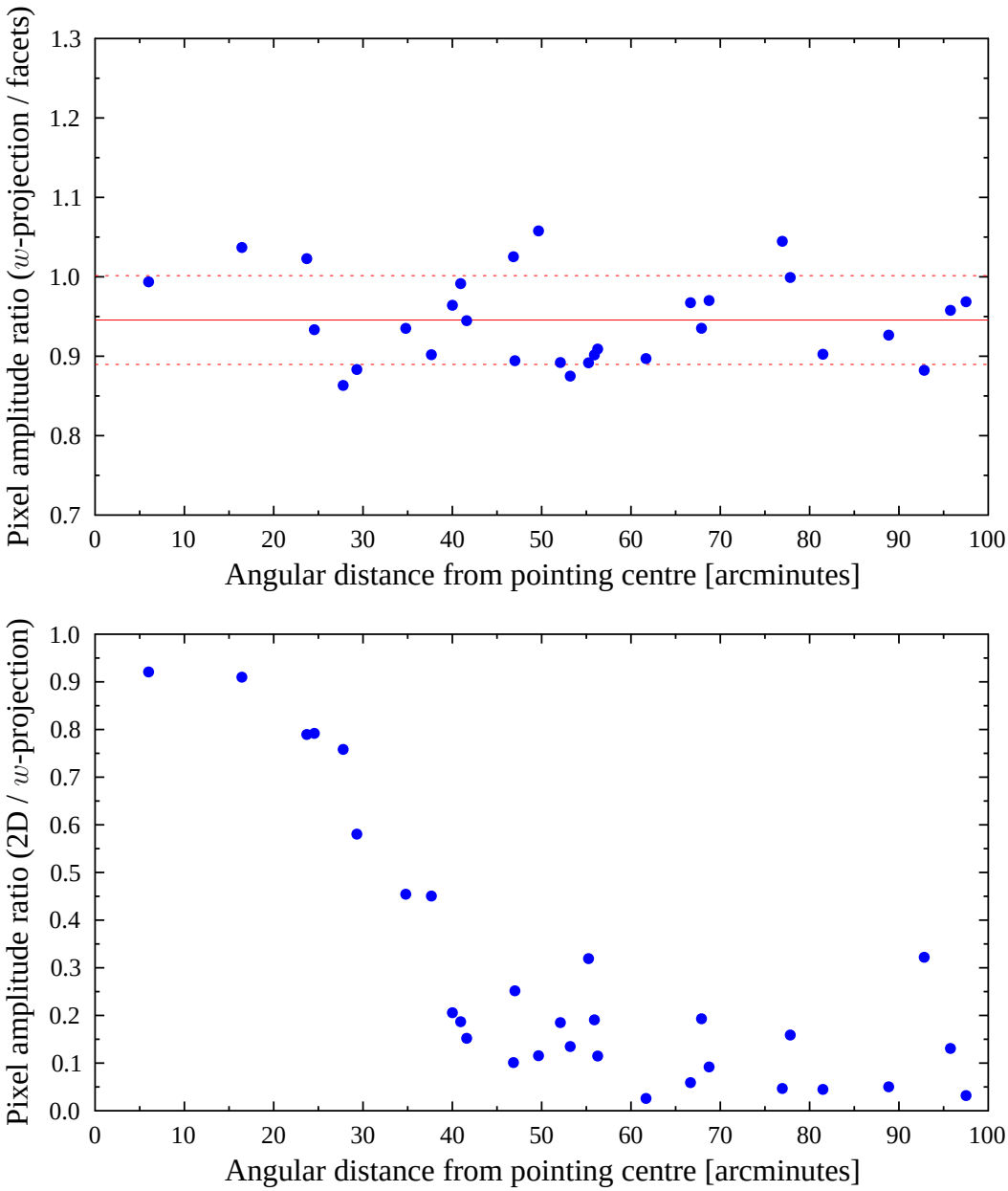


Figure 7.11: Ratio of pixel amplitudes taken for the brightest 30 sources in the GMRT dataset. (top) Ratio of source amplitudes in the w-projection and facet maps (red lines indicate the mean (solid) and standard deviation (dashed)). (bottom) Ratio of source amplitudes in the image without 3-d correction and w-projection map. This figure again verifies the numerical comparability of the w-projection algorithm implemented in *WFIT* and of the *AIPS* faceting algorithm both of which are much more successful than a standard imaging approach. The performance gains in *WFIT* explored in Chapter 6 make this the tool of choice.

Chapter 8

Conclusions

In this thesis I have investigated the application of new processing techniques to two common problems in computationally intensive astronomical data reduction. The data rates generated by new telescopes are pushing astronomical data processing towards a new regime of data intensive science. Therefore being able to effectively and efficiently extract the valuable science has never been more important and challenging. In addition, with high data rates it is quickly impractical to store data for re-examination and therefore selecting both the optimal algorithms and processing techniques for data reduction is of key importance.

In Chapter 3 I have demonstrated how the CLEAN algorithm, familiar to interferometric imaging codes, can be used in the spectral analysis of unevenly-sampled time series to reveal characteristics that would otherwise be unattainable. For this investigation I examined archival SS 433 jet speed data, recorded over a 26 year period, using the CLEAN algorithm. For comparison with CLEAN I also analysed the time-series data using a frequency folding approach and also the Lomb-Scargle periodogram approach. While the time-series in this archival data set is extremely sparsely sampled, by using CLEAN it was possible for me to rule out any strong evidence for a steady and systematic phase variation in the jet speed periodicity, a claim made by the recent literature. The main

conclusion drawn from this study was that the well known relationship between frequency instability and phase wobbling means that present data do not make it possible to definitively exclude the possibility that a variable frequency dependence of jet speed on orbital period is at play in this microquasar.

Modern interferometric telescopes mandate innovative use of computers for data processing because (i) huge data rates and sensitive detectors require complex algorithms to optimally extract the valuable science results and (ii) the rising costs of electricity and the environmental impact of excessive power consumption means that simply allocating more and more computing resources to solve the problem is impractical.

I have investigated the relevance of GPU programming to this context and in Chapters 6 and 7 I present a study of the implementation of the w-projection algorithm for wide-field interferometric imaging on GPUs. I believe this has clearly demonstrated the extremely promising potential for using GPUs in interferometric data processing as a flexible, cost effective and energy efficient solution. This situation is only likely to improve over the coming years as the electronics entertainment industry, which is responsible for driving the development of faster and lower power GPUs is a rapidly growing sector of the economy. This combined with a consumer demand for continually more complex physical simulations within computer games means that GPUs are very likely to continue growing as high performance programmable numerical processors.

I have also discussed a number of pitfalls writing code on the current generation of GPUs can impose and within the context of the w-projection algorithm have identified a certain steps in the algorithm which remain better suited to more general purpose architectures such as the CPU. Porting existing algorithms to the GPU, which I introduce in Chapter 4, can be extremely challenging, however the favourable energy efficiency of GPUs makes them a leading contender for intensive numerical computing in an era where dwindling fossil fuel resources and therefore cheap energy is of great concern. Indeed for

this reason arguably the most important metric in the deployment of new supercomputing resources is now the FLOP per watt.

Imaging of interferometric radio astronomy data shares a large number of similarities to Magnetic Resonance Imaging, where Fourier space data is recorded on spiral tracks inside a scanner and then imaged by Fourier inversion. As a result the developments I present in this thesis for cost effective and energy efficient imaging of irregularly sampled Fourier plane data also potentially have a benefit outside the astronomy community.

While investigating the application of imaging on the GPU, I have developed a modular, flexible testbed application for developing, studying and testing the deployment of GPU algorithms for interferometric imaging. This code, which I have called *WFIT*, is presented in Chapter 5.

Based on my study of the implementation of w-projection on the GPU I would identify a number of areas for future work and development:

My GPU imaging code is designed to work on offline recorded data sets, however in order to process the rate of data from an envisaged SKA size correlator (i.e. $\sim$ terabytes per minute) being able process the data at the rate it is generated is a necessity. I would therefore suggest investigating how to deploy an implementation of GPU imaging within a high data rate, pseudo-real-time, streaming framework using multiple GPUs is an important next step. The anticipated data rates from the SKA however are so extreme that even this may approach may not be sufficient. Fortunately gridding, typically the most computationally demanding step in imaging, is also largely non-interactive, so one might considering using my GPU gridding approach as a platform for investigating the porting of imaging to less flexible but higher performance, energy efficient hardware such as field programmable gate arrays (FPGAs) or even application specific integrated circuits (ASICs).

From my study of the w-projection algorithm on GPUs, I would also identify deconvo-

lution as an area of synthesis imaging which could benefit tremendously from deployment on this architecture. Deconvolution is a particularly challenging problem in wide-field imaging as the shift-variant nature of the point spread function and asymmetry of primary beams must be taken into account to obtain the highest possible dynamic range. Developing new algorithms which take into account these considerations are likely to be required for the next generation of telescopes and as this will undoubtedly be a computationally demanding endeavour, GPUs may well provide an ideal platform for their development and eventual deployment. For example, a deconvolution algorithm which evaluates the a forward model of the instrumental response per source before subtraction directly from the uv plane samples is likely to avoid many of the memory locality issues which bottleneck working with data in the image plane or on a uv grid and is therefore likely to scale extremely well.

While I believe I have demonstrated that w-projection is well suited to implementation on the GPU, one might also consider looking at other approaches to wide-field imaging, especially in the context of high data rates. The warped snapshot algorithm, which I describe in Section 2.2.5.1, is one such example. As snapshot observations are instantaneously co-planar a 2-dimensional Fourier inversion can be used for imaging, with the advantages of being able to grid with a small, fixed size convolution function. An image of the entire data set is then formed by stacking the snapshot images. The downside of this approach however is that individual snapshots would have different distortions in the image plane and while this can be corrected, it is typically too computationally demanding to be considered on the CPU. However, as the image plane correction takes the form of a coordinate transform, an algorithm which is well suited to GPUs, and as this approach avoids the issues of having to accumulate large visibility data sets prior to imaging, it may be an ideal solution to imaging the high data rate streams produced by the next generation of telescopes.

While *WFIT*, my imaging testbed application framework, has been extremely successful for investigating wide-field imaging on the GPU for the purposes of this thesis, in order to deploy a GPU imaging solution in a larger production context I believe it would be advisable to develop this code into a library. This would allow any development to be easily integrated with existing software packages across a number of platforms. When designing *WFIT* I have always had portability in mind but however converting the various processing modules into a library would be the natural next step in its development.

Through the development of innovate, computationally intensive, solutions to the challenges presented by the next generation of telescopes the future of astronomical data analysis promises to be both engaging and scientifically rich.

Bibliography

AMD Accelerated Parallel Processing specification,

`http://developer.amd.com/gpu/AMDAPPSDK/`

Bhatnagar, S., Cornwell, T. J., Golap, K., & Uson, J. M. 2008, A&A, 487, 419

Baisch, S. & Bokelmannm, G. H. R. 1999, Comput. Geosci., 25, 739

Blake, C. A., Abdalla, F. B., Bridle, S. L., & Rawlings, S. 2004, New A Rev., 48, 1063

Blanchette, J. & Summerfield, M. 2006, *C++ GUI Programming with Qt4, 2nd Edition*, Prentice Hall.

Blundell, K. M., & Bowler, M. G. 2004, ApJ, 616, L159

Blundell, K. M., & Bowler, M. G. 2005, ApJ, 622, L129

Blundell, K. M., Bowler, M. G., & Schmidtobreick, L. 2007, A&A, 474, 903

Blundell, K. M., Bowler, M. G., & Schmidtobreick, L. 2008, ApJ, 678, L47

Blundell, K., Clark, F., Perez, S. & Goodall, P., *The Global Jet Watch project*, `http://www.GlobalJetWatch.net`.

Blundell, K., Schmidtobreick, L., & Trushkin, S. 2011, arXiv:1104.2917

Briggs, D. S. 1995, *High Fidelity Deconvolution of Moderatly resolved Sources.*, New Mexico Institute of Mining and Technology.

The BrookGPU language specification,

`http://graphics.stanford.edu/projects/brookgpu/`

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., & Hanrahan, M. 2004, Brook for GPUs: stream computing on graphics hardware, ACM Trans. Graph. 23 (3), 777-786.

Calabretta, M. R., & Greisen, E. W. 2002, A&A, 395, 1077

CASA, the Common Astronomy Software Applications package,

`http://casa.nrao.edu/`

Collins, G. W., & Scher, R. W. 2002, MNRAS, 336, 1011

Cooley, J. W. & Tukey, J. W. 1965, An algorithm for the machine calculation of complex Fourier series. *Mathematical Computation*, 19:297–301

Cornwell, T. J., & Evans, K. F. 1985, A&A, 143, 77

Cornwell, T. J. 2008, IEEE Journal of Selected Topics in Signal Processing, vol. 2, issue 5, pp. 793-801, 2, 793

Cornwell, T. 2010, SKA Memo 128

Cornwell, T. J., Golap, K., & Bhatnagar, S. 2003, EVLA Memo 67

Cornwell, T. J., Golap, K., & Bhatnagar, S. 2005, Astronomical Society of the Pacific Conference Series, 345, 350

Cornwell, T. J., Golap, K., & Bhatnagar, S. 2005, Astronomical Data Analysis Software and Systems XIV, 347, 86

Cornwell, T. J., & Perley, R. A. 1992, A&A, 261, 353

Cotton, W. D. 2008, PASP, 120, 439

Croes, G. A. 1993, Astronomical Data Analysis Software and Systems II, 52, 156

Eikenberry, S. S., Cameron, P. B., Fierce, B. W., Kull, D. M., Dror, D. H., Houck, J. R., & Margon, B. 2001, ApJ, 561, 1027

Ekers, R. 2001, SKA Memo 4

Fernando, R. & Kilgard, M. J. 2003, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc.

Flynn, M.J. 1972, *Some computer organizations and their effectiveness*, *IEEE Transactions on Computing*, 21:948–960.

Fomalont, E. 1981, NEWSLETTER. NRAO NO. 3, P. 3, 1981, 3, 3

Frater, R. H., & Docherty, I. S. 1980, A&A, 84, 75

Frigo, M. & Johnson, S. G., Fastest Fourier Transform in the West.
`http://www.fftw.org`

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., & Woodall, T. S. 2004, Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104

General-Purpose Computation on Graphics Hardware, `http://www.gpgpu.org/`

Golap, K., Kemball, A., Cornwell, T., & Young, W. 2001, Astronomical Data Analysis Software and Systems X, 238, 408

Goranskii, V. P., Esipov, V. F., & Cherepashchuk, A. M. 1998, Astronomy Reports, 42, 209

Gregerson, A., 2008, Implementing Fast MRI Gridding on GPUs via CUDA, University of Wisconsin-Madison.

Greisen, E. W., 1979, VLA Memo 131

Greisen, E. 1990, *Going AIPS:* The programmers guide to the AIPS system, `http://www.aips.nrao.edu/goaips.html`

Greisen, E. 2007, The AIPS Cookbook, `http://www.aips.nrao.edu/cook.html`

Greisen, E. 2009 The FITS Interferometry Data Interchange Convention, AIPS Memo 114

Greisen, E. W., & Calabretta, M. R. 2002, A&A, 395, 1061

Greisen, E. W., & Harten, R. H. 1981, A&AS, 44, 371

Gropp, W. D., & Lusk. E. 1996, *User's Guide for* `mpich`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, ANL-96/6.

A ranking of the 500 most energy-efficient supercomputers, `http://www.green500.org`.

Harris, M. J., Coombe, G., Scheuermann, T., Lastra, A. 2002, SIGGRAPH / Eurographics Workshop on Graphics Hardware.

Harris, M., 2005, Mapping Computational Concepts to GPUs, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley Professional.

Heslop, D., & Dekkers,M. J. 2002, Physics of The Earth and Planetary Interiors, 130, 103

Högbom, J. A. 1974, A&AS, 15, 417

Ikits, M. & Magallon, M., 2010, The OpenGL extension wrangler library (GLEW), `http://glew.sourceforge.net`.

Intel Core i5 specification,
`http://www.intel.com/consumer/products/processors/corei5.htm`

Jackson, J.I. and Meyer, C.H. and Nishimura, D.G. and Macovski, A. 1991, Medical Imaging, IEEE Transactions on, 10, 3

Jaeger, S. 2008, Astronomical Data Analysis Software and Systems XVII, 394, 623

Kemp, J. C., et al. 1986, ApJ, 305, 805

Kessenic, J. 2006 The OpenGL Shading Language, version 1.2

Khronos OpenCL Working Group: The OpenCL Specification, Version 1.0, Rev. 48 (October 2009).
`http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf`

Kilgard, M. J. 2010, The OpenGL Utility Toolkit (GLUT) specification, Version 3.7,
`http://www.opengl.org/resources/libraries/glut/`

Kilgariff, E., 2005, The GeForce 6 Series GPU Architecture, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley Professional.

Kogan, L. 2005 EVLA Memo 96

Lomb, N. R. 1976, Ap&SS, 39, 447

Margon, B. 1984, ARA&A, 22, 507

McCool, M., Du Toit, S., Popa, T., Chan, B. & Moule, K. 2004, Shader algebra, ACM Trans. Graph. 23 (3), 787-795.

McCool, M., Qin, Z., & Popa, T. 2002, Shader metaprogramming, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 57-68

McMullin, J. P., Golap, K., & Myers, S. T. 2004, Astronomical Data Analysis Software and Systems (ADASS) XIII, 314, 468

The High Level Shading Language specification,
Microsoft Corporation, `http://msdn.microsoft.com`.

Microsoft DirectX Developer Center, Microsoft Corporation,
`http://msdn.microsoft.com/en-gb/directx`

Moore, G. E. 1965, *Cramming more components onto integrated circuits.*

Noordam, J. E. 1994, Frontiers of Space and Ground-Based Astronomy, 187, 533

NVIDIA Corporation. 2006, NVIDIA GeForce 8800 GPU Architectural Overview,

NVIDIA Corporation. 2008, NVIDIA GeForce GTX 200 GPU Architectural Overview

NVIDIA Corporation. 2009, NVIDIA's Next Generation CUDA Compute Architecture: FERMI

NVIDIA Corporation. 2010, The NVIDIA CUDA C Programming Guide, version 3.2.

NVIDIA Corporation. 2010, The NVIDIA Cg Languate specification,
`http://developer.nvidia.com/Cg/`.

NVIDIA Corporation. 2010, CUDA CUFFT Library

NVIDIA Corporation. 2010, CUDA C Best Practices Guide, version 3.2.

NVIDIA Corporation. 2010, CUDA Reference Manual, version 3.2.

NVIDIA Corporation. 2010, The CUDA compiler driver NVCC.

NVIDIA website, http://www.nvidia.com/.

The OpenMP consortium, http://www.openmp.org/.

Pence, W. 1999, Astronomical Data Analysis Software and Systems VIII, 172, 487

Pence, W., The C Flexible Image Transport System (FITS) library, version 3.2 http://heasarc.nasa.gov/fitsio/

Perley, R., & Clark, B. 2003, EVLA Memo 63

Prus, V. 2004, The Boost C++ program options library, http://www.boost.org/doc/

The Qt framework version 4.5, http://qt.nokia.com/

Qwt: Qt Widgets for Technical Applications, version 5, http://qwt.sourceforge.net/

Rich, J. W., de Blok, W. J. G., Cornwell, T. J., Brinks, E., Walter, F., Bagetakos, I., & Kennicutt, R. C. 2008, AJ, 136, 2897

Roberts, D. H., Lehar, J., & Dreher, J. W. 1987, AJ, 93, 968

Rost. R. J. 2006, OpenGL Shading Language, Addison-Wesley Professional

Schilizzi, R. T. 2004, Proc. SPIE, 5489, 62

Shreiner, D., Woo, M., Neider, J., & Davis, T. 2005, *OpenGL Programming Guide Sixth edition: The Official Guide to Learning OpenGL Version 2.1*. Addison Wesley, 5th edition

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., & Dongerra, J. 1996, *MPI: The Complete Reference*. MIT Press, Cambridge, MA

Sommerlade, E., Feathers, M., Lacoste, J., Lepilleur, B., Bakker, B., & Robbins, S. 2008 The CppUnit C++ unit testing framework, version 1.10.2 `http://sourceforge.net/projects/cppunit/`

Stroustrup, B. 1991, *The C++ programming, language 2nd ed*, Addison-Wesley.

The Sh metaprogramming language specification, `http://libsh.org/`

Taylor, G. B., Carilli, C. L., & Perley, R. A. 1999, *Synthesis Imaging in Radio Astronomy II.*, Astronomical Society of the Pacific Conference Series, Vol 180

Taylor, A. R. 2008, IAU Symposium, 248, 164

Thompson, C.J., Hahn, S., Oskin, M. 2002, Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture, 306, 317.

Thompson, A. R., Moran, J. M., Swenson, G. W. 2004, *Interferometry and Synthesis in Radio Astronomy.*, WILEY-VCH

Wright, M. 2004, SKA Memo 46