GPU-based Acceleration of Radio Interferometry Point Source Visibility Simulations in the MeqTrees Framework

RICHARD JONATHAN BAXTER

Dissertation presented for the degree

MASTER OF SCIENCE

in the Department of Computer Science

UNIVERSITY OF CAPE TOWN

Supervised by

Dr. Patrick Marais
Assoc. Prof. Michelle Kuttel

Abstract

Modern radio interferometer arrays are powerful tools for obtaining high resolution images of low frequency electromagnetic radiation signals in deep space. While single dish radio telescopes convert the electromagnetic radiation directly into an image of the sky (or *sky intensity map*), interferometers convert the interference patterns between dishes in the array into samples of the Fourier plane (*UV-data* or *visibilities*). A subsequent Fourier transform of the visibilities yields the image of the sky. Conversely, a sky intensity map comprising a collection of point sources can be subjected to an inverse Fourier transform to simulate the corresponding *Point Source Visibilities* (PSV). Such simulated visibilities are important for testing models of external factors that affect the accuracy of observed data, such as radio frequency interference and interaction with the ionosphere.

MeqTrees is a widely used radio interferometry calibration and simulation software package that contains a Point Source Visibility module. Unfortunately, calculation of visibilities is computationally intensive: it requires application of the same Fourier equation to many point sources across multiple frequency bands and time slots. There is great potential for this module to be accelerated by the highly parallel Single-Instruction-Multiple-Data (SIMD) architectures in modern commodity Graphics Processing Units (GPU). With many traditional high performance computing techniques requiring high entry and maintenance costs, GPUs have proven to be a cost effective and high performance parallelisation tool for SIMD problems such as PSV simulations.

This thesis presents a GPU/CUDA implementation of the Point Source Visibility calculation within the existing MeqTrees framework. For a large number of sources, this implementation achieves an 18× speed-up over the existing CPU module. With modifications to the MeqTrees memory management system to reduce overheads by incorporating GPU memory operations, speed-ups of 25× are theoretically achievable. Ignoring all serial overheads, and considering only the parallelisable sections of code, speed-ups reach up to 120×.

Acknowledgements

Plagiarism Declaration

I know the meaning of plagarism and declare that all of the work in the document, save for that which is properly acknowledged, is my own.

Contents

List of Figures

List of Tables

List of listings

Chapter 1

Introduction

Radio interferometry is the use of multiple radio-receiving elements (traditionally radio dishes, but also can be individual dipoles or phased array stations) to enhance the resolution and sensitivity of astronomy observations. While single dish telescopes convert the electromagnetic radiation directly into an image of the sky (known as the sky intensity map), interferometers calculate the interference patterns between pairs of receiving elements to produce sample points on the Fourier-plane (termed visibilities). A subsequent Fourier transform operation on a collection of samples produces the sky intensity map (Thompson et al., 2009). This process is called image synthesis.

The reverse process (conversion from a sky intensity map into the corresponding Fourier-plane visibilities) allows one to simulate what visibilities would be obtained given a certain sky model and model of instrumentation distortions as well as other effects (e.g. atmospheric effects and radio interference). This step is vital in the *fitting loop* of interferometer calibration, whereby a simulation is repeatedly run with model parameters to obtain a best fit to the data (and thus a best fit to the model parameters). A sky intensity map comprising a collection of point sources (a source is simply an observable entity in the sky) may be transformed into the corresponding point source visibilities with an inverse Fourier transform (Smirnov, 2011d). This process is called a visibility simulation in general, and a point source visibility simulation when a point source sky map is used.

Both the image synthesis and visibility simulation calculations — i.e. from visibilities to sky intensity map and from sky intensity map to visibilities — require direct Fourier transforms. This is computationally demanding as each pixel/visibility must undergo a Fourier transform.

For example, computation of a modest 256×256 pixel image from $1,000,000$ visibility samples requires $256 * 256 = 65,536$ Fourier transform calculations for *each* of the samples. The reverse (simulation) process, from an image model to visibility samples, involves a similar number of operations. Using a point source model of the sky to simulate $1,000,000$ visibility samples from a model comprising of 4000 point sources requires the computation of 4×10^9 Fourier transforms.

Extant interferometers comprise anywhere from 6 (21 base-line pairs) to 30 elements (465 base-line pairs): the Australia Telescope Compact Array has 6 elements (McKay and Wark, 2009); the Westerbork Synthesis Radio Telescope (WSRT) in the Netherlands has 14 elements (ASTRON, 2012a); the Very Large Array in New Mexico, USA has 27 elements (Associated Universities Inc, 2012), and the Giant Meterwave Radio Telescope (GMRT) in India with 30 elements (NCRA-TIFR). The LOw Frequency ARray (LOFAR) in the Netherlands has somewhat different architecture, comprising a phased array set up of $20,000$ small, cheap antennas, grouped into 40 stations (780 base-line pairs) (ASTRON, 2012b). The next generation of interferometers will be considerably larger: the Australian Square Kilometer Array Pathfinder (ASKAP) is tabled to comprise 36 elements (CSRIO, 2012); the Allen Telescope Array in California, USA, currently has 42 elements and is planned to extend to 350 antennas at completion (SETI Institute, 2012); and the MeerKAT Array in South Africa will have 64 elements (Horrell, 2012). The Square Kilometer Array (SKA) is likely to comprise both a phased array interferometer and a traditional dish interferometer with "an estimate of 2000 - 3000 ... antennas" (Dewdney et al., 2011). At 4.5 million base-line pairs, this will present a significant computational challenge.

The standard, well-explored, acceleration solution for Fourier transform calculations is to utilise the Fast Fourier Transform (FFT) algorithm. The FFT is one of the most widely-used computer algorithms in the scientific world and accelerates discrete Fourier transform calculations, reducing the complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$ (Cooley and Tukey, 1965; Brigham and Morrow, 1967). For image synthesis, a two-dimensional FFT is used to convert the visibilities into an image (Nussbaumer, 1982; Thompson et al., 2009), whereas current visibility simulation software still uses the computationally expensive direct Fourier transform method (Smirnov, 2012).

1.1 Role of High Performance Computing

Given its high computational cost and the lack of inter-dependencies in the input data, visibility simulation is a clear candidate for exploitation by High Performance Computing (HPC) and parallel processing technologies. The high computational load limits the complexity of the simulation, as only simple point source maps can be run in reasonable time. With the application of HPC technologies, these simulations can be accelerated, allowing for more testing of more complex and realistic models in shorter time frames.

The traditional HPC system is a cluster, with inter-connected CPU nodes, each responsible for a fraction of the total processing load. More recently, vector streaming systems such as as Cell (Gschwind, 2006), ClearSpeed (ClearSpeed Technology Ltd, 2012; Kozin, 2009) and MD GRAPE (IBM Research; SGI Japan), as well as GPU (Graphics Processing Unit) technology, have been developed as alternative parallel computing solutions. Instead of using a large number of serial processors that each perform one instruction on one register per clock cycle, vector processors are able to perform one instruction on multiple registers simultaneously with each clock cycle.

We focus on the GPU, the modern evolution of the graphics card. Like vector stream processors, a GPU can perform operations simultaneously on multiple data inputs using its many compute cores. In the past, these processors would only be able to perform specific rendering operations and specific 3D geometry transformations, which allowed 3D vertex information to be transformed into 2D screen coordinates. Later these processors were extended into programmable shaders which allows a programmer to chose how a vertex would be transformed. Shader-based graphics cards would often have multiple shader processors running in parallel to improve performance. Attempts to leverage these parallel shader processors to do general purpose computation on the GPU (GPGPU) were successful; however, the computation was always akin to fitting a square peg into a round hole, as general code must be written in the guise of rendering operations. On modern GPUs, these disparate shader processors were eventually unified into a single, more general, type of processor that could subsume the all of the shader tasks. Current high-end GPUs consist of over 512 cores and are able to execute general code on each of these cores in parallel.

GPUs fall into into the *Single-Instruction-Multiple-Data* (SIMD) paradigm, meaning that there is a single task that is to be applied to many data points. This is in contrast to Multiple-Instruction-Multiple-Data (MIMD) paradigm, which is applicable to problems within which

there are many different tasks to be performed on many different data points (Patterson and Hennessy, 2008). The direct Fourier transform in PSV simulations is a SIMD problem, as the same Fourier calculation must be performed on all the data points. As such, it should benefit from SIMD technologies such as vector stream processors and GPGPU (General Purpose computing on the GPU) technologies. Vector stream processors are effective at parallelising SIMD problems, but often have high entry and maintenance costs (Matsuoka et al., 2009). GPGPU technologies utilise commodity graphics hardware and offer a far more cost effective solution.

Currently leading the industry in GPU technology is nVidia and AMD, each with their own GPGPU software and hardware. nVidia's Compute Unified Device Architecture (CUDA) features a compiled high-level, C-like language for computation on their modern GeForce GPUs, whereas AMD (previously ATI) released a low-level language, FireStream, for general computation on their Radeon GPUs (AMD, 2010). AMD's current GPU efforts are focused mainly on OpenCL (Open Compute Language). OpenCL is a language interface which allows for a single piece of code to run on many different HPC hardware systems (Munshi, 2011). To date, both nVidia and AMD have OpenCL implementations, allowing for OpenCL utilisation on both vendors' hardware (Karimi et al., 2010). While OpenCL has improved significantly in recent years, and looks to become the leading platform for heterogeneous computing, CUDA remains the more prominent technology owing to its faster execution times, mature API, and wide support in the form of libraries and tools (Karimi et al., 2010).

Given the need for an accelerated implementation of PSV computation, and the wide acceptance of CUDA, we have developed a CUDA version of the PSV model present in the leading astronomy software suite, MeqTrees (Noordam and Smirnov, 2011). Our implementation exploits the SIMD nature of the PSV calculations with commodity graphics hardware, thus allowing a computationally effective and cost-effective parallelisation solution.

The MeqTrees framework has been developed to compute any ***Measurement equation*** by defining the equation as a ***Tree***. It can perform a wide variety of computations (represented as trees) and is defined by a simple Python script (Noordam and Smirnov, 2011). Its main purpose is to perform third generation calibration (3GC). In practice, recorded visibilities are subject to corruption by instrumental effects (these corruptions can be represented as per-antenna complex gain terms — i.e. an amplitude and a phase per antenna — that varies with time and frequency but not with direction). Calibration attempts to solve for these effects, and to recover the true visibilities. First generation calibration (1GC) techniques

involve switching from observations of the target with those of a known calibrator source and then back. Second generation calibration (2GC) solves for per-antenna complex gains and the unknown target sky region simultaneously, without having to break observation of the target to externally calibrate. 3GC techniques extend this with the ability to take direction-dependant gains into account. With each generation, the achievable signal-to-noise ratios of interferometers increased as well, improving sensitivity of the data. MeqTrees is designed to be flexible to allow for a multitude of 3GC instrumental and interference models (Noordam and Smirnov, 2011).

Within its many modules, MeqTrees includes a PSVs module. This module produces a simulated UV-plane based on a number of input data: interferometer setups, instrumental models, noise models and sky map models. This is essentially a simulated interferometer, and can be used to evaluate and improve the aforementioned models and for simulation in of itself. As mentioned above, however, PSV simulation is computationally intensive as it utilises a direct Fourier transform method.

1.2 Aims and Approach

The principal goal of this work is to accelerate PSV calculations performed by the current CPU PSV component of the MeqTrees framework. We aim to achieve this by exporting the CPU PSV component to the SIMD parallel architecture on modern GPUs. To enable comprehensive parallel optimisation, we focus on nVidia's CUDA devices and our code and we integrated our code into the execution paradigm of the MeqTrees software package.

We port the computational parts of the PSV module into a new CUDA PSV module, which we add to the MeqTrees framework, leaving the remainder of the framework unchanged. For the CUDA PSV module, we create a naive implementation, and then iteratively add common CUDA optimisation techniques to determine their efficacy. In particular, we explore methods to reduce the amount of off-chip memory accesses, methods such as effective use of CUDA shared memory and exploitation of CUDA memory coalescing via optimal thread organisation. We also explore processor utilisation to determine its importance in accelerating this computation. Data-sets of multiple sizes are used with two distinct interferometer setups, namely the Westerbork Synthesis Radio Telescope and the MeerKAT (Karoo Array Telescope), the Square Kilometre Array (SKA) pathfinder.

Performance of the new CUDA PSV module is benchmarked against the CPU version, to determine the magnitude of performance gain achieved by the CUDA GPU. To determine the effect that each optimisation has on the final speed-up, various CUDA optimisations are selectively removed and the code is benchmarked against the CPU version as well as other GPU versions. These benchmarks are analysed to determine the best configuration of optimisations for optimal memory and processor performance. We run our tests on nVidia's commodity Fermi architecture under numerous configurations and compare it to older hardware to gain insight as to what technological advancements in the newer hardware are most effective (such as off-chip memory caching and increased double precision floating point operation throughput).

1.3 Contributions

The main contribution is the creation of a node in MeqTrees that utilised the processing power of the GPU and the fact that it performed significantly faster than its CPU counterpart. Although it is only a working prototype of a CUDA MeqTrees node, it can be incorporated fully into the MeqTrees code-base with some additional work. This was achieved by implementing and optimising a CUDA PSV node, and discovering what technical challenges there are for future attempts at implementing GPU functionality in MeqTrees.

This work notes the importance of the use of shared memory to reduce off-chip memory accesses. The PSV problem is characterised by relatively small input vectors in comparison to the output vectors. The actual process involves a large amount of GPU memory for intermediate values, which are reduced to a smaller final output vector. Shared memory allows us to do part of the reduction in on-chip memory *before* it is written to off-chip memory, thus reducing off-chip memory writes significantly.

It is found that the serial (hence not parallelisable) section of the code becomes the main bottleneck for accelerating the PSV node. It is a primary point to be taken from this work that acceleration of the core computations alone will result in at most an order of magnitude reduction in total running time, even if the core computation is sped up a thousand fold or more. The reason behind this is that the previously small fraction of the MeqTrees overhead becomes hinderingly significant when the core computations are accelerated. Whilst this overhead is a result of MeqTrees' flexibility and wide scope, in order to achieve speed-ups of more than two orders of magnitude, this overhead should be parallelised or accelerated in

some manner.

Massive radio interferometers such as the SKA will have computational requirements multiple orders of magnitude larger than any previous interferometer. These projects will have sensitivity far exceeding anything before them, and thus will require even more precise and accurate calibration models so as to improve signal-to-noise ratio. RFI has always been a primary concern, but with 3GC calibration techniques, the need to test atmospheric interference has become just as important, as scientists and engineers strive towards better, more sensitive equipment. PSV simulations allow for testing and acceleration of these models, which enable faster turnaround and more complete simulations.

1.4   Thesis Outline

The thesis is structured as follows:

Chapter 2 reviews the basics of Radio Interferometry (specifically Point Source Visibility calculations), GPGPU technology (specifically CUDA technology) and relevant Fourier transforms, and explores the surrounding literature.

Chapter 3 explains the CUDA programming and hardware models, CUDA implemented Thrust library, and common CUDA optimisation techniques.

Chapter 4 details the techniques with which the PSV node in MeqTrees is implemented. It also outlines how MeqTrees defines equations as expression trees and specifically how PSVs are defined. Furthermore, it explains the MeqTrees execution model and how the GPU node is incorporated.

Chapter 5 details how results are measured, the metrics used and the experimental setup. Chapter 6 shows the findings of these tests with discussions thereof. Conclusions and future work are presented Chapter 7.

The Appendices present detailed explanations of image synthesis in radio interferometry (A) and PSVs (B). These support the main body of the thesis and are referred to throughout.

Chapter 2

Background

This chapter serves to give the reader the understanding of radio interferometry and GPU technology required to follow the remainder of the thesis. Radio interferometry and related concepts are defined and the radio interferometry data pipeline is outlined with an emphasis on the computational challenges that it presents. The MeqTrees framework, a central component to this work, is then introduced. The evolution of GPGPU technology is summarised in the context of High Performance Computing (HPC) technology. Furthermore, CUDA and CUDA-related technologies are introduced. An overview of some literature (especially pertaining to GPGPU technology in astronomy) is discussed in this chapter, but related literature is largely contained in the relevant chapters.

2.1   Radio Interferometry

Radio interferometry is the use of multiple radio-receiving elements to enhance the resolution and sensitivity of astronomical observations by measuring interference patterns between receiving elements. This contrasts with traditional single dish telescopes that convert the electromagnetic radiation directly into an image of the sky (or *sky intensity map*).

An interferometer measures *visibilities*, which are samples of the Fourier transform of the sky intensity map, by correlating the signals received by each element with every other element in the array. The interferometer will measure many visibilities in a snapshot observation, or over a number of hours. When plotted on the same plane, the set of visibilities produces a sampled Fourier plane of the observed region of the sky. An inverse Fourier transform (either

a direct Fourier transform or a Fast Fourier Transform (FFT)) then produces the desired observed sky intensity map.

The reverse process of producing (or simulating) visibilities from a model sky intensity map is primarily used for testing various instrumental distortion models and thereby increasing the signal to noise ratio and improving sensitivity.

2.1.1 Concepts and Definitions

A radio wave is electromagnetic radiation with a long wavelength, above 1mm and include 1 meter and even 1 kilo-meter wavelengths. Unlike optical signals that operate on shorter wavelength and can be measured by reflecting and focusing electromagnetic radiation signals by use of a reflective surface or by refracting the signal using a transparent (glass) lens, radio waves are measured by focusing the signals into a receiver with large paraboloid dishes. Since longer wavelengths reduce resolution, larger collecting areas are necessary to maintain an acceptable resolution, as given in the following linear proportion:

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

where R is the angular resolution in radian, λ is the wave-length, and D is the diameter of the collecting surface (the dish). This phenomenon is referred to as the diffraction-limit, as the resolution is limited by the diffraction of light, which at larger wavelengths, is worse. The resolution in this case is angular since signals are received over a spherical sector of the sky. The definition of angular resolution is the minimum angular distance to resolve (distinguish) two point sources as distinct object, as opposed to pixel density. A lower R is equivalent to an increase in pixel density. An increases diameter increases resolution, but also decreases the Field-of-View (FoV).

In order to obtain a high enough resolution, larger and larger radio telescopes have been manufactured in a variety of shapes. All dishes, however, essentially reflect and focus electromagnetic radiation onto a receiver, which then digitises the signal (Figure 2.1).

To date, the largest steerable radio telescopes in the world are about 100m in diameter, such as the Effelsberg 100m Radio Telescope and the slightly larger than 100m Green bank Telescope. The largest telescope is located in Arecibo, Puerto Rico; it is 305m in diameter, but is built in a natural sink-hole and is thus immovable. Currently under construction in
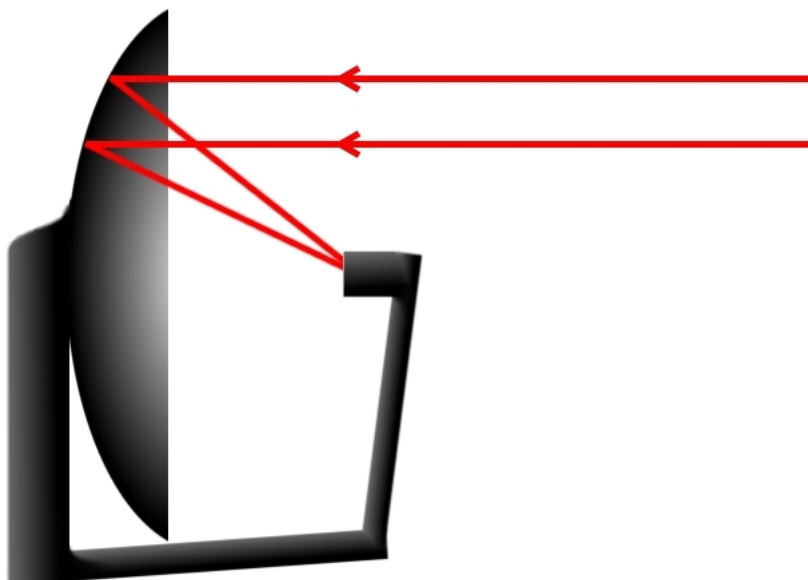
Figure 2.1: A dish with a feed horn
Incoming rays are reflected and focused onto a feed-horn, which converts the signal to a voltage.

Pingtang County, China is the Five hundred meter Aperture Spherical Telescope (FAST), built in a natural basin. All of these very large dishes, regardless of mobility, are costly to build and maintain. The engineering and monetary requirements for creating significantly larger dishes are becoming more and more infeasible with each advancement. It is thus unlikely that higher resolution images can reasonably be obtained by simply building larger dishes.

Another method for obtaining an effective larger collecting area is to use an array of smaller telescopes (as shown in 2.2) rather than a single large one. This method, known as *interferometry* or *aperture synthesis*, measures the interference patterns between dishes and allows synthesis of a telescope with an aperture equal to the greatest distance between telescopes. This changes Eqn 2.1 to:

$$R \propto \lambda/B \tag{2.2}$$

where R is the angular resolution, λ is the wave length, and B is the longest *baseline* of the array. The longest *baseline* of an array of telescopes is the longest distance between any two telescopes. Whilst there do exist optical interferometers, interferometry is most effectively used for radio signals owing to the difficulty in accurately measuring interference patterns

Figure 2.2: The Karoo Array Telescope (KAT-7)
The KAT-7 consists of seven identical 12m parabolic dishes. This array is the testbed upon which the 64 dish MeerKAT array is currently being built. (SKA South Africa, 2012; Wolleben, 2012)

in shorter wavelength signals.

Since an interferometer consists of many small dishes that can be placed far apart from each other, it has the great advantage of being able to produce higher resolution images than the traditional single dish method (see Appendix A for more details). Another advantage is that an interferometer will have a FoV based on the diameter of the individual dishes, even though the resolution is related to the longest baseline. Increase FoV means that more of the sky can be surveyed at a time, and speeds up observation time proportionately.

Unfortunately, since interferometers measure interference patterns between pairs of antennae rather than direct signals. the number of pairs in an N element interferometer is given by:

$$\frac{(N-1)N}{2} \tag{2.3}$$

As such, additional processing is required to transform the interferometer output into usable data. A notable disadvantage here is that this additional processing is very computationally expensive and may require HPC technologies in order for the interferometer to serve as a

feasible observational tool.

Whilst two telescopes of an interferometer can be placed very far apart from each other to increase resolution, the total collecting area is still very small thus limiting sensitivity. More telescopes in an interferometer array increases its total collecting area, which increases sensitivity, but increasing the number of elements also increases the computational cost.

Specifically, the computational cost is quadratic with respect to the number of elements in the array, since an interferometer measures interference between each *pair* of elements 2.3. The distance between the elements are called baselines. Any two elements constitute a baseline pair. For new and upcoming interferometers with increasing numbers of array elements, the computational cost grows significantly faster than the size of the array.

2.1.2 Radio Interferometry Data Pipeline

As an interferometer measures the interference patterns between pairs of antennae, there is not a simple direct path from received signal to the analysed data. Thus a data pipeline has developed and is as follows:

1. The array of telescopes collect their respective signals and are combined in a correlator to produce a set of *visibilities* over a number of hours

2. These visibilities are then transformed into an image of the sky using a Fourier transform. This results in a image of the sky convolved with the interferometer's Point Spread Function (PSF)

3. This image then undergoes some post-processing to remove the convolution in a process called *deconvolution.*

4. A second phase of post processing is sometimes performed to remove background noise in the data.

5. Data is analysed

Each pair of receiving elements collects one *visibility* or UV-sample over typically 0.1 to 30 seconds. Longer sample times means that more signal is received per sample, increasing sensitivity, but also means there are fewer visibilities for the observation, which leads leads to a more sparsely sampled Fourier domain and reduced ability to measure short term variability in the observed source owing to time and bandwidth smearing (see Appendix B.4). Samples

sets are collected over a few hours of observation. To give some indication of the sheer amount of visibilities that are collected, consider a modest 16 element interferometer that samples every 10 seconds over 8 hours (or $28,800$ seconds). According to Eqn 2.3, this is $\frac{16*15}{2} = 120$ pairs, each collecting over $\frac{28,800}{10} = 2,880$ time-steps, totalling $345,600$ samples.

The number of antenna pairs grows quadratically with the number of antennae. If we, in our example, double the number of elements to 32, then the number of pairs would increase to 496, over four times the original of 120. The number of samples increases to $1,428,480$ visibilities. Thus, as interferometers become larger, the computational cost becomes increasingly significant. Compounding this, advances in radio dish technology, such as increased sensitivity and reduced internal noise, allow for shorter sampling times and hence more samples per baseline pair per hour. With modern interferometers reaching over 60 elements (1770 pairs) and considering typical observing conditions of 1 second samples over 10 hours (36000 sample sets), the number of samples can balloon to over 60 million. With future interferometers like the SKA, which should contain a few thousand elements, the computational challenge becomes substantially more significant.

Adding to this already large number of visibilities, multiple frequency channels are often measured simultaneously, typically 32 or 64 channels at a time. Thus 32 or 64 times the number of visibilities. The number of visibilities recorded in an observation is

$$S = \frac{N(N-1)}{2} \times f \times t$$

where N is the number of elements in the array, f is the number of frequency channels and t the number of time-steps taken.

Mathematically, a plot of all the visibilities (called the *UV-plane*) is equivalent to a sampled version of the Fourier plane (see Figure 2.3 for an example). Thus a Fourier transform of the UV-plane results in the image plane, or *sky intensity map*. Appendix A has a detailed overview of visibility collection and the underlying mathematics.

Although a Fourier transform of the UV-plane reconstructs the sky intensity map (Figure 2.4, left), the reconstruction is not perfect . The actual reconstruction is the observed signal convolved with the interferometer's PSF (Figure 2.4, right). Whilst in optical telescopes the PSF usually has a 2D Gaussian shape, the PSF of an interferometer is dependent of the UV-plane distribution (PSF shown in Figure 2.4, middle) which is dependent on array layout and the length of time over which the observation was performed. As such, the reconstruction is
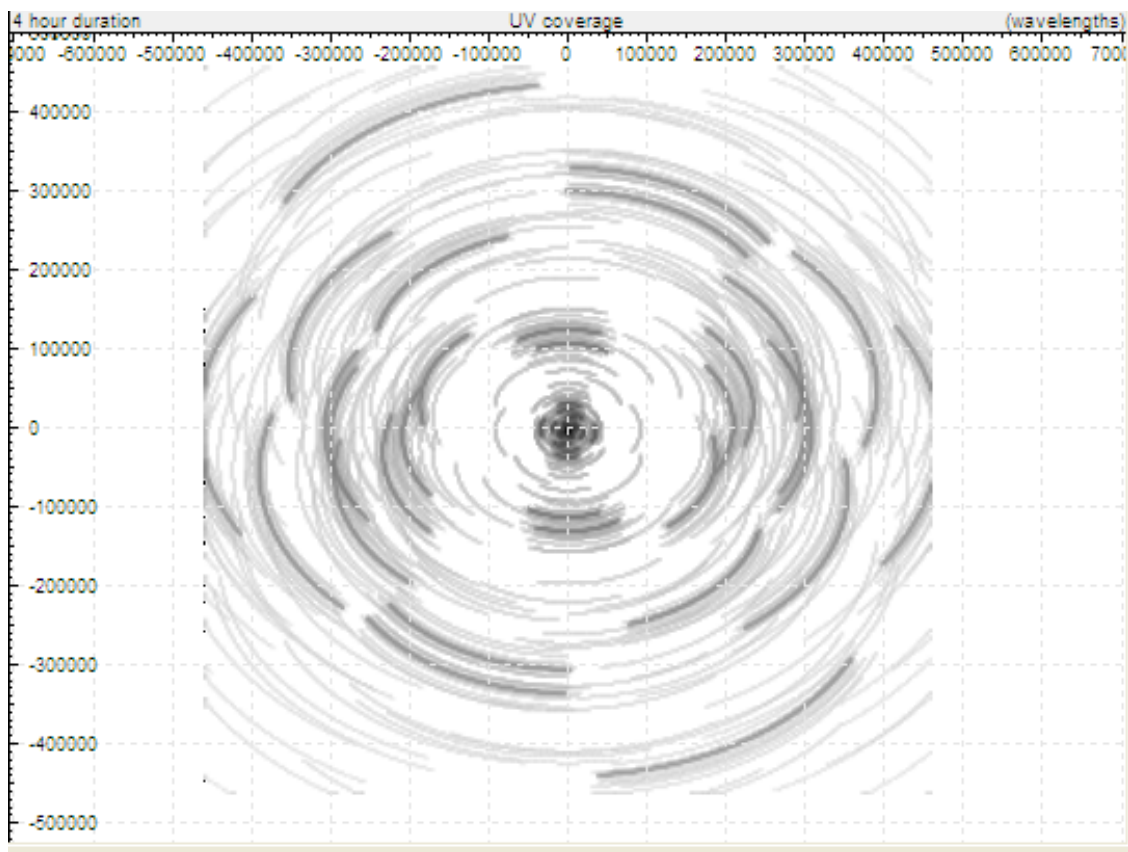
Figure 2.3: UV coverage example

UV coverage of a four-hour observation with the LOw Frequency ARray (LOFAR). While it covers a large area, there are still many gaps between the samples. If this plane were to be filled/fully sampled it would represent the Fourier Inverse of the observed sky intensity map.

called the *dirty image.*

This dirty image is then 'deconvolved' to produce a *clean image.* Internal instrumentation noise, as well as other noise caused by non-observational effects corrupts the post-convolution signal , means a straightforward inverse convolution greatly amplifies this internal noise and makes the deconvolved image unusable [1]. Thus, approximate or heuristic deconvolution algorithms are found to be more effective, and is a field of study in its own right (Taylor et al., 1999). We do not investigate or implement deconvolution in this thesis.

The most widely used technique for producing the dirty image is to re-sample the Fourier

---

[1]Internal noise in this context is strictly undesirable local noise and not background observational noise. Background noise from other sources in the sky will be preserved by a direct inversion.

14
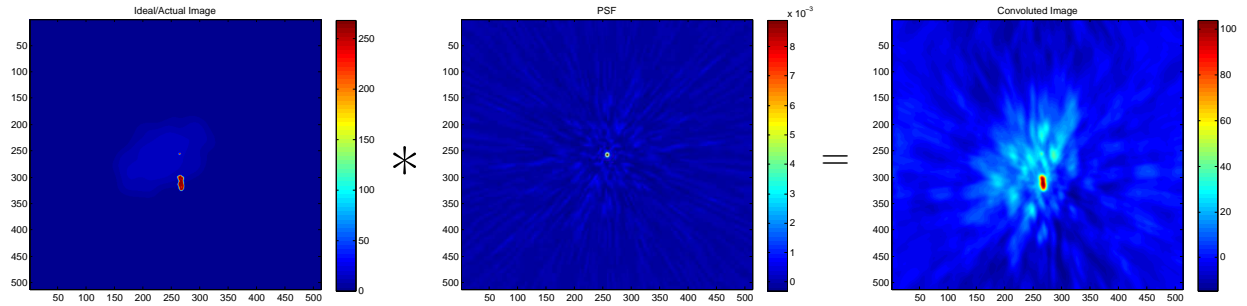
Figure 2.4: Example of an intensity map affected by a PSF
Left: Actual Intensity map of the sky. Middle: Point spread function of the interferometer. Right: The convolution of the two. This is the data that the interferometer produces (minus any noise).

samples onto a uniform grid and then perform a Fast Fourier Transform (FFT) (Thompson et al., 2001). For image synthesis, a two-dimensional FFT (2D FFT) is used to convert the visibilities into an image (Nussbaumer, 1982; Thompson et al., 2009). Since the FFT requires discretised data, the visibilities are discretised (or 'gridded') such that they lie on a uniform grid. This usually involves some form of transformation or convolution function (Sault and Wieringa, 1994; Rau et al., 2009; Thompson et al., 2009).

Using an FFT is faster than calculating the direct Fourier transform on a per-sample basis. To demonstrate the difference in computational complexity, consider a data set of S samples. Using a per-sample transform based on a direct Fourier transform, an $M \times M$ image ,where M represents the pixel dimensions of the image, must be calculated for each of the S samples, which requires $\mathcal{O}(M^2)$ running time per sample — $\mathcal{O}(M^2 S)$ in total. These S images are summed (or *reduced*) to a single image; this operation also has a running time of $\mathcal{O}(M^2 S)$.

For the direct Fourier transform (non-FFT) method of image synthesis, we need to calculate P pixels:

$$P = \frac{N(N-1)}{2} \times M^2 \times f \times t$$

where N is the number of elements in the array, f is the number of frequency channels, and t the number of time-steps taken.

The visibilities can be discretised onto a uniform $M \times M$ grid/image using an FFT. The FFT algorithm runs in $\mathcal{O}(N \log(N))$ for a data set of N values. An FFT on a 2D $N \times M$ data set is equivalent to performing N 1D FFTs on each of the rows (each row an M-length vector) and then M 1D FFTs on each column (each column an N-length vector). This operation

requires a $\mathcal{O}(M \cdot N \log(N) + N \cdot M \log(M)) = \mathcal{O}(MN \log(MN))$ running time.

A 2D FFT on an $M \times M$ grid would run in $\mathcal{O}(M^2 \log(M))$ with an added $\mathcal{O}(S)$ step for discretisation of each source. The resulting run time is $\mathcal{O}(S + M^2 \log(M))$, which is faster than the direct method of $\mathcal{O}(M^2 S)$ for large values of S.

The FFT, while faster, leaves the resultant image-plane slightly distorted owing to the cyclic nature of the FFT, which causes PSF *side-lobes* to wrap around the image. Additionally, the gridding step (discretisation) intrinsically distorts the input data thus causing slight aliasing in the resultant image (Thompson et al., 2009). Almost all FFT implementations use some form of padding to alleviate the cyclic *ringing* and a post-process to reduce the aliasing caused by the gridding process (Thompson et al., 2009; Taylor et al., 1999).

2.1.3 Simulating Point Source Visibilities

Image synthesis is the act of transforming visibilities to a sky intensity map. The reverse process — producing visibilities using a sky intensity map — is called *visibility simulation.* Simulated visibilities are important for testing (and understanding) instrumental models, something that is necessary to increase signal-to-noise ratios and to obtain more reliable and sensitive data (Smirnov, 2011d).

In order to simulate visibilities, we need a sky intensity map upon which an inverse Fourier transform can be performed. The simplest methods use a sky intensity map comprising a collection of *point sources*, as this requires a simple Fourier transform (Gaussians can also be used). In the MeqTrees software suite, PSVs are computed with a direct Fourier transform (Smirnov, 2012). Theoretically, a sky map can be discretised (or gridded) and use an FFT to obtain the visibilities. This discretisation of data and the effects of this are in early stages of exploration. Appendix B gives a detailed breakdown of the PSV equation.

PSV data changes depending on the layout of the interferometer, the measured frequency bands, the number of time intervals, and the point source model of the sky. Increasing the number of receiving elements in the telescope array results in more visibilities but comes with a concomitant quadratic increase in the processing required to calculate visibilities. For an N element interferometer simulating P point sources, the number of visibilities to be calculated is

$$S = \frac{N(N+1)}{2} \times P \times f \times t$$

where f is the number of frequency channels and t the number of time-steps taken.

Each one of these visibilities V must be described in terms of which antennae pair they derive from, q and q, at what time is was measured, t and the frequency channel, f. This equation, called the Radio Interferometric Measurement Equation (RIME) is derived from the antennae voltages at time t. While the classic RIME is defined in detail in Appendix A, its formulation made it difficult to extent to take into account various noise models, a more modern and flexible model has been developed (Smirnov, 2011a,b,c,d) and reads:

$$
\begin{aligned}
V'_{pq}(t,\nu) &= \sum_s \mathrm{sinc}\frac{\Delta\boldsymbol{\Psi}(\nu_m)}{2}\mathrm{sinc}\frac{\Delta\boldsymbol{\Phi}(t_m)}{2}K_{sp}(t,\nu)B_s K_{sq}^H(t,\nu) \\
&= \sum_s \mathrm{sinc}\frac{\Delta\boldsymbol{\Psi}(\nu_m)}{2}\mathrm{sinc}\frac{\Delta\boldsymbol{\Phi}(t_m)}{2}B_s e^{-2\pi i\frac{\nu}{c}(\boldsymbol{u}_{tpq}\boldsymbol{\sigma}_s)}
\end{aligned}
\tag{2.4}
$$

where $V_{pq}(t,\nu)$ is complex visibility between antennae p and q at time t in frequency channel f. Other terms in this equation are explained in full in Appendix C.

2.1.4 MeqTrees

MeqTrees (Noordam and Smirnov, 2011) is a software package used for so called third generation calibration (3GC) of radio interferometers and the reverse problem of visibility simulation. In general, it is a software package for calculating any Measurement EQuation specified as an *Expression Tree*. A measurement equation is an equation that relates the measured signal to the observation. This is covered in more detail in Chapter 4. Expression trees are defined by the user with Python scripts, making MeqTrees easy and flexible to use for most radio interferometric calculations. MeqTrees has been used extensively for LOFAR, WSRT, and MeerKAT and is planned for use in the SKA (Noordam and Smirnov, 2011) .

This thesis focuses on the implementation of a MeqTrees 'node' that calculates PSVs. A CPU version of this node already exists in the MeqTrees framework and it is this node against which our GPU results are bench-marked. This 'PSV node' is based on the Radio Interferometry Measurement Equation (RIME), which is detailed in Appendix B.

2.1.5 Parallelism Opportunities

In both image synthesis and visibility simulation, we see similar characteristics. Both perform a Fourier transform for each of the pixels/visibilities that need to be calculated. Indeed, the actual equation of each is the Fourier inverse of the other. For each case all the calculations are the same, but with different input data. This lends itself to a solution employing the Single-Instruction-Multiple-Data (SIMD) paradigm (Patterson and Hennessy, 2008).

Within the domain of high performance parallel computing, there are several ways to approach SIMD problems, most of which require the use of specialised hardware such as Cell (Gschwind, 2006), ClearSpeed (ClearSpeed Technology Ltd, 2012; Kozin, 2009), and MD GRAPE (IBM Research; SGI Japan). These solutions can be very expensive involving high initial investments, specialised maintenance, and often high power-consumption rates (Matsuoka et al., 2009). In 1999 nVidia coined the term *Graphics Processing Unit* (GPU) to describe the first graphics card in its GeForce range, the GeForce 256 (nVidia Corporation, 1999). Advancements in GPU technology culminated in 2006 when nVidia released its *Computer Unified Device Architecture* (CUDA) devices (nVidia Corporation, 2006). CUDA exposes the powerful computation ability of the GPU by allowing the user to control the GPU with general code (nVidia Corporation, 2011b) rather than with graphics operations, as had previously been the case (Owens et al., 2008). With lower entrance costs, less specialised maintenance, and reduced power consumption (Matsuoka et al., 2009), CUDA has opened up the field of General Purpose computing on the GPU (GPGPU). Other stream processing technologies are available, for example OpenCL (Munshi, 2011), ATI's Close to Metal (CTM) (Hensley, 2007), AMD's FireStream (AMD, 2010), and MicroSoft's DirectCompute (Microsoft Corporation, 2010). Many of these technologies over generalise the GPU hardware so that it is not fully utilised or are still coming into full maturity as a useful API for GPGPU computing (Karimi et al., 2010).

Our selected HPC platform is commodity nVidia GPGPU system for its effective cost to processing power compared to other systems that require large initial and continued maintenance costs, it's SIMD processing model and because of nVidia's mature API and framework compared to other similar GPGPU technologies, as discussed in Section 1.1.

2.2 Graphics Processing Units

Graphics Processing Units (GPUs), or 'graphics cards', were developed for the computer gaming industry, where many pixel and vertex operations must be performed in parallel to render a 3D scene with detailed geometry at an adequate rate (Owens et al., 2008).

Initially, these processors could only perform fixed rendering operations and specific 3D geometry transformations, i.e. to transform 3D vertex information into 2D screen pixel *fragments*. Fixed processors were later extended into programmable *shaders*, which enabled programmers to control the GPU via OpenGL or DirectX API calls (Peercy et al., 2000). Shader code allows programmers to choose how a vertex is transformed (with a vertex shader) and how the final pixel stream is drawn to the screen (with a fragment shader). Shader-based graphics cards often have multiple shader processors to improve performance as there are many vertices and fragments on which to operate, but only one vertex and fragment shader program. This Single-Instruction-Multiple-Data (SIMD) model is central to all modern GPU devices.

Initial attempts to leverage shader processors to do general purpose computation on the GPU (GPGPU) demonstrated impressive computational gain. Unfortunately performing such computation requires general code to be written in the guise of rendering operations using graphics primitives such as polygons, textures, and fragments. This adds an unnecessary level of abstraction that the programmer has to account for, involving unintuitive render operations and multiple shader processors with different characteristics. Memory management is arguably even harder, as the programmer had little or no access to the underlying memory hierarchy of shader based GPUs (Buck et al., 2004).

Attempts to abstract these operations were first successfully implemented in the *BrookGPU* API (Buck et al., 2004). BrookGPU converts *Brook language* code into Cg code, a shader language, that can be executed on any hardware that is OpenGL or DirectX compatible.

The disparate shaders were eventually unified into a single, more general, highly parallel type of processor that subsumed all of the shader tasks. This was called the *unified* shader model and modern GPUs contain large numbers of these processors. As a result there is increasing interest in harnessing these low-cost devices for more general purpose computing. The unified shader model allowed for more modern APIs, such as OpenCL (Open Compute Language) (Munshi, 2011). OpenCL abstracts code to be run on a variety of HPC platforms, including multicore CPUs, Intel's Cell processors and most GPUs. Both nVidia and ATI

have released OpenCL implementations for their respective GPU lines.

Besides OpenCL, the two major GPU vendors (nVidia and AMD) have each released their own APIs and GPU compilers for their respective hardware. Unlike OpenCL, these APIs are developed in-house and thus are better able to utilise the specific hardware characteristics of their respective devices. nVidia's CUDA is a standalone API that runs on proprietary nVidia CUDA-enabled GPUs, and like OpenCL it uses a SIMD model for general purpose computation. CUDA is written in a C-like language which is interoperable with standard C and C++ but has bindings and libraries for many other languages. ATI released *Close-to-Metal* (Hensley, 2007), a low-level programming interface that allows programmers to access low-level instructions of ATI GPUs. This developed into *FireStream* (AMD, 2010), released by ATI's successor AMD. Both these ATI/AMD technologies were short-lived and current GPGPU efforts from AMD are focused on its OpenCL SDK.

More recently, nVidia has also released an OpenCL SDK, which with sufficient tweaking, is no slower than standard CUDA code (Karimi et al., 2010). Even though these tweaks are CUDA specific (and will not run as fast on an AMD device, for example), this shows promise for OpenCL as a heterogeneous HPC framework that is able to exploit various devices to their full ability. Nevertheless, at the beginning of this thesis, CUDA was clearly the best GPGPU technology available.

Although OpenCL has improved significantly, CUDA remains the more widely used technology owing to its mature API and wide support in the form of libraries and tools. For example cuFFT for Fast Fourier Transforms, cuBLAS for Basic Linear Algebra Solutions, NPP (nVidia Performance Primitives) for image and video processing and many others largely geared at mathematical and statistical problem solving (nVidia Corporation, 2012c). A recent development incorporated in the official CUDA SDK is *Thrust*, a productivity-based CUDA library which generalizes many common CUDA problems into easy-to-code C++ function calls (Hoberock and Bell, 2012; nVidia Corporation, 2011a,b, 2012c). While Thrust is not as efficient as a custom-made CUDA code in terms of GPU utilisation and memory usage, it takes good advantage of CUDA hardware and in most cases is far easier to code than custom CUDA code.

CUDA is able to achieve significant performance owing to its use of many lightweight threads, which run on its many *compute cores*. Standard CPU threads (such as POSIX threads) are run on multi-core systems with each core executing one thread at a time. If there are more threads than cores, active threads can be switched in turn with inactive ones. This form of

thread switching required a costly *context switch*. A thread's *context* differs from system to system but all contain at least a set of register values and an instruction pointer. Essentially, the thread's context consists of all the information needed by the thread to continue executing after it has been inactive. CUDA's thread model schedules many thousands of threads and runs these in groups of 32. The 32 thread groups (called *warps*) run in lock-step on multiple CUDA cores. This is in contrast to CPU threads which run independently on each core; and whose contexts, once started, resides in on-chip memory until the threads have finished execution.

Characteristics of CPU threads include expensive but infrequent context switches and memory latency hiding by use of multiple levels of cache on each CPU core. This model allows for what is called *task-parallelism*, as each thread can execute its own task independent of any other thread's task. In contrast, CUDA threads are characterised by limited caching ability but instantaneous and frequent context switches to hide memory latency, and although they are able to execute on blocks of data per clock cycle, they have limitations when it comes to executing divergent code. CUDA threads exhibit what is called *data-parallelism* as it is suited to computation of a single task upon many data points.

Chapter 3 goes into more detail on the CUDA programming and hardware models, the Thrust library, and common CUDA optimisation techniques.

CUDA FFT implementations: Owing to the importance of the Fast Fourier Transform (FFT) algorithm to the scientific community, HPC implementations of the FFT have been explored using shader based GPUs (Moreland and Angel, 2003), Field Programmable Gate Arrays (FPGAs) (Duan et al., 2011), and OpenCL (Li et al., 2011). There has been extensive interest in porting the FFT to CUDA, and many focus areas have been explored within this sub-field: precision (Govindaraju et al., 2008; Qi et al., 2011), heterogeneous CPU-GPU implementations (Ogata et al., 2008), large-scale FFTs (Chen et al., 2010), 3D FFTs (Nukada et al., 2008), and tuning schemes to yield as much performance as possible (Gu et al., 2010; Dotsenko et al., 2011).

CUDA in Astronomy: Within the field of astronomy, GPGPU technology has already been used to accelerate computationally intensive algorithms (Thompson et al., 2009). For example, a CUDA GPU implementation of gravitational lensing calculations runs roughly two orders of magnitude faster compared to a single core CPU implementation (Bate et al.,

2010). Monte Carlo dust temperature simulations by Jonsson and Primack (2009) produce speed-ups of up to 69 times over the CPU using CUDA hardware. Ford (2008) used CUDA hardware to accelerate calculations of Kepler's Equation for Exoplanet Searches and obtained speed-ups also in the $100\times$ range, approximately. More recently, Clark et al. (2011) utilises 79% of modern GPU's theoretical bandwidth for cross correlation calculations.

Although GPGPU techniques for calculating Fast Fourier Transforms (FFT) have been widely explored, there seems to be little research on GPGPU calculations of direct Fourier transforms.

2.3 Summary

This chapter outlines radio interferometry and how an interferometer (an array of radio dishes) is used to produce an image of the sky. Simulating direct interferometer output from a model image is known as *Point Source Visibility* (PSV) simulation and is an important procedure in the calibration loop; however, it is very computationally expensive. Fortunately, PSV simulations follow an SIMD paradigm, which is well explored in high performance computing technology.

In particular GPGPU technology has shown itself to be effective in accelerating SIMD problems, with the added advantage of low entry and maintenance costs associated with commodity hardware.

It follows that radio interferometry PSV calculations can benefit from GPGPU technology and techniques in terms of performance gain as well as cost effectiveness.

Chapter 3

CUDA

Modern GPUs possess a large number of simple Single-Instruction-Multiple-Data (SIMD) processors that can be harnessed for general purpose computing. In recent years, this task has been made easier with the development of general application programming interfaces for GPUs such as nVidia's *Compute Unified Device Architecture* (CUDA) technology, an SIMD model for general-purpose computation on nVidia commodity GPU hardware (Hoberock and Bell, 2012; nVidia Corporation, 2011a,b).

This chapter introduces CUDA device hardware, memory, and execution model. CUDA's unique execution centres around the use of its many lightweight threads. How these threads are executed on CUDA hardware and how they access the CUDA memory hierarchy is explained. This will give the detailed information needed to make sense of the optimisations and considerations explored in the design and implementation of our PSV node.

3.1   CUDA Programming Model

CUDA supports a C-like syntax for its code that is interoperable with standard C and C++. CUDA code is compiled and then 'deployed' to a *CUDA capable* device. The code is scheduled for execution by thousands of lightweight threads. These threads are divided amongst the device's many compute cores. Although it is claimed that CUDA code is designed to run on any CUDA capable device, in reality this is only partially true. A CUDA GPU has a *compute capability (CC)*, (1.0, 1.1, 1.2, 1.3, 2.0 or 2.1) with each CC generation backwards compatible with devices of a lower CC (for example a CC 2.1 device will run code designed
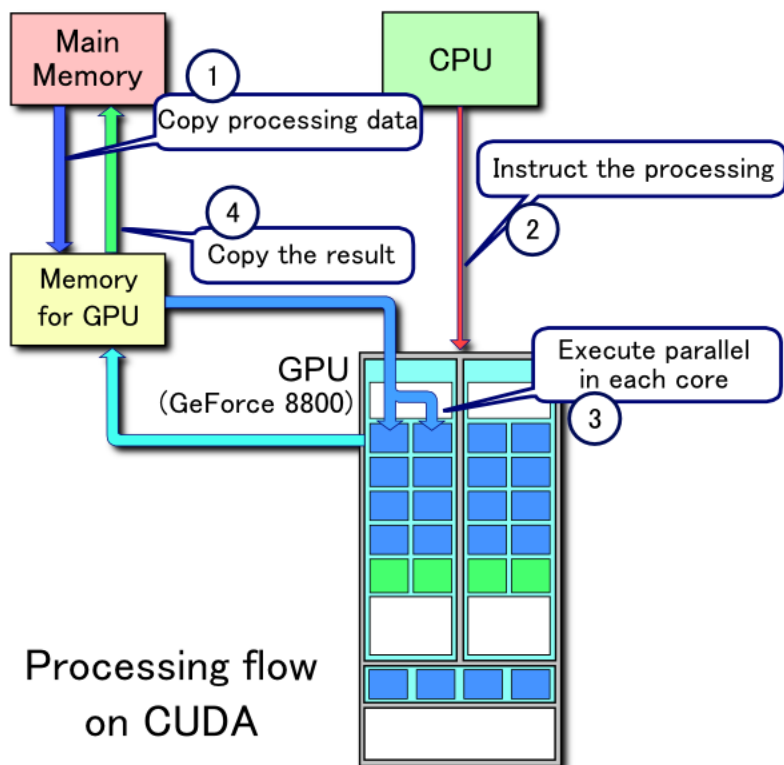
Figure 3.1: CUDA Processing Flow

This is a canonical example of CUDA processing flow. First, data that is to be processed is copied to the GPU device. CUDA code is compiled by the CPU and emphdeployed to the GPU and then executed in parallel on the many compute cores. The processed data is then copied back to the CPU.

for a CC 1.3 device). Devices with CC 1.3 or greater have double precision floating point operations. CC 2.x introduced automatic global memory caching and faster double precision operations (only half the speed of single precision performance, as opposed to 1.x devices which only have one-eight double precision performance).

The CUDA architecture defines two hardware abstractions: the *device*, which is a CUDA-enabled GPU, and the *host*, which is the machine on which the device resides. In order to execute code on the device, a C-like function (called a *kernel*) is executed or invoked. The kernel executes simultaneously on the many lightweight threads of the CUDA device. Data needs to be transferred to and from the device before and after, respectively, executing a kernel function (as shown in Figure 3.1).

There are two main considerations to a CUDA implementation: how data is transferred (memory management) and how this data is operated on (execution model). Unlike CPUs,

which use multiple levels of cache to hide memory access latency (and hence to ensure the processor is always being utilised), CUDA relies on its many lightweight threads and instantaneous thread context switching to swap between threads that are waiting for memory access requests and threads that require processing. This leads to scheduling characteristics and a memory hierarchy that contrasts sharply with a traditional multi-core CPU system.

3.1.1 The CUDA Execution Model

To understand the CUDA execution model, we first need to understand how GPU hardware and software components interact and how GPU instructions are scheduled. While seemingly unintuitive at first glance, the CUDA software model mirrors the characteristics of the hardware. Multiple schedulers are used to quickly switch between the many runnable threads. Threads are grouped into *blocks* and blocks are arranged into a *grid*. The hardware contains a number of *Streaming Multiprocessors* (SMs) that will be assigned a number of blocks. The SM is then responsible for scheduling execution of all the threads of all its blocks. The user does not control the schedulers directly, but has full control over the thread and block layout. These layouts along with other factors, explained below, allow the programmer to indirectly but deterministically affect efficient GPU utilisation.

Software Hierarchy - The Grid, Blocks, and Threads: CUDA code defines C-style functions called *kernels*, which are executed in parallel by thousands of lightweight threads on the many cores of the CUDA GPU. Threads are grouped into blocks, with up to 512 (CC 1.3) or 1024 (CC 2.x) threads per block. Blocks are grouped into a grid, which contains up to $65535 \times 65535 \times 65535$ blocks. Figure 3.2 shows an example grid/block layout. The execution of each block is independent of any other block, with no guarantee of block execution order and no direct mechanism for inter-block communication. Threads within the same block can communicate via *shared memory* (see Section 3.1.2).

Thread-Block Layout and Indexing: Blocks of threads are organised in 1-, 2-, or 3-dimensions and will be interchangeably referred to throughout as *blocks* or *thread-blocks*. Each thread-block dimension can be any size within certain CC specific limitations, but may not exceed 512 (CC 1.3) or 1024 (CC 2.x) threads per block. The grid (of thread-blocks) is also organised into 1-, 2-, or 3-dimensions. Grid dimensions can be any size up to 65535,
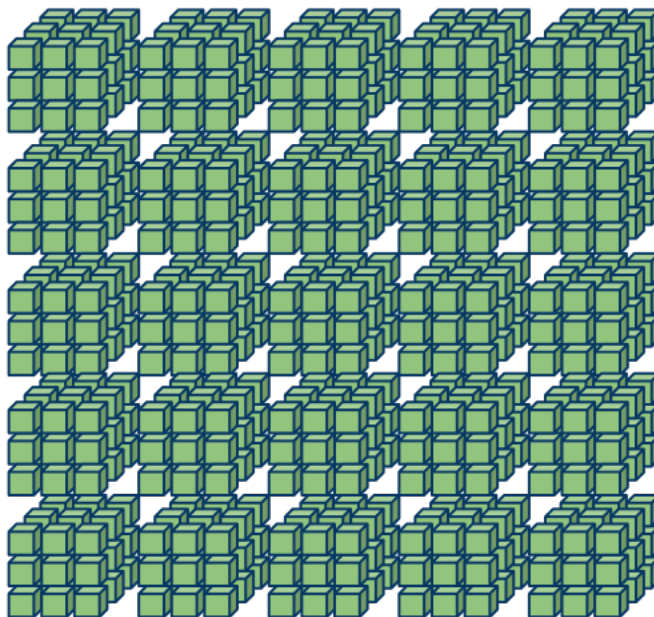
Figure 3.2: Example Grid/Block Layout
In the above, each green cube represents a thread*. In this case, there are* $5 \times 5 = 25$
blocks*. Each block contains* $3 \times 3 \times 3 = 27$ *threads.*

meaning that a total of 65535^3 blocks can be held in a grid. The layout of the grid is simply a programming convenience since, as will be seen below, blocks are scheduled on an SM in an undefined order. The block layout, however, can have significant effects on performance.

When a kernel is run, all the threads will run the exact same CUDA code. Thread indexing is thus vital in controlling a specific thread's execution path and the data it operates on. CUDA exposes two important variables, `threadIdx` and `gridIdx`, to allow the program to determine which thread from which block is currently undergoing execution. CUDA also uses `blockDim` and `gridDim` variables to determine the block and grid layout. Listing 3.1 shows an example CUDA kernel that squares all the values of a (512×512) array. If the programmer were to specify a thread-block size of, say, (8×4), they would need to specify a (64×128) grid. As standard CUDA arrays are 1-dimensional only, a multi-dimensional array would need to be *flattened* into a 1D array. Standard arrays are stored in *global* memory that resides on the DRAM of the GPU (see Section 3.1.2 for details). There are, however, non-global memory types available in CUDA which allow for multi-dimensional array access.

```
1  __global__ ExampleKernel(float* array, float w, float h) {
2
3      // find index of this thread
4      int indexX = threadIdx.x + (blockIdx.x*blockDim.x);
5      int indexY = threadIdx.y + (blockIdx.y*blockDim.y);
6
7      // we need to work out the flattened index for the flattened array
8      int flattenedIndex = indexX+(indexY*w);
9
10     // we read the value from the array, square it, and write it back
11     float value = array[flattenedIndex];
12     array[flattenedIndex] = value * value;
13 }
```

Listing 3.1: Example CUDA Kernel
This CUDA kernel will take a $w \times h$ flattened array and square its values.

Hardware - The GPU, Streaming Multiprocessors, and Cores: The CUDA execution model arose directly from the design of the CUDA hardware. A CUDA GPU contains a number of *Streaming Multiprocessors* (SMs), each comprising a number of scalar processors (SPs) or *cores* (either 8, 32, or 48 cores per SM). There is thus a direct mapping between the GPU hardware and the software components of CUDA: GPU-device to grid, SM to block, SP/core to thread (Figure 3.3).

nVidia's CUDA GPU cores were developed, in part, by combining vertex, geometry, and fragment shaders into a unified shader unit capable of executing all the necessary shader functionality. This unified shader unit became a single 32bit scalar processor — a CUDA compute core. These cores are organised into groups of 8 (CC 1.x), 32 (CC 2.0) or 48 (CC 2.1) and collectively define a Streaming Multiprocessor (SM).

Any CUDA architecture will include a number of SMs, each being largely independent of any other SM (nVidia Corporation, 2011b). This allows a CUDA GPU to scale by simply adding more SMs to the GPU chip. Indeed, the principal difference between low-end and high-end GPUs of the same series is usually the number of SMs on the chip and the amount of on-board DRAM.

Each SM contains a number of CUDA cores and a number of special function units (SFUs). These SFUs allow CUDA cores access to fast calculation of square-root, exponential, sine, cosine, and other commonly used mathematical operations. Each SM also has banks of shared memory, cached constant memory, and local registers that are shared between its cores. SMs

27
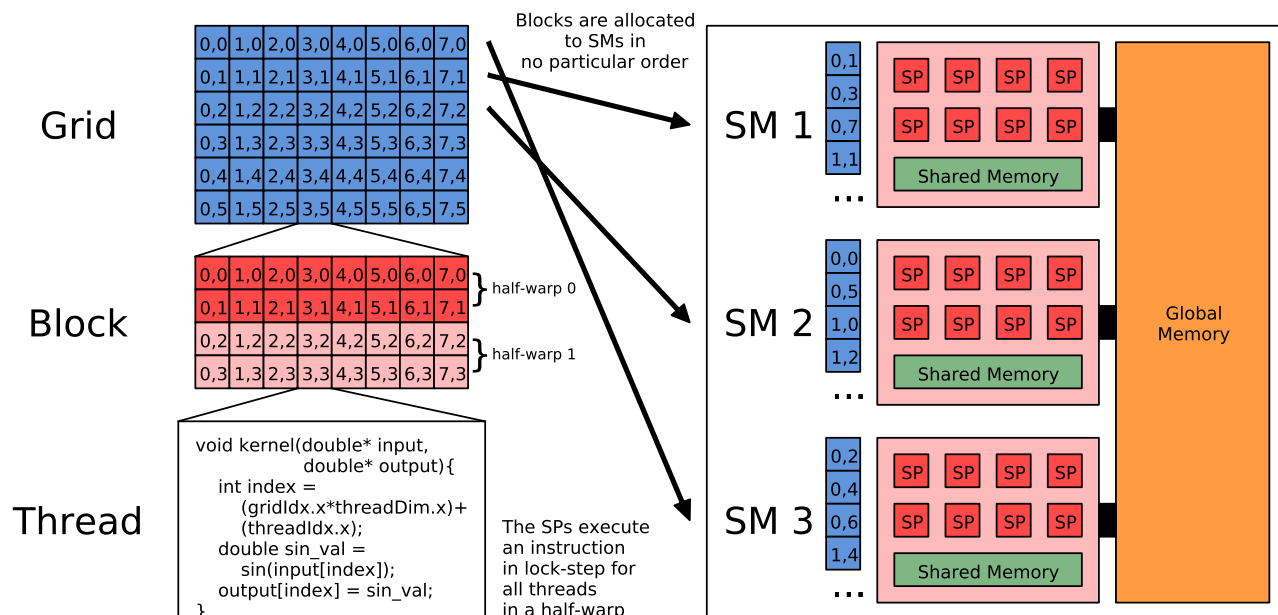
Figure 3.3: Thread/Block/Grid hierarchy relation to the GPU/SM/SP hierarchy

The blocks in a grid are allocated to the various Streaming Multiprocessors (SMs) in no particular order. The blocks are split into half-warps of 16 threads each. Thus each SM has a number of half-warps which it can execute at any one time. Each SM has a number of Scalar Processors (SPs) or 'cores'. The SM will select a half-warp and execute the kernel code on the SPs in lock-step (on each SP per thread). If threads within a block need to communicate, 16kB of on-chip shared memory is available. All threads can access global memory at any time.

have one or more double precision floating point unit processors for 64bit instructions. 2.x CC devices have L1 cache for global memory. Hardware specifications are given in Table 3.1.

Thread, Warp, and Block Scheduling: For any kernel execution, the numbers of thread-blocks and threads per block are statically specified before the kernel is invoked. When a kernel is executed, the CUDA device's *block-scheduler* schedules a number of thread-blocks to run on each of the SMs. Each SM splits its assigned thread-blocks into groups of 32 threads, called *warps* (for CC 2.x), or groups of 16 threads, called *half-warps* (for CC 1.x). For brevity, we say each SM holds b blocks resident at any one time and each block has t threads, which are divided into $w = \lceil t/32 \rceil$ warps. Thus, each SM holds $b \times w$ warps resident at any one time. We denote the i-th warp by w_i.

Compute Capability		2.1	2.0	1.3
CUDA cores	per SM	48	32	8
Special Function Unit	per SM	8	4	2
Double prec. FPU	per SM	2 clock-cycles		1
32bit registers	per SM	32K		16K
Threads	per Block	1024		512
x-dimension	per Block	1024		512
y-dimension	per Block	1024		512
z-dimension	per Block	64		
Grid	dimensionality	$3\ (x \times y \times z)$		$2\ (x \times y)$
Block	dimensionality	$3\ (x \times y \times z)$		
Warp schedulers	per SM	2		1
Instructions issued per cycle	per scheduler	2	1	1

Table 3.1: Compute Capability Specific Specifications

All nVidia devices operate in the same way, although devices with a different Compute Capability (CC) will have different hardware characteristics. Most notably the 2.x range of devices allows for larger thread blocks, has more cores per SM, more Special Function Units (SFUs) per SM (for functions such as sin, cos, sqrt etc), 3D grid allocations and additional warp schedulers. In the case of double precision operations, 1.3 devices have a dedicated double precision Floating Point Unit (FPU) whereas 2.x devices compute double precision operations built into the cores, but these take two clock-cycles to complete instead of one.

The SM needs to execute all the instructions of the CUDA code on all threads of each block. It does this by running an instruction on all threads of a warp in lock-step. This means that if, say, the SM is executing instruction 10 of warp w_5, then (simultaneously) core 0 executes instruction 10 of thread 0 of warp w_5, core 1 executes instruction 10 of thread 1 of warp w_5 and so on, for all threads in a warp.

Each warp has its own instruction counter and its own registers (its *context*), meaning that if instruction 10 of warp w_5 is executed, on the next clock cycle instruction 16 of warp w_0 can be performed, after which instruction 11 of warp w_5 can then be executed. This instantaneous *context switch* contrasts with classic 'heavy-weight' CPU thread switch, in which the instruction counter and registers must be copied to and from the CPU every time a thread is switched, incurring a delay. CUDA context switches incur no overhead as they involve only a switch to the next scheduled warp's instruction pointer and registers, both of which are already in on-chip memory (Figure 3.4).

This ability to instantaneously switch contexts between resident warps allows for effective
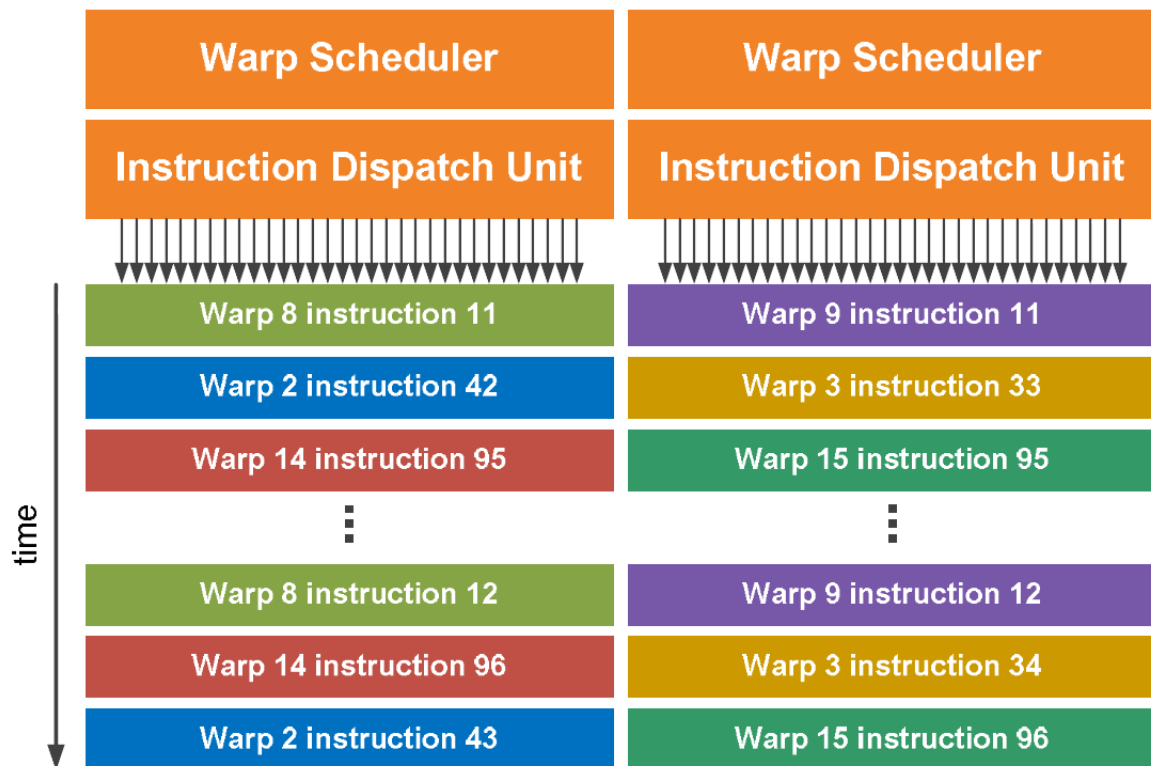
Figure 3.4: Example Dual Warp Scheduler

Schedulers need only ensure that warps belonging to the same block do not execute out of order, otherwise there is the possibility of blocks with threads that are ahead or behind other threads in its block. Beyond this restriction, any warp can be called by any scheduler at each clock cycle. In this case, warp 8 and 9 are of the same block and thus instruction 11 must be completed for those warps before proceeding to instruction 12. Similarly for warps 14 and 15. Warps 2 and 3 are in separate blocks and thus their instructions can be scheduled independently.

hiding of the memory access latency because the scheduler can switch from a warp that is waiting on a memory request to a warp that has operations to be executed. This characteristic assumes that there will always be a warp available that has instructions waiting to be executed. This is not always the case and the programmer should thus be aware this form of memory access latency hiding is enabled by supplying sufficient threads and thread-blocks to the CUDA GPU.

For a CC 1.x device, with only 8 CUDA cores per SM, the same instruction of a warp is sequentially run twice on each core so that all 16 threads of the half-warp are consistent (as it cannot leave some of the threads of a half-warp an instruction ahead or behind). During these two cycles, the warp scheduler is able to schedule the next warp for processing. For a CC 2.0 device, with 32 cores per SM, the warp can saturate all the cores and complete in one clock cycle. For a CC 2.1 device, with 48 cores per SM, two instructions are scheduled per clock cycle and thus all 48 cores can be saturated by one and a half warps. Details can be found in the nVidia Programming Guide (nVidia Corporation, 2011b).

Figure 3.3 shows a simplified visual layout of the CUDA hardware and execution model.

Flow Control and Divergent Code: In the case of a flow control instruction ('if', 'switch', 'do', 'while' or 'for') whereby the instruction path of some threads might diverge from that of others (i.e. *branch divergence* between threads in a warp), all execution paths are executed sequentially rather than in parallel. In other words, when an 'if' statement is encountered, the 'true' code-branch is executed first, followed by the 'false' code-branch; they are *not* executed at the same time as might be done in a traditional multi-core system. Threads that diverge down the 'true' branch will simply ignore instructions when the 'false' branch is being executed and the 'false' threads will ignore instructions in the 'true' branch.

Fortunately, if all threads in a warp follow the same execution path (i.e. all threads evaluate to 'true' or all evaluate to 'false'), only that particular branch ('true' or 'false') of code will be executed. If a single thread diverges from the rest, both branches will be executed. This is due to the lock-step fashion in which instructions are executed. Different threads are simply unable to perform different instructions on the same clock cycle.

Since blocks of threads are divided into half-warps (16 threads on 1.x) or full-warps (32 threads on 2.x), all 'if' statements in a half-warp /full-warp should evaluate to the same value and all loops should execute the same number of times to avoid a branch. This need

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

Table 3.2: *Register and Shared memory have very fast, on-chip access but is limited in size. Global, Constant, and Texture Memory reside off-chip, which is slower but far larger and have scope over all threads over all blocks for the entirety of the application's lifetime. Although constant and texture memory are read-only, they are both cached. Texture memory has support for 1D, 2D, and 3D spatial caching and constant memory is optimised for concurrent multiple thread access. Local memory* is effectively the same as global memory, except that it is only used if the register memory is over-allocated. Register allocations then 'spill over' into local memory. The slow off-chip access of register spilling should be avoided if possible.*
*For 2.x devices, global and local memory are cached.

not mean that all threads in a block need to follow the same execution path, it is only the half-warp/full-warp that must have consistent flow-control.

3.1.2   CUDA Memory Hierarchy

The CUDA memory hierarchy offers several memory types with different scope and caching characteristics. Table 3.2 summarises the types of memory.

Memory on a CUDA device is divided into two broad categories: *on-chip* and *off-chip* memory. On-chip memory is analogous to a CPU chip's register and cache memory: it is fast (one clock cycle to access) but limited in size: (approx 100kB - 1MB on the CPU). A CUDA GPU has 16kB-64kB of on-chip memory per SM. Off-chip memory refers to the on-board DRAM that accompanies all CUDA chips. This memory is far slower to access than on-chip memory (400-600 clock cycles to access) but is far larger (up to 2GB).

Off-chip memory has four forms: standard read/write global memory, 1D/2D/3D spatially cached read-only texture memory, cached read-only constant memory, and local memory for register overallocation (see Table 3.2). Any thread from any block can access any off-chip memory location. In the case of global memory, if a number of threads access a contiguous

block of memory simultaneously, these requests are *coalesced* into aligned 128bit memory requests instead. This could, for instance, reduce four 32bit requests into a single 128bit request. Figure 3.5 shows this effect in greater detail.

On-chip memory is split into *register memory* and *shared memory*, with 2.x devices featuring a 48kB of L1 cache to global memory. Each SM has 16kB of shared memory that is split equally amongst the resident blocks. 2.x devices allow use of a portion of L1 cache as shared memory, which increases shared memory size to 48kB and reduces L1 cache to 16kB. Each SM also has 16kB-32kB of register space that is split equally amongst the threads of the resident blocks. Registers are again subdivided between threads of each resident block. If there is not enough register space, registers are *spilled* to off-chip *local memory*, with a catastrophic increase in memory access time.

3.2 Thrust

Thrust is a productivity-based CUDA library which generalizes many common CUDA problems into easy-to-code C++ function calls, intended to mimic the coding conventions of the Standard Template Library (STL) (Hoberock and Bell, 2012). CUDA arrays are defined as `device_vector`s and all Thrust functions take as parameters a number of iterators and possibly a functor that operates on these iterators. For instance, if we have an `device_vector<int>` a vector with an iterator `device_vector<int>::iterator a_itt`, we can use a function call of the form `thrust::reduce(a_itt, plus<int>)` where `plus<int>` is the integer addition functor/function. To sort the array we would use `thrust::sort(a_tt)`. This is far easier to program than creating a whole CUDA kernel from scratch, and is already largely optimised.

In order to accommodate constants, special permutations, and transformations there are (respectively) constant, permutation, and transformation iterators as well as a zip iterator to combine two or more vectors/iterators into a tuple vector. Chains of iterators can be used to form more complicated operations. For instance, if we have a number of 2D points in vectors `x` and `y`, we could use a zip iterator to iterate over them as a tuple (`x, y`). If we need to iterate over every third point, we must layer the zip iterator inside a permutation iterator. Unfortunately as code complexity increases, Thrust becomes unwieldy to code and debug.

Aligned and sequential

Addresses: 96 128 160 192 224 256 288

Threads: 0 ... 31

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Aligned and non-sequential

Addresses: 96 128 160 192 224 256 288

Threads: 0 ... 31

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Misaligned and sequential

Addresses: 96 128 160 192 224 256 288

Threads: 0 ... 31

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	7 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224 1 x 32B at 256	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

Figure 3.5: Coalesced Global Memory Access

Images taken from nVidia's CUDA C Programming Guide "Examples of Global Memory Accesses by a Warp, 4-Byte Word per Thread, and Associated Memory Transactions Based on Compute Capability" (nVidia Corporation, 2011b). If the access is sequential and 64byte/128byte aligned, all devices exhibit coalescing. CC 2.x devices have the added advantage of cached global access, and exhibit the most favourable coalescing characteristics as they are able to handle misaligned and non-sequential accesses better than older devices. CC 1.0 and 1.1 devices are not considered in this work owing to their lack of double precision support, but nonetheless exhibit the most fragile of coalescing characteristics, being unable to coalesce either misaligned or non-sequential accesses.

3.3   General GPU Optimisation Techniques

Naive CUDA implementations usually result in at least a 2-3 times speed-up over fairly optimised CPU code. Gaining significant speed-ups requires careful optimisation that considers the CUDA hardware and execution model. There are a number of common CUDA optimisation techniques that apply to a wide range of computational problems.

The *CUDA C Best Practices Guide* (nVidia Corporation, 2011a) contains a comprehensive overview of CUDA related optimisations and techniques to maximize performance. The following subsections are taken largely from this guide.

This section is split into roughly two parts, first part focusing on memory-based optimisations, and the second part focusing on processor-based optimisations.

3.3.1   Memory-Based Optimisations

As with many HPC technologies, memory throughput is as significant a bottleneck as processor utilisation as low memory throughput leads to processors wasting valuable processing time waiting for memory requests. CUDA is no different and data transfer from both CPU RAM to GPU on-board memery and GPU on-board memory to GPU on-chip memory can be a source of significant performance bottlenecks.

Latency Hiding using Asynchronous Transfers: Many CUDA computations require multiple kernel invocations. More often than not this requires (i) data to be copied from device to host, (ii) some processing on the GPU followed by (iii) a potential copy back of data from device to host. This process may need to be repeated several times.

While data is being copied to and from the host, the GPU sits idle. In order to utilise the GPU during this time, asynchronous data transfer can be used. This allows for the GPU to continue processing one set of data while another set of data is being copied to or from the host. Asynchronous transfers are well supported in CUDA and allows for hiding of significant memory transfer latency between host and device. This does assume, however, that the device has sufficient processing jobs to cover the memory transfer latency and sufficient GPU DRAM to store multiple datasets (as at least two data sets are required, one to be processed by the kernel and one to be copied).

Global Memory Coalescing through Sequential Memory Access: An important concept in global memory access is that of organising data stored in global memory into 32, 64, or 128bit aligned sequential segments, since the SMs request data from addresses in either 32, 64, or 128bit transactions (depending on the CC) and *coalesce* these memory transactions. If a number of threads in the half-warp (or warp) requests global memory data from sequential memory addresses in the same instruction, these requests are combined or coalesced into fewer 64bit or 128bit requests containing the data requests of two or more threads in one. Un-cached global memory requests incur a $400 - 600$ clock cycle delay. Thus by decreasing the number of memory requests from global memory though coalescing, off-chip memory throughput can be drastically improved.

The exact rules for coalescing can be found in Appendix G.3.2 and G.4.2 of the CUDA Programming Guide (nVidia Corporation, 2011a). A visual example is given in Figure 3.5

Banked Shared Memory Access: Shared memory is organised into 16 (1.x) or 32 (2.x) banks. Each bank can be accessed simultaneously by the SM's cores once per clock cycle. This means 16/32 potential shared memory reads per cycle. The memory banks are organised so that each successive 32bit word is assigned to each successive bank.

This is especially important for how a warp accesses shared memory. Say we have a warp that wishes to access shared memory, and that shared memory has 32 banks. If each thread in a warp accesses its own unique memory bank then all those requests will be fulfilled in the same clock cycle. However if 2 threads access different words in the same bank, this request must be serialised into 2 separate requests and will take 2 cycles to complete. This is known as a two-way bank conflict. Figure 3.6 shows how strided access or access of contiguous 64bit data-types can cause a two-way bank conflict.

Using Shared Memory to Reduce Global Memory Writes: During kernel execution a common optimisation strategy to reduce the number of global memory accesses is to use shared memory rather than global memory to do intermediate processing. In general, shared
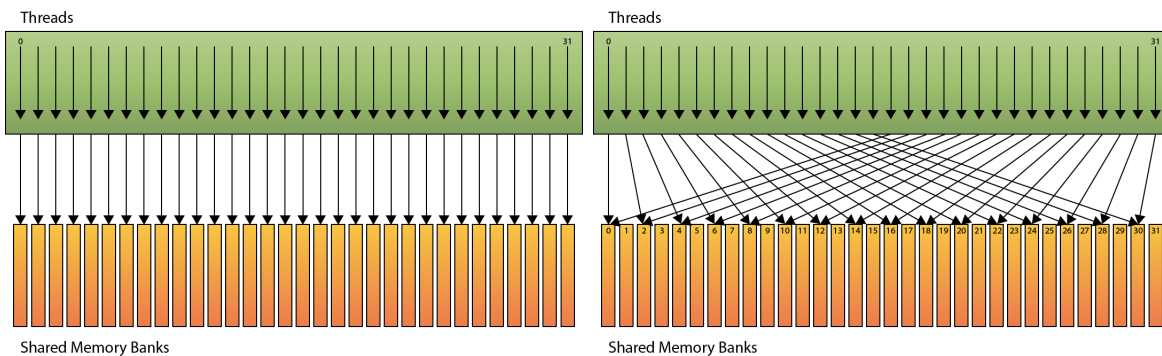
Figure 3.6: Shared Memory Banked Access
Both examples show the threads at the top accessing the shared memory banks at the bottom. The left example shows each thread accessing each consecutive 32bit shared memory bank (thread 0 accesses address 0, thread 1 accesses address 1, etc.). In this case, there are no bank conflicts and all 32 operations occur in the same clock cycle. However, on the right, each thread accesses every second 32bit element (either accessing non-sequentially or accessing contiguous 64bit data-types).In this case, thread 0 accesses address 0, thread 1 accesses address 2 and so on, except that now thread 16 will access address 32, which is in the same bank as address 0. Thus a two-way bank conflict is encountered and 2 clock cycles are required.

memory is used as follows:

1. Transfer data from global memory into fast, on-chip, shared memory

2. Process this data using shared memory

3. Copy back the processed data from shared memory to global memory

A shared memory solution usually results in reduced global memory reads and writes, and this can speed-up execution time, especially for memory-bound problems. Because of the limited size of shared memory, data may need to be processed in batches. Most problems can benefit from the use of shared memory to reduce the number of global memory accesses. However, this strategy must be tailored to the memory access characteristics of problem.

3.3.2 Processor-Based Optimisations: Occupancy

Each SM can hold a limited number of thread-blocks in its on-chip memory at any given time. Since all the *resident* (or active) blocks on an SM are in on-chip memory, switching between them is a matter of switching a pointer, which is instantaneous. However, for

technical reasons, the number of active blocks actually resident on an SM might be less than the maximum it can hold.

This concept is broadly expressed as *occupancy*. Low occupancy means that there are fewer resident blocks on the SM and thus limits the number of warps that can be scheduled and utilised by an SM's warp scheduler. A simple definition of occupancy is the *actual* number of blocks that can be resident on an SM at any given time over the *maximum* number of blocks that can be resident on an SM at any give time. The actual definition is more detailed — the reader is referred to nVidia Programming Guide (nVidia Corporation, 2011b) and the CUDA C Best Practices Guide (nVidia Corporation, 2011a). Nonetheless a fairly detailed explanation with examples is given below.

The maximum number of resident blocks per SM is determined by the block size. An SM can only hold a certain number of warps resident (32 warps for CC 1.3, 48 for CC 2.x) and each block will use a certain number of warps, w. The maximum number of resident blocks per SM is therefore $\lfloor \frac{w}{32} \rfloor$ for CC 1.3 and $\lfloor \frac{w}{48} \rfloor$ for CC 2.x. The actual number of resident blocks per SM is determined by the following three factors:

Warps per SM hardware limitation : Each SM can only manage a certain number of warps at any one time (32 warps for CC 1.3, 48 for CC 2.x). For all CCs, SMs can only manage 8 blocks. If there are many threads per block and thus many warps per block, the SM will not be able to hold as many blocks. For example, if the blocks have 16 warps (512 threads) each, and the SM can only hold 48 warps, then the SM can only hold $\lfloor \frac{48}{16} \rfloor = 3$ blocks. On the other hand, if each block has only 2 warps (64 threads), then the SM can technically hold up to $\lfloor \frac{48}{2} \rfloor = 24$ blocks. An SM can only hold a maximum of 8 blocks, regardless of the maximum theoretical number of warps it can hold. Thus only 8 blocks (16 warps) can be held. This represents an under-utilisation of the SM (only $\frac{16}{48} = 33\%$).

Shared memory limitations: Every block requests an amount of shared memory before a kernel invocation. Every SM, however, has a limited number of shared memory to divide amongst its resident blocks (16kB for CC 1.3, 48kB or 16kB for CC 2.x). If too much shared memory is requested per block, then the SM can only accommodate a reduced number of blocks at any time. For example if each block requests 12kB of shared memory, and the SM only has 48kB, then only $\lfloor \frac{48 \, \text{kB}}{12 \, \text{kB}} \rfloor = 3$ blocks can be active at any given time. However, if only 3kB per block is requested, then $\lfloor \frac{48 \, \text{kB}}{3 \, \text{kB}} \rfloor = 12$ blocks can be active. Since hardware

limits the number of resident blocks on an SM to 8, the ability to allocate 12 blocks to an SM means that a total of 4 blocks worth of shared memory ($4 \times 3\text{kB} = 12\text{kB}$ in this example) could be allocated with no loss in occupancy.

Limited number of register space per SM: Each thread requires a certain number of registers (depending on the kernel code) but there is only a limited amount of register space per SM (16kB for CC 1.3, 32kB for CC 2.x). If a kernel uses too many registers it will cause a decrease in the number of active threads that the SM can concurrently run. For example, if each thread requires 45 registers and each block contains 256 threads, then the block requires $20 * 256 = 5120$ registers. On a CC 1.3 device, only $\lfloor \frac{16384}{5120} \rfloor = 3$ blocks can be resident on the SM. On a CC 2.x device, $\lfloor \frac{32768}{5120} \rfloor = 6$ blocks per SM can be resident. Increasing the number of threads per block decreases occupancy as more registers are needed. Decreasing the number of threads per block might increase occupancy, but at the risk of under-utilising the SM.

Any of these three factors will limit the number of blocks that can be resident on an SM, and thus limit the occupancy. Higher occupancy usually means better performance. However, 100% occupancy does not necessary mean that the best block/thread layout has been used. Similarly, a low occupancy does not necessarily mean that the solution can be improved.

3.3.3 Occupancy-Related Considerations

Spatial Organisation and Optimal Thread-Block Size

Splitting up the computation into blocks and threads allows for spatial organisation of the computation. This not only has the advantage of making more intuitive sense to the programmer, but also allows spatial caching to come into effect, especially when texture memory is used. Consider an image in which each pixel must undergo some form of transform. The image can be split into 16×16 sub-images, with each sub-image assigned to a block with 16×16 threads, and each thread operating on a single pixel. Spatial caching ensures that for any query pixel, the neighbouring pixels are cached. Since CUDA enables each thread to access its location within its block, as well as its block's location in the grid, a pixel's location and the location of its neighbouring pixels can be calculated.

Latency Hiding

GPUs use fast context switching and a very large number of schedulable threads to ensure their cores are always occupied with executable instructions. This is mainly used to hide memory access latency and differs from the model used by CPUs, in which large, multiple-level caches are used hide memory latency.

Reading and writing to the GPUs DRAM can incur a 400-600 clock cycle latency. This latency can be hidden by occupying the SM with a sufficient number of threads that the warp scheduler may call upon to occupy the CUDA cores. The number of threads needed to hide global memory access latency depends on how many global accesses are performed and the number of threads/blocks that are active on the SM.

It must be stressed that too many global memory accesses cannot be hidden by thread scheduling alone. It is thus always a good idea to reduce global memory accesses whenever possible.

Block Size

There are no simple rules to specify the number of blocks and threads that should be used for any particular problem. Unfortunately, different sized grids and blocks usually have a drastic effect on performance.

Larger blocks mean that the limited available registers and shared memory must be divided amongst more threads. This can reduce occupancy on the SM, and runs the risk of *register spilling*. Register spilling occurs when there is insufficient on-chip space for registers and slow, off-chip memory is used instead. This is a very undesirable outcome as local memory incurs the same 400-600 clock cycle delay that global memory incurs.

Conversely, too few threads per block means that the SM might be under utilised and also will not have enough warps available to hide global memory access latency effectively. A concurrent consideration is that the dimensions of the block will have significant effects on memory access patterns, which in turn affects coalescing and caching efficacy.

Careful planning and a fair degree of testing is required to find the optimal block size for a particular problem. Two important guiding 'rules' can, however, be followed: (i) The number of threads in a block should be a multiple of the warp size, otherwise the last warp

in every block will have unused threads, and thus unused cores. (ii) A larger number of smaller blocks is generally preferable to a smaller number of larger blocks in order to better occupy the SMs, as well as to ensure that the grid has enough blocks to allocate to all the SMs on the GPU.

Grid Size

Generally the grid needs to contain enough blocks to occupy all the SMs. An insufficient number of blocks means that the SMs might not be allocated enough blocks to run at full occupancy. Larger grids incur no extra overhead, so grids can and should be as large as possible.

If we have a GPU with S SMs, with each SM being able to hold B blocks resident at any time, the grid need only be larger than $S * B$ in order to ensure that all SMs are maximally occupied.

Chapter 4

Design and Implementation

This chapter introduces the MeqTrees execution model and briefly outlines the Thrust PSV prototype. The main body of the chapter is the description and discussion of the CUDA design and optimisation considerations such as memory requirements, effective usage of shared memory, and multidimensional array indexing. The remainder of the chapter outlines and details the numerous steps in the CUDA implementation and explores issues that arose during development.

4.1   Problem Definition

An interferometer, typically an array of radio dishes, will produce a set of *visibilities* by feeding it's individual voltage outputs to a correlator which correlates signals from pairs of dishes. Visibilities are collected by each pair of antennae every few seconds over a period of several hours. These visibilities are also filtered into various frequency buckets.

As we are simulating visibilities rather than observing them, we have to simulate a visibility for each pair of antennae for each time-step and for each frequency. We do this by means of a *point source sky intensity map*. The sky intensity map is simply the brightness of the observed section of the sky. We are given a sky intensity map comprising many *point sources*. A source is simply any observable entity in the sky. A point source is, as the name implies, a source that is described by an infinitely small, but observable, dot. This contrasts with a complex source which has a definite (or resolved) shape. A point source sky intensity map

is defined as a list of point sources, where each point source s, has a location $\boldsymbol{\sigma}_s$ and a brightness $\boldsymbol{B}_s$.

The following equation defines a *visibility* between two interferometer antennae, p and q, for S point sources at time t and frequency ν, as defined in Appendix B:

$$
\begin{aligned}
V'_{pq}(t,\nu) &= \sum_S \operatorname{sinc}\frac{\Delta\boldsymbol{\Psi}(\nu_m)}{2}\operatorname{sinc}\frac{\Delta\boldsymbol{\Phi}(t_m)}{2}K_{sp}(t,\nu)B_s K_{sq}^H(t,\nu) \\
&= \sum_S \operatorname{sinc}\frac{\Delta\boldsymbol{\Psi}(\nu_m)}{2}\operatorname{sinc}\frac{\Delta\boldsymbol{\Phi}(t_m)}{2}B_s e^{-2\pi i\frac{\nu}{c}(\boldsymbol{u}_{tpq}\boldsymbol{\sigma}_s)}
\end{aligned}
\tag{4.1}
$$

Here c is the speed of light, $\boldsymbol{u}_{tpq}$ is the relative position of antennae p and q at time t, and $\operatorname{sinc}\frac{\Delta\boldsymbol{\Psi}(\nu_m)}{2}$ and $\operatorname{sinc}\frac{\Delta\boldsymbol{\Phi}(t_m)}{2}$ are smearing factors defined in Appendix B.4.

In the MeqTrees framework, the t values are partitioned into groups of typically 16 or 32 (but this is user definable), which we will call *time-slot groups*. Visibilities for each antenna pair (p,q) and for each time-slot group are calculated sequentially. Thus we are tasked with creating a MeqTrees node that calculates $V_{pq}(t,\nu)$ for each t in the time-slot group and for all ν.

Our approach is to decompose this PSV calculation by splitting the problem into its three intrinsic dimensions — *sources* (S), *time-slot group* (T), and *frequency* (F). The same visibility calculation is applied to each point source s, across each time-slot t, and frequency-band ν. This is a Single-Instruction-Multiple-Data (SIMD) computation which is well-suited to a GPU implementation.

To tackle this problem we assign a single GPU thread to each point in this 3D data-set of size $(T \times F \times S)$. Each thread calculates the visibility for its specific (t,ν,s) sample. To optimise this problem, we calculate multiple sources per thread in order to leverage shared memory and reduce global memory accesses. Another pass is run to reduce the data-set over S, summing the values to a $(T \times F)$ data-set. Figure 4.1 shows an example of this data-cube.
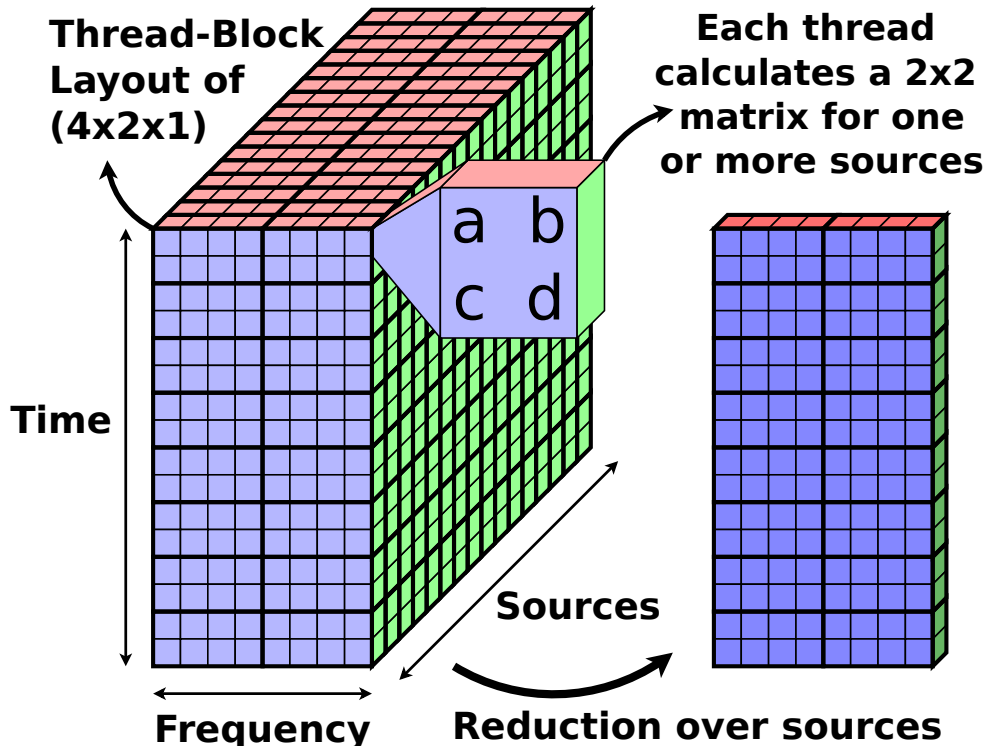
Figure 4.1: Visual representation of the PSV data-cube
In this case, we have a data-cube sized $16 \times 8 \times 32$ for the three intrinsic dimensions $(T \times F \times S)$. This is split into 2×8 thread-blocks, with each thread in a block calculating the 2×2 PSV matrix for one source or the sum of matrices for multiple sources. The matrix is actually flattened in memory, represented in one dimension $\{a,b,c,d\}$. A reduction step sums the data-cube over the source dimension, resulting in a $T \times F$ array of 2×2 matrices.

4.2 MeqTrees Execution Model

The MeqTrees framework defines functions in *Measurement EQuation Trees* or expression trees (Noordam and Smirnov, 2011). For example, Figure 4.2a shows Eqn (4.1) expressed as a tree in MeqTrees. This abstract use of nodes to form an equation tree allows for easy construction of the complicated equations needed in radio interferometry.

The user might define only one tree, or they might instead define a 'forest' of trees that are executed in any desired order. All MeqTrees nodes and forests are specified in Python scripts and are thus relatively easy to define.

```
1  def run(timeslot_size, num_timeslot, antennae_pairs):
2      # for each timeslot block (0-7, 8-15, 16-24, ...)
3      for t in range(0, num_timeslots, timeslot_size):
4          # for each antennae pair
5          for p in antennae_pairs:
6              end_t = min(num_timeslots, t+timeslot_size)
7              # run the Node for pair p, over [t, ..., end_t-1] timeslots
8              calc_visibilities  (p, t, end_t)
```

Listing 4.1: Python-style pseudo-code of PSV execution in MeqTrees
`calc_visibilities(p, t, end_t)` will run the PSV tree (normal or monolithic) which calculates the visibilities for antennae pair `p` over timeslots `[t, end_t-1]` for all frequencies.

4.2.1 Point Source Visibility Execution Model in MeqTrees

MeqTrees can run calculations by defining a forest of trees (see Figures 4.2a and 4.2b for examples). The manner in which these forests are defined is completely up to the user. In the case of our PSV calculations, each tree in the forest represents one antenna pair and a range of time-slot groups. During computation, each tree is executed sequentially, as described below. Listing 4.1 shows the execution model in pseudo-code.

For example, if we have five antennae $\{0, 1, 2, 3, 4\}$, the PSV for the antenna pair $(0, 1)$ would be completed, then $(0, 2)$ and so on until all pairs are calculated. Self pairs such as $(0, 0)$, $(1, 1)$ etc, equate to 0 and are simply ignored (see Appendix A.7).

Each PSV node only computes a maximum of 32 time slots per antenna pair, thus splitting each antenna pair's calculations into groups of 32 (which we call time-slot groups). Groups of 32 time-slots are a limitation set in the MeqTrees framework. Given this limitation, with the above calculation for five antennae, $\{0, 1, 2, 3, 4\}$ over 50 timeslots, MeqTrees would calculate the base-pair $(0, 0)$ over $t \in [0, 31]$, then $(0, 1)$ over $t \in [0, 31]$ until all base pairs are completed. It would then continue to the next time-slots group: $(0, 0)$ over $t \in [32, 50]$, then $(0, 1)$ over $t \in [32, 50]$ and so on.

Point Source Visibility Tree/Node: The MeqTrees framework contains two versions of the PSV module: A traditional tree (Figure 4.2a) and a single node 'tree' (Figure 4.2b). The single node tree foregoes MeqTrees's usual method of equation definition (a tree of nodes) and combines all the calculations into one node. This is less intuitive but reduces

(a)

(b)
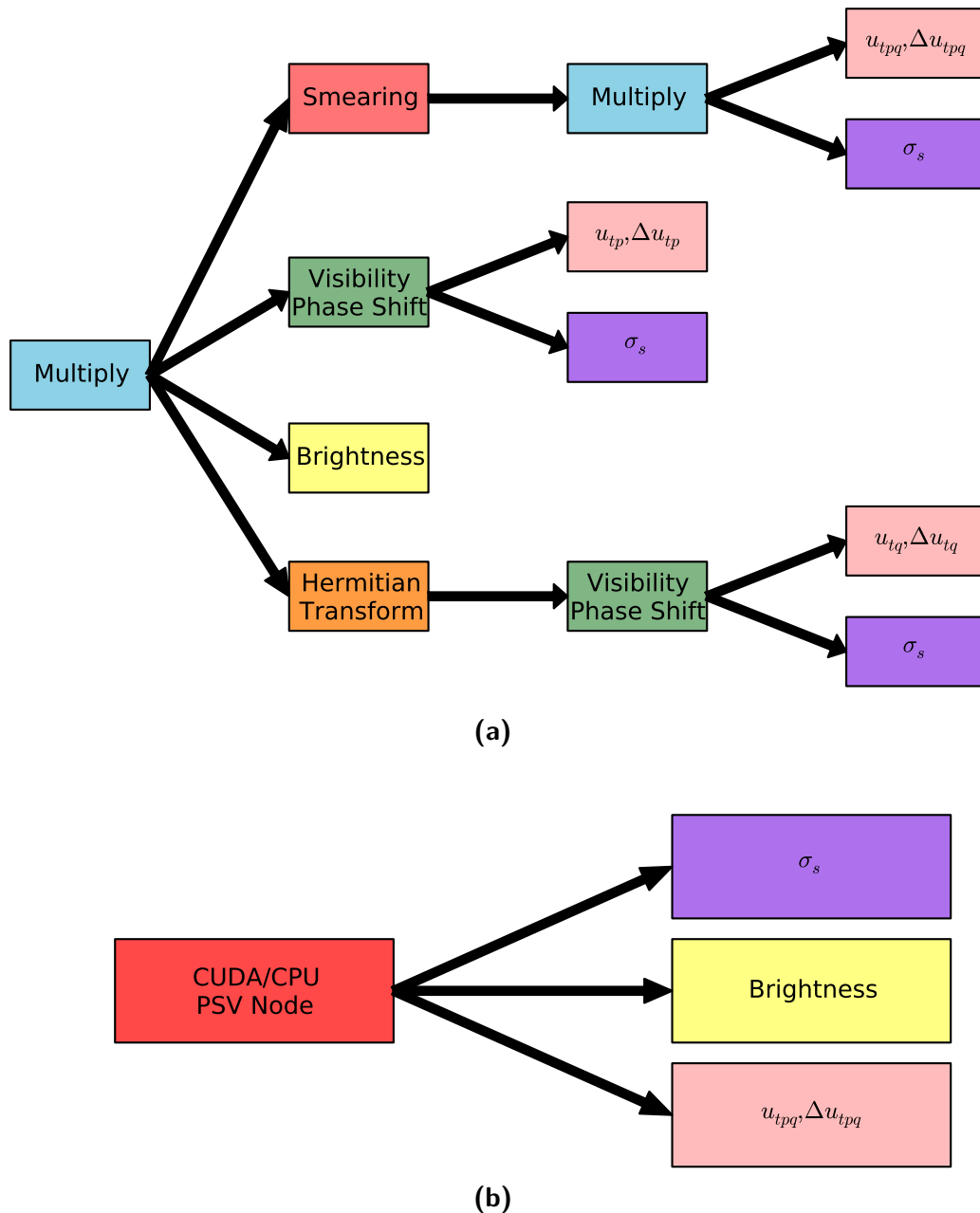
Figure 4.2: GPU/CPU PSV Measurement Equation Trees

(a) Tree expansion of Point Source Visibility Calculation representing the calculation $S_{pq}K_{sp}(t,\nu)B_sK_{sq}^H(t_m,\nu_m)$. The many simple nodes show how flexible MeqTrees is, allowing a user to design a problem flow visually. However, this increased number of nodes does create some MeqTrees overhead. (b) Tree expansion of Point Source Visibility Calculation for CUDA implementation. Compared to the original tree (a) this is far simpler, thus reducing overheads. However, the inner workings of this PSV node are far more complicated.

node-to-node overheads that are present in MeqTrees.

Our implementation is a port of the single node PSV module. This was done to avoid the extra node-to-node overheads, and to avoid the need for multiple GPU implementations of more general computations (such as signal phase-shifts, Hermitian transforms, matrix multiplication, etc.), which lie beyond the scope of this work. The inner workings of the CUDA version of the single node are described by the flow chart in Figure 4.3 and detailed in Section 4.5.

4.2.2 MeqTrees Memory Management

MEQtrees has a "straightforward but very powerful scheme of *dependency tracking* [that] allows a node to figure out when a result may be usefully cached [...]" (Noordam and Smirnov, 2011). This allows for persistent storage of data in CPU RAM, which in turn gives easy access to that data from any other node. Unfortunately the mechanisms that allow for persistent allocation and storage are only implemented for CPU memory and not GPU memory. This leaves all GPU node executions agnostic to any previous allocations on any other GPU node.

Whilst a GPU node does not have direct access to data allocated on CPU RAM, it can be connected with a child node that preallocates that data, which it does have access to. An analogous allocation node for data allocated to the GPU RAM is not supported and would require changes to the internal workings of MeqTrees or at very least an undesirable and potentially messy workaround. Porting the memory allocation to work with GPU memory would be an important addition for any future potential MEQtrees GPU implementation. This is left for future work.

4.3 Thrust Prototype Implementation

We initially created a PSV implementation using Thrust but quickly discovered that the PSV's many inputs and the order in which they need to be permuted caused Thrust's layered iterator approach to become too unwieldy to code and debug. The custom CUDA implementation includes the calculation of 'uncorrupted visibility' equations, augmented with 'time/bandwidth smearing'. Time/bandwidth smearing was never attempted with Thrust,

only the uncorrupted visibility equation, see Appendix Eqn (B.10). At the time this work was undertaken, Thrust's reduce-by-key function demanded so much memory that the Thrust implementation needed up to four times more memory than that of our custom CUDA PSV implementation.

Listing 4.2 shows the way in which Thrust performs transformations using functors, which is a class function that accepts one input (which may be a tuple, allowing for multiple inputs) and produces one output. The Thrust transform takes an input array and calculates the output for each element in parallel using the functor. Our problem has many inputs, most of which must be computed for each element in other input arrays since we are essentially calculating a 3D cube of data. The output is dependent on on multiple inputs in various combinations and leads to the need for multiple layers of Thrust iterators.

The Thrust prototype was implemented as follows: a PSV functor was created that takes as parameters $\boldsymbol{\nu} \in \mathbf{R}, \boldsymbol{u_{tp}} \in \mathbf{R}^3, \boldsymbol{\sigma_s} \in \mathbf{R}^3, \boldsymbol{B} \in \mathbf{R}$ (an 8-tuple) and calculates the visibility as in Eqn (B.10). This has to be done for all $\nu \times t \times s$ permutations. First, the 8 inputs are placed into a zip iterator. Each of these input arrays must be permuted in a certain order such that all $\nu \times t \times s$ are covered. Permutation iterators require that the permutations are stored on the GPU device. This takes up valuable GPU memory. The workaround for this is to use a counting iterator (an iterator that counts through the natural numbers: $0, 1, 2, ...$) and to wrap that in a transform iterator that converts the counting iterator to the appropriate indices required by the permutation iterator. This way the permutation iterator is referencing the index as calculated 'on the fly' by the transformed counting iterator, rather than pre-calculating the indices and storing them on the GPU.

Below shows an enumeration of the different Thrust iterator layers as they are added. We start with a number of input vectors stored in GPU memory `device_vector<double>`, each of which has an iterator `device_vector<double>::iterator` (which we shall call `DoubleIterator`)

1. Start with a counting iterator $(0, 1, 2, 3, 4...)$
   - `thrust::counting_iterator<int>`

2. This must be transformed so that the flattened multi-dimensional arrays are permuted correctly, the transformation is called `IndexFlatten`:
   - `thrust::transform_iterator<IndexFlatten, thrust::counting_iterator<int>>`
   We shall call this `TransformedIndexIterator`

3. We permute along each `device_vector<double>`:
   - `thrust::permutation_iterator<DoubleIterator, TransformedIndexIterator>`
     We shall call this `TransformedPermuteIterator`

4. We define a `TransformedPermuteIterator` for each of the input vectors, `freq` ($\nu \in \mathbf{R}$), `u`, `v`, `w`, ($\boldsymbol{u_{tp}} \in \mathbf{R}^3$) `l`, `m`, `n` ($\boldsymbol{\sigma}_s \in \mathbf{R}^3$) and `brightness` ($\boldsymbol{B} \in \mathbf{R}$) and zip them together into a tuple:
   - `thrust::zip_iterator(freq, u, v, w, l, m, n, brightness)`

5. We transform this zipped tuple iterator to obtain an array of unreduced output (`unreduced_output`) using the `PsvFunctor` as described in Listing 4.2:
   - `thrust::transform(zipped_data_iterator, unreduced_output, PsvFunctor())`

6. This output is reduced (or summed) over the sources using:
   - `thrust::reduce_by_key(source_key_iterator, unreduced_output, output, equals<double2>, add<double2>)`

7. We now have our final `output` vector, which is copied back to the CPU and stored

This implementation was successful but only yields a 4× speed-up compared to the CPU version. Whilst Thrust has proven itself a good high-level programming library for performance code on CUDA GPUs, it is the opinion of this author that the multiple layers required for this problem makes implementation and debugging just as challenging as custom CUDA code. It was thus decided that the custom CUDA version held more potential for better performance.

4.4 CUDA Design considerations

Our CUDA implementation requires careful consideration of the execution model and GPU architecture in order to obtain maximum parallel execution speeds. For this work, CUDA devices of compute capability (CC) 1.3 or greater are required, since double precision floating point operations are not supported on lower CCs. CC 2.x devices are recommended since they have features such as global memory caching and better double precision floating point operation performance, but our algorithm can be compiled with CC 1.3 and higher.

```
1  typedef thrust::tuple< double, double, double, double, // freq, u, v, w
2                         double, double, double, double2 // l, m, n, B
3                       > PSVTuple;
4  struct PsvFunctor {
5    double _2pi_over_c = -0.0000000209584502;
6      double2 operator()(PSVTuple t) {
7
8      #define FREQ thrust::get<0>(t)
9      #define D_U  thrust::get<1>(t)
10     #define D_V  thrust::get<2>(t)
11     #define D_W  thrust::get<3>(t)
12     #define D_L  thrust::get<4>(t)
13     #define D_M  thrust::get<5>(t)
14     #define D_N  thrust::get<6>(t)
15     #define D_B  thrust::get<7>(t)
16
17     double argument = _2pi_over_c*FREQ*(D_U*D_L+D_V*D_M+D_W*D_N);
18     double realVal = sin(argument);
19     double imagVal = cos(argument);
20
21     double2 psv;
22     psv.x = D_B.y*realVal + D_B.x*imagVal;
23     psv.y = D_B.y*imagVal + D_B.x*realVal;
24
25     return psv;
26   }
27 }
```

Listing 4.2: Thrust PSV Functor

An outline of the Thrust PSV functor used to calculate the visibilities. Thrust's transform function usually takes in a number of separate arrays, which are coupled or 'zipped' into an array of tuples. Each element of the zipped array is run on a separate thread on the CUDA device to produce an output array. In our case the 8 input arrays are far smaller than the output array. For instance, each element in l, m, and n must be multiplied by each element in freq. It would be a huge cost on the limited memory to make an array that has multiple copies of freq, so we permute over the arrays instead. Note that this example excludes certain important lines of Thrust code for readability.

4.4.1 GPU Memory Requirements

Available on-board GPU RAM is far smaller in size than CPU system RAM, with only around $1 - 2$GB available on commodity GPUs. For this reason GPU implementations usually have to pay close attention to the amount of memory used, and to create workarounds when the problem cannot fit entirely into GPU memory.

For PSV calculations, the size of the problem dimensions (i.e. the number of frequency bands, the number of time-slots, and the number of sources), changes the required amount of GPU memory. To specify, the number of double precision floating point numbers required to be held in GPU memory (N) is:

$$
\begin{aligned}
N &= |\boldsymbol{u}| + |\Delta\boldsymbol{u}| + |\boldsymbol{\Phi}| + |\boldsymbol{\Psi}| + |\boldsymbol{\sigma}| + \quad|\boldsymbol{B}| \quad + |\boldsymbol{\nu}| + |\boldsymbol{t}| + \quad|\boldsymbol{o}| \quad + \quad |\boldsymbol{i}| \\
&= 3t + \quad 3t \quad + \quad \nu \quad + \quad t \quad + 3s + 2(4)s\nu + \nu \quad + \quad t \quad + 2(4)\nu t + 2(4)s\nu t \\
&= 8t + 2\nu + 3s + 8s\nu + 8\nu t + 8s\nu t \\
&= 16768 + 16902s \quad\quad\quad\quad\quad\quad\quad\quad\quad \text{for } t = 32 \text{ and } \nu = 64
\end{aligned}
\tag{4.2}
$$

where the relative distance between antennas, $\boldsymbol{u}$ and its change over time $\Delta\boldsymbol{u}$, the smear factors, $\boldsymbol{\Phi}$ and $\boldsymbol{\Psi}$, the direction of the source $\boldsymbol{\sigma}$, the brightness of the source $\boldsymbol{B}$, the frequency, $\boldsymbol{\nu}$, and time, $\boldsymbol{t}$ are input vectors with values as defined in Appendix B, $\boldsymbol{o}$ is the output vector, $\boldsymbol{i}$ is the intermediate-output vector, and $|\boldsymbol{x}|$ is the number of elements in vector $\boldsymbol{x}$.

Timeslots (t) and frequency bands (ν) are the least problematic dimensions. While realistic measurement sets may have thousands of time and frequency, we simplify the scope by processing a series of smaller chunks, in the order of 32-64 slots per each dimension. Since calibration solves for gains that vary over short time frames and frequency scales, we have good operational justification for this split and thus have no reason to deal with chunks any larger.

as the time dimension is limited to 32 buckets and frequency to 64 in the MeqTrees framework. These numbers are not expected to increase in the foreseeable future.

The number of sources, however, *is* expected to grow significantly. The more sources that can be simulated, the better and more general the final simulation can be. The $\boldsymbol{t}$ and $\boldsymbol{\nu}$ vectors are small in relation to the number of sources, which can exceed $5,000$. If the amount of memory required to run the simulation exceeds the amount of memory on the GPU, we split

the problem into manageable sections and sequentially process each section.

Assuming that we are using double precision floating point numbers (64-bits or 8-bytes each) and we have a GPU with $g = 1\text{GB} = 2^{30}$bytes of on-board memory, with 32 time-slots and 64 frequency bands, we can compute:

$$
\begin{aligned}
g &> 8N \\
2^{30} &> 8(16768 + 16902s) \\
s &< \frac{2^{30} - 8(16768)}{8(16902)} = 7939 \text{ sources}
\end{aligned}
\tag{4.3}
$$

Consequently, if we wish to compute $10,000$ sources, the first $7,939$ sources are calculated with one kernel invocation after which the result for the remaining $2,061$ are calculated with a second kernel invocation. These extra kernel invocations do affect overall performance and are an unavoidable consequence of the limited on-board memory.

Fortunately, as demonstrated below and detailed in the next section, calculating *multiple* sources per CUDA thread reduces the amount of required output memory. Say we calculate m sources per thread, we now reduce the amount of space needed to store m sources ($m \times t \times \nu$) to the amount of space we would use to store just one ($t \times \nu$). This reduces the size of the intermediate-output (i) memory by a factor of m, which results in a decrease of floating point numbers we need to store:

$$
\begin{aligned}
N &= |\boldsymbol{u}| + |\Delta\boldsymbol{u}| + |\boldsymbol{\Phi}| + |\boldsymbol{\Psi}| + |\boldsymbol{\sigma}| + \quad|\boldsymbol{B}|\quad + |\boldsymbol{\nu}| + |\boldsymbol{t}| + \quad|\boldsymbol{o}|\quad + \quad|\boldsymbol{i}| \\
&= 3t \; + \quad 3t \quad + \; \nu \; + \; t \; + \; 3s \; + \; 2(4)s\nu \; + \; \nu \; + \; t \; + 2(4)\nu t + \frac{2(4)s\nu t}{m} \\
&= 8t + 2\nu + 3s + 8s\nu + 8\nu t + \frac{8}{m}s\nu t \\
&= 16768 + (515 + \frac{16387}{m})s \qquad\qquad\qquad \text{for } t = 32 \text{ and } \nu = 64 \\
&= 16768 + 2563s \qquad\qquad\qquad\qquad\qquad\quad \text{for } m = 8
\end{aligned}
\tag{4.4}
$$

This modifies Eqn (4.3):

$$
\begin{aligned}
g &> 8N \\
2^{30} &> 8(16768 + 2563s) \\
s &< \frac{2^{30} - 8(16768)}{8(2563)} = 6,539 \text{ source slots}
\end{aligned}
\tag{4.5}
$$

Since in this case we are calculating $m = 8$ sources per thread, we can now calculate $6539m = 52,312$ sources without needing to split the problem into multiple invocations.

Use of multiple sources drastically decreases global memory usage and, as will be seen in the next section, improves run times.

4.4.2 Shared Memory and Multiple Sources per Thread

Shared memory is fast on-chip memory accessible by all threads in a block. It enables inter-thread communication but is limited in size ($16 - 48$kB). Threads from different blocks cannot see outside their shared memory allocation, so classic shared-memory message passing techniques do not apply. The great advantage of shared memory is that memory accesses are as fast as register accesses — 1 or 2 clock cycles per access — whereas global memory access incurs a $400 - 600$ clock cycle wait. The CUDA GPU attempts to hide this global memory access latency, as well as read-after-write latency, by swapping out those blocks of threads that are waiting for memory requests with blocks of threads that can perform computation.

Reducing latency requires that there are enough runnable threads available to the scheduler. However, creating too many threads means that the already limited register and shared memory space must be further divided among these many threads. Reducing the number of global memory accesses is generally a good way to obtain faster code execution (nVidia Corporation, 2011b).

A common strategy is to utilise shared memory as much as possible for GPU calculations, and write to global memory only after the final value has been calculated. This is the strategy we adopt for our Visibility Kernel, the kernel that performs the majority of the computation. Without any modifications, the kernel writes to global memory 4 times, since it calculates one source (a 2×2 matrix). We opt instead to calculate multiple sources (m) per thread, which requires that we write $4m$ values to global memory. This by itself does not affect execution times, since we are performing m times as many global memory accesses while using m times fewer threads. However, when using shared memory, we calculate the values for m sources but store the $4m$ values in shared memory instead of global memory. These $4m$ values are reduced (summed) to 4 values in shared memory. At the end of the kernel, only the 4 reduced values are written to global memory instead of all $4m$, thus reducing the number of global memory writes by a factor of m.

Array	Access	Dimensions				Total Dimensions
		Time	Freq.	Source	Matrix	
u	Input	✔				1 (t)
v		✔				1 (t)
w		✔				1 (t)
Δu		✔				1 (t)
Δv		✔				1 (t)
Δw		✔				1 (t)
Φ			✔			1 (ν)
Ψ		✔				1 (t)
σ			✔$(\times 3)$			1 (3ν)
B			✔	✔	✔	3 $(\nu \times s \times 4)$
ν			✔			1 (ν)
t		✔				1 (t)
i	Output	✔	✔	✔	✔	4 $(t \times \nu \times s \times 4)$
o		✔		✔	✔	3 $(t \times s \times 4)$

Table 4.1: Dimensions of Input and Output Vectors
This problem can be framed as computation of a 4D data 'cube', with different inputs given the different cube dimensions (time, frequency, source, and matrix elements). Alternatively, it can be framed as computation of a 3D data-cube of 2×2 matrices (which are represented as a 4×1 array in memory). The inconsistency with σ being an interleaved array of 3-tuples and u, v, w being $3 \times$1D arrays is due to the way in which those data are stored by MeqTrees, as explained in Section 4.5.1. u, v, w values are already in separate contiguous arrays and interleaving them into one array on the CPU is a relatively slow operation.

Furthermore, as shown in the previous section, calculating multiple sources per thread has the added benefit of using less global memory compared to calculating a single source per thread. Instead of allocating storage space for s sources (1 thread storing results for 1 source), we only need to allocate space for s/m sources (1 thread storing results for m sources).

4.4.3   Input and Output Array Indexing

As CUDA does not support multi-dimensional arrays outside of texture memory (see Section 4.4.4 below), such arrays need to be flattened into a 1D array. Many of the input and output arrays are multi-dimensional, as shown in Table 4.1.

Because of the large number of multidimensional arrays, a level of abstraction was employed for indexing flattened arrays. Consider accessing element B[5][200][2] from the array

`B[f][s][j]`. We would have to calculate the element index as `(5*s*j)+(200*j)+2`, where `s` is the number of sources, and `j`$(= 4)$ is the number of elements in a 2×2 matrix. This index calculation becomes problematic as we have to rewrite this line of code every time we need to access an array. The primary issue arises when we wish to change the order of the dimensions (say, to `B[j][f][s]`), as we would have to rewrite every line that indexes `B`. This is arduous and error-prone; thus an abstraction was found to be very useful in development.

In our implementation, the function `getMultiDimIndex(a,aTotal, b, bTotal, ...)` is used in conjunction with various permutations (for example `get_B_index(s, sT, f, fT, j, 4)`, `get_output_index(t, tT, f, fT, j, 4)`, etc). Slight speed-ups might be gained from manually inserting the flattened index code rather than using abstracted function calls. However, defining these functions as *inline* functions means that the compiler eliminates any slight function call overheads. More importantly, abstracting this operation saves a lot of time in terms of coding productivity and sanity.

4.4.4 Texture and Surface Memory

Texture memory was utilised in early iterations to reduce un-cached off-chip memory reads. Texture memory can be allocated as 1-,2-,or 3-dimensional arrays with spatial caching. This was of particular relevance to CC 1.3 devices, which unlike CC 2.x devices do not feature automatic global memory caching. Since texture memory is read-only, we could only use it for accessing input arrays.

As shown in Section 4.4.1, the size of the output arrays far exceeds that of the input arrays. We found that texture memory, even on CC 1.3 devices, has a negligible effect on total run times as most of the run time is taken up by writing results to the large output arrays. We thus focused on optimisations that reduced off-chip memory *writes* rather than reduced off-chip memory *reads*.

CC 2.x devices support *surface memory* that has similar spatial caching characteristics to texture memory, but may also be written to. This is useful for problems in which intermediate results are read back into memory with complicated and unpredictable dependencies (such as iterated N-body simulations). Our implementation only has one such instance whereby intermediate data is read by the kernel (a reduction step right at the end). The reduction kernel might benefit from surface memory's spatial caching; however, this was not explored as the reduction kernel's run time is relatively small, thus optimisations for the visibility

kernel were considered first.

4.5 CUDA Parallel decomposition

Three CUDA kernels are required for our GPU implementation of the PSV algorithm: a visibility kernel, a reduction kernel, and a re-order kernel. After the input data is copied to the GPU device, the visibility kernel calculates all of the visibilities over all of the sources. The reduction kernel then sums (reduces) the data over all the sources. Finally, the reorder kernel packs the data into contiguous memory so that it can be copied back to CPU RAM with one streaming command.

The steps required for this process are summarised below:

1. Allocate GPU global memory for inputs and outputs

2. Reorder and copy Brightness ($\boldsymbol{B}_s$) and direction ($\boldsymbol{\sigma}_s$) data from CPU to GPU

3. Copy all other data (arrays for $\boldsymbol{u}_{pq}, \Delta\boldsymbol{u}_{pq}, \boldsymbol{t}, \boldsymbol{\nu}$, $\boldsymbol{\Phi}$ and $\boldsymbol{\Psi}$) from CPU to GPU

4. Zero GPU global memory for output and intermediate output

5. Run CUDA kernel: calculate visibilities for each source (over time and frequency) and store in intermediate output

6. Run CUDA kernel: Reduce/add all visibilities over all sources

7. Run CUDA kernel: Reorder and add to final output

8. Copy data back to CPU

Figure 4.3 shows a flow chart of the CUDA PSV node execution.

4.5.1 Memory allocation

At the start of node execution, memory on the GPU needs to be explicitly allocated, after which data can be copied to that space. The amount of memory allocated depends on the number of timeslots, frequency bands, and most importantly, the number of sources that we wish to compute. Details on the exact space requirements are given in Section 4.4.1.
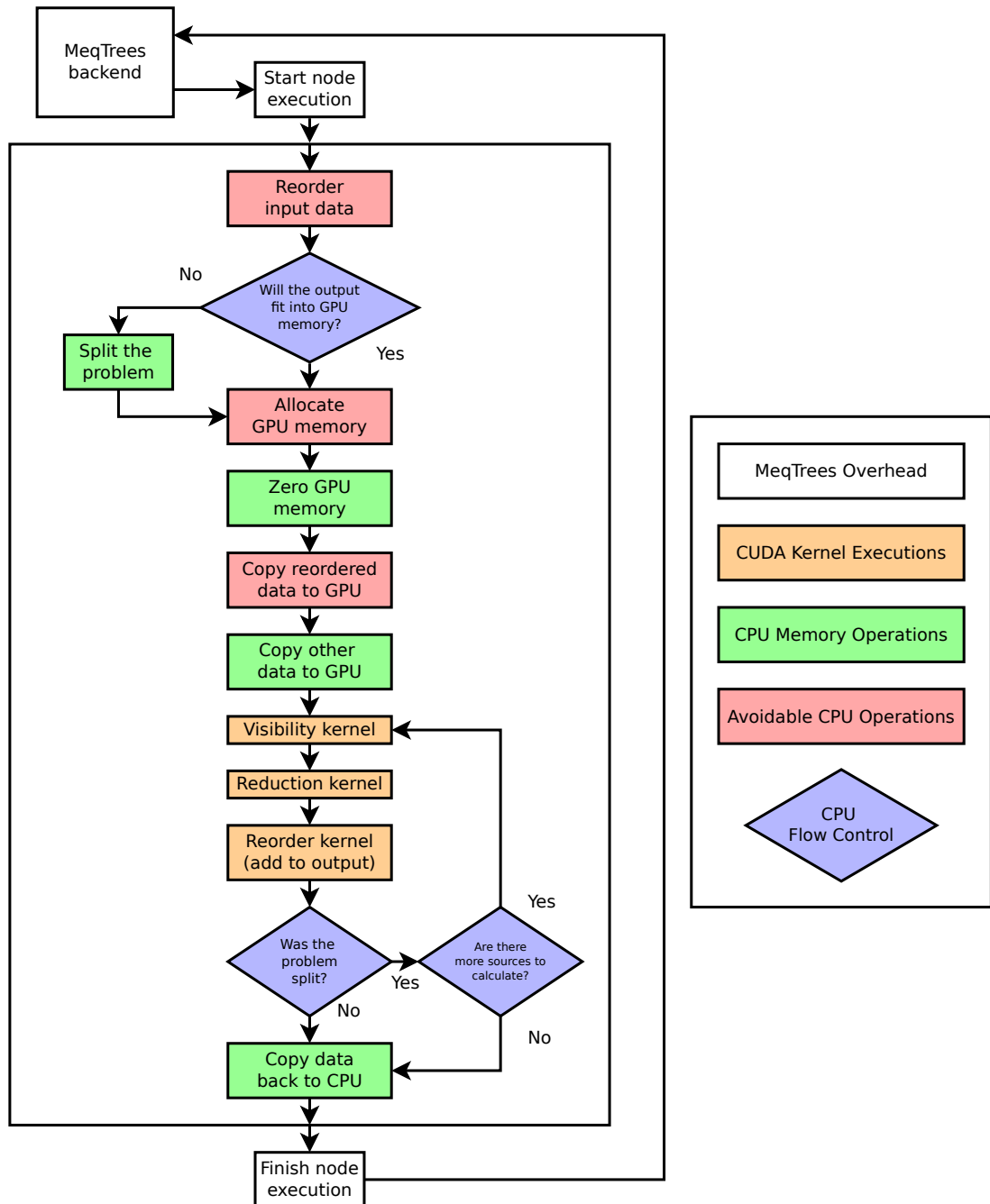
Figure 4.3: Flowchart of the execution of the PSV node
Green and red squares represent CPU operations, with red squares representing operations that are needlessly re-run every node execution, instead of once at the beginning of the simulation. Orange squares represent GPU operations.

As mentioned in Section 4.2.2 a disadvantage of porting the MeqTrees PSV calculations to GPU is the lack of persistent inter-node GPU memory. The PSV node is run once per antenna pair and once per time-slot group. The only input vectors that are dependent on the node are $\boldsymbol{u}_{pq}$, $\Delta\boldsymbol{u}_{pq}$, $\boldsymbol{t}$, $\boldsymbol{\Phi}$, and $\boldsymbol{\Psi}$. The vectors $\boldsymbol{\nu}$, B_s, and $\boldsymbol{\sigma}_s$ should not require reassignment to the GPU every node execution. However, since there is no way to allocate and then pass those vectors from node-to-node, $\boldsymbol{\nu}$, B_s, and $\boldsymbol{\sigma}_s$ must be reallocated every time. This causes significant increases in total run time.

Reordering $\boldsymbol{B}$ and $\boldsymbol{\sigma}$:   The $\boldsymbol{B}_s$ and $\boldsymbol{\sigma}_s$ input vectors require re-ordering to effectively exploit memory access coalescing on the GPU. One important consideration here is that $\boldsymbol{B}_s$ and $\boldsymbol{\sigma}_s$ are stored in MeqTrees as *Vells* or *VellSets* in such a way that they must be extracted element by element. The inner working of Vells and VellSets will not be explained here and the reader is referred to Noordam and Smirnov (2011) for more details. Furthermore, these VellSets do not necessarily organise their data so as to exploit the access characteristics of CUDA memory. However, since we have to extract the values for $\boldsymbol{B}_s$ and $\boldsymbol{\sigma}_s$ anyway, we simply place these values in our own array in the preferred order during the extraction process.

The principle reason we wish to re-order the data is to exploit CUDA's global memory *coalescing*. CUDA runs threads in groups of 32 or 16, called *warps* or *half-warps* (respectively). All threads in a warp/half-warp execute in lock-step. Naively, if the warp/half-warp performs a memory request, 32/16 simultaneous memory requests are issued, one for each thread. However, CUDA hardware is designed to coalesce these requests into a smaller number of 128-bit and 64-bit requests (the exact coalescing characteristics depends on the compute capability). An example of how reordering affects memory coalescing is shown in Figure 4.4 and discussed in Section 4.5.4. Details of coalescing can be found in Appendix F.3.2 and F.4.2 of the CUDA programmers' handbook (nVidia Corporation, 2011b).

A significant problem associated with VellSet extraction is speed: it can be very slow, especially for a large number of sources. Since each GPU node execution is independent of previous execution of any other GPU node, $\boldsymbol{B}_s$ and $\boldsymbol{\sigma}_s$ must be extracted and reordered for all s, *every time* a GPU node is called rather than just once at the beginning of the simulation. Section 6.3 shows how this increases raw computation time by up to 30%.
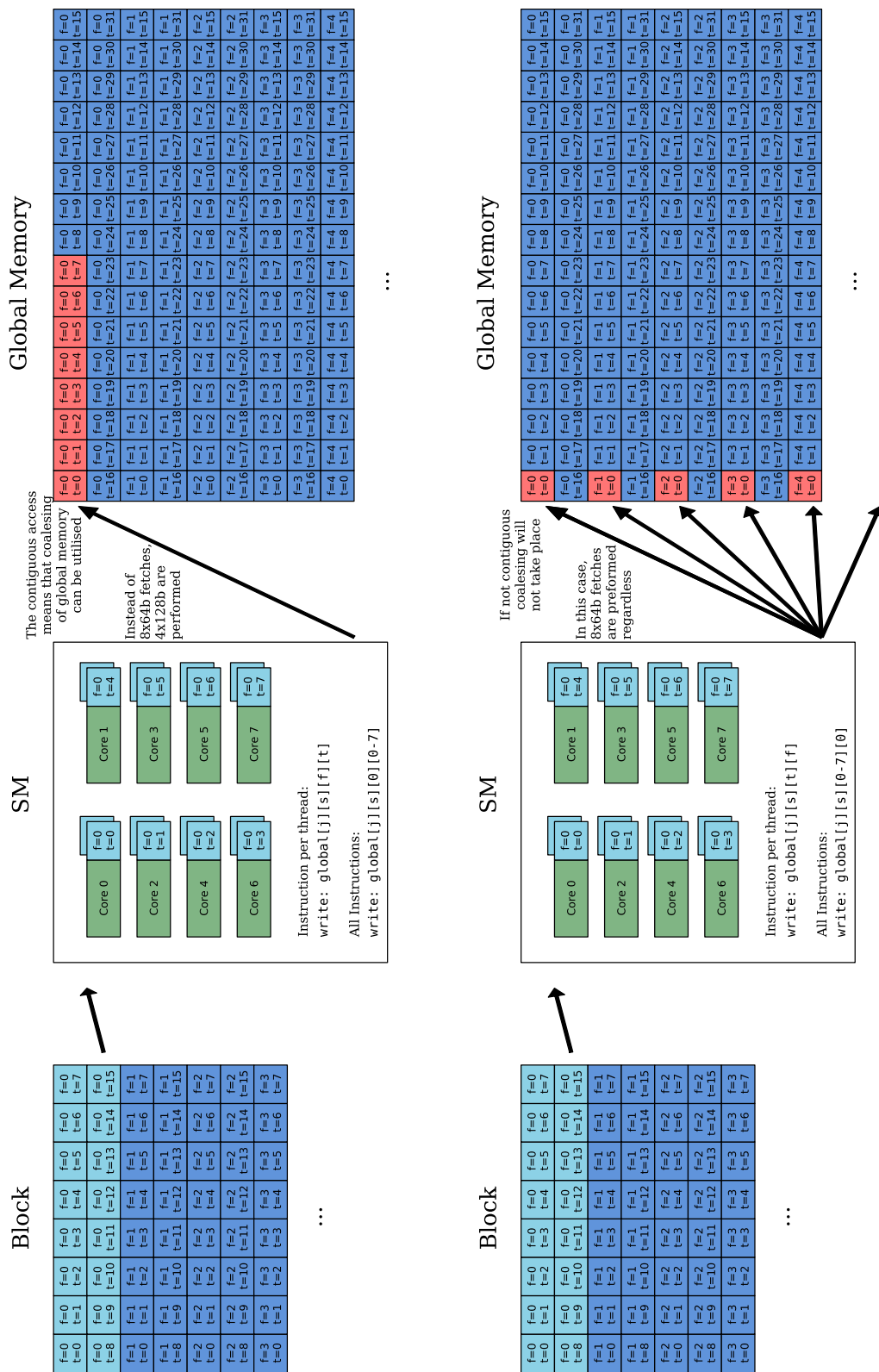
Figure 4.4: How reordering data affects memory access patterns
This figure demonstrates how different array orders can greatly affect coalescing effectiveness. The top figure shows the standard memory layout of global[j][s][f][t]. In conjunction with the block layout this leads to each thread accessing a contiguous memory location which allows coalescing to occur. The bottom figure shows the same block layout but with the global memory layout of global[j][s][f][t] (t and f switched), this fragments memory access and limits coalescing.

Copying remainder of data: Unlike $\boldsymbol{B}_s$ and $\boldsymbol{\sigma}_s$, which only need to be copied onto the GPU once per simulation, input vectors for $\boldsymbol{u}_{tqp}, \Delta\boldsymbol{u}_{tpq}, \boldsymbol{\Phi}$, and $\boldsymbol{\Psi}$ (as well as the vectors holding values for $\boldsymbol{\nu}$ and $\boldsymbol{t}$) must be recopied every node execution. Fortunately, these vectors are small in comparison to $\boldsymbol{B}_s$ and $\boldsymbol{\sigma}_s$, so while this copying operation is unavoidable, it is not very costly.

Intermediate-output and output memory: Results must be stored in GPU memory then copied back to the CPU once execution is complete. As such, we have to allocate enough memory for the final output $(\nu \times t \times 4)$ vector. If we performed these calculations serially we would write the result for source s_0 to this result vector, then calculate source s_1 and add that to the final result vector and so on for the remaining sources. Since we are performing the calculations of all of the sources in parallel, we cannot have a single vector of dimension $(\nu \times t \times 4)$ to which we add the values, as this would lead to multiple race-conditions and invalid results. Instead we need to allocate s vectors of dimension $(\nu \times t \times 4)$ vectors, so that each source can written to its own vector. The result is reduced using a parallel logarithmic reduction algorithm — see Section 4.5.5.

Instead of computing s sources in parallel, one by one, sources are bucketed into groups of m sources and s/m groups of sources are computed in parallel, with each group calculating m sources serially. This provides significant performance gains — see Section 4.4.2 for details.

4.5.2 Visibility Kernel

The visibility kernel is the kernel that performs the calculation of the visibilities described in Eqn (4.1). Since this kernel has the largest running time of all our kernels, the most effective optimisations are implemented here.

Figure 4.1 shows the decomposition of the problem along its three dimensions (time, frequency, and sources). Each individual thread is responsible for one PSV calculation at one timeslot, one frequency, and for one source. These threads are grouped into a number of thread-blocks, with each thread-block responsible for a certain cross section of the data cube.

In our initial design, each CUDA thread calculated the result for one source at one frequency and one timeslot. However, we found that calculating multiple (m) sources for one frequency and one timeslot per thread utilised the processors more efficiently. Once calculated, the

threads stores the visibility (or m visibilities) in global memory.

As mentioned in the previous chapter, the Streaming Multiprocessors (SMs) of a GPU can hold a limited number of blocks resident at any one time. To maximise computational throughput, we must ensure that the SMs of the GPU are always fully occupied. In order to ensure this, we have to have a sufficient number of blocks in total so as to fill the SMs. We have control over how we split our problem up, meaning that we can partition the problem into as many or few blocks as we wish. We need only work out what number of blocks is optimal.

We start by calculating the minimum number of blocks required for full utilisation. This number is dependent the number of block per SM (b), which is determined by the occupancy. It is also dependent on the number of SMs per GPU device (r). Hence, the minimum number of thread-blocks required is br.

We must also calculate the number of blocks the kernel will allocate. This is determined by (i) the number of sources, time-slots and frequency bands (s, t, ν), which represents the number of points in our data cube; (ii) the size of the thread-block (z), since each thread in a block computes one point of the data cube (we allocate $st\nu$ threads, and thus allocate $\frac{st\nu}{z}$ thread-blocks) and; (iii) the number of sources calculated per thread, we actually compute m sources per thread, meaning that we require $\frac{st\nu}{m}$ threads, hence we only allocate $\frac{st\nu}{zm}$ thread-blocks.

Comparing the two values, we require that $br \leq \frac{st\nu}{zm}$, or $s \geq \frac{brzm}{t\nu}$. z is the size of the thread-block and represents a subsection of the t and ν dimensions, and as such cannot be larger than $t\nu$, in other words $z \leq t\nu$, or $1 \geq \frac{z}{t\nu}$. We now find that $s \geq \frac{brzm}{t\nu} \geq brm$.

We find that the number of sources (s) must be larger than the product of the number of blocks per SM (b), the number of SMs per device (r), and the number of sources calculated per thread (m). We consider that the maximum number of blocks per SM is $b = 8$, and since no current or planned CUDA GPU exceeds $r = 48$ SMs per device, we can conservatively say $s \geq (8)(48)m$, or $s \geq 384m$.

So we finally find that the minimum number of sources required to fully occupy the GPU is dependent on the value of m. If we calculate $m = 32$ sources per thread, we require at least $12,288$ sources to fill the SMs, whereas $m = 2$ would only need a minimum of 768. We see that smaller problems could be hindered by a high m value, which would reduce the number of blocks we allocate, whereas larger problems would not benefit from lower m values.

In practice we find that b can range all the way from 1 to 8, depending on occupancy. A low b value means even less total thread-blocks are required for full utilisation of the SMs. The effect of different m values is explored in the Section 6.2 in the Results Chapter.

4.5.3 Visibility Kernel Execution Steps

The visibility kernel (Listing 4.3) performs the following tasks (at the following lines):

Ln 9: Calculate thread index

Ln 17: Define and clear shared memory

Ln 20: For each source:

    Ln 22: Calculate the smear factor

    Ln 32: Calculate the exponential

    Ln 37: Calculate the matrix and accumulate to shared memory

Ln 42: After all sources are processed, write the accumulation to global memory

The simplified kernel omits some uninteresting lines of code, mainly macro definitions that switch between use of shared and global memory, and switch between use of multiple sources or a single source. If the kernel does not use shared memory, the results are accumulated straight to global memory instead of shared memory which obviously forgoes the write back from shared to global memory at the end of the kernel. If the kernel is calculating only one source, `s_start` and `s_end` (line 12) are not calculated and the loop at line 20 is omitted.

Calculating a Thread's Location: Each thread resides within a 2D block, stored as $1 \times f_b \times t_b$ (`blockDim` in code). These blocks reside within a 3D grid, $s_g \times f_g \times t_g$. Each thread is responsible for calculating the visibilities of a number of sources, over one time-slot and over one frequency-slot. Therefore we need to determine exactly which time-slot (`t_i`, line 10), frequency-slot (`f_i`, line 11), and source range (`s_start`, `s_end`, line 12) we are evaluating. In Listing 4.3, line 9 to 12 we can calculate these values using the thread's location within the block (`threadIdx`), the dimensions of the block (`blockDim`), the block's location within the grid (`blockIdx`), and how many sources per thread should be computed (`srcs_per_thread`).

```
1   CUDAPointSourceVisibilityKernel(      // Inputs
2       /* σ_s(R³),  B_s(C⁴),  t(R),  ν(R)      */ double3* lmn, double2* B, double* time, double* freq,
3       /* (u_tpq,  v_tpq,  w_tpq),  (Δu_tpq,  Δv_tpq,  Δw_tpq) */ double*  uvw, double* duvw,
4       /* ΔΨ(ν_m)/2,  ΔΦ(ν_m)/2                 */ double*  df_over_2, double* f_dt_over_2,
5       /* array sizes                        */ int ntime, int nfreq, int nsrcs, int srcs_per_thread,
6       /* global memory output              */ double2* intermediate_output_complex ) {
7
8       // Calculate thread index and shared memory index
9       int s_i = ((blockIdx.x*blockDim.x) + threadIdx.x); // input index, source
10      int t_i = ((blockIdx.y*blockDim.y) + threadIdx.y); // input index, time
11      int f_i = ((blockIdx.z*blockDim.z) + threadIdx.z); // input index, freq
12      int s_start = s_i*srcs_per_thread; int s_end = s_start + srcs_per_thread;
13
14      int t_si = threadIdx.y; // shared (memory) output index, time
15      int f_si = threadIdx.z; // shared (memory) output index, freq
16
17      __shared__ double2 shared_mem [];   // declare shared memory ([blockDim.y][blockDim.z][4])
18      ... // clear shared memory (need a 2×2 double2 matrix for each thread in the thread-block)
19
20      for (int s = s_start ; s < s_end && s < nsrcs ; ++s) {
21          // Calculate smearing factor with (−2π/c)(u_tpq · σ_s) and (−2π/c)(Δu_tpq · σ_s)
22          double smearFactor = 1.0;
23          double argument  = _2pi_over_c * ( u[t]*lmn[s].x + v[t]*lmn[s].y + w[t]*lmn[s].z);
24          double dargument = _2pi_over_c * (du[t]*lmn[s].x +dv[t]*lmn[s].y +dw[t]*lmn[s].z);
25
26          double dphi = d_f_dt_over_2[t] * dargument;
27          if (dphi != 0.0) smearFactor = sin(dphi)/dphi;   // sinc(ΔΦ(ν_m)/2)
28          double dpsi = d_df_over_2[f] * argument;
29          if (dpsi != 0.0) smearFactor *= sin(dpsi)/dpsi; // sinc(ΔΨ(ν_m)/2)
30
31          // Calculate the PSV exponential term, exp(−2π i (ν/c)(u_tpq σ_s))
32          double realVal; double imagVal;
33          sincos(d_freq[f]*argument, &realVal, &imagVal);
34          for (int j = 0 ; j < 4 ; ++j) {
35              int b_i          = get_B_index(s, f_i, j);
36              int share_index = get_shared_mem_index(t_si, f_si, j);
37              shared_mem[share_index].x += (d_B[b_i].y*realVal + d_B[b_i].x*imagVal) * smearFactor;
38              shared_mem[share_index].y += (d_B[b_i].y*imagVal − d_B[b_i].x*realVal) * smearFactor;
39          }
40      }
41      // Copy data in shared memory to global memory
42      for (int j = 0 ; j < 4 ; ++j) {
43          int global_index = get_intermediate_output_index(s_i, t, f, j);
44          int share_index  = get_shared_mem_index(t_si, f_si, j);
45          d_intermediate_output_complex[global_index] = shared_mem[share_index];
46      }
47  }
```

Listing 4.3: Simplified visibility kernel code

Visibility kernel code formatted for readability. Some operations are left out for brevity.
This code is what each thread will execute and calculates multiple sources per thread
using shared memory.

Defining and Clearing Shared Memory: Each block is allowed a certain amount of shared memory for all of its threads to use. For our implementation we need 4 'double2's per thread per thread-block for the 2×2 matrix (a `double2` is a 128byte data-type consisting of 2×64byte `doubles`, used to represent a complex number). The shared memory array will thus be `4*threadIdx.y*threadIdx.z` (`threadIdx.x` is always 1). All these values must be initialised to zero otherwise the reduced values will be undefined.

Calculating Smear Factor: As described in Section B.4, the unmodified visibility function assumes that the signal is measured at a constant time t at a certain frequency f. This is not actually the case as the signal is measured over a frequency band, affecting the phase coherence, and measured over some period of time, affecting the relative location of the antennae shifts. As a result the accuracy of the visibility equation is reduced. To correct for this a *smearing factor*, M_{tfs}, is multiplied against the visibility.

$$M_{tfs} = \operatorname{sinc}\frac{\Delta\mathbf{\Psi}}{2}\operatorname{sinc}\frac{\Delta\mathbf{\Phi}}{2} \tag{4.6}$$

where

$$t_m = (t_0 + t_1)/2, \nu_m = (\nu_0 + \nu_1)/2,$$

$$\Delta\mathbf{\Psi} = \arg V_{pq}(t_1, \nu_m) - \arg V_{pq}(t_0, \nu_m),$$

$$\Delta\mathbf{\Phi} = \arg V_{pq}(t_m, \nu_1) - \arg V_{pq}(t_m, \nu_0)$$

Here $\arg(\cdot)$ denotes the complex argument or complex angle and $\operatorname{sinc}(\cdot)$ is defined as

$$\operatorname{sinc}(x) \;=\; \begin{cases} \frac{\sin x}{x} & \text{if } x \neq 0 \\ 1 & \text{if } x = 0 \end{cases}$$

(Smirnov, 2011a; Thompson et al., 2001; Taylor et al., 1999)

The argument of the visibility equation is:

$$\arg V_{pq}(t, \nu) = \arg\left[\exp\left(\frac{-2\pi i}{c}\nu(\boldsymbol{u}_t \cdot \boldsymbol{\sigma}_s)\right)\right] = \frac{-2\pi i}{c}\nu(\boldsymbol{u}_t \cdot \boldsymbol{\sigma}_s)$$

From this we calculate:

$$
\begin{aligned}
\Delta\Psi &= \arg\, V_{pq}(t_1, \nu_m) - \arg\, V_{pq}(t_0, \nu_m) \\
&= \left(\frac{-2\pi i}{c} \nu_m(\boldsymbol{u}_{t_1} \cdot \boldsymbol{\sigma}_s) \right) - \left(\frac{-2\pi i}{c} \nu_m(\boldsymbol{u}_{t_0} \cdot \boldsymbol{\sigma}_s) \right) \\
&= \frac{-2\pi i}{c} \nu_m(\Delta t \, |\boldsymbol{u}_{\Delta t}| \cdot \boldsymbol{\sigma}_s) \\
&= \nu_m \, \Delta t \cdot \frac{-2\pi i}{c} (\Delta\boldsymbol{u}_t \cdot \boldsymbol{\sigma}_s) \\
\Delta\Phi &= \arg\, V_{pq}(t_m, \nu_1) - \arg\, V_{pq}(t_m, \nu_0) \\
&= \frac{-2\pi i}{c} \nu_1(\boldsymbol{u}_{t_m} \cdot \boldsymbol{\sigma}_s) - \frac{-2\pi i}{c} \nu_0(\boldsymbol{u}_{t_m} \cdot \boldsymbol{\sigma}_s) \\
&= \Delta\nu \cdot \frac{-2\pi i}{c} (\boldsymbol{u}_{t_m} \cdot \boldsymbol{\sigma}_s)
\end{aligned}
\tag{4.7}
$$

Before kernel execution, we pre-calculate arrays holding values for $\frac{\nu_m \Delta t}{2}$ (`f_dt_over_2`) and $\frac{\Delta\nu}{2}$ (`df_over_2`) and use those to calculate the smearing term, $\mathrm{sinc}(\frac{\Delta\Psi}{2})\mathrm{sinc}(\frac{\Delta\Phi}{2})$. This is shown at lines 22 - 29 in Listing 4.3.

Calculating Phase-Shift: This is the visibility calculation and is expressed by:

$$
P_{tfs} = \exp\left(\frac{-2\pi i}{c} f(\boldsymbol{u}_t \cdot \boldsymbol{\sigma}_s) \right)
$$

In Listing 4.3 (line 23) the argument to the exponential is calculated:

```
_2pi_over_c * freq[f](u[t]*lmn[s].x + v[t]*lmn[s].y + w[t]*lmn[s].z)
```

where `u[]`, `v[]`, `w[]` are the arrays holding the relative antennae location at different times, `lmn[]` is the array holding the directions ($\boldsymbol{\sigma}$) of the different sources, `freq[]` is an array of frequency slot values, and `_2pi_over_c` is -2π over the speed of light (c)

To calculate $P_{tfs} = e^{ik}$ (for $k = \frac{-2\pi f(\boldsymbol{u}_t \cdot \boldsymbol{\sigma}_s)}{c}$), we use Euler's formula $P = \cos(k) + i\sin(k)$

In the kernel one call to the `sincos()` function is used instead of two separate `sin()` and `cos()` function calls (line 33)

Writing to shared/global memory: If shared memory is used, the visibility matrix for each source, `s`, is calculated and added to `shared_mem[j][f][t]`. At the end of the kernel,

this sum is added to `global_memory[j][s][f][t]` (lines 37, 38). If shared memory is not used, the results for each source `s`, are added straight to `global_memory[j][s][f][t]`.

Since the global memory array is organised in the order `global_memory[j][s][f][t]`, the time dimension is most tightly packed. This tight packing in conjunction with the block layout allows for memory coalescing to take place. In our code, we make use of multiple `get_*_index` functions (lines 37, 38, 43, 44) in order to easily experiment with different array orderings.

4.5.4   Visibility Kernel Tuning

As mentioned before, different thread-block layouts will result in different occupancies and different memory access characteristics. Occupancy is entirely determined by the *size* of the thread-block layout, and not by the dimensions (see Section 4.6.2). Memory access patterns, on the other hand, are very sensitive to changes in thread-block layout dimensions. Thread-block dimensions are in time and then frequency ($t \times f$). The number of timeslots are chunked in MeqTrees to 32 and the number of frequency channels are chunked to 64.

This means our maximum thread-block layout is 32×64, which is 2048 threads. As CUDA does not support more than 1024 thread per block, we must reduce one or both dimensions. We find another limitation using shared memory, where using any more than 256 thread per block means that more than the maximum 16kB of shared memory per SM is required (see Section 4.6.2). If we utilise CUDA's ability to expand this shared memory to 48kB (at the cost of L1 cache) this limit is only extended to 512 threads per block.

Furthermore, more threads per block does not necessarily lead to better performance. Thus, we test a variety of thread-block layouts of various sizes to determine what characteristics (namely size of the time dimension, frequency dimension, and overall size of the thread-block) are responsible for faster or slower execution times.

It would be ideal to use an auto-tuning system that would determine an optimal thread-block layout, probably employing some heuristic parameter search. We leave any form of auto-tuning for future work and instead manually test a wide range of different thread-block layouts. We do this because our primary goal is to determine which thread-block characteristics cause which effects on execution time of this prototype. It is too much to cover every single thread-block layout thus we only test in powers of two $(1, 2, 4, 8, 16, 32, 64)$

and we do not cover all permutations of this. As will be seen in Section 4.6.2, certain configurations allow for larger or smaller thread-blocks, which alters the range of thread-blocks we test.

4.5.5 Reduction Kernel

Reduction allows one to combine (or reduce) many values into a single value. One of the simplest forms of reduction is summation, whereby all values are added together to give one value. For our work in particular, we have an intermediate output vector of dimension $(s/m) \times \nu \times t$. In other words, we have (s/m) vectors of dimension $\nu \times t$ which must be summed together. We call this 'reducing the vector over (s/m)'.

In CUDA all thread-blocks calculating visibilities are executed in parallel and in an undefined order. Consequently, to reduce values using a single global total would generate multiple race-conditions and produce inconsistent results. Thus, we perform a two step, calculate-and-then-reduce method rather than simply adding the visibilities to an array as we calculate the values. The reduction step is implemented on the GPU rather than on the CPU since the data already resides on the GPU, and takes far less time to execute on the GPU than on the CPU.

The reduction kernel uses summation to combine the visibilities of all the sources into one value. The intermediate results are stored in an $(s/m) \times f \times t \times 4$ vector which must be reduced to a $t \times f \times 4$ vector. Reduction on the CPU means that all $(s/m) \times f \times t \times 4$ values need to be copied to CPU memory and then reduced serially, adding the values together one-by-one. It is much faster to reduce on the GPU where the array already resides. We use a logarithmic parallel reduction algorithm (Nickolls et al., 2008), which repeatedly adds half of the array to the other half, until only one element remains. The steps are as follows:

1. For N element array

2. For each $(2n)^{\text{th}}$ element, add the $(2n+1)^{\text{th}}$ element to it

3. We now have an $\lceil N/2 \rceil$ element array

if $\lceil N/2 \rceil = 1$

    4. There is one element in the array — it is the reduction/sum

else $\lceil N/2 \rceil > 1$

4. Repeat step 1 with the $\lceil N/2 \rceil$ elements

After the first iteration, $\lceil N/2 \rceil$ elements remain. Repeat this to get $\lceil N/4 \rceil$ elements, then again to get $\lceil N/8 \rceil$ elements and so on until only one element remains. Similar to the visibility kernel, each thread is responsible for one frequency, one time-slot, and two sources (the $(2n)^{\text{th}}$ and $(2n + 1)^{\text{th}}$ source). The reduction kernel is run once for each reduction step, each successive kernel run reducing the number of elements by half. This completes the reduction in $\lceil \log N \rceil$ steps. See Figure 4.5 for a worked example.

4.5.6 Reorder Kernel

This kernel ensures that the reduced output is contiguous in memory and hence can be copied back with just one copy command. If reordering is not done, the reduced output will be fragmented in memory, necessitating multiple copy operations.

Fragmentation occurs as a result of the memory order used to exploit CUDA's global memory coalescing. The intermediate/unreduced output is structured as a $4 \times s \times f \times t$ flattened array (see Section 4.4.3). When reduced, the array dimensions becomes $4 \times 1 \times f \times t$, and the array is not necessarily contiguous in memory (see Figure 4.5). The reorder kernel compacts and reorders the non-contiguous $4 \times 1 \times f \times t$ array into a contiguous $t \times f \times 4$ array. Copying back to the CPU can now be performed in a single streaming command.

4.6 CUDA Implementation Issues

The following section is split into two primary considerations: memory management and processor management. Our problem is characterised by a small amount of input data with a relatively large amount of output data. Throughout the chapter the various memory write optimisations and considerations have been discussed.

We start this section with a recap of the memory management issues that were detailed throughout the chapter. Following this we discuss the other major component: how to keep the many CUDA processors occupied with instructions. We do this by discussing our implementation's *occupancy* (defined in Section 3.3.2) and what factors affect it. Although not as large a consideration considering the large memory requirements of this problem,
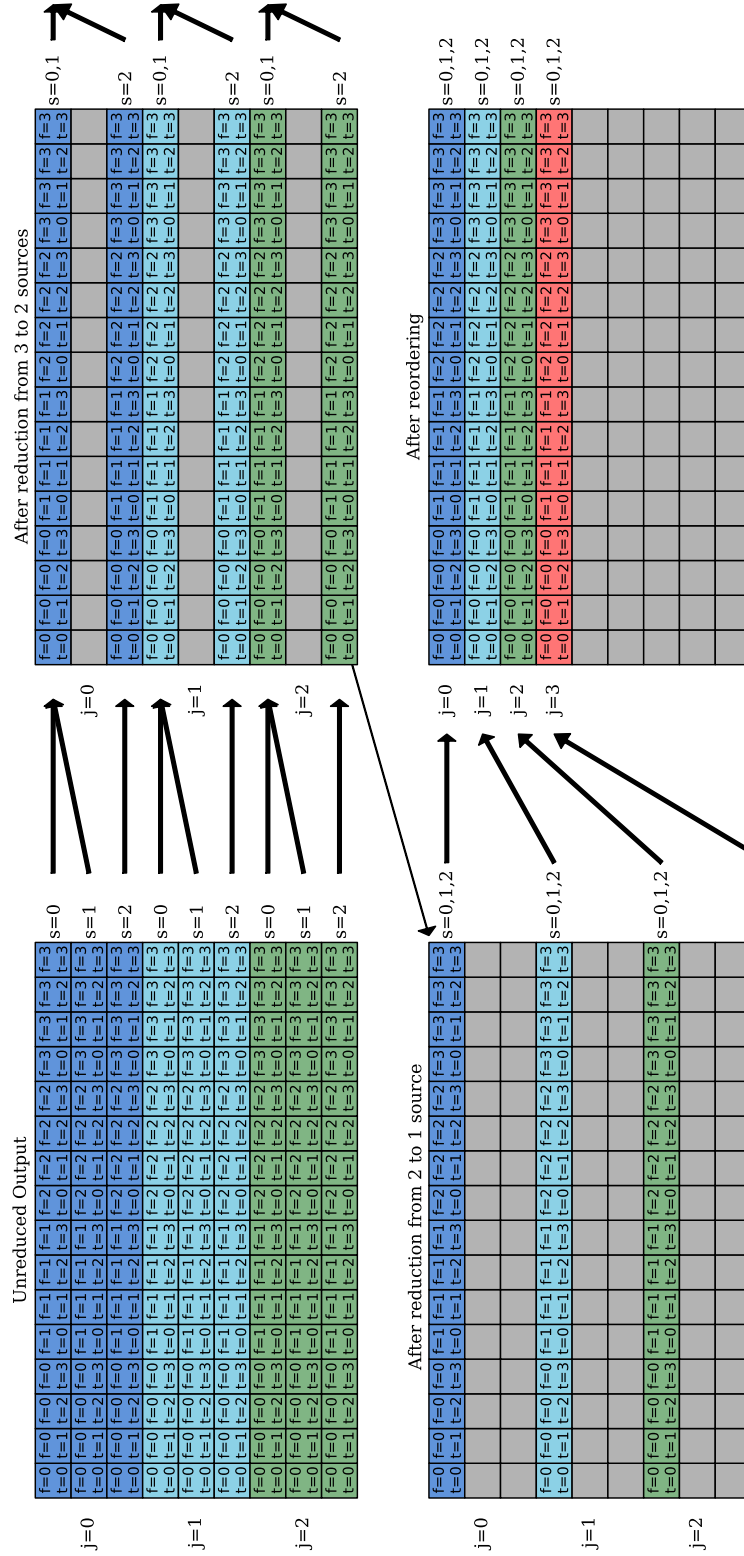
Figure 4.5: Example of reduction and reordering of unreduced output to a reduced form

The data is initially unreduced, stored in an array of dimensions $intermediate\_output[j][s][f][t]$, with $t = 4$ and $f = 4$ in this simplified example. The $s = 3$ sources go through two reduction steps to reduce the array to a fragmented $[j][1][f][t]$ arrangement. The reordering step ensures that this reduced output is optimally fitted into the final output array, $output[j][f][t]$.

we analyse our implementation with respect to occupancy and how this might improve computational throughput.

4.6.1 Memory Management

For most CUDA implementations, memory management can be as large a factor as processing speed in determining computational throughput. Porting visibility calculations to the GPU has its own unique memory considerations.

Part of MeqTrees's flexibility arises from the manner in which it defines its calculations. Nearly any mathematical expression or measurement equation can be built using a tree of nodes (see Figure 4.2a for an example). As mentioned previously, our implementation replaces the many-node expression of the visibility equation with a single node, thus reducing overheads significantly. The CUDA version is essentially a port of the single-node CPU version. Unless otherwise stated, when we refer to the *node* we mean this single-node GPU implementation.

As discussed in Section 4.5.1 we note that the manner in which MeqTrees passes data from node to node, understandably, does not apply to the GPU memory management model. This would require either modification of the inner working of MeqTrees or an undesirable workaround and is left for future work.

4.6.2 Occupancy

Our kernel implementation, particularly the visibility kernel, is register-heavy and utilises a large amount of shared memory. This has a significant effect on the occupancy of the SMs as it limits the number of blocks that can be active/resident on an SM at any given time.

The major factors that determine occupancy are (i) the number of registers the kernel needs, (ii) the amount of shared memory per thread-block requested, and (iii) the number of threads per thread-block (refer to Section 3.3.2 for details on how occupancy is determined). The number of registers can only be changed by changing the code (they are fixed otherwise). In our implementation, we require 64 bytes of shared memory per thread. In many problems, shared memory is shared amongst the threads and can be independent of the number of threads per block; however, as we are simply using shared memory to store output data, the

amount needed per block is directly proportional to the number of threads in the block. The number of threads per block is variable and affects occupancy by changing the amount of shared memory required and the amount of registers required. The different configurations are given in Table 4.2.

CC 1.3 devices are limited to 512 threads per block. In our implementation there are not enough registers or shared memory to run the kernel with 512 threads. With a 2.x device, while there is enough register space, the amount of shared memory still limits occupancy. 2.x devices have 48kB of L1-cache for global memory and, like 1.3 devices, 16kB of shared memory. This can be configured such that 48kB is reserved for shared memory with only 16kB of L1 cache. In our implementation, the extra shared memory eases the significant shared memory pressure, allowing thread-blocks of 512 threads. If we attempt to exploit all the shared memory (768 threads), we run out of registers, as shown in Table 4.2.

Higher occupancy usually means better performance. This was, however, not observed in our implementations, where higher occupancy had a negligible effect on performance. See Section 6.5 for more details.

Shared Memory Limits: Each thread-block calculates visibilities over a range of time-slots, T, and a range of frequency bins, F, with each thread calculating the visibility for one time-slot and one frequency bin. Each visibility is a 2×2 complex-valued matrix and as such requires 8 double precision floating-point numbers (64 bytes). Thus, each block required $T \times F \times 64$ bytes of shared memory to store its array of complex doubles. More threads per block means that the required shared memory per SM increases. Too many threads and the SM will not be able to allocate enough of its limited shared memory. In the current implementation, the amount of shared memory cannot be reduced. Thus shared memory limits on occupancy cannot be eased without a different approach.

Register Pressure: The large number of registers (30-45) can be partly attributed to the use of double precision floats (which takes up twice as much register space as floats) and to the size of the visibility kernel code itself. Attempts were made to reduce the number of registers per thread by avoiding unnecessary declaration of variables in the kernel. Reduction by only 4-6 registers is possible. For the most part the CUDA compiler optimises away any minor changes in CUDA code, making manual register reduction mostly redundant. Unfortunately, upon closer investigation, the number of registers per thread would have to be reduced from

CC	Reg's	Threads /Block	Sh.m /Block	Sh.m /SM	Occ.	Blocks/SM limited by:			Warps /Block	Warps/SM	
						Warps/SM	Reg's/SM	Sh.m/SM		Active	Max
2.x	45 (2.1)	32	2048 b	48kB	33%	**8**	22	24	1	8	48
		64	4096 b		33%	**8**	11	12	2	16	
		128	8192 b		42%	8	**5**	6	4	20	
		256	16384 b		33%	6	**2**	3	8	16	
		512	32768 b		33%	3	**1**	**1**	16	16	
		768	49152 b		**0%**	2	**0**	1	32	**0**	
	46 (2.0)	32	2048 b	16kB	17%	**8**	22	**8**	1	8	
		64	4096 b		17%	8	11	**4**	2	8	
		128	8192 b		17%	8	5	**2**	4	8	
		256	16384 b		17%	6	2	**1**	8	8	
		512	32768 b		**0%**	3	1	**0**	16	**0**	
	31 (2.1)	32	0 b	16kB (un- used)	17%	**8**	32	N/A	1	8	
		64			33%	**8**	16		2	16	
		128			67%	**8**	**8**		4	32	
		256			67%	6	**4**		8	32	
	32 (2.0)	512			67%	3	**2**		16	32	
		768			50%	2	**1**		24	24	
		1024			67%	**1**	**1**		32	32	
1.3	48	32	2048 b	16kB	25%	8	**5**	8	1	1	32
		64	4096 b		25%	8	5	**4**	2	2	
		128	8192 b		25%	8	**2**	**2**	4	4	
		256	16384 b		25%	4	**1**	**1**	8	8	
		512	32768 b		**0%**	2	**0**	**0**	16	16	
	32	32	0b	16kB (un- used)	25%	**8**	**8**	N/A	1	8	
		64			50%	**8**	**8**		2	16	
		128			50%	8	**4**		4	16	
		256			50%	4	**2**		8	16	
		512			50%	2	**1**		8	16	

Table 4.2: Effect of Different Configurations on Occupancy
2.x devices have a choice between a 16/48kB or a 48/16kB split for shared memory/L1 cache. When using the larger amount of shared memory per SM (48kB), the occupancy bottleneck shifts from shared memory to register pressure (the minimum blocks per SM for each configuration is shown in **red**). *With fewer threads per block, less registers per block are required, meaning that occupancy is then limited to the maximum allowable blocks/warps per SM, a hardware limitation. For each CC and shared memory configuration, we see that each has an upper limit to the number of threads per block. This is mostly caused by shared memory limits, but also by register pressure. We can forgo use of shared memory to increase occupancy, but we lose the significant performance gains.*

45 to 32 to have any effect on occupancy, making minor register reduction largely redundant.

Splitting the Kernels

A solution to the above mentioned register pressure problem is to split the kernel into multiple sequential runs, with each run performing a part of the execution. Since each kernel is responsible for fewer calculations, less registers are required per kernel. For our problem, it is possible to calculate each smearing factor in its own kernel and then multiply those by the visibilities in another kernel and vice-versa.

In general, there are a number of *Jones* matrices against which each visibility must be multiplied, as explained in Section B.2.1. A prototype kernel was implemented that performed all matrix multiplication operations within a single kernel. This required too many registers, causing register spilling and very poor performance. A potential solution is to split the problem up such that each kernel does the multiplication step of one Jones matrix. Splitting the kernels allows for a variable number of Jones matrices in the visibility equation, but, as we show below, nullifies our current shared memory optimisation.

The main problem encountered when exploring the use of multiple kernels is that we can no longer exploit shared memory, as we are no longer able to evaluate multiple sources per thread.

Currently each thread calculates the summed visibilities with smearing factor for m sources, i.e. $S_1V_1 + S_2V_2 + ... + S_mV_m$, where S_i the ith smearing term and V_i is the ith visibility. If we split this up we have to first calculate $S_1, S_2, ..., S_m$ with a smearing factor kernel, and then $V_1, V_2, ..., V_m$ with a visibility kernel, multiplying them to get $S_1V_1, S_2V_2, ..., S_mV_m$. These must then be summed to get our result: $S_1V_1 + S_2V_2 + ... + S_mV_m$.

However, this approach means that we use more memory per source as we need storage space for each S_iV_i value instead of storage space for each summation, $\sum_{i=1}^{m} S_iV_i$, only. See Section 4.4.1 for exact memory usage characteristic for shared memory compared to no shared memory.

Furthermore, we lose the memory throughput advantage of using shared memory. Normally we only have to write 4 values to global memory, as we reduce the m visibilities in shared memory. With multiple kernels, we must reduce the values only *after* all the visibilities have been calculated (i.e. after all the kernels have executed), meaning we cannot reduce the m

visibilities in shared memory.

The other major issue is that any decrease in execution time achieved by reducing the register pressure must be offset by the overhead cost of the extra kernel invocations.

These considerations are left for future work as they are more applicable to the larger project of a CUDA implementation of PSV simulation using *multiple* Jones terms, which is beyond the scope of this thesis.

4.7   Summary

There are many design considerations that are required to implement an efficient CUDA kernel, without which much of the power of CUDA hardware would remain unexploited. Chief among these, for our problem at least, are CUDA hardware optimisations that reduce global memory writes, in particular coalescing and the use of shared memory for intermediate calculations. Processor-based optimisations focused around the occupancy metric are not of primary concern, as will be seen in Chapter 6 (Results).

Chapter 5

Evaluation Methods

Having detailed the implementation of the CUDA PSV node, we now evaluate the efficacy of our design choices and optimisations. This chapter outlines the evaluation process and the experimental setup for testing. Our principal analysis metric is the *speed-up* of the CUDA PVS node over the CPU PSV node: essentially how many times faster the GPU version runs compared to a CPU version. Our main goal is to determine how effective our various optimisation and design choices are at accelerating the PSV node and which factors hinder acceleration.

In the previous chapter, we noted that MeqTrees splits the computational problem up into sequential sections, each section representing one run of our PSV node. It is *only* these computational sections that we parallelise. Between these computational sections, MeqTrees performs a number of *overhead* operations required for the subsequent computational section of code, e.g. traversing the equation trees, copying input and allocating output memory, among other things (see Figure 5.1). These 'MeqTrees overheads' are part of the core mechanics of the MeqTrees framework and were not considered for acceleration in this project. As such, we refer to these sections of code as the 'non-parallelisable' section of code (although outside of this work they could theoretically be parallelised) as they are serial as far as we are concerned. In contrast, we refer to the computational sections of code as 'parallelisable'.

Our CUDA PSV implementation is validated and benchmarked against the CPU PSV node already in the MeqTrees framework (Smirnov, 2012). Both nodes operate in the same manner within MeqTrees and require the same input, excepting that our PSV node lacks certain advanced functionality. When comparing run times of the GPU and CPU nodes, identical

input parameters are used for both, and both are run under the same conditions and on the same computing equipment.

The two primary factors which affect the overall run time of a PSV simulation are the number of antennae used and the number of point sources simulated. Secondary factors include the number of frequency bands and the number of time-steps; however these are constant for all our tests. We used two sets of antennae for our tests: the Westerbork Synthesis Radio Telescope (WSRT) (14 antennae and thus 91 base-pairs), and the planned MeerKAT array (64 antennae and thus 2016 base-pairs). Note that each interferometer setup measures over different times: the WSRT measures 72 time-steps, while the MeerKAT array has a longer exposure of 480 time-steps. An artificial sky model comprising a uniform square grid of point sources is used in all our simulation tests. A 'gridded' sky model is not realistic, but if we consider the RIME equation, Eqn (4.1), we see that the locations of the sources ($\boldsymbol{\sigma}_s$) are independent of the computational complexity. Hence, only the number of sources, and *not* their locations, affect the running time of the simulation.

We divide our run time into three logical sections, one computational and two overhead sections, with the intent of analysing the effect of each. We do this by defining speed-ups that selectively and deliberately include and exclude certain overheads. Besides the effect of overheads, we wish to determine how effective our optimisations and design choices are. We test over many configurations of our implementation, namely various thread-block layouts, different shared memory usages, problem sizes, and interferometers. We have detailed our experimental setup and how we have run all our tests. The following chapter gives the results of these tests and highlights expected (and unexpected) results. We show which cases lead to speed-ups (or lack thereof), which of our optimisations prove to be effective and which are shown to be ineffective. We discuss these findings with reference to the CUDA hardware and the MeqTrees software to determine the root cause of each of our design considerations.

5.1   Code Classification

Figure 5.1 shows the execution time-line of a simulation run, and demonstrates how Meqtrees switches back and forth between our computational node and its internal 'MeqTrees over-head' operations. Whereas Figure 4.3, in the previous chapter, details the internal workings of our PSV node. We note three different divisions of the PSV node: the kernel executions (orange in Figures 4.3 and 5.1), the memory operations (green), and the avoidable memory

operations (red). Avoidable memory operations are those memory operations that could have been amortised if MeqTrees contained GPU-based memory management. Currently they are needlessly redone in each PSV node (typically thousands of times), whereas it should be done once for the whole simulation.

We divide the total execution time of a simulation run into three distinct sections: the PSV node run time (excluding the avoidable memory operations), the avoidable memory operations, and the MeqTrees overhead. These three sections of code are timed separately and constitute the total run time completely; there are no other contribution to the total running time.

The 'MeqTrees overhead' or back-end execution is responsible for managing memory and calling the relevant nodes (our CUDA PSV node in this case), represented in Figure 5.1 by the purple 'M' blocks. A second class of overhead arises from the avoidable reordering and copying operations that are performed for each PSV node execution. For our implementation, there are a number of arrays allocated on the GPU and data that must be reordered on the CPU and copied to GPU memory. A substantial fraction of this data is needlessly reallocated, reordered, and recopied, as these operations are performed every time the node is called, but only is required to be performed once at the beginning of the simulation. This cost could be avoided by modifying the underlying MeqTrees framework (red in Figure 5.1)

Although we cannot amortise the unnecessary reorder and recopy operations in this work, we are able to measure their contribution to the total running time of the GPU simulation. When measuring core computational times, we subtract these contributions from the GPU times in order to estimate the time it would have taken if these operations were not performed. We note that we *only* do this when comparing the computational sections of code. Unless otherwise stated, we leave the total running times unaffected.

It is important to emphasise that the speed-up of the GPU PSV node is in comparison to a *single* core of a CPU node, although MeqTrees does contain multi-core capabilities, a meaningful comparison between this and a GPU version would be difficult.

5.1.1   Timing Metrics

As we have split our execution times into three distinct sections, we define three timing metrics:
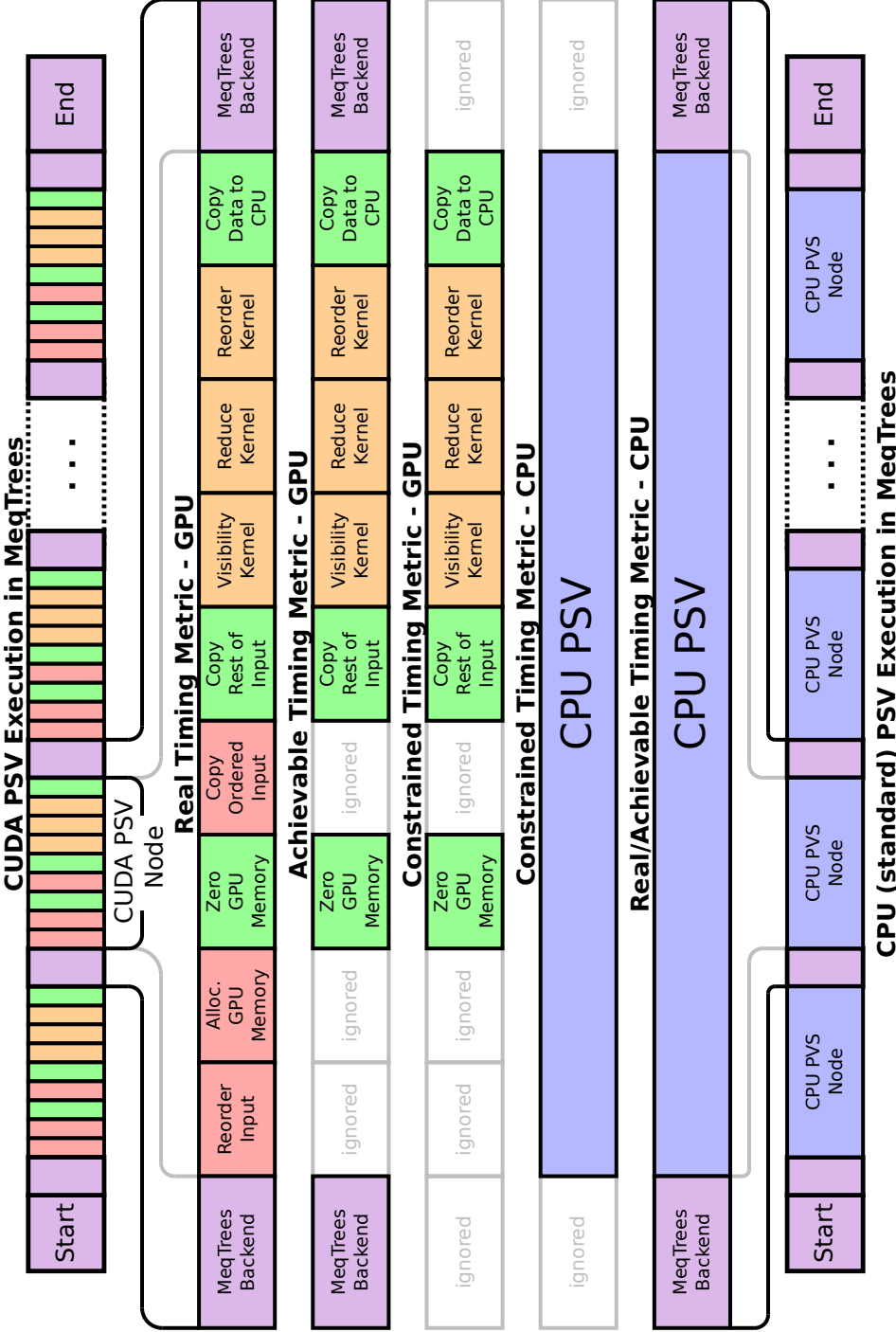
Figure 5.1: Breakdown of the different timing metrics used in a MeqTrees simulation run.
This shows how MeqTrees executes its nodes. It flips between computational component and the MeqTrees overhead (purple) many times until processing is complete. We divide the GPU version into two distinct sections: (i) the necessary memory operations (green) and the kernel executions (orange), and (ii) the avoidable memory operations (red). The CPU version is the standard version that comes with MeqTrees (blue). For the GPU version we show the three timing metrics used, the Real timing metric consists of the entire run time of the simulation. The Achievable timing metric is the Real time without the avoidable memory operations. The Constrained time is the time of the computational sections of code with necessary memory operations.

1. *Real* Time/Speed-up:
   Total GPU time versus Total CPU time.

2. *Achievable* Time/Speed-up:
   Total GPU time, excluding unnecessary memory operations, versus Total CPU time.

3. *Constrained* Time/Speed-up:
   Total GPU time, excluding unnecessary memory operations, and excluding MeqTrees overhead, versus Total CPU time excluding MeqTrees overhead.

The first metric, *Real* speed-up, is the actual speed-up achieved and represents the most basic comparison between CPU and GPU run times. The *Achievable* speed-up is the speed-up if we were to ignore the avoidable memory allocation and copy operations and shows what could be achieved given additional GPU memory management in MeqTrees. This represents an estimation of speed-ups that could be achieved in future work. The third, *Constrained* speed-up, directly compares GPU to CPU computation times and represents the speed-up gain of only the computational aspects of the execution. Whilst *Constrained* speed-up is not practically achievable, it gives a better metric for how much the GPU accelerates the core computation.

5.2 Testing Parameters

Problem size and number of antennae have a clear effect on run times. Larger datasets and more antennae lead to slower run times, but greater speed-up on the GPU. We focus on effects independent of problem size, particularly our optimisation and design choices: (i) The thread-block layout used in conjunction with our array packing order, which affects exploitation of global memory coalescing; (ii) use of shared memory to perform intermediate operations, whereas they would normally be done on global memory; (iii) use of newer hardware, namely a CC 2.0 device versus a CC 1.3 device, the former of which affords us a choice of L1 cache to global memory or three times the shared memory of CC 1.3 devices as well as increased double precision floating point operation throughput compared to CC 1.3 devices; and (iv) the performance when using a smaller interferometer to see if our acceleration is as effective with small as well as large datasets.

CUDA groups its many lightweight threads into 1D, 2D, or 3D thread-blocks with user-defined dimensions. We employ a 2D thread-block layout comprising a time and frequency

dimension. This, in conjunction with optimal input and output array packing, can have an effect on memory throughput, as coalescing effects require contiguous data. Different layouts also have different processor occupancy characteristics. We test over a multitude of thread-block layouts to determine its effect on coalescing and occupancy.

Comparison of shared memory versus global memory essentially tests the use of our shared memory optimisation (Section 4.4.2) versus the standard version that does not employ any shared memory optimisation. As the number of sources computed per thread, m is key to the shared memory optimisation, we not only test our implementation with and without use of shared memory, but also test different m values. Early tests showed that use of texture memory had no clear effect. Consequently, we did not use this type of memory in our final version. As discussed in Section 4.4.4, surface memory, which is similar to texture memory, was not applicable to our implementation.

Tests are performed on nVidia GeForce GTX 285 and GTX 470 devices. The GTX 285 has a compute capability of 1.3 whilst, the newer GTX 470 is a 2.0 device. Whilst the GTX 470 is constituted from newer technology, both have a similar peak estimated theoretical GFLOP performance of around 1.1 GFLOPs. However, the GTX 470 has a lower memory bandwidth at 133.9 GB/s compared to 159.0 GB/s of the GTX 285. Despite the GTX 285 being top of its range[1] and the GTX 470 the second in its range, the GTX 470s newer computation technology ensured that it outperformed the GTX 285 significantly (nVidia Corporation, 2012a,b).

We test with CC 2.x devices as it allows for switching between L1 cache to shared memory configurations. There is 64kB of on-chip memory per streaming multiprocessor, part of which is reserved for shared memory and part for L1 cache to global memory. The standard distribution is 16kB for shared memory and 48kB for L1 cache, but the user can opt to switch this configuration to 48kB of shared memory and 16kB of L1 cache, which would afford more shared memory, but at the risk of less cache and hence less cache hits. This will allow us to test efficacy of global memory L1 cache. We can also test if tripling the amount of shared memory has any effect as well, as more shared memory allows for higher occupancy which could increase processor utilisation. Since our implementation makes exclusive use of double precision floating point operations, we test the increased double precision performance of the CC 2.x devices over CC 1.3 devices.

As well as the MeerKAT array (2016 baseline pairs and 480 time steps), we test the smaller

---

[1]Excluding the GTX 295 which contains two GPU chips on one card.

WSRT interferometer (91 baseline pairs and 72 time steps). We configure MeqTrees to split the PSV simulation into groups of 32 times steps per baseline pair, thus the MeerKAT array has 30240 'pair groups' (2016 pairs over 15 time slot groups) and WSRT has 273 'pair groups' (91 pairs over 3 time slot groups). When a PSV node is called, GPU or CPU, it will calculate the visibilities of only one of these groups. The PSV nodes are called sequentially (see Figure 5.1) and it is these individual PSV nodes that are accelerated, not the simulation as a whole. We test if simulations that have fewer 'pair groups', and thus fewer PSV node calls, run at the same rate (groups per second) as simulations with many 'pair groups'.

5.2.1 Experimental Setup

Our experiments were run on a computer with an i5 2.66GHz CPU (2 hyper-threaded cores, 4 logical cores) with 3GB of DDR RAM. This machine also has an nVidia GTX 470 CUDA GPU with 1GB of on-board RAM. The operating system used is a 32-bit Ubuntu 11.10 standard desktop installation. We initially used an nVidia GTX 285 CUDA GPU (1GB RAM), which has a compute capability (CC) of 1.3, but this was swapped out for the newer 470 device with a CC of 2.0. We report how these different devices performed along with the main body of our results.

We ran all our tests from the command line with the *meqtrees-pipeliner* available in the MeqTrees software suite (Smirnov, 2012), with the simulation configuration specified via command line parameters. Configuration determines, among other things, which interferometer setup is being used (in our case, WSRT or MeerKAT), the list of test cases to run, whether CPU or GPU is being used, and — if the GPU is used — what thread-block layouts to test with.

The two simulated interferometers differ in three primary areas: firstly the number of antennae, and hence the number of pairs to process (91 base-pairs for WSRT versus 2016 base-pairs for MeerKAT); secondly, the location of these antennae; and thirdly, the amount of time-steps each computes.

Other configurable options are the number of sources calculated per thread, the shared memory to global L1 cache layout (16/48kB or 48/16kB), whether to use the shared memory optimisation, and different array packing orders.

For instance, we run a configuration that will execute a number of simulations, each using a

thread-block layout from a list ($\{32 \times 1, 32 \times 2, 32 \times 4, ...\}$) and data-set size from another list ($\{1, 9, 25, 49, 81, ...\}$). Other options are configured before hand, such as use of shared memory how many sources per thread and L1 cache size versus shared memory size. The script will take the first problem size (1 source) and simulate it with each of the given thread-blocks, it will then simulate the second size (9 sources) with all the thread-block, and so on until all problem sizes have been processed. For each run we recorded the total time, the computational time, and the overhead times.

Additionally, owing to the long time it takes to run a single configuration over all necessary thread-blocks and problem sizes (usually around 2 days of constant computation per configuration) we only perform one test run per configuration instead of averaging multiple runs. We tested the variance in run times between multiple test runs of the same configuration. We found negligible differences, even for long test run configurations, but we could not do this for all configurations in reasonable time.

Chapter 6

Results and Discussion

In this chapter, we present the results of the evaluation experiments. We explore the effects of different thread-block layouts, memory packing, shared memory, the MeqTrees overheads, compute capability (CC) 2.x CUDA hardware, L1 global caching, occupancy, and testing on smaller datasets such as the WSRT array. We also discuss less significant results, such as our Thrust prototype, use of CUDA texture and surface memory, and the effect of unnecessary reorder and recopy overheads in the PSV.

The Thrust PSV node prototype was found to be much slower than the custom-developed CUDA kernel showing 4× speed-up for Thrust versus 16× speed-up for the custom kernel. We found that the high dimensionality of the problem made it a poor fit for Thrust's abstracted problem-solving functions and did not explore this approach further.

For our custom CUDA kernel, the best configurations made use of shared memory. More optimal thread-block layouts were characterised by a larger time dimension as this allowed it to take better advantage of coalescing owing to our specific array packing order. Use of 48kB of shared memory increased the occupancy but did not show significantly faster run times than the standard configuration with 16kB.

We find that use of newer CUDA hardware (CC 2.x) had the expected outcome of running faster than comparable hardware of the previous generation (CC 1.3). This is due to a multitude of new features that reduce the number of off-chip memory accesses and utilise the hardware more efficiently for double precision floating point operations.

As mentioned in the previous chapter, we test with the 64 element MeerKAT array (2016

baseline pairs) and the 14 element WSRT interferometer (91 baseline pairs). The size of the dataset has a large impact on performance. We note that the composition of the dataset does not have an effect on run times, only the size of the dataset. A larger dataset is better exploited by the GPU hardware's parallelism. As such, we find that speed-ups increase as problem sizes increase, but converge past a certain dataset size. Similarly the interferometer setup used has an effect; we find smaller interferometers are not as well exploited by GPU hardware.

When comparing configurations, it is important to note that we find consistent differences between all three metrics (described in Section 5.2). If one configuration is faster or slower than another, it is the same along all three metrics. This is since only the core computations are accelerated, whilst all other sections of code run in the same time. Thus when core computations fun faster or slower, the whole simulation runs faster or slower, respectively and in all cases.

6.1 Impact of Array Ordering and Thread Layout

Thread-blocks are organised in a 2D layout, the size of which affects the final running time, specifically affecting the ability of the kernel to take advantage of coalescing effects. Figure 6.1a shows the running times of the CPU version compared to the GPU version. The large disparity makes for difficult comparison, hence our decision to analyse the speed-ups rather than the times themselves. We are better able to see the differences in Figure 6.1b, which shows just the GPU run times over all the thread-block layouts. In Figure 6.1c we show the real speed-ups achieved over a number of thread-block layouts. In Figure 6.1d shows the constrained speed-up, and represents an exaggerated version of the real speed-ups.

We can see that thread layout plays a significant role in achieving faster run times: time × frequency layouts with more than 32 threads that have a larger time dimension perform better overall. We see that layouts with a time dimension of greater than or equal to 8 perform the best. This arises from the way in which the data is packed and then accessed: all arrays are ordered such that 'time' is the final dimension and the arrays are thus contiguous in memory when accessed over 'time'. Threads run in lock-step with 16 thread *half-warps*, and each of the 16 threads reads a global memory location simultaneously. Since our array order is sequential, the half-warp requests contiguous memory locations and these multiple requests thus coalesced into fewer request. This is covered in detail in Sections 4.5.1, 4.5.4
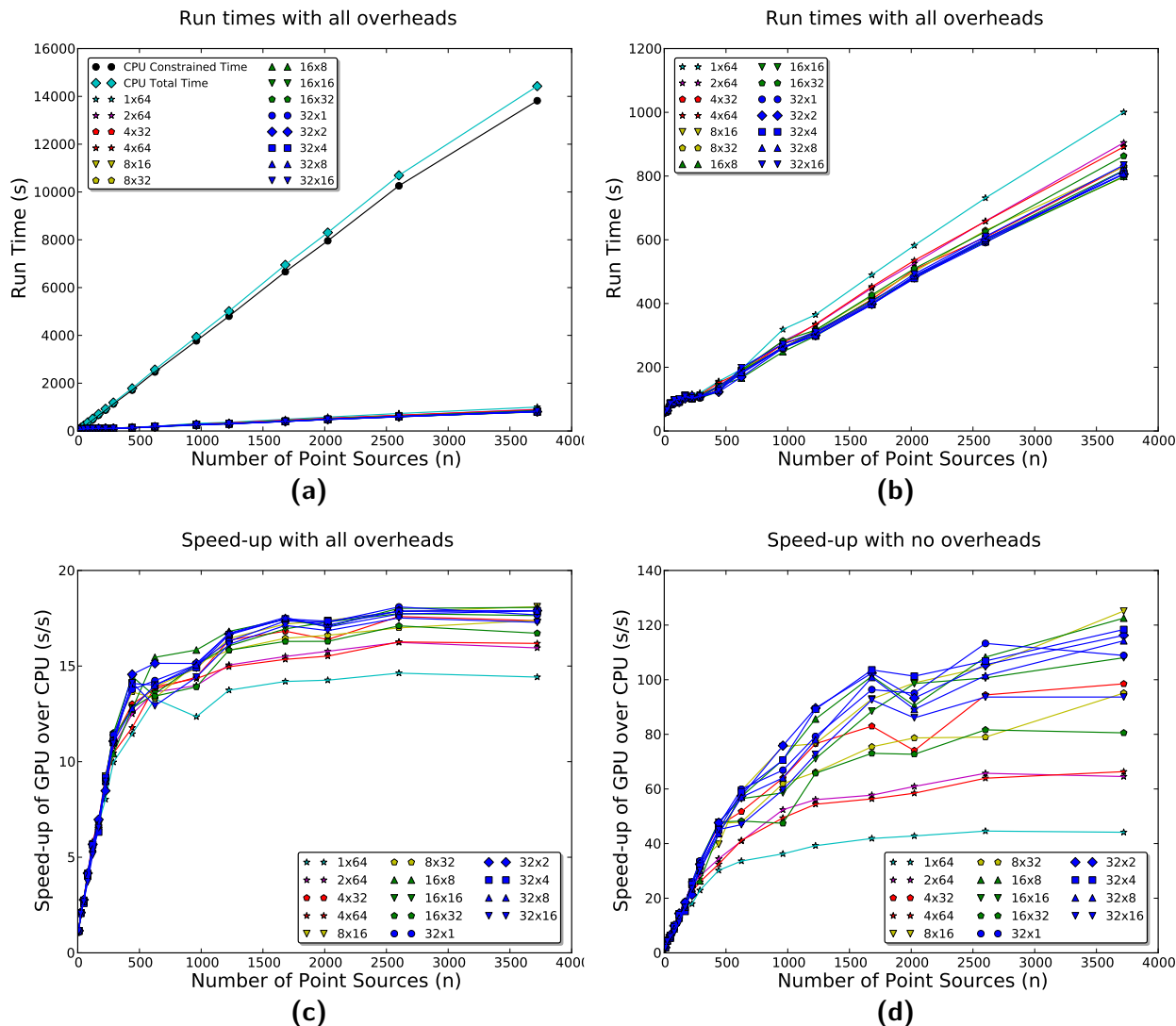
Figure 6.1: Real and Constrained Run Times and Speed-ups for all thread-block layouts (MeerKAT array, nVidia GTX 470)

*Thread layouts are displayed as $(t \times \nu)$ and colour coded by time dimension. (a) The running time of the simulation of all thread block layouts. (b) The same with running times of the CPU simulation times included. (c) The* Real *speed-ups achieved. (d) The* Constrained *or computational speed-ups achieved. Thread-block layouts with a time dimensions of* 8 *or greater benefit the most from memory coalescing.*

and Figure 4.4. As shown in Figure 6.1d, when the time dimension falls below 8 threads, computational speed-up starts to dip, as threads are no longer requesting as many contiguous memory locations simultaneously and coalescing is only partially exploited. Although the top half of the test runs differ by 30%, in terms of real time speed-ups the difference is much more subtle (Figure 6.1c).

6.2 Shared, Surface, and Texture Memory

The use of shared memory results in a further 30% speed-up (Figure 6.2a). For computation time only (*Constrained* speed-up), the improvement is close to 80% for large numbers of point sources (Figure 6.2b). The use of shared memory in conjunction with calculation of multiple sources per thread reduces the number of global memory writes significantly, leading to significant performance gains. Calculation of multiple sources per thread has the advantage of decreasing global memory writes, but also reduces the number of threads available to the scheduler, which in turn restricts its ability to use context switches to hide latency. For large problems, we find that increasing the number of sources per thread increases performance, but that any more than 16 sources per thread shows no appreciable benefit.

As discussed in Section 4.4.2, the number of sources calculated per thread (m) affects the number of global memory writes; however, m also affects the total number of blocks that are allocated to the grid, with a larger m value benefiting simulation of larger data sets. In Figure 6.2, we see that speed-ups improve with larger m values, but shows no improvement past $m = 16$. Note that computing one source per thread ($m = 1$), finish in similar times to runs that used no shared memory at all. This is due to the fact that the shared memory optimisation relies on reducing the number of global memory accesses by performing intermediate calculations on the shared memory. As expected, with $m = 1$, we have the same number of global memory accesses as the non-shared memory case, resulting in no performance benefit and even a slight decrease in run time because of increased complexity of the shared memory optimisation code.
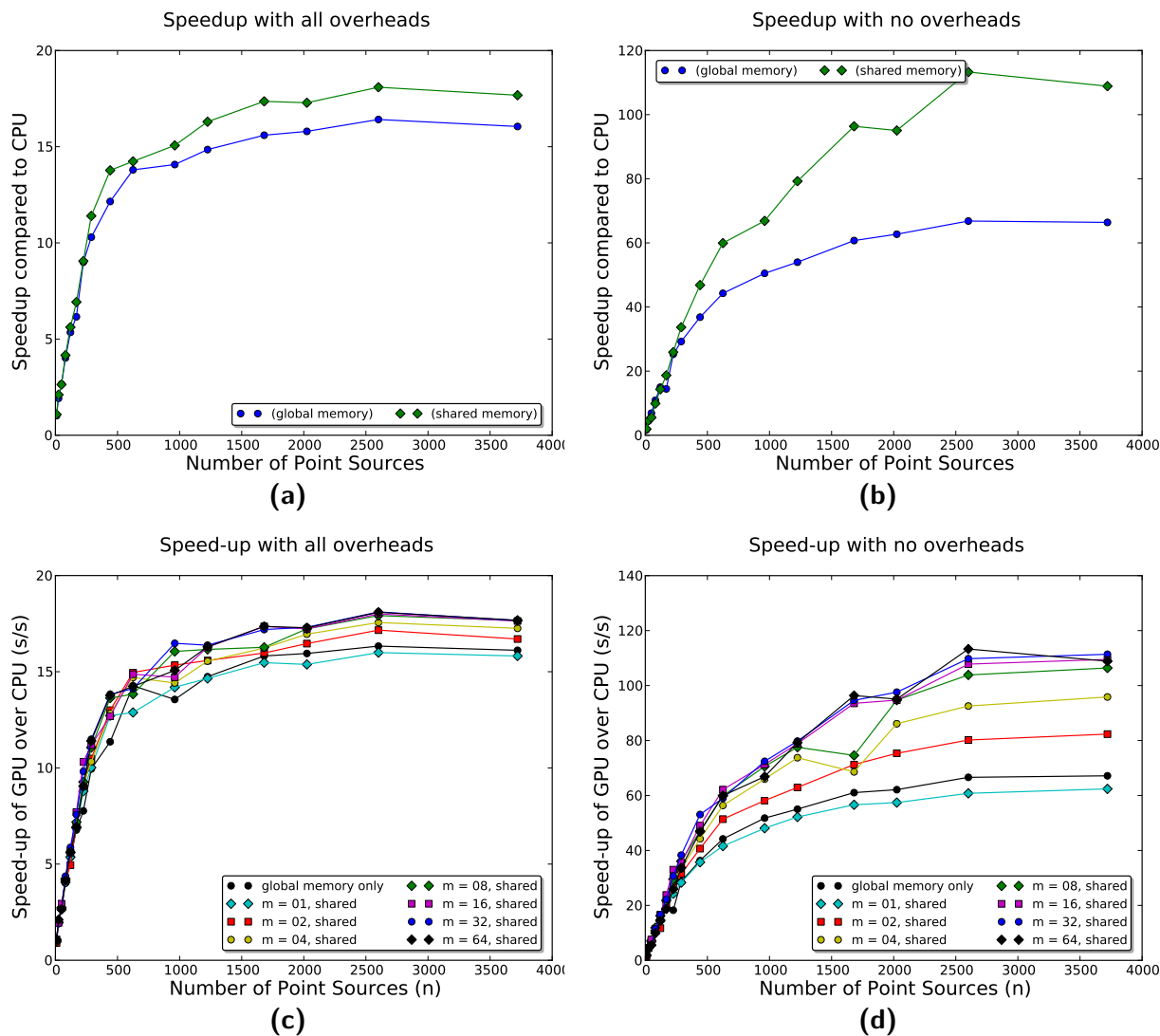
Figure 6.2: Comparison of speed-up relative to CPU for shared memory versus global memory (MeerKAT array, nVidia GTX 470 & Intel i5 2.66GHz)

Thread-block layout 32×1. (a) Real speed-ups of shared versus global memory, (b) Constrained speed-ups of shared versus global memory. Although the difference between shared memory and global memory usage is relatively small for Real times, there is actually an 80% improvement on the computational times. (c) Real speed-ups of differing m values. (d) Constrained speed-ups of differing m values. Note how the $m = 1$ speed-ups are slightly lower than that of global memory only.
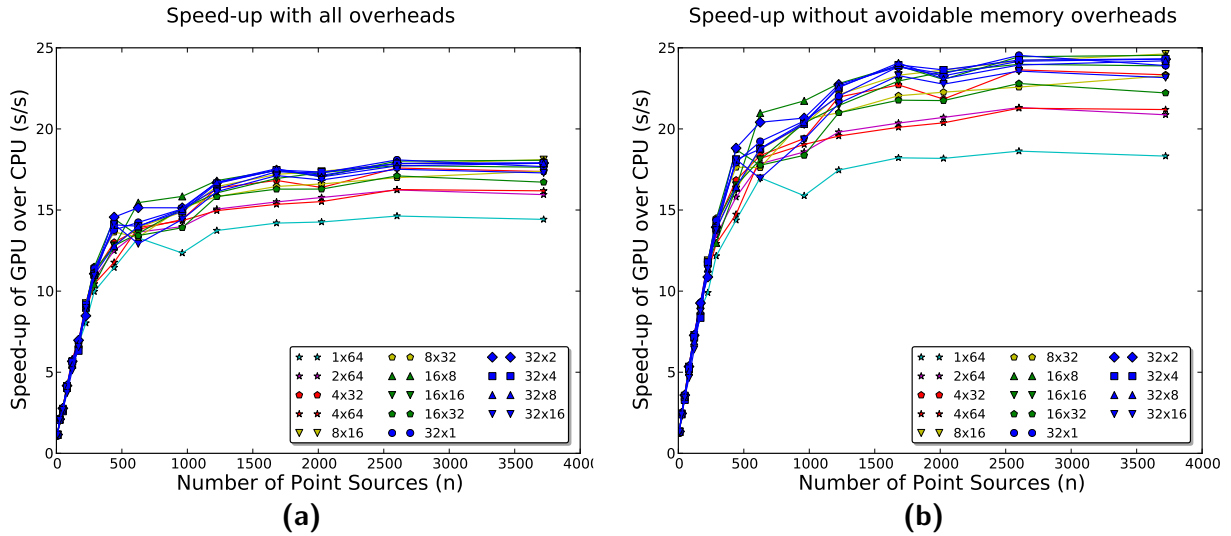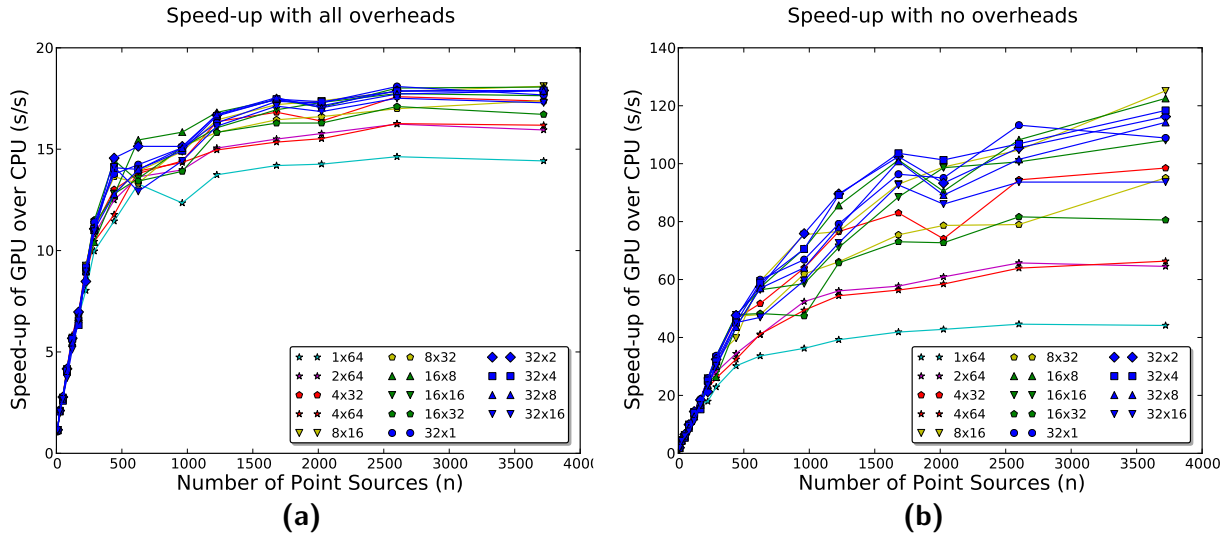
Figure 6.3: Real and Achievable speed-ups for all thread block layouts (MeerKAT array, nVidia GTX 470 & Intel i5 2.66GHz)
(a) Real *speed-up of all the thread-block layouts.* *(b)* Achievable *speed-up of all the thread-block layouts.* Achievable *speed-up calculated by subtracting the contributions of copy and reordering operations that should be amortised into the whole simulation. These operations are needlessly rerun for every node execution.*

6.3 Impact of MeqTrees Overheads

Figure 6.3 shows the speed-up results for all tested thread-block layouts with and without the unnecessary copy and reorder operations that the GPU version performed, as explained in Section 4.5.1. Whereas we get $18\times$ speed-up in the normal case (Figure 6.3a), if we remove the time taken to perform the recopy and reorder operations, we could achieve up to $24\times$ speed-up (Figure 6.3b). Although we cannot be sure that this would be the outcome were we to implement a PSV node that amortised these unnecessary operations, a significant portion of the *total* run time, $20\% - 25\%$, is taken up by these operations and they could be amortised. A possible avenue of exploration besides upgrading the MeqTrees internal working, would be to implement an 'overlord' node that could manage the GPU memory pointers and keep them persistent across all PSV node executions. This would not be a general solution, as an overlord node would probably have to be created for each specific MeqTrees problem. This avenue was not explored in this work.

Considering only the running time of the parallelisable sections of code, without MeqTrees overhead, a *Constrained* or 'core' speed-up of more than $110\times$ is possible (Figure 6.4). This

Figure 6.4: Real and Constrained speed-ups for all thread-block layouts (MeerKAT array, nVidia GTX 470 & Intel i5 2.66GHz)
This graph shows the speed-up results for all tested thread block layouts with and without all overheads. The Constrained *or core speed-up metric is a measure of how much faster the parallalisable sections of code performed, whereas the real speed-up is a measure of speed-ups an end user would experience.*

large difference in speed-up points to a bottleneck in the MeqTrees framework. Under the CPU version, MeqTrees overhead accounts for an acceptable 4% of the total running time (Figure 6.5a - black line). However, with the faster running time of the GPU, MeqTrees overhead constitutes up to half the running time in the CUDA PSV node (see Figure 6.5a).

An interesting corollary of this discussion is the maximum theoretical speed-up based on the percentage of run time that is taken up by MeqTrees overhead. The percentage overhead asymptotes at between $40\% - 60\%$ for the GPU executions (depending on the exact GPU run) and around 5% for the CPU execution (Figure 6.5a). If we were to reduce the running time of the computational sections of code to 0 seconds, we find that by a corollary of Amdahl's Law we would only be able to achieve a maximum speed-up of $25\times$ (Figure 6.5b - black line).

Amdahl's Law states that for a parallelised program, of which a ratio of $B \in [0,1]$ is strictly serial and thus $1 - B$ is fully parallelisable, then the time it takes for n parallel threads to complete execution of that program is given by $T(n) = T(1)(B + \frac{1-B}{n})$, where $T(1)$ is the time it would take to run the program with 1 thread (serially). A corollary of this law is that we can define the theoretical speed-up for n threads, which we called $S(n) = \frac{T(1)}{T(n)} = \frac{1}{B + (1-B)/n}$.

MeqTrees Overhead as a percentage of Total Time

Peak Theoretical Speed-up

(a)

(b)
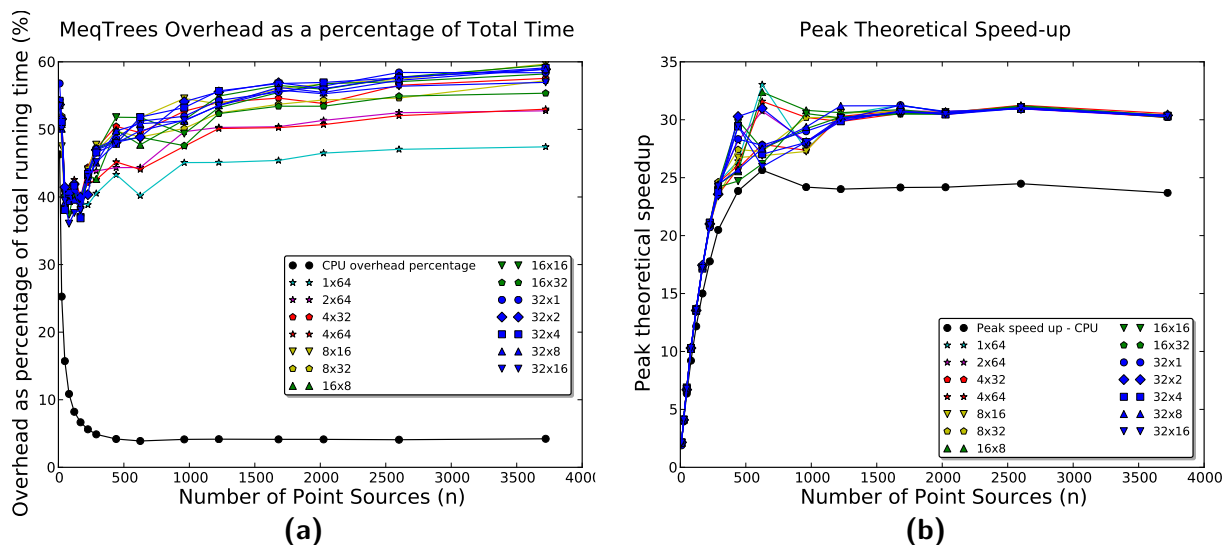
Figure 6.5: Percentage time of MeqTrees overhead compared to total running time (MeerKAT array, nVidia GTX 470 and Intel i5 2.66GHz)

(a) The percentage of runtime taken up by the Meqtrees overhead. (b) The maximum theoretical speed-up, calculated by assuming that the computational parts of the GPU run time takes 0 seconds (a theoretical infinite Constrained *speed-up).*

Increasing n towards infinity,

$$P(B) = \lim_{n \to \inf} \frac{1}{B + \frac{1-B}{n}} = \frac{1}{B}$$

gives us the peak theoretical speed-up, beyond which no further speed-up is possible without modifying B, how much of the program is parallelisable Amdahl (1967).

Considering that the overhead takes up approximately 4% of the total running time, or a ratio of 0.04. If the CPU simulation runs in x seconds, then overheads (non-parallelisable sections of code) account for $0.04x$ seconds. If we now assume that we can reduce the parallelisable (constrained) run time to zero, we have a total run time of $0.04x$. Calculating the speed-up we find it is $\frac{x}{0.04x} = \frac{1}{0.04} = 25 \times$. In terms of Amdahl, the peak theoretical speed-up $P(0.04) = \frac{1}{0.04} = 25 \times$

The MeqTrees overhead is calculated by subtracting the node running time from the total running time. We expected that the MeqTrees overhead would be the same in both the CPU and GPU test runs. However, we found that the MeqTrees overhead measured in the CPU run was approximately 25% greater than the MeqTrees overhead in the GPU versions (Figures 6.6a and 6.6b). This extra overhead time is not significant for competitively large

90
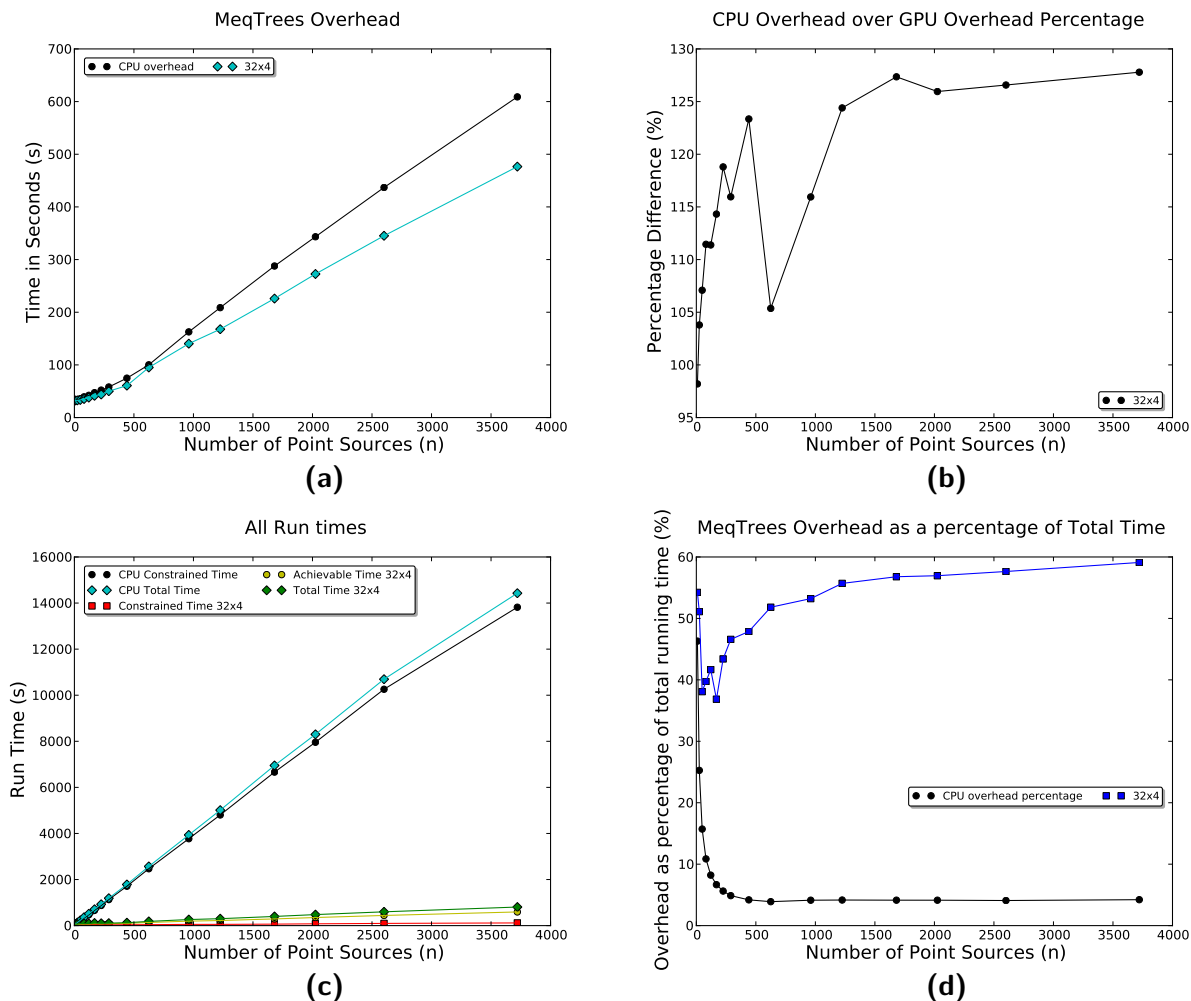
Figure 6.6: Discrepancy between CPU and GPU overheads

(a) The run times of the MeqTrees overheads in the CPU and GPU versions and (b) the percentage difference between them. Whilst the overheads should be exactly the same for both the CPU and GPU implementations, we find that they differ by 25%. (c) The total running times for CPU and GPU versions and (d) the percentage of time taken up by MeqTrees overhead for CPU and GPU versions. Considering the total run time of the CPU, the overhead discrepancy is small; however, for the GPU run time this discrepancy can be significant.

CPU run times (Figures 6.6c), as the total fraction of overheads compared to run time is only about 5% of run time: meaning a 25% discrepancy causes only a 1.25% difference in running time. However, as a fraction of GPU run times, overheads are as much as 60% of the total run time (Figure 6.6d), a discrepancy of 25% leads to an overall discrepancy of up to as much as 15%.

We were unable to determine the cause of this discrepancy. We measure MeqTrees overheads by subtracting the computational time from the total running time. We could be tempted to modify our timing metrics to conservatively adjust for this discrepancy, but we do not for the following reasons: the *Real* speed-up is comparison of the total running times reported by the operating system and thus there is no need for modification. The *Constrained* speed-up ignores all overheads and measures the effect of the computational section of code only, thus this discrepancy does not need to be accounted for. The theoretical maximum speed-up is very sensitive to the overhead run time values, and thus sensitive to this discrepancy. We therefore include the peak speed-up calculated given the slightly reduced overhead times Figure 6.5b (non black lines). The reduced overheads lead to a slightly higher theoretical maximum speed-up, converging at 30× for large test cases.

This implies that without speeding up the MeqTrees overhead, we will be unable to achieve greater than 30× speed-up. As mentioned before, our version can speed up the core calculations by two orders of magnitude. We find that overheads account for *more than half* of the total running time in the GPU case. So whereas prior to our GPU implementation, it was clear that the calculations (the parallelisable sections of code) were a significant fraction of run time, now both the overheads and the calculations are equally significant. Additional significant speed-ups could be achieved by further speeding up the parallelisable sections of code, but also by reducing overhead time.

6.4   Performance for the WSRT Array

The Westerbork Synthesis Radio Telescope (WSRT) is a 14 element array interferometer (91 pairs), smaller than the MeerKAT interferometer which has 64 elements (2016 pairs). While they have a different number of pairs to process, the rate at which each set is processed should remain the same for both. Assuming all other factors (number of sources, number of total time-steps, number of frequency band) are the same, MeerKAT simulates over 480 time slots (15 groups of 32) and WSRT into 72 time-steps (3 groups of 32 with the third

group comprising of 8 actual time-steps and 24 padded dummy time-steps that equate to 0, as we configure MeqTrees to compute in 32 time-step batches). The speed-up values in Figure 6.7 show that with the smaller dataset, smaller speed-ups are achieved. We would expect both datasets to run at similar rates as they only differ by the number of times the PSV node is called, and not the computational load of each PSV node call.

In the MeqTrees execution model (Section 4.2.1) we noted that MeqTrees splits the problem over groups of antenna pairs over 32 time-steps. Therefore, MeerKAT calculates 2016 pairs over 15 time slot groups (30240 'pair groups') and WSRT only 91 pairs over 3 time slot groups (273 'pair groups'). In Table 6.1 we show the running times per pair group. We find that, for the CPU version, the MeerKAT simulations run almost twice as fast per pair group than the WSRT simulations across both computation code and overhead code. The GPU code runs the MeerKAT version 3 times faster than the smaller WSRT. If we consider the computational/parallelisable component of the program only, this increases to 4 times, with overheads scaling in a similar fashion to the CPU version.

As discussed in Section 4.2.1, the visibility calculations for each antennae pair are executed in serial. This should imply that GPU utilisation remains as efficient, no matter the number of 'pair groups'. However, as mentioned above, we find that the larger MeerKAT array runs two to three times faster per antennae pair as the smaller WSRT array for both the CPU and GPU runs.

Ideally we should run further tests on additional arrays or additional sizes to determine the exact nature of the difference in efficiency. If we find that efficiency increases proportionately with array size, as the two data-points *might* imply, then this implies a number of potential causes. Caching characteristics on the GPU could be more efficient for larger cases. Owing to the padding in the WSRT input data, there might be divergent code. Also, time taken for kernel invocations is proportionately shorter for larger test cases, owing to longer kernel running times. Any combination of these and possibly other as yet discovered effects could be at play. Full investigation of these considerations are left for future work when many other array layouts can be compared and efficiency characteristics based on array size, dataset information order on disk and file size of these datasets among others.
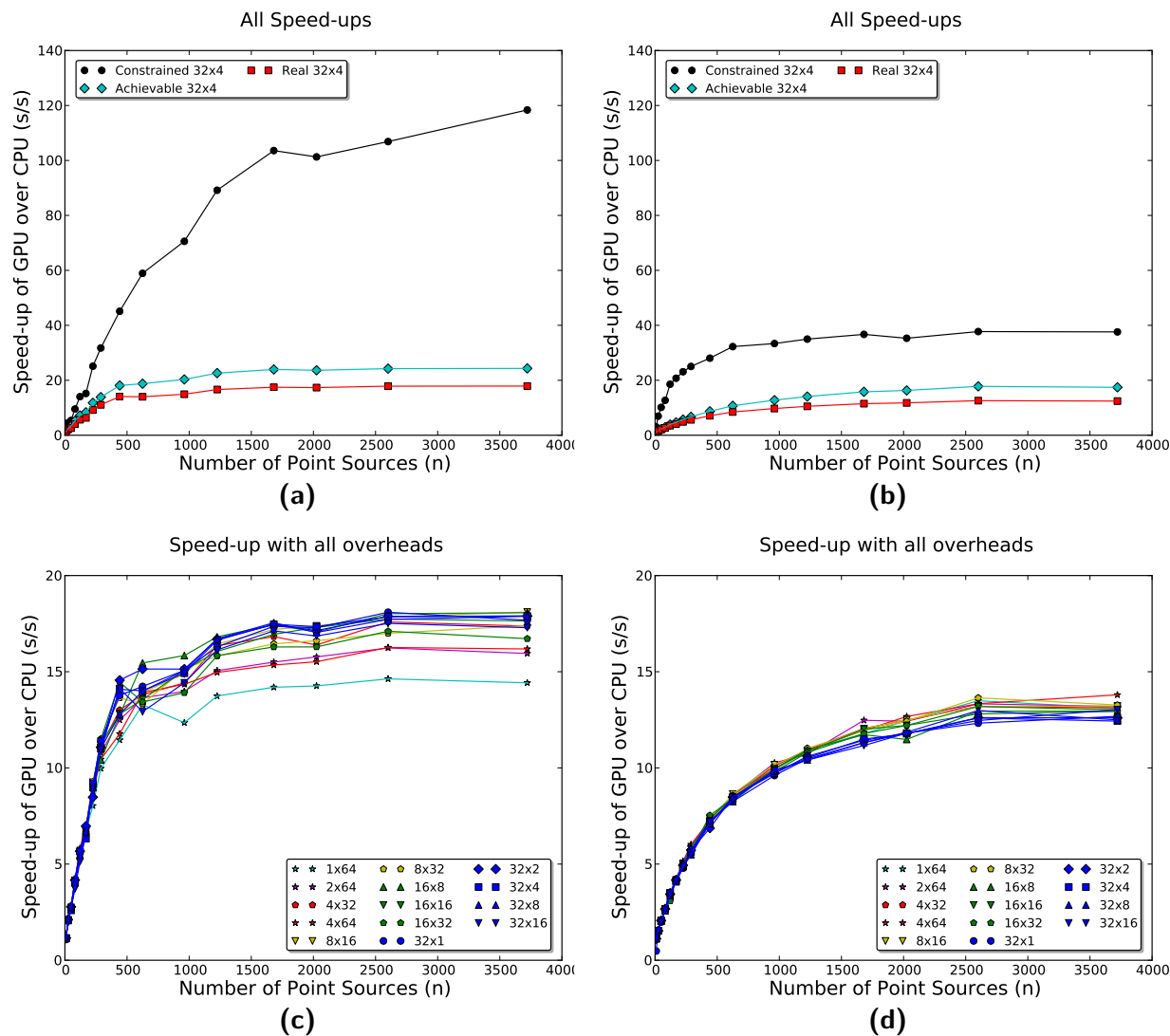
Figure 6.7: Speed-ups of PSV calculations with the smaller Westerbork Synthesis Radio Telescope compared to the MeerKAT array (WSRT array/MeerKAT array, nVidia GTX 470 versus Intel i5 2.66GHz)

The smaller WSRT array (14 elements — b, d) does not perform as well with the larger MeerKAT array (64 element — a, c). The overhead still takes up a large proportion of running time (40%) which limits any real speed-up. Although the WRST array has less impressive speed-ups ($9\times$ from $16\times$ and $35\times$ from $120\times$) compared to MeerKAT array, it still runs significantly faster than the CPU code.

			MeerKAT	WSRT	Difference
		Pairs	2016	91	
		time-steps	480	72	
		Time slot groups	15	3	
		Pair groups	30240	273	
CPU		Run Time	14427.67 s	252.65 s	
		per group	0.4771 s	0.9254 s	51.55%
		Core Time	13818.72 s	241.35 s	
		per group	0.4569 s	0.8840 s	51.68%
		Overhead Time	608.95 s	11.3 s	
		per group	0.0201 s	0.0413 s	48.66%
GPU (32×4)		Run Time	806.21 s	20.32 s	
		per group	0.0266 s	0.0744 s	35.75%
		Core Time	329.69 s	12.24 s	
		per group	0.0109 s	0.0448 s	24.33%
		Overhead Time	476.51 s	8.08 s	
		per group	0.0157 s	0.0295 s	53.22%

Table 6.1: Running times per time slot group per pair (MeerKAT and WSRT array, nVidia GTX 470 versus Intel i5 2.66GHz)
When we adjust the total, core, and overhead running times, we see that the larger dataset runs faster than the smaller dataset. Furthermore, this effect seems to be even more pronounced in the GPU version's core running time, although the overheads of both the CPU and GPU version show a similar change in computation rate.

6.5   Effect of Occupancy, L1-Cache, and CC 2.0

Occupancy does not have a significant effect on speed-up (Table 6.2); far more benefit accrues from other considerations, such as the use of shared memory versus global memory, which results in an increase from a $16.34\times$ to an $18.10\times$ speed-up. In the case of the *Constrained* speed-up, the computational part of the simulation runs almost twice as fast when using shared memory against using global memory only ($125.14\times$ at 17% occupancy versus $69.9\times$ at 67% occupancy). Table 6.2 shows that tests with with similar occupancies show no noticeable correlation between performance and occupancy, for example we find that there are runs at 17% occupancy with speed-ups as disparate as $108.87\times$ and $125.14\times$, and runs at 33% occupancy with similar speed-ups of $104.55\times$ and $124.21\times$.

In all cases (Table 6.2) the fastest run time happened to have the highest occupancy. That the values are so similar along all occupancies suggests other factors play a more important

| | | | | Best Speed-ups | | | |
| | | | | *Real* speed-up | *Achievable* speed-up | *Constrained* speed-up | Best |
CC	Threads /Block	Sh.m /SM	Occ.	All overheads	Some overheads	No overheads	block dims
	32		33%	17.57	23.75	110.04	32×1
	64		33%	17.89	24.24	118.46	32×2
	128	48kB	42%	18.00	24.48	124.21	8×16
	256		33%	17.68	23.95	110.68	32×8
	512		33%	17.64	23.78	104.55	32×16
	32		17%	17.67	23.90	108.87	32×1
	64	16kB	17%	17.89	24.30	116.25	32×2
2.0	128		17%	18.10	24.62	125.14	8×16
	256		17%	17.88	24.19	114.25	32×8
	32		17%	16.05	21.03	66.38	32×1
	64	16kB	33%	15.90	20.74	63.05	32×2
	128	(un-	67%	15.76	20.55	61.61	32×4
	256	used)	67%	16.34	21.51	69.99	8×32
	512		67%	16.07	21.14	67.71	32×16
	32		25%	16.19	21.22	71.42	32×1
	64	16kB	25%	16.61	21.90	79.40	32×2
	128		25%	16.21	21.33	71.11	16×8
	32	16kB	25%	13.85	17.42	38.51	32×1
1.3	64	(un-	50%	15.17	19.59	51.76	32×2
	128	used)	50%	15.45	20.00	54.27	32×4
	256		50%	15.25	19.73	52.98	32×8

**Table 6.2: Effect of occupancy on speed-up**

This table outlines the speed-ups achieved given certain occupancies. Occupancy (Occ.) is entirely determined by (i) the Compute Capability (CC) of the device (ii) the amount of shared memory available and (iii) the number of threads per block. For each thread-block size, we show the speed-up results of the best thread-block layout.

role. This can be most clearly seen for the CC 2.0 device where no shared memory is used, in which the layout with 17% occupancy performs slightly better than the 33% and 66% occupancy layouts. There is no apparent correlation between occupancy and speed-ups, if there were they would be minor in comparison to other considerations such as the use of shared memory, which almost doubles performance.

CC 2.x devices allow for a choice of 16kB/48kB or 48kB/16kB configurations for shared memory/L1 cache (respectively). With 48kB of shared memory, instead of 16kB the occupancy increases up to two times. We see (Table 6.2) that there is little difference between the two configurations, with a negligible difference in speed-ups with the 8×16 thread-block layout between 48kB versus 16kB (124.21$\times$ versus 125.14$\times$ respectively) despite the difference in occupancy (42% versus 17% respectively). Figure 6.8 shows the difference between use of the two global L1 cache versus shared memory configurations for both *Real* and *Constrained* speed-ups. The two configurations show no discernible difference in speed-ups. This implies that L1 cache could have little to no effect on speed-ups. Which would be expected as our input memory operations are tiny in comparison to our output memory operations. This could also mean that the advantage gained from the increased shared memory is cancelled by the disadvantage of less cache.

Figure 6.9 shows the difference between a CC 1.3 (GTX 285) and CC 2.0 (GTX 470) devices. The overall *Real* speed-up increases from 16.61$\times$ to 18.10$\times$ (Table 6.2). We see that with all expected overheads present in MeqTrees, the *Achievable* speed-up increases from 21.90$\times$ to 24.62$\times$. Although only a 10% difference, consider that the *Constrained*/core execution times are almost twice as fast, going from 79.40$\times$ to 125.14$\times$ faster. The primary difference between the cards is the newer technology as they both have comparable performance statistics, with GTX 285 slightly outperforming the GTX 470. CC 2.x devices have a number of additional features, chief among these is the global memory L1-cache and the increase double precision performance. We found earlier that the amount of L1-cache has little effect on performance, thus we can conclude that the extra double precision performance has at least a significant effect.

CC 1.3 devices have a double precision unit for each SM (8 cores) they can perform double precision calculations at 1/8th the speed compared single precision performance. CC 2.x devices forego a special double precision unit and incorporate the operation inside the CUDA cores, however they take 2 clock cycles to complete a double precision operation instead of one. This means that double precision performance is 1/2 that of floating point performance
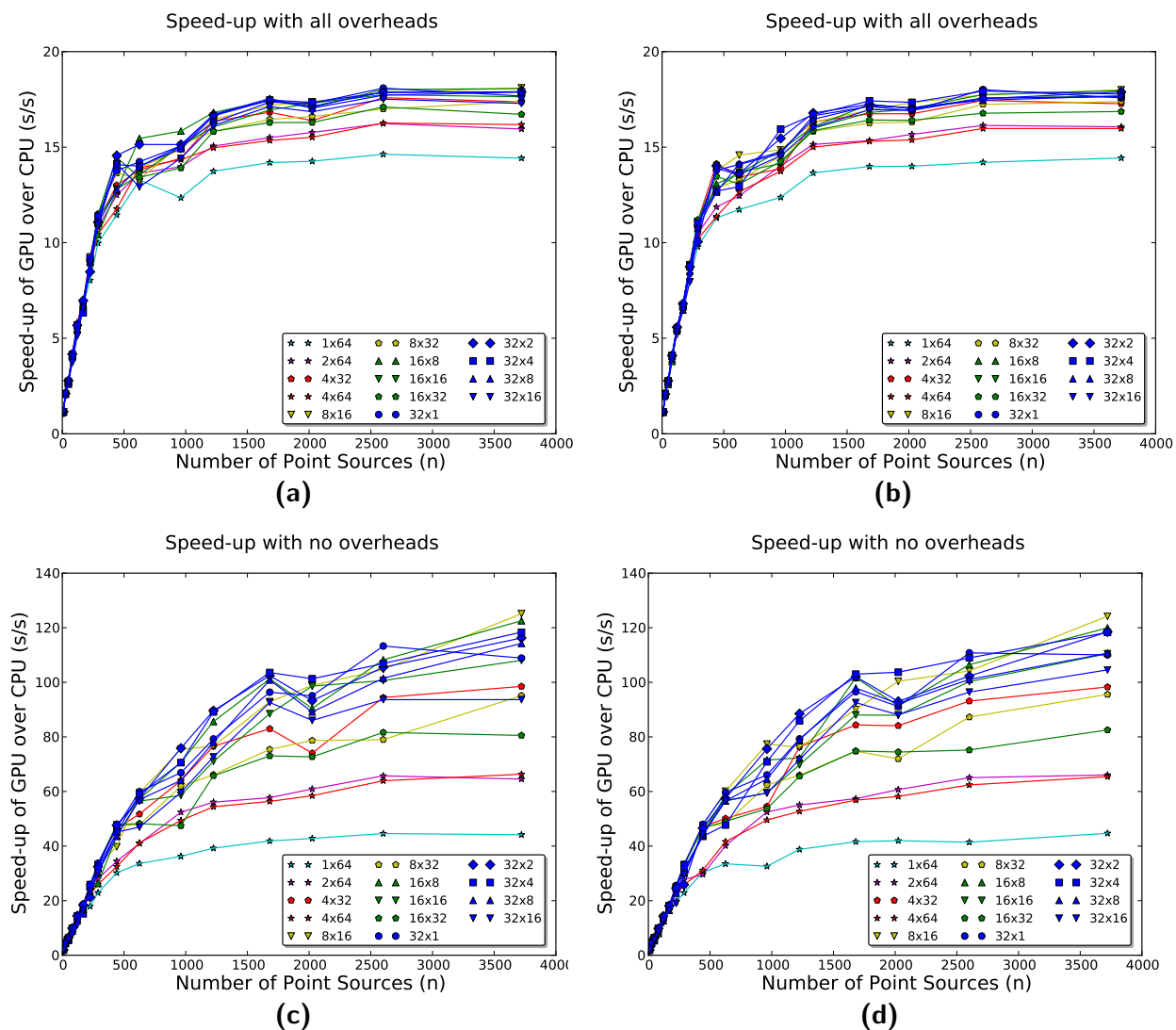
Figure 6.8: Effect on speed-up of 16kB versus 48kB shared memory configuration (MeerKAT array, nVidia GTX 470 and Intel i5 2.66GHz)
*(a) 16kB versus (b) 48kB shared memory configuration (*Real *speed-up). (c) 16kB versus (d) 48kB shared memory configuration (*Constrained *speed-up). When considering the* Real *and* Constrained *times, the spread is almost identical and as such we are unable to come to any noteworthy conclusions based on this data alone.*
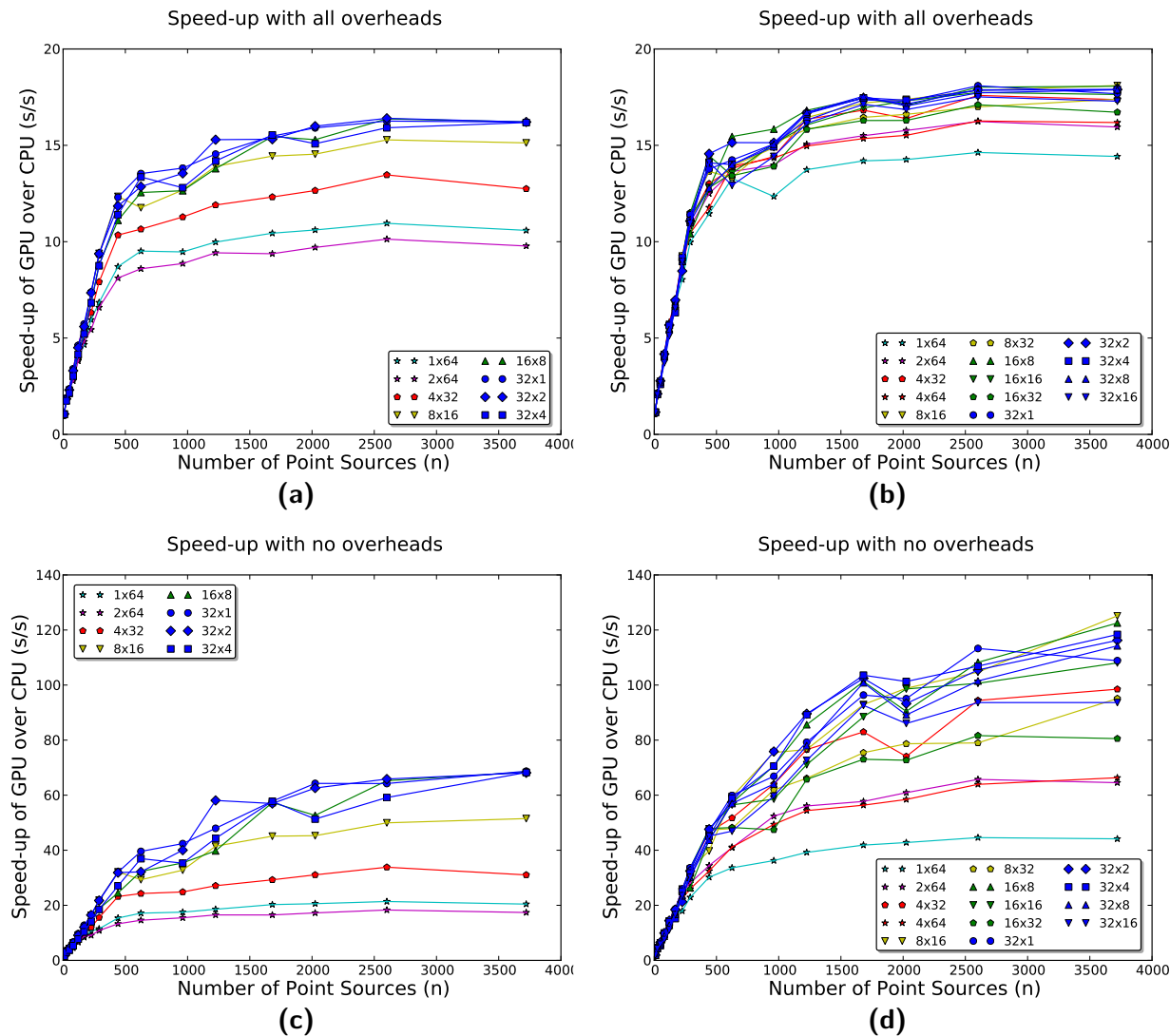
Figure 6.9: Comparison of results for GTX 285 vs GTX 470, Real and Constrained speed-up (MeerKAT array, nVidia GTX 285/GTX 470 and Intel i5 2.66GHz)

(a) Real *speed-ups of the GTX 285 and (b) GTX 470. (c)* Constrained *speed-ups of the GTX 285 and (d) GTX 470. The GTX 470 shows a tighter grouping than the GTX 285 in* Real *speed-up. But we see from the* Constrained *speed-up that the GTX 470 outperforms in all cases, with about double the performance of the GTX 285.*

99

(as confirmed by Nikolaj et al. (2009) in the Fermi white-paper). This should have given us a 4-fold increase in performance, instead of the 2-fold increase seen in Figure 6.9. Further investigation shows that the GF100 architecture (which is the base of the commercial GTX 470 and GTX 480) limits its double precision performance to only 1/4th of its floating point performance (Wasson). This explains why the GTX 470 only performs twice as fast as the GTX 285. If we were to test these results on a Tesla device with similar memory bandwidth and computational characteristics, such as the High Performance Computing focused Tesla M2070 (150 GB/s, 1030 GFLOPS) (nVidia Corporation, 2012a), we would expect performance twice as fast as the GTX 470.

Although the core computations will double in speed, as mentioned above (in Section 6.3), the MeqTrees overheads would limit any real speed-up values. For instance, a speed-up of $18.00\times$ (with core speed-up of $124.21\times$) would only improve to $19.34\times$ (with core speed-up of $248.42\times$) if we were to halve run time of the computational part of the code with a Tesla device.

6.6   Discussion

Here we evaluated the performance of our GPU implementation under a number of speed-up metrics, each intending to show different aspects of parallel scaling. Most importantly, we show the *Real* speed-ups, to determine how fast the program is for practical usage. We achieved speed-ups of around $18\times$ for large data sets, a significant improvement over the CPU code: turning a 9-hour simulation into a 30 minute simulation.

We further calculate that we could reach speed-ups of $25\times$ if we were to forego unnecessary reorder and copy operations. For just the core computations, we reach speed-ups of $120\times$.

The MeqTrees overhead, while acceptably small for the CPU version, become a serious impediment when accelerating the core computations on a GPU. This bottleneck limits any real speed-ups that an end user could experience to about $30\times$, despite an acceleration of two orders of magnitude to the core computation. To further accelerate overall run times, the MeqTrees overhead would have to be accelerated in some manner as well.

Two optimisations that are key to achieving significant speed-ups in CUDA: the effective use of shared memory and the improved double precision performance gained from newer hardware. Use of shared memory allows us to reduce the number of global memory reads

and writes. Since the PSV simulation requires many memory operations, this shows a two-fold increase in performance. The introduction of CUDA's compute capable 2.x devices also improves our performance. Although global caching is a major feature of CC 2.x, it does not necessarily improve our results. The main boost comes from the improved double precision floating point performance of the CC 2.x's. We noticed that with two devices of similar memory and computation characteristics, but different compute capabilities, that the newer hardware outperformed the older by a factor of two. This could be improved to a factor of four if HPC focused Tesla GPUs were used.

We conclude that significant GPU acceleration of point source visibility calculations can indeed by achieved by CUDA devices. Although this is a 'proof of concept' prototype and not ready for use in public, in its current state it sets a platform for future work. Although our Thrust implementation is not successful, it does highlight that development time is an important factor if this project is to be made academically viable. Significant speed-up of PSV calculations means that more point sources can be simulated in the same amount of time, allowing for more realistic simulations, which aids in the development of more accurate radio frequency interference and atmospheric models. Accurate modelling of radio interference effects is key to third-generation calibration techniques, which will allow for interferometers to see fainter and more far off sources than with previous calibration techniques. More generally, this speed-up shows that SIMD problems in the MeqTrees framework can be accelerated with the use of GPU by at least an order of magnitude without modification to the framework, albeit with a significant investment in development time. In order to fully exploit GPU hardware, a workaround or modification to the MeqTrees framework might be required. This said, there is potential for encompassing GPU integration into more nodes in the MeqTrees framework, or create a general system within MeqTrees to offload intensive SIMD computational sections of a tree from the CPU to the GPU.

Chapter 7

Conclusions

Accelerating execution times of Point Source Visibility (PSV) simulations results in the ability to perform more substantial PSV simulations in the same amount of time. This, in turn, allows for more complex sky models with more point sources and simulation of larger interferometer arrays. More complex sky models with many point sources enables more accurate and realistic testing of more accurate calibration models. It also allows for larger interferometer arrays to be tested within reasonable time frames. However we see that modern and forthcoming interferometers will consist of many more array elements, up to two orders of magnitude more antennae. With this increase in antennae, the computational cost increases as well, demanding an increase in the ability to execute the required simulations within acceptable running times.

Our work set out to apply CUDA technology to PSV calculations in the MeqTrees framework. This project explored areas of radio interferometry, specifically PSV calculations, MeqTrees, GPGPU technology, and CUDA hardware, which we brought together to create a PSV simulation node on commodity GPU hardware that runs an order of magnitude faster than its CPU counterpart.

There are a number of points, both technical and qualitative, to take away from this work. Chief among the technical points is the reduction of global memory throughput, which significantly improved overall performance. We also learn of the difficulties involved in development, trading off performance for development speed and vice-versa. Another key point is that MeqTrees is not designed with GPUs in mind, the inherent dynamic and flexible model of the MeqTrees framework clashes with the GPU model which benefits from

a more static and rigid implementations. This is not to say that GPU integration is not appropriate for MeqTrees — indeed, in the case of PSV calculations, it was found show many computational benefits — just that it is a non-trivial problem.

The project started as an exploration into image synthesis on the GPU, where image synthesis is the transformation of visibilities into an image of the sky. We achieved around a 20× speed-up compared to a custom CPU image synthesis implementation. We discovered that the reverse problem, visibility simulation, from an image of the sky to simulated visibilities, was a key feature in the MeqTrees framework, and it was decided that this problem would have more impact on the academic community. Fortunately, visibility simulation is simply an inverse Fourier transform, an almost identical problem to what had been addressed in the image synthesis project.

For development of this new PSV simulation, we had access to the MeqTrees framework, with an already developed and academically utilised PSV node. Writing a GPU node to compare against this held potential for greater impact than an image synthesis application. A major disadvantage to the image synthesis project was that we did not have any standard CPU-based package to compare our results to. Moving to a PSV simulation project meant that we could now validated our data against CPU data using the standard software package, MeqTrees.

We also had to tackle the problem of simulating multiple frequency bands and multiple time slots. While this increased the complexity of the problem, it afforded us an opportunity to decompose the problem in 3 dimensions (time, frequency and sources), which neatly matched the CUDA software model.

Our naive implementation was tested on the nVidia GTX 285 with the smaller WSRT dataset and achieved a reasonable 8× speed-up over the CPU version. We soon considered the effect of array ordering on coalescing, and modified the code to exploit it. This only boosted performance to about $11 - 12\times$ speed-up.

We also attempted to prototype a Thrust version of the PSV node. We found that development was not as tricky as custom CUDA code as there were less CUDA-specific nuances that needed to be considered. However, development was far from smooth, as many complicated errors stemming from the multiple Thrust iterator layers slowed development speed, making it almost as complex (albeit in a different manner) as coding an optimised CUDA kernel. This combined with poor performance meant that we decided to focus on the custom version

instead.

We upgraded our hardware to the new GTX 470 which afforded us global memory caching, better double precision floating point performance and extra shared memory. It was at this time that we explored the use of shared memory which improved results to around $14 - 15\times$ speed-up. Although not much higher than the $12\times$ speed-up we had before, we had achieved a $40\times$ core speed-up.

When testing with the larger MeerKAT dataset we found that results could reach even higher values, achieving an $18\times$ speed-up, and more importantly, a $120\times$ core speed-up. We realised that real speed-ups were increasing at a slow rate, which lead to our discovery that the bottleneck lay no longer with the computation, but with the overheads. The exact bottleneck we encountered is described by Amdahl's law, as the non-parallelisable sections of code, the running time of which makes up a constant ratio of the total running time, limits the maximum theoretical speed-up achievable.

Part of these overheads included many reordering and recopying operations performed by our GPU PSV nodes and thus we have emphasised that proper GPU memory management for MeqTrees should be the primary focus of any further GPU implementations within MeqTrees. Whilst a memory management module could be implemented by the current unmodified MeqTrees framework, this is a messy and potentially unhelpful approach to the more general problem of bringing the computational efficiency of the GPU to MeqTrees. Both exploration of a memory management module implemented on top of MeqTrees and modification of the MeqTrees internal working to incorporate GPU memory management, is left for future work.

Another substantial problem that we encountered was the need for the PSV module be general enough to compute PSV with multiple *Jones* arguments, which are 2×2 matrices that each describe a linear effect on an incoming radio signal (see A for details). MeqTrees allows for the PSV equation to be modified with multiple Jones matrices to describe propagation effects.

We attempted to implement multiple Jones matrices with our kernel; however this was unsuccessful and slow because of excessive register pressure brought on by the complexity of the equation. We attempted a prototype implementation with multiple kernel invocations to relieve this pressure. Whilst this worked, the multiple invocations are not compatible with our shared memory optimisation. This and the extra CUDA overhead brought on by extra kernel invocations was slow and also proved unsuccessful. It was decided that the problem

of generalising the PSV node to incorporate multiple Jones terms was a project that should be left for future work.

No multi-core implementations were attempted in this work. It would be out of the scope of this project to speculate about what performance gains might be achieved by any multi-core implementation. Whilst we do not focus on any potential effects of a multi-core implementation, many of our discussions can be transferred from the GPU sphere to the multi-core CPU sphere. We could theorise what a multi-core implementation might do: for instance an 8-core machine running at ideal linear speed-ups might produce an 8-fold acceleration ($8\times$ speed-up). As this would speed up the parallelisable sections of code only, and not the overheads, this would amount to about a $6\times$ speed-up overall. We found that (Section 6.3) with 4% overheads in the CPU version, we can never achieve a greater than $30\times$ *Real* speed-up, no matter the method of acceleration. So even, say a 128-node compute cluster, accelerating the computational sections of code by $128\times$, would only achieve a single order of magnitude ($18\times$) faster run times overall, similar to our CUDA acceleration. This is only if the serial overheads cannot be accelerated as well, offloading the overheads to an HPC platform, be that GPU or multiple cores, could result in speed-ups of greater than two orders of magnitude over the current implementation. Exploration of MeqTrees overhead acceleration is left for future work.

With this, we conclude the work presented in this thesis. The PSV acceleration problem, characterised by a requirement for a large amount of off-chip memory for storage of intermediate results, benefited most from optimisations that reduced throughput to global memory. Optimisations such as use of shared memory to perform intermediate operations in on-chip memory, rather than storing it first in global memory and operating on it at a later stage. We also pack memory in an optimal manner as to exploit CUDA's global memory coalescing. We have achieved significant $18\times$ speed-up over a single core CPU implementation using modern commodity GPU hardware. We have discovered the importance that the overheads, or more generally the serial sections of code, can have on this sort of problem. In our case, no amount of acceleration of the parallelisable sections of code would lead to anything more than an order of magnitude speed-up ($30\times$), noting that we were able to achieve acceleration of two orders of magnitude ($120\times$) for the core computations over the CPU counterpart. Although non-trivial to code, we find that benefits can be gained from CUDA-based acceleration, especially given an appropriate memory management model.

There are rich veins of possibilities that could stem from the work presented here. Not all

problems are suited to SIMD solutions (such as the way in which PSV simulations are), but many data-intensive problems are present in radio interferometry and in MeqTrees. The challenge in making any potential GPU implementation MeqTrees implementation lies primarily in reducing the overheads by at least the same rate at which core computations are accelerated, and accelerating the computational nodes whilst retaining the great flexibility MeqTrees provides. These are not simple tasks but this thesis hopes it has shown a tentative first step towards achieving such a feat.

References

AMD. AMD Press Release, 2010. URL `http://www.amd.com/us/press-releases/pages/firestream-peak-performance-2010june23.aspx`.

G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483—-485, Atlantic City, New Jersey, 1967. ACM. URL `http://doi.acm.org/10.1145/1465482.1465560`.

Associated Universities Inc. NRAO Very Large Array, 2012. URL `http://www.vla.nrao.edu/`.

ASTRON. LOFAR Technical Information — ASTRON, 2012a. URL `http://www.astron.nl/radio-observatory/astronomers/technical-information/lofar-technical-information`.

ASTRON. LOFAR TECHNICAL INFORMATION — ASTRON, 2012b. URL `http://www.astron.nl/radio-observatory/astronomers/technical-information/lofar-technical-information`.

N. F. Bate, C. J. Fluke, B. R. Barsdell, H. Garsden, and G. F. Lewis. Computational advances in gravitational microlensing: a comparison of CPU, GPU, and parallel, large data codes. *New Astronomy*, 15(8):11, 2010. URL `http://arxiv.org/abs/1005.5198`.

E. O. Brigham and R. E. Morrow. The Fast Fourier Transform. *Proceedings of the IEEE*, 490(12):490–494, 1967. ISSN 00189235. doi: 10.1109/MSPEC.1967.5217220. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5217220`.

I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs : Stream Computing on Graphics Hardware. *ACM Trans Graph*, 23(3):

777–786, 2004. ISSN 07300301. doi: 10.1145/1015706.1015800. URL `http://portal.acm.org/citation.cfm?id=1015800`.

B. F. Burke and F. Graham-Smith. *An Introduction to Radio Astronomy: Second Edition.* Cambridge University Press, 2002. URL `http://adsabs.harvard.edu/abs/2002ira..book.....B`.

Y. Chen, X. Cui, and H. Mei. Large-scale FFT on GPU clusters. *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 315–324, 2010. doi: 10.1145/1810085.1810128. URL `http://portal.acm.org/citation.cfm?doid=1810085.1810128`.

M. A. Clark, P. C. La Plante, and L. J. Greenhill. Accelerating Radio Astronomy Cross-Correlation with Graphics Processing Units. *International Journal of High Performance Computing ApplicationsL*, page 36, 2011. URL `http://arxiv.org/abs/1107.4264`.

ClearSpeed Technology Ltd. ClearSpeed, 2012. URL `http://www.clearspeed.com/`.

J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965. URL `http://www.jstor.org/stable/2003354`.

CSRIO. ASKAP Home, 2012. URL `http://www.atnf.csiro.au/projects/askap/index.html`.

P. E. Dewdney, P. J. Hall, R. T. Schilizzi, and T. J. L. W. Lazio. The Square Kilometre Array. *Proceedings of the IEEE*, 97(8):5, 2011. URL `http://arxiv.org/abs/0910.0632`.

Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju. Auto-tuning of fast fourier transform on graphics processors. *Compute*, 11:257–266, 2011. doi: 10.1145/1941553.1941589. URL `http://impact.crhc.illinois.edu/ftp/conference/sara2011.pdf`.

B. Duan, W. Wang, X. Li, C. Zhang, P. Zhang, and N. Sun. Floating-point mixed-radix FFT core generation for FPGA and comparison with GPU and CPU. In *2011 International Conference on Field-Programmable Technology*, pages 1–6. High Performance Computer Research Center, Institute of Computing Technology, Chinese Academy of Sciences, IEEE, 2011. ISBN 9781457717390. doi: 10.1109/FPT.2011.6132672. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6132672`.

E. B. Ford. Parallel Algorithm for Solving Kepler's Equation on Graphics Processing Units:

Application to Analysis of Doppler Exoplanet Searches. *New Astronomy*, 14(4):19, 2008. URL `http://arxiv.org/abs/0812.2976`.

N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete Fourier transforms on graphics processors. *2008 SC International Conference for High Performance Computing Networking Storage and Analysis*, (November): 1–12, 2008. doi: 10.1109/SC.2008.5213922. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5213922`.

M. Gschwind. Chip multiprocessing and the cell broadband engine. *Proceedings of the 3rd conference on Computing frontiers CF 06*, 23921:1–8, 2006. doi: 10.1145/1128022.1128023. URL `http://portal.acm.org/citation.cfm?doid=1128022.1128023`.

L. Gu, X. Li, and J. Siegel. An empirically tuned 2D and 3D FFT library on CUDA GPU. *Tensor*, pages 305–314, 2010. doi: 10.1145/1810085.1810127. URL `http://portal.acm.org/citation.cfm?id=1810127`.

J. Hensley. ATI CTM overview. In *ACM SIGGRAPH 2007 courses*, page 7, New York, NY¡ USA, 2007. ACM.

J. Hoberock and N. Bell. thrust - Code at the speed of light - Google Project Hosting, 2012. URL `http://code.google.com/p/thrust/`.

J. Horrell. MeerKAT Specifications and Science, 2012. URL `http://public.ska.ac.za/meerkat`.

IBM Research. IBM Research — Projects — MD-GRAPE. URL `http://www.research.ibm.com/grape/`.

R. C. Jones. A New Calculus for the Treatment of Optical Systems. *Journal of the Optical Society of America*, 31(7):488, 1941. ISSN 00303941. doi: 10.1364/JOSA.31.000488. URL `http://www.opticsinfobase.org/abstract.cfm?URI=josa-31-7-488`.

P. Jonsson and J. Primack. Accelerating Dust Temperature Calculations with Graphics Processing Units. *New Astronomy*, 15(6):7, 2009. URL `http://arxiv.org/abs/0907.3768`.

K. Karimi, N. G. Dickson, and F. Hamze. A Performance Comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581*, cs.PF(1):12, 2010. URL `http://arxiv.org/abs/1005.2581`.

REFERENCES

I. Kozin. Evaluation of ClearSpeed Accelerators for HPC. Technical Report August, Science and Technology Facilities Council, 2009. URL `http://epubs.cclrc.ac.uk/work-details?w=50895`.

Y. Li, Y. Zhang, H. Jia, G. Long, and K. Wang. Automatic FFT Performance Tuning on OpenCL GPUs, 2011. ISSN 15219097. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6121282`.

S. Matsuoka, T. Aoki, T. Endo, A. Nukada, T. Kato, and A. Hasegawa. GPU accelerated computing-from hype to mainstream, the rebirth of vector computing. *Journal of Physics Conference Series*, 180:012043, 2009. ISSN 17426596. doi: 10.1088/1742-6596/180/1/012043. URL `http://stacks.iop.org/1742-6596/180/i=1/a=012043?key=crossref.7f9087c19a1aff0f99f069a20e84e998`.

N. McKay and R. Wark. Australia Telescope Compact Array, 2009. URL `http://www.narrabri.atnf.csiro.au/`.

Microsoft Corporation. DirectX11 DirectCompute, 2010. URL `http://www.microsoftpdc.com/2009/P09-16`.

K. Moreland and E. Angel. The FFT on a GPU. *Computing*, 2003(2003):112–119, 2003. URL `http://portal.acm.org/citation.cfm?id=844174.844191&type=series`.

A. Munshi. OpenCL Specification. Technical Report 2, Khronos OpenCL Working Group, 2011. URL `http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:OpenCL+Specification#0`.

NCRA-TIFR. Introducing GMRT. URL `http://www.ncra.tifr.res.in/ncra/gmrt/about-gmrt/introducing-gmrt-1`.

J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008. ISSN 15427730. doi: 10.1145/1365490.1365500. URL `http://portal.acm.org/citation.cfm?doid=1365490.1365500`.

L. Nikolaj, V. Osipov, and P. Sanders. Fermi Architecture White Paper. Technical report, nVidia Corperation, 2009. URL `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

J. E. Noordam and O. M. Smirnov. The MeqTrees software system and its use for third-

REFERENCES

generation calibration of radio interferometers. *Astronomy & Astrophysics*, 524(15013): 15, 2011. URL `http://arxiv.org/abs/1101.1745`.

A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. *2008 SC International Conference for High Performance Computing Networking Storage and Analysis*, 11pages(November):1–11, 2008. ISSN 1873426X. doi: 10.1109/SC.2008.5213210. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5213210`.

H. Nussbaumer. Fast Fourier transform and convolution algorithms. *Berlin and New York, Springer-Verlag(Springer Series in Information Sciences.*, 2, 1982.

nVidia Corporation. GeForce 256, 1999. URL `http://www.nvidia.com/page/geforce256.html`.

nVidia Corporation. Press Release, NVIDIA Unveils CUDA-The GPU Computing Revolution Begins, 2006. URL `http://www.nvidia.com/object/IO_37226.html`.

nVidia Corporation. Cuda C best practices guide. Technical Report May, NVIDIA Corporation, 2011a. URL `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf`.

nVidia Corporation. NVIDIA CUDA C Programming Guide Version 4.0. Technical report, Santa Clara, CA, 2011b. URL `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`.

nVidia Corporation. GeForce GTX 470 — Specifications — GeForce, 2012a. URL `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-470/specifications`.

nVidia Corporation. GeForce GTX 285 — Specifications — GeForce, 2012b. URL `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-285/specifications`.

nVidia Corporation. GPU-Accelerated Libraries — NVIDIA Developer Zone, 2012c. URL `http://developer.nvidia.com/cuda/gpu-accelerated-libraries`.

Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka. An efficient, model-based CPU-GPU heterogeneous FFT library. In *The Seventeenth International Heterogeneity in Computing Workshop*, pages 1–10, 2008.

J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. ISSN 00189219. doi: 10.1109/

REFERENCES

JPROC.2008.917757. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4490127`.

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Number 0. Morgan Kaufmann, 2008. ISBN 1558604286. URL `http://www.loc.gov/catdir/description/els032/97016050.html`.

M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multi-pass programmable shading. *Proceedings of the 27th annual conference on Computer graphics and interactive techniques SIGGRAPH 00*, pages 425–432, 2000. doi: 10.1145/344779.344976. URL `http://portal.acm.org/citation.cfm?doid=344779.344976`.

S. Qi, X. Wang, and S. Shi. Mixed Precision Method for GPU-based FFT. In *2011 14th IEEE International Conference on Computational Science and Engineering*, pages 580–586. IEEE, 2011. ISBN 9781457709746. doi: 10.1109/CSE.2011.103. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6062934`.

U. Rau, S. Bhatnagar, M. A. Voronkov, and T. J. Cornwell. Advances in Calibration and Imaging Techniques in Radio Interferometry. *Proceedings of the IEEE*, 97(8):1472–1481, 2009. URL `http://arxiv.org/abs/0902.0817`.

R. Sault and M. Wieringa. Multi-frequency synthesis techniques in radio interferometric imaging. *Astronomy and Astrophysics Supplement Series*, 108:585–594, 1994.

SETI Institute. Technical Overview — SETI Institute, 2012. URL `http://www.seti.org/seti-institute/project/technical-overview/details/technical-overview`.

L. SGI Japan. Completion of a one-petaflops computer system for simulation of molecular dynamics. URL `http://www.riken.jp/engn/r-world/info/release/press/2006/060619/index.html`.

SKA South Africa. MeerKAT, 2012. URL `http://www.ska.ac.za/meerkat/index.php`.

O. M. Smirnov. Revisiting the radio interferometer measurement equation. IV. A generalized tensor formalism. *Astronomy & Astrophysics*, 527(16434):16, 2011a. URL `http://arxiv.org/abs/1106.0579`.

O. M. Smirnov. Revisiting the radio interferometer measurement equation. III. Addressing direction-dependent effects in 21 cm WSRT observations of 3C 147. *Astronomy & Astrophysics*, 527(16435):11, 2011b. URL `http://arxiv.org/abs/1101.1768`.

REFERENCES

O. M. Smirnov. Revisiting the radio interferometer measurement equation. II. Calibration and direction-dependent effects. *Astronomy & Astrophysics*, 527(16434):11, 2011c. URL `http://arxiv.org/abs/1101.1765`.

O. M. Smirnov. Revisiting the radio interferometer measurement equation. I. A full-sky Jones formalism. *Astronomy & Astrophysics*, 527(16082):12, 2011d. URL `http://arxiv.org/abs/1101.1764`.

O. M. Smirnov. BuildingTimba/CheckingOut - MeqTrees Wiki, 2012. URL `http://www.astron.nl/meqwiki/BuildingTimba/CheckingOut`.

G. B. Taylor, C. L. Carilli, and R. A. Perley. *Synthesis Imaging in Radio Astronomy II*, volume 180 of *Astronomical Society of the Pacific Conference Series*. Astronomical Society of the Pacific, 1999. ISBN 0937707236. URL `http://adsabs.harvard.edu/abs/1999ASPC..180.....T`.

A. C. Thompson, C. J. Fluke, D. G. Barnes, and B. R. Barsdell. Teraflop per second gravitational lensing ray-shooting using graphics processing units. *New Astronomy*, 61 (May 2009):21, 2009. URL `http://arxiv.org/abs/0905.2453`.

A. R. Thompson, J. M. Moran, and G. W. Swenson. *Interferometry and Synthesis in Radio Astronomy*. Wiley-Interscience, 2001. ISBN 9780471254. URL `http://books.google.com/books?hl=en&lr=&id=AwBN5bpuEU0C&pgis=1`.

S. Wasson. GF100 graphics architecture unveiled - The Tech Report. URL `http://techreport.com/review/18332/gf100-graphics-architecture-unveiled/5`.

M. Wolleben. KAT-7, 2012. URL `http://www.ska.ac.za/media/kat7.php`.

Appendix A

Basics of Interferometry

All explanations in the following subsections are taken either directly or indirectly from Lecture 1 of *Synthesis Imaging in Radio astronomy II* by Taylor et al. (1999) in conjucntion with additional understandings from *Interferometry and synthesis in radio astronomy* (Thompson et al., 2001) and *An Introduction to Radio Astronomy, Second Edition* (Burke and Graham-Smith, 2002). Many details are left out as a matter of pragmatism. Please refer directly to these books for a more robust explanation.

A.1 General Direction

The end goal of this section is to derive a function of the *observed intensity* of incoming electromagnetic radiation by use of a synthesised aperture using two signal receivers (two radio dishes).

First, the phenomenon being observed is mathematically formulated as an electromagnetic field at the location of the phenomenon. Then, after some simplifying assumptions are made, the electromagnetic field at the location of observation is derived.

Using this information, the *Spatial Coherence Function* is derived in terms of the observed intensity of the field. At this point, it needs to be noted that the Spatial Coherence Function contains values that an interferometer measures.

After this, the Spatial Coherence Function undergoes a Fourier inversion to make it a function of the observed intensity (the value we wanted in the first place). Then real-world limitations

are taken into consideration. Such as discretisation and sampleing of the signal, declination of the observed source, adjustment for curvature of the earth and multi element (more than 2) arrays.

A.2 The Spatial Coherence Function

A.2.1 The Observed Electric Field

Say we are to measure some astrophysical phenomenon at location $\mathbf{R}$ (boldface to denote a vector quantity). This phenomenon, we assume, emits electromagnetic radiation and thus we can say it "causes a time-variable electric field" denoted by $\mathbf{E}(\mathbf{R}, t)$.

As time variable fields are 'inconvenient' to work with mathematically, we only consider a finite time interval of this field. Instead of $\mathbf{E}(\mathbf{R}, t)$, we use the coefficients of the real part of the sum of the Fourier series of $\mathbf{E}(\mathbf{R}, t)$ (Taylor et al., 1999) [1]. We call this $\mathbf{E}_\nu(\mathbf{R})$, thus eliminating the time variable. Note that $\mathbf{E}_\nu(\mathbf{R})$ is complex.

If $\mathbf{r}$ denotes the location of measurement, or *test-location*, we can now formulate $\mathbf{E}_\nu(\mathbf{r})$ by joining the fields produced at the test location:

$$\mathbf{E}_\nu(\mathbf{r}) = \iiint \mathbf{E}_\nu(\mathbf{R}) P_\nu(\mathbf{R}, \mathbf{r}) \, dx \, dy \, dz \tag{A.1}$$

Where $P_\nu(\mathbf{R}, \mathbf{r})$ describes how the electric field at $\mathbf{R}$ influences the electric field at $\mathbf{r}$ (called the *propagator*). The integral is taken over all of space.

A.2.2 Simplifying Assumptions

At this point, some simplifying assumptions are introduced. Firstly, and purely as a mathematical convenience, all electromagnetic radiation is treated as a scalar field, not as a vector field [2]. Mathematically, this means $\mathbf{E}_\nu(\mathbf{R})$ becomes $E_\nu(\mathbf{R})$. This can be accounted for and is so in practice, but explanation of this not be included in this background chapter. Refer

---

[1] This is due to the linearity of Maxwell's equations.

[2] This allows for scalar multiplication as opposed to vector multiplication and effectively is likened to ignoring polarisation phenomena (Taylor et al., 1999)

to Lecture 1, Section 5.2 of *Synthesis Imaging in Radio astronomy II* (Taylor et al., 1999) for explanation of the polerization adjustment.

The second is that all points of interest are a very long way away. This denies us almost all ability to describe depth of the phenomenon, but allows us to formulate surface brightness on the conceptual *celestial sphere*. A very large sphere of radius $|\mathbf{R}|$, that contains no additional radiation within it. We only consider the radiation on the surface of this sphere, denoted by $\mathcal{E}_\nu(\mathbf{R})$

The third assumption is that the celestial sphere is empty. Thus, in this case, according to Huygen's Principle, the propagator ($P_\nu(\mathbf{R}, \mathbf{r})$ in Eqn A.1) takes a particularly simple form

We can now say:

$$E_\nu(\mathbf{r}) = \int \mathcal{E}_\nu(\mathbf{R}) \frac{e^{2\pi i \nu |\mathbf{R}-\mathbf{r}|/c}}{|\mathbf{R}-\mathbf{r}|} \, dS \tag{A.2}$$

Where c is the speed of light, S is the surface of the celestial sphere and $\mathcal{E}_\nu(\mathbf{R})$ is the distribution of the electric field on the surface of the celestial sphere.

A.2.3   Correlation of the Field

In order to progress we formulate the correlation of the field at points $\mathbf{r}_1$ and $\mathbf{r}_2$. We define the *correlation of the field* as

$$V_\nu(\mathbf{r}_1, \mathbf{r}_2) = \langle \mathbf{E}_\nu(\mathbf{r}_1)\mathbf{E}_\nu^*(\mathbf{r}_2)\rangle$$

where the angled brackets denotes an average over time and the asterisk denotes the complex conjugate. By the first assumption we can thus say

$$V_\nu(\mathbf{r}_1, \mathbf{r}_2) = \langle E_\nu(\mathbf{r}_1)E_\nu^*(\mathbf{r}_2)\rangle$$

$V_\nu(\mathbf{r}_1, \mathbf{r}_2)$ is the signal that the radio dishes collect after they have been correlated, basically the raw data this thesis is concerned with. The first dish collects $\langle E_\nu(\mathbf{r}_1)\rangle$, which is the signal received at that dish averaged over time by the correlator. The second dish collects a similar averaged signal, $\langle E_\nu(\mathbf{r}_2)\rangle$ at the exact same time and this is then conjugated. These are multiplied together to get

$$\langle E_\nu(\mathbf{r}_1)\rangle\langle E_\nu^*(\mathbf{r}_2)\rangle = \langle E_\nu(\mathbf{r}_1)E_\nu^*(\mathbf{r}_2)\rangle = V_\nu(\mathbf{r}_1, \mathbf{r}_2)$$

Let us expand $V_\nu(\mathbf{r}_1, \mathbf{r}_2)$ using Eqn A.2 to substitute $E_\nu(\mathbf{r})$ we get:

$$
\begin{aligned}
V_\nu(\mathbf{r}_1, \mathbf{r}_2) &= \langle E_\nu(\mathbf{r}_1) E_\nu^*(\mathbf{r}_2) \rangle \\
&= \langle \int \mathcal{E}_\nu(\mathbf{R}_1) \frac{e^{2\pi i \nu |\mathbf{R}_1 - \mathbf{r}_1|/c}}{|\mathbf{R}_1 - \mathbf{r}_1|} \, dS \int \mathcal{E}_\nu^*(\mathbf{R}_2) \frac{e^{-2\pi i \nu |\mathbf{R}_2 - \mathbf{r}_2|/c}}{|\mathbf{R}_2 - \mathbf{r}_2|} \, dS \rangle
\end{aligned} \tag{A.3}
$$

We wish to combine these two integrals into a double integral, for this reason we replace S, the surface of the celestial sphere with two dummy spheres, S_1 and S_2, centered around dish one and two respectively. We now obtain:

$$
\begin{aligned}
V_\nu(\mathbf{r}_1, \mathbf{r}_2) &= \langle \int \mathcal{E}_\nu(\mathbf{R}_1) \frac{e^{2\pi i \nu |\mathbf{R}_1 - \mathbf{r}_1|/c}}{|\mathbf{R}_1 - \mathbf{r}_1|} \, dS_1 \int \mathcal{E}_\nu^*(\mathbf{R}_2) \frac{e^{-2\pi i \nu |\mathbf{R}_2 - \mathbf{r}_2|/c}}{|\mathbf{R}_2 - \mathbf{r}_2|} \, dS_2 \rangle \\
&= \langle \iint \mathcal{E}_\nu(\mathbf{R}_1) \mathcal{E}_\nu^*(\mathbf{R}_2) \frac{e^{2\pi i \nu |\mathbf{R}_1 - \mathbf{r}_1|/c}}{|\mathbf{R}_1 - \mathbf{r}_1|} \frac{e^{-2\pi i \nu |\mathbf{R}_2 - \mathbf{r}_2|/c}}{|\mathbf{R}_2 - \mathbf{r}_2|} \, dS_1 \, dS_2 \rangle
\end{aligned} \tag{A.4}
$$

A.2.4 Spatial Coherence Function

At this point, a fourth simplifying assumption is introduced; that radiation from astronomical objects are not spatially coherent. What this means is that the integral over the two celestial spheres, $\mathcal{E}_\nu(\mathbf{R}_1)$ and $\mathcal{E}_\nu(\mathbf{R}_2)$, can collapse into a single integral over S. We also note that if $\mathbf{R}_1 \neq \mathbf{R}_2$ then $\langle \mathcal{E}_\nu(\mathbf{R}_1) \mathcal{E}_\nu^*(\mathbf{R}_2) \rangle = 0$, which logically implies that if $\langle \mathcal{E}_\nu(\mathbf{R}_1) \mathcal{E}_\nu^*(\mathbf{R}_2) \rangle \neq 0$ then $\mathbf{R}_1 = \mathbf{R}_2$. We know that $\langle \mathcal{E}_\nu(\mathbf{R}_1) \mathcal{E}_\nu^*(\mathbf{R}_2) \rangle \neq 0$ as this would imply we receive no signal, so we now can say $\mathbf{R} = \mathbf{R}_1 = \mathbf{R}_2$. So:

Continuing from Eqn A.4:

$$
\begin{aligned}
V_\nu(\mathbf{r}_1, \mathbf{r}_2) &= \langle \iint \mathcal{E}_\nu(\mathbf{R}_1) \mathcal{E}_\nu^*(\mathbf{R}_2) \frac{e^{2\pi i \nu |\mathbf{R}_1 - \mathbf{r}_1|/c}}{|\mathbf{R}_1 - \mathbf{r}_1|} \frac{e^{-2\pi i \nu |\mathbf{R}_2 - \mathbf{r}_2|/c}}{|\mathbf{R}_2 - \mathbf{r}_2|} \, dS_1 \, dS_2 \rangle \\
&= \langle \int \mathcal{E}_\nu(\mathbf{R}) \mathcal{E}_\nu^*(\mathbf{R}) |\mathbf{R}^2| \frac{e^{2\pi i \nu |\mathbf{R}_1 - \mathbf{r}_1|/c}}{|\mathbf{R} - \mathbf{r}_1|} \frac{e^{-2\pi i \nu |\mathbf{R} - \mathbf{r}_2|/c}}{|\mathbf{R} - \mathbf{r}_2|} \, dS \rangle
\end{aligned} \tag{A.5}
$$

APPENDIX A. BASICS OF INTERFEROMETRY

We do some tidying up,

$$
\begin{aligned}
V_\nu(\mathbf{r}_1, \mathbf{r}_2) &= \langle \int \mathcal{E}_\nu(\mathbf{R})\mathcal{E}_\nu^*(\mathbf{R})|\mathbf{R}^2|\frac{e^{2\pi i\nu|\mathbf{R}-\mathbf{r}_1|/c}}{|\mathbf{R}-\mathbf{r}_1|}\frac{e^{-2\pi i\nu|\mathbf{R}-\mathbf{r}_2|/c}}{|\mathbf{R}-\mathbf{r}_2|}\, dS \rangle \\
&= \int \langle \mathcal{E}_\nu(\mathbf{R})\mathcal{E}_\nu^*(\mathbf{R})|\mathbf{R}^2|\frac{e^{2\pi i\nu|\mathbf{R}-\mathbf{r}_1|/c}}{|\mathbf{R}-\mathbf{r}_1|}\frac{e^{-2\pi i\nu|\mathbf{R}-\mathbf{r}_2|/c}}{|\mathbf{R}-\mathbf{r}_2|} \rangle\, dS \\
&= \int \langle \mathcal{E}_\nu(\mathbf{R})\mathcal{E}_\nu^*(\mathbf{R})\rangle|\mathbf{R}^2|\frac{e^{2\pi i\nu|\mathbf{R}-\mathbf{r}_1|/c}}{|\mathbf{R}-\mathbf{r}_1|}\frac{e^{-2\pi i\nu|\mathbf{R}-\mathbf{r}_2|/c}}{|\mathbf{R}-\mathbf{r}_2|}\, dS
\end{aligned} \tag{A.6}
$$

and we get

$$
V_\nu(\mathbf{r}_1, \mathbf{r}_2) = \int \langle |\mathcal{E}_\nu(\mathbf{R})|\rangle^2|\mathbf{R}^2|\frac{e^{2\pi i\nu|\mathbf{R}-\mathbf{r}_1|/c}}{|\mathbf{R}-\mathbf{r}_1|}\frac{e^{-2\pi i\nu|\mathbf{R}-\mathbf{r}_2|/c}}{|\mathbf{R}-\mathbf{r}_2|}\, dS \tag{A.7}
$$

We now write $\mathbf{s}$ for $\frac{\mathbf{R}}{|\mathbf{R}|}$ and the observed intensity, $I_\nu(\mathbf{s})$, for $\langle \mathcal{E}_\nu(\mathbf{s})\rangle^2|\mathbf{R}^2| = \langle \mathcal{E}_\nu(\mathbf{R})\rangle^2|\mathbf{R}^2|$. Using the second assumption (great distance to sources and celestial sphere) we can now formulate the equation in terms of a 2-dimensional angle [3], instead of a 3-dimensional position. The integral is now taken over the solid-angle space at a distance of $\mathbf{R}$ rather than over a sphere inside of normal 3-dimensional space. This means dS is replaced by $|\mathbf{R}^2|d\Omega$.

We again use the second assumption (massive distance to observed source) to neglect small terms in the order of $|\mathbf{r}/\mathbf{R}|$. $\mathbf{R}$ is in the order of light-years, whereas $\mathbf{r}$ is in the order of thousands of kilometres. Thus, we can say $|\mathbf{R}-\mathbf{r}| \approx |\mathbf{R}|$.

From Eqn A.7:

$$
\begin{aligned}
V_\nu(\mathbf{r}_1, \mathbf{r}_2) &= \int I_\nu(\mathbf{s})\frac{e^{2\pi i\nu|\mathbf{R}-\mathbf{r}_1|/c}}{|\mathbf{R}-\mathbf{r}_1|}\frac{e^{-2\pi i\nu|\mathbf{R}-\mathbf{r}_2|/c}}{|\mathbf{R}-\mathbf{r}_2|}\, dS \\
&\approx \int I_\nu(\mathbf{s})\frac{e^{2\pi i\nu|\mathbf{R}-\mathbf{r}_1|/c}}{|\mathbf{R}|}\frac{e^{-2\pi i\nu|\mathbf{R}-\mathbf{r}_2|/c}}{|\mathbf{R}|}\, dS \\
&= \int I_\nu(\mathbf{s})\frac{e^{-2\pi i\nu\mathbf{s}\cdot(\mathbf{r}_1-\mathbf{r}_2)/c}}{|\mathbf{R}|^2}\, dS \\
&= \int I_\nu(\mathbf{s})\frac{e^{-2\pi i\nu\mathbf{s}\cdot(\mathbf{r}_1-\mathbf{r}_2)/c}}{|\mathbf{R}|^2}|\mathbf{R}^2|d\Omega \\
&= \int I_\nu(\mathbf{s})e^{-2\pi i\nu\mathbf{s}\cdot(\mathbf{r}_1-\mathbf{r}_2)/c}\, d\Omega
\end{aligned}
$$

---

[3] Or solid-angle, measured in steradians. This is analogous to the 1-dimensional radian.

and we get

$$V_\nu(\mathbf{r}_1, \mathbf{r}_2) = \int I_\nu(\mathbf{s})e^{-2\pi i\nu\mathbf{s}\cdot(\mathbf{r}_1-\mathbf{r}_2)/c}\, d\Omega \qquad\qquad (A.8)$$

Eqn A.8 only depends on the difference of the vectors $(\mathbf{r}_1 - \mathbf{r}_2)$ and not their absolute positions. If we formulate the function $V_\nu(\mathbf{r}_1, \mathbf{r}_2)$ as this difference, say $V_\nu(\mathbf{r}_d)$ for $\mathbf{r}_d = \mathbf{r}_1 - \mathbf{r}_2$, then we call this function, the *spatial coherence function* of the field $E_\nu(\mathbf{r})$.

A.2.5 Fourier Inversion of the Spatial Coherence Function of the field $E_\nu(\mathbf{r})$

The fifth and final assumption is implemented in order to make *spacial coherence function* of the field $E_\nu(\mathbf{r})$ invertible by Fourier Transform. There are actually two alternative forms of this. Both project what is a spherical measurement onto a plane. One onto a plane centred at the observation location, the other centred on the edge of the celestial sphere. Some explanations given are not fully robust, for a full explanation refer to *Synthesis Imaging in Radio Astronomy* (Taylor et al., 1999). Namely Lectures 1, 2 and 19.

A.2.6 Spherical Measurements Projected onto a Plane

The first variation of the fith assumption is that we can to confine measurements to a plane centered on the observation location going through $\mathbf{r}_1$ and $\mathbf{r}_2$ [4]. This allows us to work on a 2D plane, rather than in 3D space. Since we may choose the plane, we choose the one perpendicular to $\mathbf{s}$ passing through the centre of the celestial sphere. We transform $\mathbf{r}_d$ into wavelength form, so that $\mathbf{r}_1 - \mathbf{r}_2 = (u, v, w \equiv 0)\lambda$, w is equivilant to 0 as it is perpendicular to $\mathbf{s}$. This coordinate system this yields $\mathbf{s}$ as $(l, m, \sqrt{1 - l^2 - m^2})$

We may now make further changes to Eqn A.8

$$V_\nu(\mathbf{r}_1, \mathbf{r}_2) = \int I_\nu(\mathbf{s})e^{-2\pi i\nu\mathbf{s}\cdot(\mathbf{r}_1-\mathbf{r}_2)/c}\, d\Omega$$

---

[4]If the plane through $\mathbf{r}_1$ and $\mathbf{r}_2$ is not perpendicular to $\mathbf{s}$, the original formulation of $\mathbf{E}_\nu(R)$ as $\mathbf{E}(R, t)$ can be adjusted by adding a time delay τ. With the correct delay, any $\mathbf{r}$ can falls on the plane perpendicular to $\mathbf{s}$.

Transforming to wavelength form and noting that $\lambda = c/\nu$

$$V_\nu(\lambda) = \int I_\nu(\mathbf{s}) e^{-2\pi i (l, m, \sqrt{1-l^2-m^2}) \cdot (u, v, 0)} \, d\Omega$$

Expanding λ to (u, v, w) and $\mathbf{s}$ to $(l, m, \sqrt{1-l^2-m^2})$

$$V_\nu(u, v, w \equiv 0) = \int I_\nu(l, m, \sqrt{1-l^2-m^2}) \frac{e^{-2\pi i (ul+vm)}}{\sqrt{1-l^2-m^2}} \, d\Omega$$

The integral is adjusted to reflect this, the w term is removed from V_ν as it will always be 0 (by the fourth assumption)

$$V_\nu(u, v) = \iint I_\nu(l, m, \sqrt{1-l^2-m^2}) \frac{e^{-2\pi i (ul+vm)}}{\sqrt{1-l^2-m^2}} \, dl \, dm$$

Since we are confining I_ν to a plane it makes no sense for it to have a third term, we can say $I_\nu(l, m, \sqrt{1-l^2-m^2}) = I'_\nu(l, m)$

$$V_\nu(u, v) = \iint I'_\nu(l, m) \frac{e^{-2\pi i (ul+vm)}}{\sqrt{1-l^2-m^2}} \, dl \, dm$$

We can now absorb the square-root into the intensity function to reformulate it as the modifidied intensity $I_\nu(l, m) = I'_\nu(l, m)/\sqrt{1-l^2-m^2}$. Thus,

$$V_\nu(u, v) = \iint I_\nu(l, m) e^{-2\pi i (ul+vm)} \, dl \, dm \tag{A.9}$$

A.2.7 Observed Sources Contained in a Small Region of the Sky

We now take a different route and assume that we are receiving electromagnetic radiation from a sufficiently small section of the sky. This means we are only receiving around a small arc σ. We ignore everything outside of $\mathbf{s} = \mathbf{s}_0 + \sigma$. Since $\mathbf{s}$ and $\mathbf{s}_0$ are unit length vectors, we can say

$$
\begin{aligned}
1 = |\mathbf{s}| = \mathbf{s} \cdot \mathbf{s} &= \mathbf{s}_0 \cdot \mathbf{s}_0 \\
&= (\mathbf{s}_0 + \sigma) \cdot (\mathbf{s}_0 + \sigma) \\
&= \mathbf{s}_0 \cdot \mathbf{s}_0 + 2\mathbf{s}_0 \cdot \sigma + \sigma \cdot \sigma \\
&\approx 1 + 2\mathbf{s}_0 \cdot \sigma
\end{aligned}
$$

Since $2\mathbf{s}_0 \cdot \sigma \approx 0$, we can say $\mathbf{s}_0$ and σ are perpendicular within a negligible range. We now use a coordinate system whereby $\mathbf{s}_0 = (0, 0, 1)$. This implies $\mathbf{r}_1 - \mathbf{r}_2 = (u, v, w)c/\nu$ and that $\mathbf{s} = (l, m, 1)$.

Thus we take Eqn A.8

$$V_\nu(\mathbf{r}_1, \mathbf{r}_2) = \int I_\nu(\mathbf{s})e^{-2\pi i \nu \mathbf{s} \cdot (\mathbf{r}_1 - \mathbf{r}_2)/c}\, d\Omega$$

as $\mathbf{s} \cdot (\mathbf{r}_1 - \mathbf{r}_2) = (l, m, 1) \cdot (u, v, w)c/\nu = (ul + vm + w)c/\nu$, we get

$$V_\nu'(u, v, w) = \int I_\nu(l, m, 1)e^{-2\pi i(ul + vm + w)}\, d\Omega$$

$$= \int I_\nu(l, m)e^{-2\pi i(ul + vm)}e^{-2\pi i w}\, d\Omega$$

$$= e^{-2\pi i w} \int I_\nu(l, m)e^{-2\pi i(ul + vm)}\, d\Omega$$

We adjust the integral to reflect the new coordinate system

$$V_\nu'(u, v, w) = e^{-2\pi i w} \iint I_\nu(l, m)e^{-2\pi i(ul + vm)}\, dl\, dm$$

We now define V_ν to absorb the floating exponential, $V_\nu(u, v, w) = V_\nu'(u, v, w)e^{-2\pi i w}$. As w is independent we can write $V_\nu(u, v, w)$ as $V_\nu(u, v)$ and we now get

$$V_\nu(u, v) = \iint I_\nu(l, m)e^{-2\pi i(ul + vm)}\, dl\, dm \tag{A.10}$$

A.2.8 The Fourier Transform

With either assumption, we get a function with which we can take the Fourier transform. Turning

$$V_\nu(u, v) = \iint I_\nu(l, m)e^{-2\pi i(ul + vm)}\, dl\, dm \tag{A.11}$$

into

$$I_\nu(l, m) = \iint V_\nu(u, v)e^{2\pi i(ul + vm)}\, du\, dv \tag{A.12}$$

In Eqn A.3 we defined V_ν, the correlation of the field. V_ν is the data that a two element radio telescope array outputs. I_ν is the observed intensity, or *intensity distribution*. So we have succeeded in formulating our desired output in terms of the input available to us.

There are volumes of maths behind interferometry including how to adjust for noise, atmospheric conditions and some of the assumptions we made above. The maths given here is sufficient for the purposes of this thesis.

A.3   Real World Adjustments and Consideration

A.3.1   Local Adjustments for Angle of Declination

To observe sources that are not a zenith, one needs to angle the telescopes appropriately. This has two effects, firstly it introduces a time delay as one of the telescopes receives the signal slightly later than the other. Secondly, the original baseline B is also shortened as the projected distance between telescopes is shortened (see Figure A.1).
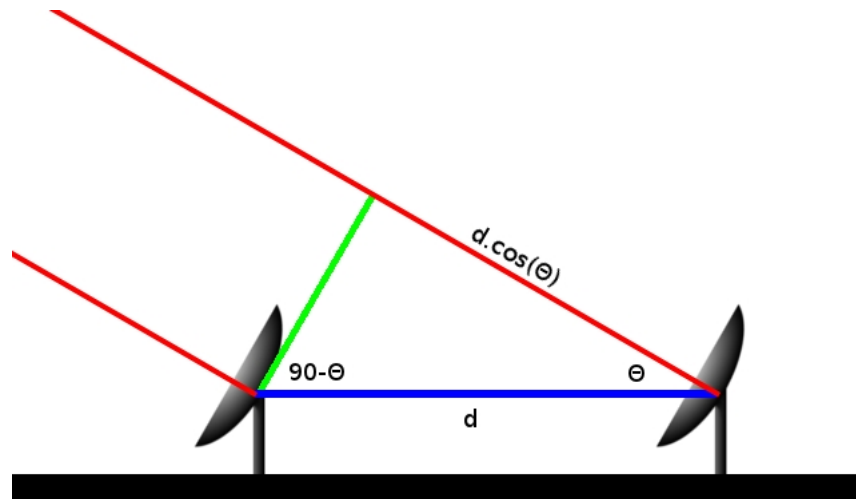

Figure A.1: Affect of declination on time delay in receiving signal
Two dishes d meters away from each other. When not pointed towards zenith (straight up), the signal from the source (red) must travel an extra $d\cos(\theta)$ meters for the further dish, this results in a $c \cdot d\cos(\theta)$ seconds (where c is the speed of light) delay that the further dish must account for.

A.3.2 Intercontinental Adjustments

As can be seen by in Figure A.2, similar considerations are needed when the baseline spans continents. The maths becomes slightly more complicated to account for curvature of the earth but the same considerations as the local case apply. An additional consideration for intercontinental distances used in VBLI is that there might be angles at which the earth obscures the signal from one of the telescopes, reducing accuracy and sometime making VBLI impossible.
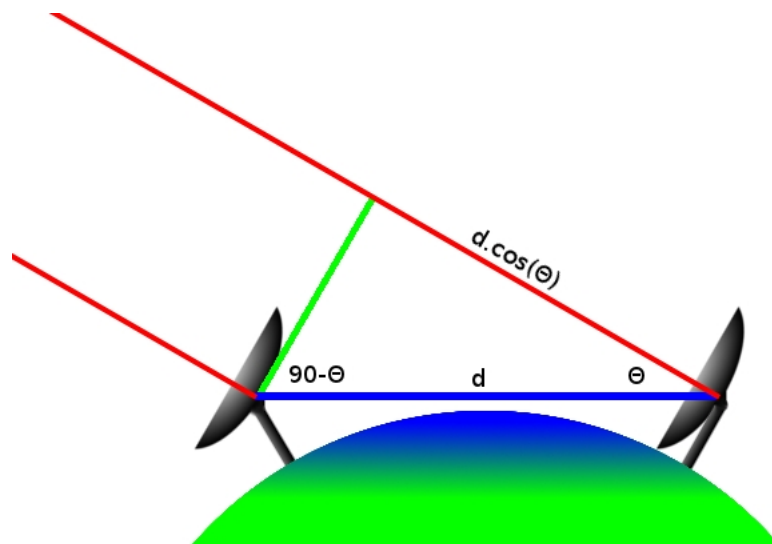
Figure A.2: Affect of earth's curvature on time delay in receiving signal
For very long distances between dishes, the curvature of the earth affects the angle at which the dishes are pointed. The rightmost dish is pointed almost at the horizon, whereat the leftmost dish is only at a slight angle. Otherwise the mathematics remains the same as is shown in figure A.1

A.4 The UV-Plane: Larger Arrays and the Rotation of the Earth

In the literature the sampled Fourier plane, V_ν, is called the *UV-plane*. In reality, measurements are subject to an additional noise component due to thermal noise in the receiver or in the sky; this manifests itself as Gaussian noise in the real and imaginary part of the measures visibility. We denote this effect with V_ν'. A two element array taking one reading

will give us a single complex-number at u_1, v_1 on the UV-plane. In other-words we have a value for $V_\nu(u_1, v_1)$. If we were to visualise the distribution of V_ν' it would look like in Figure A.3(d).

This is only a single uv-sample, real world interferometers have thousands and even hundreds of thousands of samples of the UV-plane. The following sections explain how interferometers create more samples, and just as importantly, how to cover the uv-plane as widely and completely as possible. Using more than two elements means that more points are sampled ($\frac{N(N+1)}{2}$ samples for an N element interferometer) and sampling over time means that $\frac{N(N+1)}{2}$ samples can be collected every second. How the elements are arranged and what direction the elements are pointed has a significant effect on the final uv-plane *coverage.*

Certain characteristics of the *uv-coverage* results in certain characteristics of the resultant dirty image when transformed. The further out that the uv-plane is covered, the better small, high resolution sources can be detected. Covering the uv-plane closer to the center reveals more diffuse large sources. A combination of wide and close samples are needed to detect both the compact and diffuse sources. Covering the uv-plane more uniformly means that the produced dirty image of the sky will more accurately represent the true intensity of sky. The more dense the coverage, the more sensitive the dirty image.

Please note that all the UV-plane images in this section are basic simulations and real world UV-planes take slightly different forms. The u-v plane is measured in wavelengths λ, more commonly kilo-wavelengths. For these contrived examples we assume the units in the figures are in meters. The UV-plane is also given in meters, but is usually converted to the appropriate wavelength scale. So for instance, for an observed signal with $\lambda = 4$m, 100m converts to 25 wavelengths. For $\lambda = 20$cm $= 0.2$m, 100m converts to 500 wavelength.

In Figure A.3(d) you can see there are two points, but we only have one sample. This is because we may use V_ν' formulated with $\mathbf{r}_1 - \mathbf{r}_2$ as well as $\mathbf{r}_2 - \mathbf{r}_1$. We can say $V_\nu' = V_{\nu 1,2} \cup V_{\nu 2,1}$ where $V_{\nu i,j}$ is the UV-plane formulated by $\mathbf{r}_i - \mathbf{r}_j$.
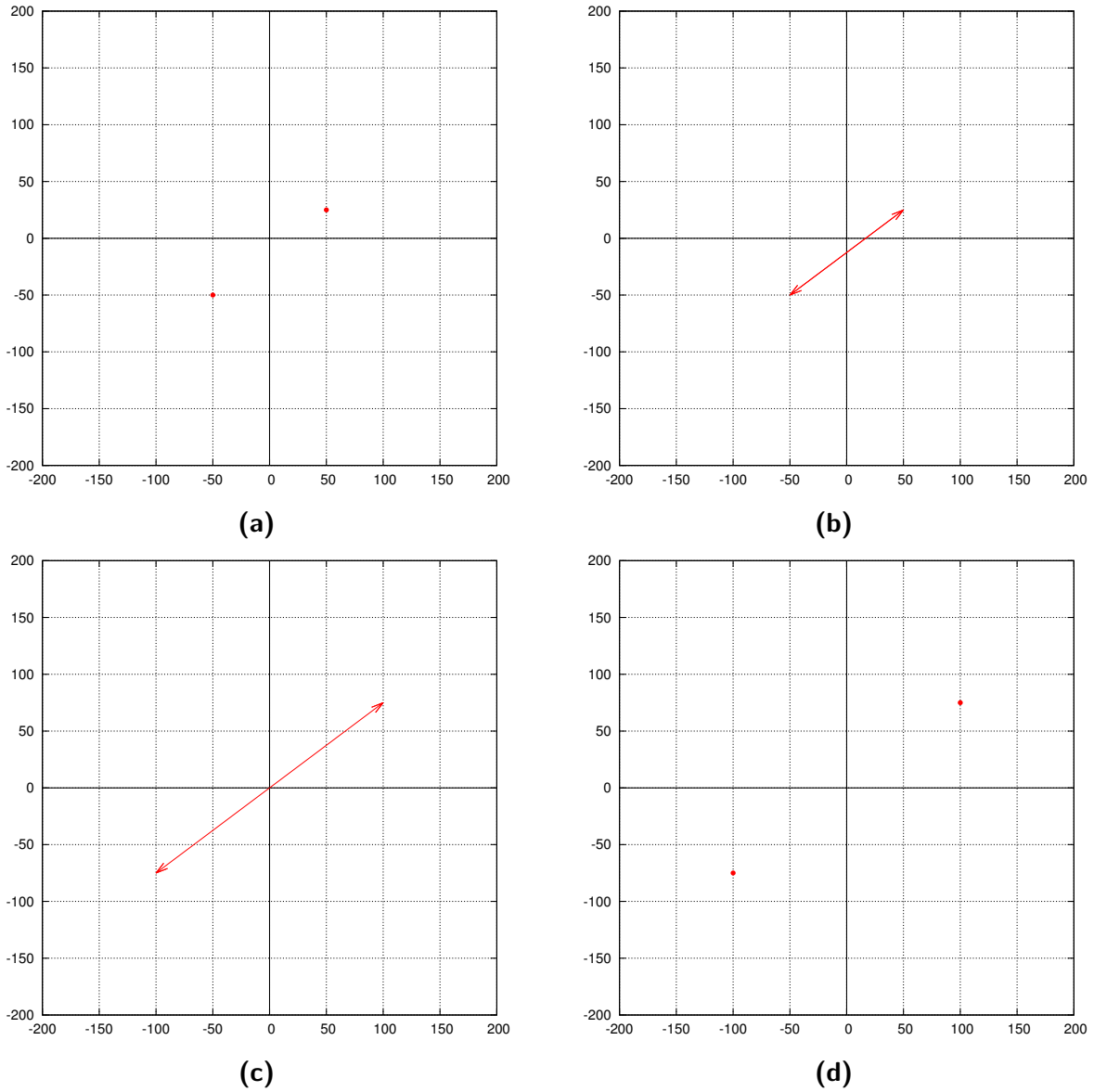
Figure A.3: A UV-plane of a 2 element interferometer

(a) Simulated locations of two antennas, r_1 and r_2, in real world space. (b) Two overlapping vectors, one from r_1 to r_2 and r_2 to r_1. (c) The vectors $r_2 - r_1$ and $r_2 - r_1$. (d) The resultant UV-plane distribution with a sample at $r_2 - r_1$ and by symetry, $r_1 - r_2$

A.4.1 More-Than-Two Element Arrays

We extend the definition of V'_ν to accommodate more receivers. We now say for N receivers that

$$V'_\nu = \bigcup_{i=1}^{N} \bigcup_{j=1}^{N} V_{\nu i,j} \tag{A.13}$$

If it is the case that $i = j$, then $V_{\nu i,j} = V_{\nu i,i} = 0$

As seen in Figure A.4, with the 3 element array, the effect on the UV-plane is immediately apparent. So we have a sample point for each pair of receivers. This results in $\frac{N(N-1)}{2}$ samples for an array sized N. This fortunate property implies that the number of sample points grows at a quadratically at the number of receivers grow. For a 25 element array we have $\frac{25 \times 26}{2} = 325$ samples, for 50 telescopes, 1275 samples. For 100 elements the number of samples rises to 5050, for 300, it is 45150 samples.

Even larger arrays produce a far better distribution, as seen in the 15 element array in Figure A.7.

## A.4.2 Covering the UV-Plane Using the Rotation of the Earth

Samples taken by a radio interferometer are usually over a time period of 1 to 10 seconds. So over an 8 hour period, about 3000 to 30,000 samples sets can be taken. Naively one might think that an 8 hour sample would yield the same results as we would be sampling the same point on the UV-plane 3000 times. The usefulness of taking all these samples over a long time period becomes apparent when considering how the earth's rotation changes the relative location of the interferometer elements. This causes a slight change in angle to the source, which causes the UV-plane to be sampled at a slightly different location each time (Fig. A.6). Thus, we gain far greater UV-plane coverage (Fig. A.7).

One can intuitively visualise that a giant theoretical radio dish the shape and size of the UV-plane would be equivalent to an entire array. Thus a wider diameter of this dish (a wider baseline) would result in a higher resolution. Better coverage in the middle of this theoretical dish means that there is more "collecting area" and thus more electromagnetic radiation can be received.
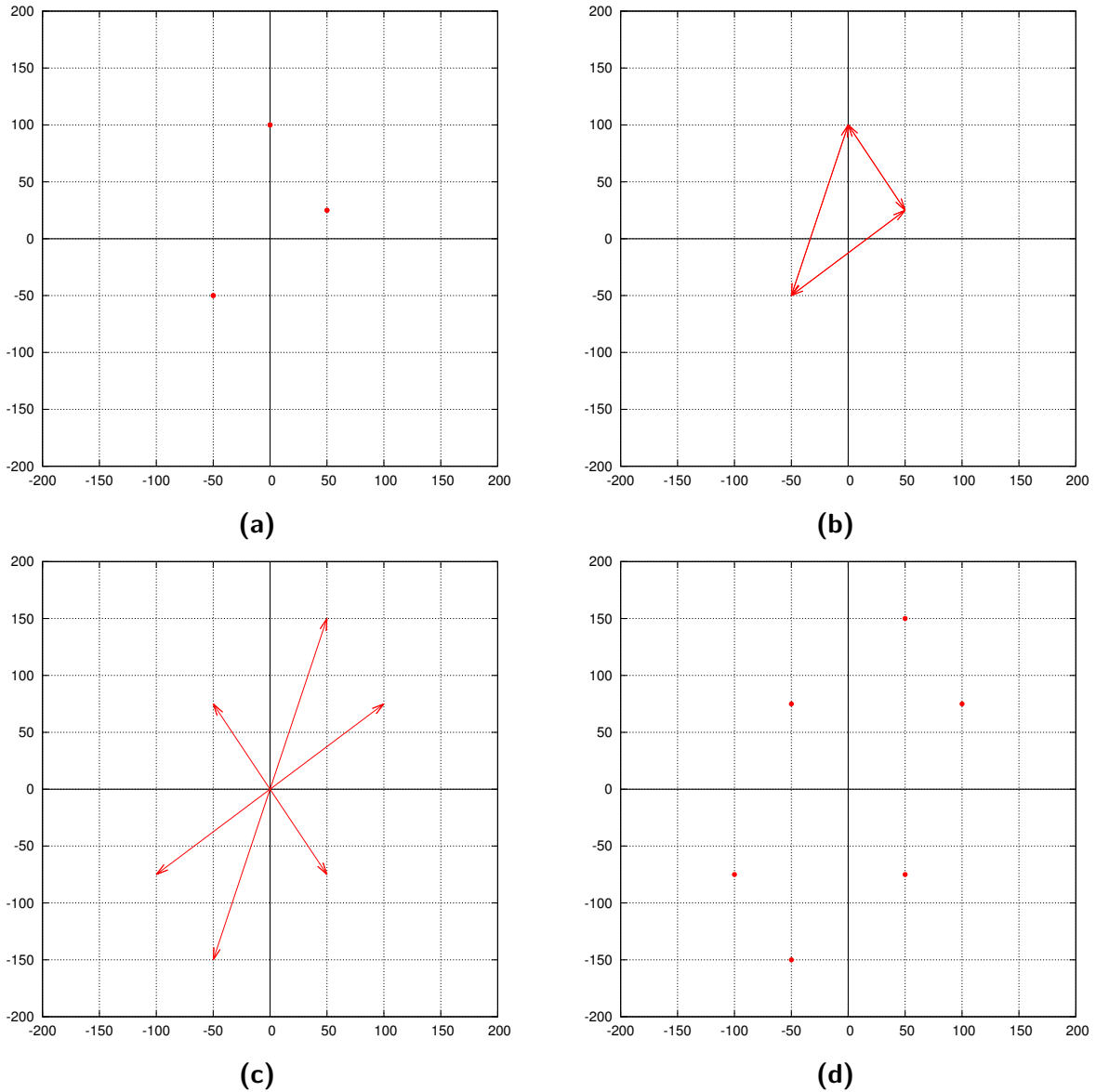
Figure A.4: A UV-plane of a 3 element interferometer
(a) Simulated locations of antennas in real world space. (b) Vectors from each array to each other array. (c) The same as (b), but centred at **0**. *(d) The resultant UV-plane distribution with a sample for each antenna pair, with symmetry.*
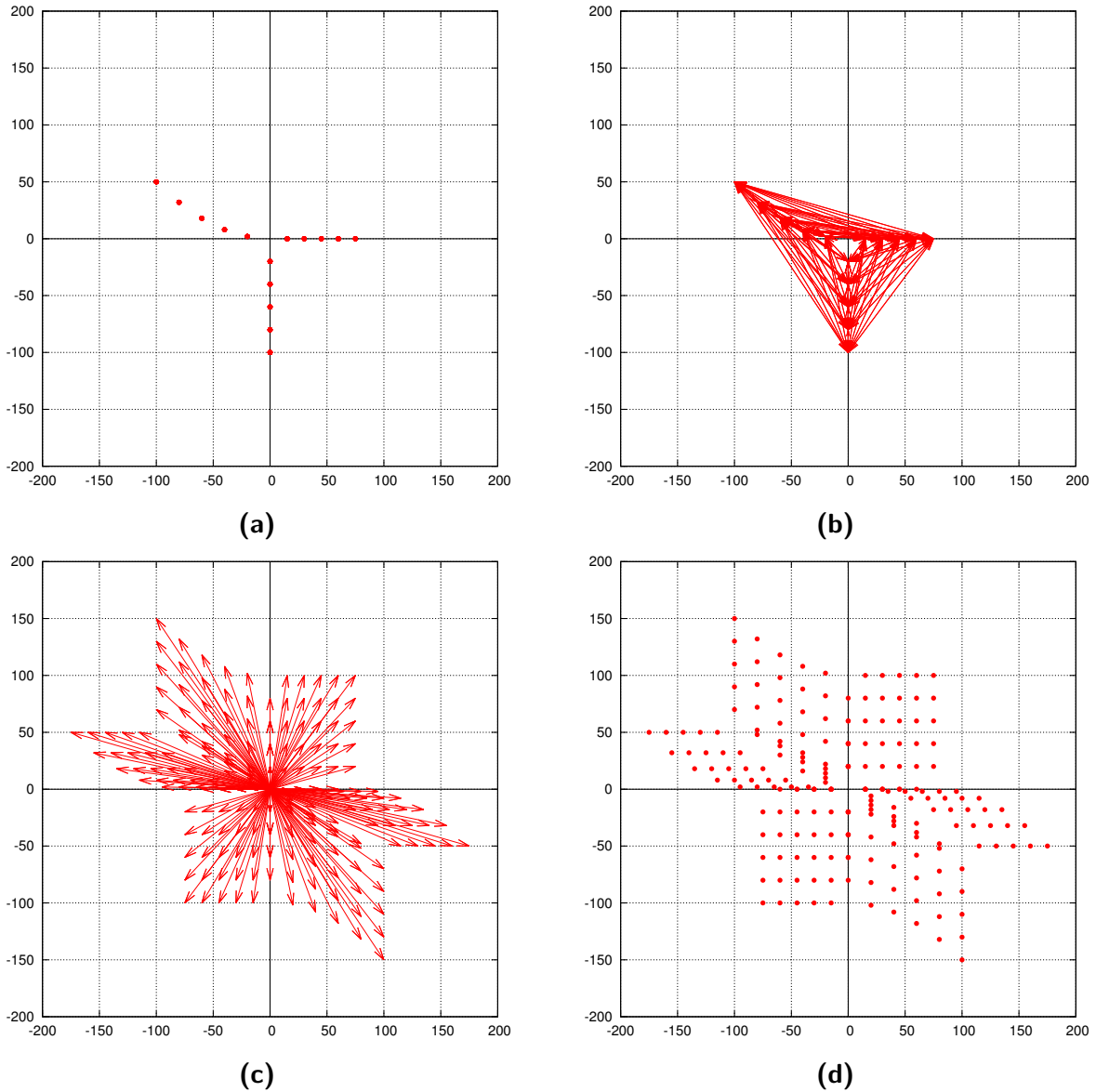
Figure A.5: A UV-plane of a 15 element interferometer
(a) Simulated locations of antennas in real world space. (b) Vectors from each array to each other array. (c) The same as (b), but centred at $\mathbf{0}$. (d) The resultant UV-plane distribution with a sample for each antenna pair, with symmetry.
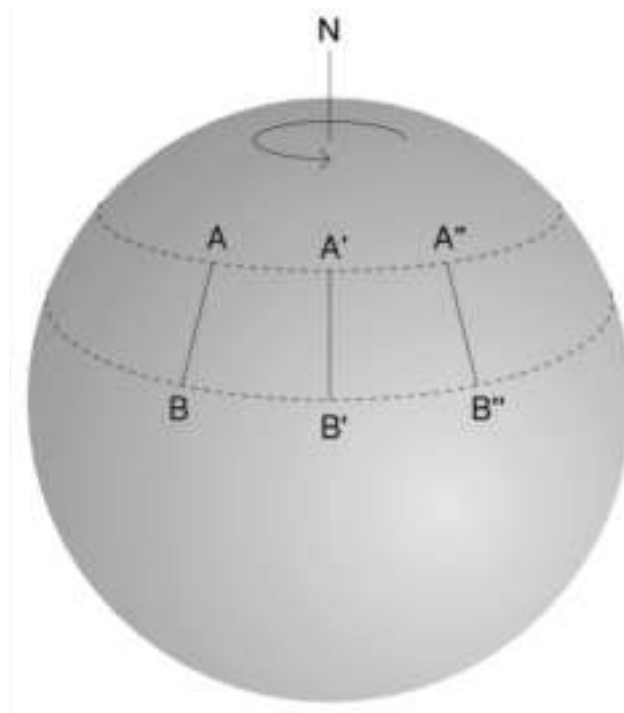
Figure A.6: A simple example of the effect of the earth's rotation
As the earth rotates, the relative vector between antenna A and B change.

As mentioned above, samples are usually taken in a 1 to 10 second interval. Any shorter and the noise-to-signal ratio will increase to beyond acceptable levels [5], as longer sampling time amortises internal noise by increasing the amount of signal recieved (observed noise is not amortised in this way). Too long will result in fewer sample points and hence less UV-coverage. Engineers are constantly striving to reduce noise in observations and thus shorten the sampling time. This results in more samples and hence greater computation time. To be conservative, say we have a 25 element array taking 10 second samples, this results in about one million samples. Future planned arrays like the SKA Africa will have 3,000 elements (Dewdney et al., 2011). With a 1 second sample time over 8 hours this results in 129,643,200,000 (one hundred billion) samples!

---

[5]Though shorter and shorter intervals are being achieved with new technology

A.4.3   Declination of Observed Signal

The angle of declination of the source has an important effect to the u-v coverage. Although the way in which the radio receivers are distributed relative to each other has a direct effect on the UV-plane's distribution, the direction the array is facing will also have an effect. The lower the declination of the source being tracked, the more squashed the distribution will be. As is demonstrated by in Figure A.8. The reason for the squashed distribution is clear from Figure A.2 whereby the angle at which the telescopes are pointed causes baseline B to shorten to the projected baseline $B \cos \theta$ where θ is the angle from zenith.

A.5   Other Considerations

A.5.1   Hermitian Nature of the Correlation of the Field:

In $V_{\nu i,j}$ the signal from the jth radio dish is conjugated, and since in $V_{\nu j,i}$ the signal from the ith radio dish is conjugated it can be shown that $V_{\nu j,i}(u,v) = \overline{V_{\nu i,j}(u,v)}$. This makes the real part of V'_ν even since $Re(V_{\nu j,i}) = Re(V_{\nu i,j})$ and its imaginary part is odd since $Im(V_{\nu j,i}) = Im(-V_{\nu i,j})$. V'_ν and is therefor a Hermitian function. As a direct result, we can say that I_ν is a real-valued function. This point has some minor implications later on but also confirms that I_ν is a physical quantity.

A.5.2   The Sampling Function

To simplify the maths, rather than creating a modified version of V'_ν that represents the discretely sampled nature of the UV-plane, we introduce a sampling function $S(u,v)$. $S(u,v)$ is equal to zero everywhere, except on values of u,v where the UV-plane is sampled. S is such that the integral over it is equal to 1. We modify Eqn A.12 and introduce I_ν^D, the observed intensity affected by discrete sampling, commonly called the *dirty image.*

$$I_\nu^D(l,m) = \iint V'_\nu(u,v)S(u,v)e^{2\pi i(ul+vm)}\,du\,dv \qquad (A.14)$$

Due to the nature of S, we may formulate it as a type of *Dirac-Delta* function. A Dirac Delta function or distribution satisfies these two properties:

$$\delta(x) = \begin{cases} +\infty, & x = 0 \\ 0, & x \neq 0 \end{cases} \tag{A.15a}$$

$$\int\limits_{-\infty}^{\infty} \delta(x)\, dx = 1 \tag{A.15b}$$

Similarly, a 2D version can be defined

$$\delta(x, y) = \begin{cases} +\infty, & x = 0 \text{ and } y = 0 \\ 0, & x \neq 0 \text{ or } y \neq 0 \end{cases} \tag{A.16a}$$

$$\int\limits_{-\infty}^{\infty} \int\limits_{-\infty}^{\infty} \delta(x, y)\, dx\, dy = 1 \tag{A.16b}$$

To more formerly define S, we use an (offset) 2D Dirac-Delta function, δ, to describe the sampling of the UV-plane:

$$S(u, v) = \sum_{k=1}^{M} \delta(u - u_k, v - v_k) \tag{A.17}$$

Thus we have defined $S(u, v)$ that mathematically picks point samples from the Fourier plane. We use this sampling function in the next section to define the computational methods for calculating the image plane.
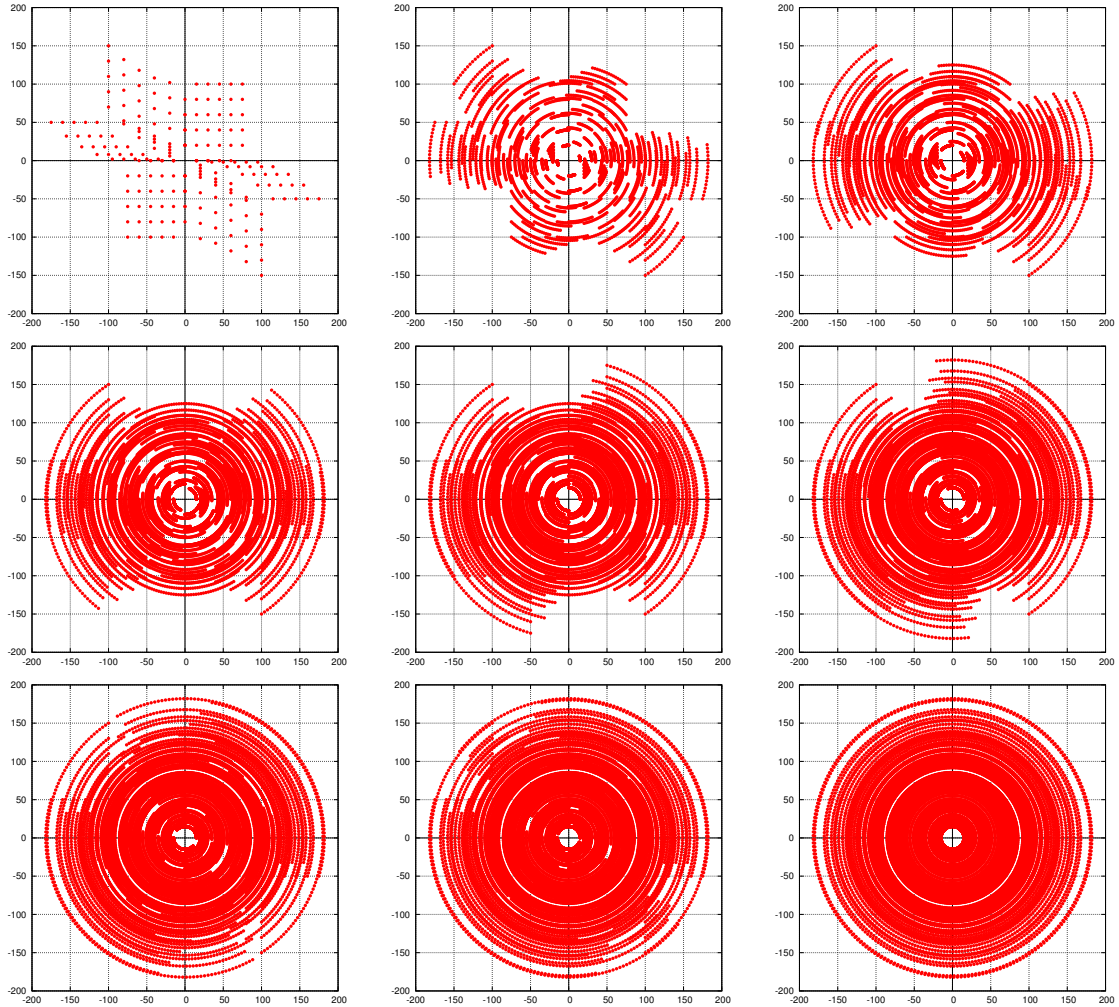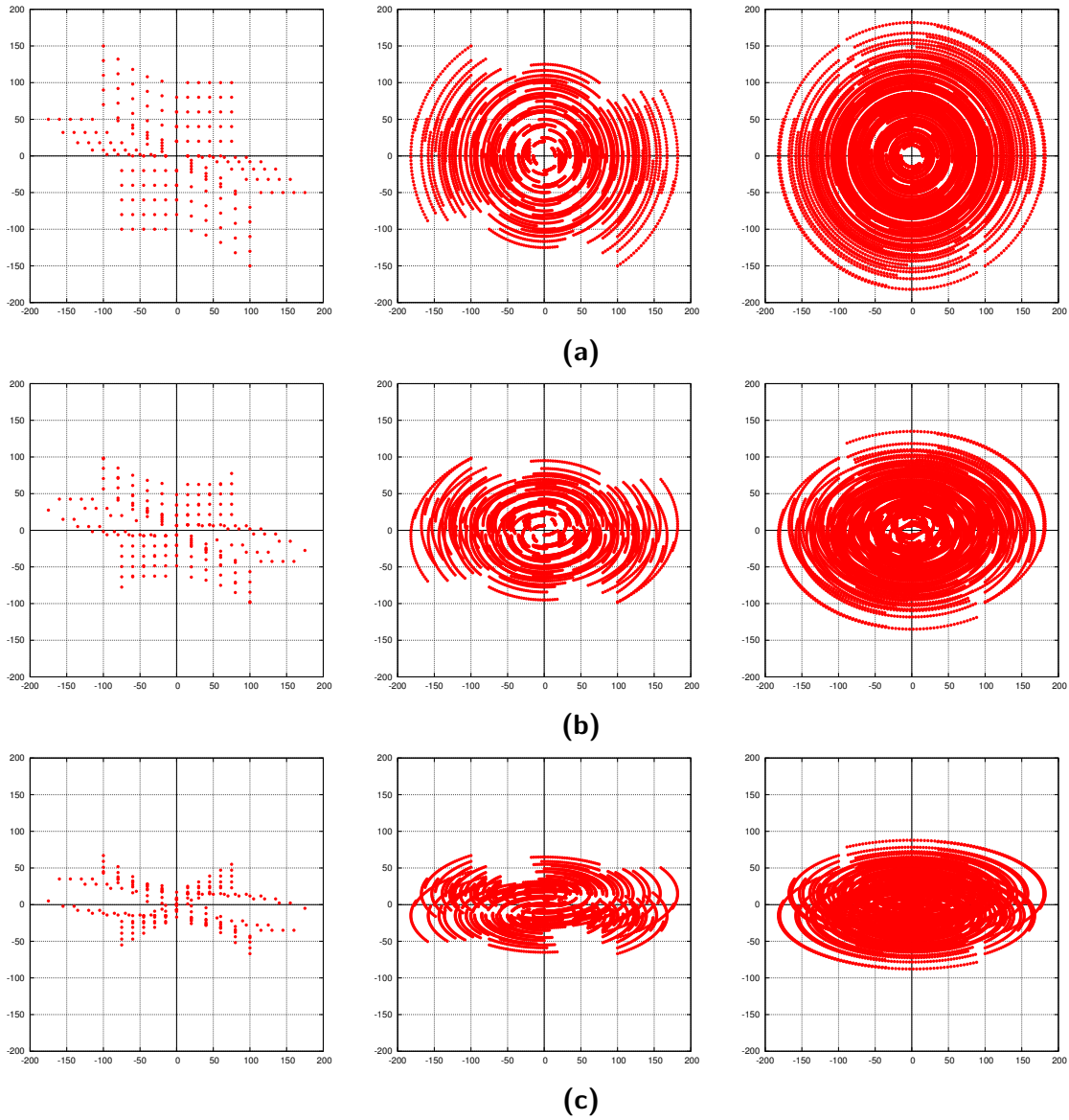
Figure A.7: The UV-distribution over time
As the earth rotates, samples can be taken at slightly different angles each time. In this simulated set the samples are quite far spaced. The distance between samples are usually far smaller in real life.

Figure A.8: A demonstration of the UV-plane at different declinations
(a) Shows the UV-plane over some time at zennith at various times. (b) and (c) Is the same distribution run for the same time but at a lower declination.

Appendix B

Point Source Visibilities

In order to simulate the visibilities an interferometer would produce, one needs to understand the relationship between the sky-intensity map (the brightness of the sky) and the signal received by the observing elements. Unlike single dish telescopes, interferometers do not have a straightforward relation and require correlation, Fourier transform, phase shifts, propogation effects and other factors. This complex relationship for interferometers is known as a *Measurement Equation*

Smirnov's paper series on the *Radio Interferometry Measurement Equation* (RIME) reformulates the classic radio interferometry visibility equation into a more robust and general equation based on *Jones matrices.*(Smirnov, 2011a,b,c,d) An overview of the equation is given in the remainder of this chapter.

Interferometers essentially work in pairs, each pair at each point in time produces one *visibility.* A visibility represents a single sampled point on the Fourier plane of the sky-intensity map. The relative position of the two antennae dictate where on the Fourier plane the sample will fall. By cross correlating the input voltages of the two antennae, the value of the sample is obtained.

How many samples we obtain depends on how long we sample the sky for and how many telescope pairs we have. A more robust explanation is given in Appendix A.4

We will first define the output of the interferometers in terms of its voltages. We then assume there is single point in the sky that we are observing and derive the measurement equation for just that. We then extend the formulation to multiple sources, multiple time-slots and

multiple frequencies. We finally explain and account for time/bandwidth smearing.

B.1   Formulation of the Interferometer Equipment

Interferometric observations are obtained by measuring the interference patterns between two receiving elements (*antennae*), p and q, each of which outputs a voltage ν_p and ν_q. The voltage is proportional to the amplitude of electromagnetic radiation received. These measurements are taken and averaged in the corrolator over a short time interval, typically 1-10 seconds. These time-averaged signals are respectively called $\langle \nu_p \rangle$ and $\langle \nu_q \rangle$. Various frequency bands are measured simultaneously.

In reality the signals are usually recieved using two receptors (either up/left or left/right circular) and each antenna will therefore have two voltage outputs $\boldsymbol{\nu} = (\nu_l, \nu_r)$. When two outputs are correlated, it produces four outputs, $\langle \nu_{pl} \nu_{ql}^* \rangle, \langle \nu_{pl} \nu_{qr}^* \rangle, \langle \nu_{pr} \nu_{ql}^* \rangle, \langle \nu_{pr} \nu_{qr}^* \rangle$, better represented in matrix form:

$$V_{pq} = 2 \begin{pmatrix} \langle \nu_{pl} \nu_{ql}^* \rangle \langle \nu_{pl} \nu_{qr}^* \rangle \\ \langle \nu_{pr} \nu_{ql}^* \rangle \langle \nu_{pr} \nu_{qr}^* \rangle \end{pmatrix} = 2 \left\langle \begin{pmatrix} \nu_{pl} \\ \nu_{pr} \end{pmatrix} (\nu_{ql}^* \nu_{qr}^*) \right\rangle = 2 \langle \boldsymbol{\nu}_p \boldsymbol{\nu}_q^H \rangle \tag{B.1}$$

where $*$ denotes the complex conjugate. [1]

B.2   A single uncorrupted point source

We assume the existence of a quasi-monochromatic signal, a single point in the sky fixed in time and space. We define some orthonormal coordinate system $x - y - z$ where the z axis runs along the path of *propagation*, i.e. from source to antenna. Call this source

$$\boldsymbol{e} = \begin{pmatrix} e_x \\ e_y \end{pmatrix} \tag{B.2}$$

where $\boldsymbol{e}$ is a complex vector. Assuming that the signal path is linear, one can represent a linear transform as a 2×2 matrix, $\boldsymbol{J}$ (a *Jones* matrix (Jones, 1941)). With this, one can take

---

[1] The factor of 2 adjustment is not important for this work's purposes, but is kept as a matter of referential accuracy.

into account multiple (linear) effects along the signal path with multiple Jones matrices:

$$\boldsymbol{e}' = \boldsymbol{J}_n\boldsymbol{J}_{n-1}...\boldsymbol{J}_1\boldsymbol{e} = \boldsymbol{J}\boldsymbol{e} \tag{B.3}$$

We now refer back to Eqn (B.1), and assume the voltage received by the antenna is the original signal $\boldsymbol{e}$ affected by a number of linear transformations, $\boldsymbol{J}$:

$$\boldsymbol{\nu} = \begin{pmatrix} \nu_l \\ \nu_r \end{pmatrix} = \boldsymbol{J}\boldsymbol{e} \tag{B.4}$$

Since p and q are not in the same location, the signal travels along different paths to get them. We use $\boldsymbol{J}_p$ and $\boldsymbol{J}_q$ to represent the separate linear transformations for the separate paths the signal travels down to antennae p and q, we expand Eqn (B.1) using Eqns (B.2) and (B.4) and obtain:

$$\begin{aligned} V_{pq} &= 2\boldsymbol{J}_p \begin{pmatrix} \langle e_x e_x^*\rangle\langle e_x e_y^*\rangle \\ \langle e_y e_x^*\rangle\langle e_y e_y^*\rangle \end{pmatrix} \boldsymbol{J}_q \\ &= \boldsymbol{J}_p B \boldsymbol{J}_q^H \end{aligned} \tag{B.5}$$

where $B = \begin{pmatrix} \langle e_x e_x^*\rangle\langle e_x e_y^*\rangle \\ \langle e_y e_x^*\rangle\langle e_y e_y^*\rangle \end{pmatrix}$ is called the *brightness matrix*.

If the antennae p and q were receive the same signal at precisly the same time, the visibility equation would simply be $V_{pq} = B$. However, because of the declination of the signal, there is always a delay and hence a *phase-delay* is introduced (see Appendix A.3.1), as follows. For antennae p, q at locations $\boldsymbol{u}_p = (u_p, v_p, w_p)$ and $\boldsymbol{u}_q = (u_q, v_q, w_q)$ respectively and point source propagating in direction $\boldsymbol{\sigma} = (l, m, n = \sqrt{1 - l^2 - m^2})$, the phase difference is defined as:

$$\kappa_p = 2\pi i\lambda^{-1}(u_p l + v_p m + w_p(n - 1)) \tag{B.6}$$

where λ is the wavelength of the signal. This phase-shift effect can be defined as a 2×2 matrix:

$$K_p = \begin{pmatrix} e^{-ik_p} & 0 \\ 0 & e^{-ik_p} \end{pmatrix} = e^{-ik_p}I_2,$$

where I_2 is the 2×2 identity matrix. [2]

---

[2]When K_p is multiplied by another matrix, it is equivalent to a scalar multiplication by e^{-ik_p}.

The phase-shifted RIME then becomes

$$V_{pq} = K_p B K_q^H \tag{B.7}$$

When this is expanded, we obtain a more canonical version of the visibility function (Thompson et al., 2001):

$$
\begin{aligned}
V_{pq} &= e^{-ik_p} B e^{ik_q} = B e^{-ik_p + ik_q} \\
&= B e^{-2\pi i \lambda^{-1}(u_{pq}l + v_{pq}m + w_{pq}(n-1))}
\end{aligned}
\tag{B.8}
$$

where λ is the wavelength and $\boldsymbol{u}_{pq} = \boldsymbol{u}_p - \boldsymbol{u}_q$.

B.2.1 A single corrupted point source

In the real world, signals are corrupted by a variety of effects. Whilst important, this consideration was not in the scope of this work and is left for future work. In cases where there are multiple corrupting effects the Jones matrix might look something like:

$$\boldsymbol{J}_{sp} = \boldsymbol{G}_p \boldsymbol{E}_{sp} K_{sp}$$

and the full RIME would look as such (Smirnov, 2011a):

$$V_{pq} = \boldsymbol{J}_p B \boldsymbol{J}_q^H = \boldsymbol{G}_p \boldsymbol{E}_p K_{sp} B K_q^H \boldsymbol{E}_q^H \boldsymbol{G}_q^H$$

where K, $\boldsymbol{E}$ and $\boldsymbol{G}$ are 2×2 Jones matrices representing a linear transform. Full details can be found in Smirnov's paper series on RIME (Smirnov, 2011a).

B.3 Multiple sources, times, and frequencies

This basic single uncorrupted point source RIME (Equation B.7) can be extended to incorporate multiple time, t, and frequency, ν, bands, as:

$$V_{pq}(t, \nu) = K_p(t, \nu) B K_q^H(t, \nu) \tag{B.9}$$

where

$$K_p(t, \nu) = e^{-2\pi i \frac{\nu}{c}(\boldsymbol{u}_{tp} \cdot \boldsymbol{\sigma})}$$

and c is the speed of light. The term $\boldsymbol{u}_{tp}$, now represents the relative change in location of the antenna over time.

Simulation of multiple point sources in the sky is easily achieved as a sum of the visibility function over all N sources

$$V_{pq}(t, \nu) = \sum_s K_p(t, \nu) B K_q^H(t, \nu) \tag{B.10}$$

where

$$K_{sp}(t, \nu) = e^{-2\pi i \frac{\nu}{c}(\boldsymbol{u}_{tp} \cdot \boldsymbol{\sigma}_s)}$$

The term $\boldsymbol{\sigma}_s$ represents the direction to source s.

B.4   Time and bandwidth smearing

Visibilities are measured over an averaged time period, $[t_0, t_1]$, and frequency band, $[\nu_0, \nu_1]$. The RIME formulation in Equation B.10 calculates the visibilities as if $t = t_0$ and $\nu = \nu_0$ rather than using the range $[t_0, t_1] \times [\nu_0, \nu_1]$. This integration results in a loss of measured amplitude, which is known in interferometry as time and bandwidth smearing. This must be accounted for in the calculation in order to evaluate predicted visibilities accurately. Equation B.10 is reformulated as an integral (Smirnov, 2011d):

$$\begin{aligned}
\langle V'_{pq} \rangle &= \frac{1}{\Delta t \Delta \nu} \int_{t_0}^{t_1}\!\!\int_{\nu_0}^{\nu_1} V_{pq}(t, \nu) \, d\nu \, dt \\
&= \frac{1}{\Delta t \Delta \nu} \int_{t_0}^{t_1}\!\!\int_{\nu_0}^{\nu_1} K_{sp}(t, \nu) B_s K_{sq}^H(t, \nu) \, d\nu \, dt
\end{aligned} \tag{B.11}$$

If this integration is not used in the calculation the result is a loss of measured signal and is a common known problem in interferometry known as *time/bandwidth smearing*.[3] If it is assumed that the time and frequency change linearly over a bucket, the integral can be

---

[3]It is also known as *time/bandwidth decorrelation*. For phase delay calculations (K_p in RIME) this effect is referred to as *smearing*. For any other linear transform or effect it is more commonly referred to as *decorrelation*.

calculated with the following:

$$\int_0^{x_0} e^{ix}\, dx = \mathrm{sinc}\frac{x_0}{2} e^{i\frac{x_0}{2}}$$

Using this integral approximate, the visibility equation with *time/bandwidth smearing* for a single source becomes:

$$\langle V'_{pq}\rangle \simeq \mathrm{sinc}\frac{\Delta\mathbf{\Psi}}{2}\mathrm{sinc}\frac{\Delta\mathbf{\Phi}}{2}V_{pq}(t_m,\nu_m) \tag{B.12}$$

where

$$t_m = (t_0+t_1)/2, \nu_m = (\nu_0+\nu_1)/2,$$

$$\Delta\mathbf{\Psi} = \arg V_{pq}(t_1,\nu_m) - \arg V_{pq}(t_0,\nu_m),$$

$$\Delta\mathbf{\Phi} = \arg V_{pq}(t_m,\nu_1) - \arg V_{pq}(t_m,\nu_0),$$

arg denotes the complex argument or complex angle (Smirnov, 2011a; Thompson et al., 2001; Taylor et al., 1999), and sinc is defined as $\mathrm{sinc}(x) = \begin{cases} \frac{\sin x}{x} & \text{if } x \neq 0 \\ 1 & \text{if } x = 0 \end{cases}$

B.5   The Final Visibility Equation

The full visibility equation reads:

$$\begin{aligned}
V'_{pq}(t,\nu) &= \sum_s \mathrm{sinc}\frac{\Delta\mathbf{\Psi}(\nu_m)}{2}\mathrm{sinc}\frac{\Delta\mathbf{\Phi}(t_m)}{2}K_{sp}(t,\nu)B_s K_{sq}^H(t,\nu)\\
&= \sum_s \mathrm{sinc}\frac{\Delta\mathbf{\Psi}(\nu_m)}{2}\mathrm{sinc}\frac{\Delta\mathbf{\Phi}(t_m)}{2}B_s e^{-2\pi i\frac{\nu}{c}(\boldsymbol{u}_{tpq}\boldsymbol{\sigma}_s)}
\end{aligned} \tag{B.13}$$

In this way, the RIME equation is extended to incorporate multiple point sources, multiple time, frequency bands as well as account for time/frequency bandwidth smearing. Whilst Smirnov's formulations can also account for various interference effects and Direction-Dependent-Effects (something classical "pre-RIME" formulations struggled to incorporate), this work only implements the time/frequency smearing factor, one of the primary causes for loss of measured signal in RIME.

Equation (B.13) is the one that will be accelerated using GPGPU methods.