AN OPTIMIZED OPENGL CONVOLUTIONAL GRIDDING ALGORITHM FOR THE SQUARE KILOMETRE ARRAY

A THESIS SUBMITTED TO AUCKLAND UNIVERSITY OF TECHNOLOGY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF COMPUTER AND INFORMATION SCIENCES

Supervisors
Dr. Seth Hall
Dr. Andrew Ensor

November 2018

By
Adam Robert Campbell
School of Engineering, Computer and Mathematical Sciences

Abstract

Convolutional gridding is a technique used in synthesis imaging for the reconstruction of images from spatial-frequency data. *Gridding*, as it is referred to, has many applications, some of which include medical imaging (magnetic resonance imaging, computerized tomography), synthetic aperture sonar imaging, and radio interferometry. Gridding is a computationally expensive procedure, with modern gridding algorithms utilizing many-core accelerators to achieve high performance. The Square Kilometre Array (SKA) will eventually become the worlds largest radio interferometer ever constructed, generating enormous volumes of spatial-frequency data. Therefore, an optimized gridding algorithm is needed. Existing gridding solutions demonstrate the use of graphics processing units and heterogeneous computing libraries to achieve optimized gridding performance. However, these solutions are often dependant on additional data processing mechanisms to maximize gridding throughput. This thesis explores an implementation of the W-Projection convolutional gridding algorithm which utilizes Open Graphics Library and the graphics rendering pipeline; informally titled the Hall-Ensor-Campbell (HEC) gridder. It was hypothesized that efficient and effective gridding could be achieved and general performance improved by conducting gridding operations in a graphics rendering environment. A design science approach was employed to design, develop, and evaluate the efficacy of a graphics based gridding solution. Performance of the HEC gridder was comparatively evaluated against a leading convolutional gridding algorithm. It was found that efficient and effective graphics based gridding is feasible with the use of vertex point sprites, accumulative fragment blending, and a custom implementation of textured W-Projection kernels.

Contents

List of Tables

List of Figures

List of Code Snippets

Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a university or other institution of higher learning.

Signature of student

Acknowledgements

I would first like to sincerely thank my supervisory team; Dr. Seth Hall and Dr. Andrew Ensor. Their valuable expertise, personal guidance, and patience have made all the difference during the undertaking of this complex research topic. I would also like to thank them for the many litres of Running Horse coffee over the past year; the next 12 months are on me.

I would also like to thank my partner Nikita, her loving support has kept me sane during challenging phases of my research. Additionally, a thanks to my good friends Brendan and Luis for their encouraging words.

Last but not least, I would like to thank my parents; Sharon and John, and my grandparents; Bob and Dorothy. Their support has made it possible for me to excel in achieving a higher education.

Adam R. Campbell
November 30, 2018

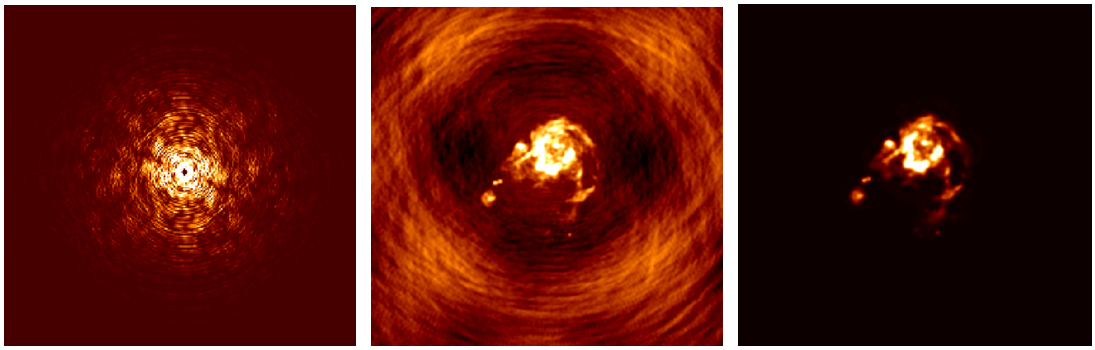
Chapter 1

Introduction

Synthesis imaging, also referred to as *aperture synthesis*, is an advanced imaging technique used for the production of images from arbitrarily sampled spatial-frequency data. Traditionally, an array of sensors is used to measure samples of electromagnetic radiation emitted by some observable source. These samples are progressively transformed into an image using a series of complex algorithms, in a process referred to as *imaging*.

One of the dominant algorithms used in the imaging process is the convolutional gridding algorithm, or *gridder*. The process of gridding observes the integration of spatial-frequency data samples into a two-dimensional regular Fourier domain grid. General smoothing functions are typically used to perform this integration, in the form of two-dimensional convolution kernels. Other algorithms in the imaging pipeline are responsible for the correction, enhancement, and cleaning of gridded data, to ensure high quality images are synthesized. Of the algorithms used in imaging, convolutional gridding is described as the most computationally expensive algorithm in the pipeline (Romein, 2012).

Gridding has since been adopted by a number of scientific disciplines due to its effectiveness in synthesizing high quality images. Radio astronomers utilize gridding in conjunction with radio interferometry, when studying astronomical bodies which lay beyond the realms of the visible spectrum of light. Figure 1.1 demonstrates the gridding of astronomical data (left), which is transformed using an inverse Fourier transform (mid), and is then cleaned to obtain the true image of the astronomical source (right).



(a) Gridded Samples **(b)** Dirty Image **(c)** True Image

Figure 1.1: Synthesis imaging in radio astronomy (Wise, 2013)

In the health sector, gridding is commonly used in magnetic resonance imaging (Schomberg & Timmer, 1995) and computerized tomography (Jackson et al., 1991) to synthesize image *slices* of the human form. These image slices allow specialists to accurately diagnose underlying disease or abnormalities, which would otherwise require invasive surgery. Synthetic aperture sonar systems utilize gridding to map and observe the ocean floor (McKay et al., 2017).

Recently, there has been an increase in research for optimized convolutional gridding algorithms. This stems from the active development of the Square Kilometre Array (SKA), which will eventually become the worlds largest radio telescope ever constructed. Featuring a total collection area of one square kilometre, it is anticipated that the SKA will produce tremendous volumes of observational data. Upwards of 720 gigabytes per

second is expected during the first phase of the telescope (10% of the total capacity). The imaging pipeline for the SKA is expected to be as optimized as possible to deal with these extreme volumes of incoming data, which must incorporate optimized convolutional gridding.

Hardware accelerators, such as the graphics processing unit (GPU) has become commonplace for parallelizing convolutional gridding algorithms to achieve improved performance (Humphreys & Cornwell, 2011). This has introduced new challenges to gridding, including effective concurrency management and maintaining high gridding throughput. Various algorithm designs and performance improvement techniques have ensued. Specialized threaded work-distribution techniques have been demonstrated to manage concurrency during gridding (Romein, 2012), which is improved with the coarsening of threads (Merry, 2016b). Gridding throughput can be improved through z-order tiling of the Fourier domain grid (Du Toit, 2017), in order to maximize grid locality during convolutional gridding (Veenboer et al., 2017).

However, the majority of parallelized convolutional gridding algorithms demonstrates the need for additional data processing in order to achieve optimized gridding performance. These data processing techniques include searching (Edgar et al., 2010), sorting (Du Toit, 2017), or compression/elimination (Muscat, 2014) of observational data. Evidently, this unnecessary computational overhead is undesired by the SKA, as it only contributes to the already extensive financial costs needed to produce, operate, and maintain the radio telescope.

The various algorithms presented in gridding literature typically utilize general-purpose computing libraries, such as Compute Unified Device Architecture (CUDA), Open Computing Language (OpenCL), Open Multi-Processing (OpenMP), to name a few. There appears to be an absence of gridding solutions which utilize traditional GPU based computing; i.e. graphics rendering. It is speculated that the graphics rendering pipeline has the necessary functionality to both simplify convolutional gridding, and

improve general gridding performance. Edgar et al. (2010) states that preliminary experimentation with graphics based gridding using Open Graphics Library (OpenGL) demonstrates feasibility, but was discontinued due to a lack of computer graphics experience by engineers (Edgar et al., 2010). Romein (2012) suggests that texture based convolution kernels would benefit from hardware accelerated interpolation offered by graphics rendering hardware. He also suggests textures could demonstrate improved image quality, and improved execution times for convolutional gridding when efficient texture caching is used (Romein, 2012).

The speculation surrounding graphics based convolutional gridding is of interest to demonstrate whether it provides a feasible gridding solution for the SKA. It would be ideal to maximize gridding performance where possible, and the rendering pipeline is thought to be a viable option. Furthermore, the pipeline provides a breadth of built-in functionality which may simplify the overall process of convolutional gridding.

1.1 Statement of the Problem

Current gridding solutions demonstrate a reliance on additional data processing to achieve optimized gridding performance. These processing mechanisms includes sorting, searching, compression, or elimination of observational data either before or during the gridding process. Such mechanisms contribute to the already expensive computational needs of convolutional gridding, and are undesired for the SKA.

The SKA requires an efficient and effective convolutional gridding algorithm. One which supports wide-field imaging, as the SKA will facilitate long baseline interferometry. An ideal solution should demonstrate efficient and effective gridding capabilities with no reliance on additional data processing mechanisms to achieve optimized performance. This is needed as the SKA anticipates extremely large volumes of observational data which cannot be efficiently cached for offline processing.

The SKA would stand to benefit financially from optimized convolutional gridding. The overall production, operation, and maintenance costs for the SKA would be compensated by a reduction in unnecessary computational overhead. In the long run, use of an optimized gridding algorithm has the potential to save large sums of money.

1.2 Aim and Scope

The aim of this research is to implement and demonstrate the effectiveness of graphically rendered convolutional gridding using OpenGL and the graphics rendering pipeline. A secondary aim of this research is to integrate graphics rendering optimizations where possible. This involves the identification of gridding features which can be optimized to improve performance, and subsequent implementation of said optimizations.

Formally, these aims are represented by the following two research questions:

1. *How can OpenGL and the graphics rendering pipeline be used to facilitate convolutional gridding?*
2. *What optimizations can be implemented to improve the performance of graphics based convolutional gridding?*

It is important to note that the definition of *performance* could be interpreted as vague. Therefore, the definition of performance used in this thesis is defined as follows to ensure there is no misunderstanding or ambiguity of its use relative to gridding:

Gridding Speed: Defines how expeditiously the algorithm can complete one single gridding cycle

Gridding Precision: Defines the level of accuracy achieved by the algorithm, measured as relative error

Memory Utilization: Defines the requisite memory footprint for caching a set of convolution kernels within the GPU

The scope of the research performed is limited to the design, development, and evaluation of a functional OpenGL based convolutional gridding algorithm. The W-Projection gridding algorithm (Cornwell et al., 2008) will be implemented to provide correction for non-coplanar baselines[1], and provide adequate support for wide-field imaging[2] for the SKA. Evaluating the efficacy of the proposed algorithm will be performed by comparing gridding performance against a leading W-Projection convolutional gridding algorithm.

1.3 Significance of the Study

The significance of this study will demonstrate the practical implications of utilizing graphics rendering technology to facilitate convolutional gridding. There exists a norm in the development of high performance software; one which observes the default adoption of general-purpose tools such as CUDA or OpenCL. This study will demonstrate that high performance computing problems can still be effectively solved with the use of native graphics rendering, and in some situations can outperform the general-purpose approach.

The findings of this research will benefit gridding algorithm researchers, by demonstrating rendering techniques which simplify data parallelization, and will show how *thinking outside of the box* can result in effective high-performance solutions.

The development of a functional convolutional gridding algorithm will contribute to the body of knowledge by exhibiting an alternative approach to gridding. This contribution is not only beneficial for the radio astronomy community and the Square

[1] Non-coplanar baselines refers to the situation where baseline recievers do not lay on a common plane.

[2] Wide-field imaging refers to a situation where the portion of the Celestial Sphere under observation is not planar.

Kilometre Array - whom of which is this research targets - but also towards other disciplines which utilize gridding algorithms.

1.4 Thesis Structure

This chapter has stated that an outstanding high performance computing problem exists for the Square Kilometre Array. A hypothesis was presented, theorizing the feasibility of the graphics rendering pipeline to facilitate optimized convolutional gridding. Two research questions have been proposed which require the design, development, and evaluation of a graphics based gridding algorithm. It has been stated that a GPU accelerated gridding solution will be implemented using OpenGL and the graphics rendering pipeline.

Chapter 2 introduces the necessary background information for this research; including an overview of existing hardware accelerated gridding algorithms, and information relative to radio interferometry based imaging.

Chapter 3 describes the application of the design science research methodology relative to the conducting of this research, including an overview of the gridding performance evaluation methods.

The implementation of the gridding algorithm will be thoroughly discussed in Chapter 4; including the design search process, configuration, and operation of the gridding solution. Several variations of custom fragment shader are presented; demonstrating improved shader design to achieve optimized gridding performance.

Chapter 5 demonstrates and evaluates the performance and efficacy of the proposed gridding solution. Gridding performance is measured against a leading CUDA based convolutional gridding algorithm.

Chapter 6 will discuss the findings of the research obtained from the evaluation of the gridding algorithm. Additionally, Chapter 6 will highlight critical aspects of

graphics based gridding which demonstrate improved performance, and to discuss why some rendering techniques reduce performance and how this can be mitigated.

Finally, Chapter 7 will conclude the thesis, and define additional work to be performed in the near future.

Figure 1.2 presents a high-level overview of the main computing elements for the OpenGL convolutional gridding algorithm.

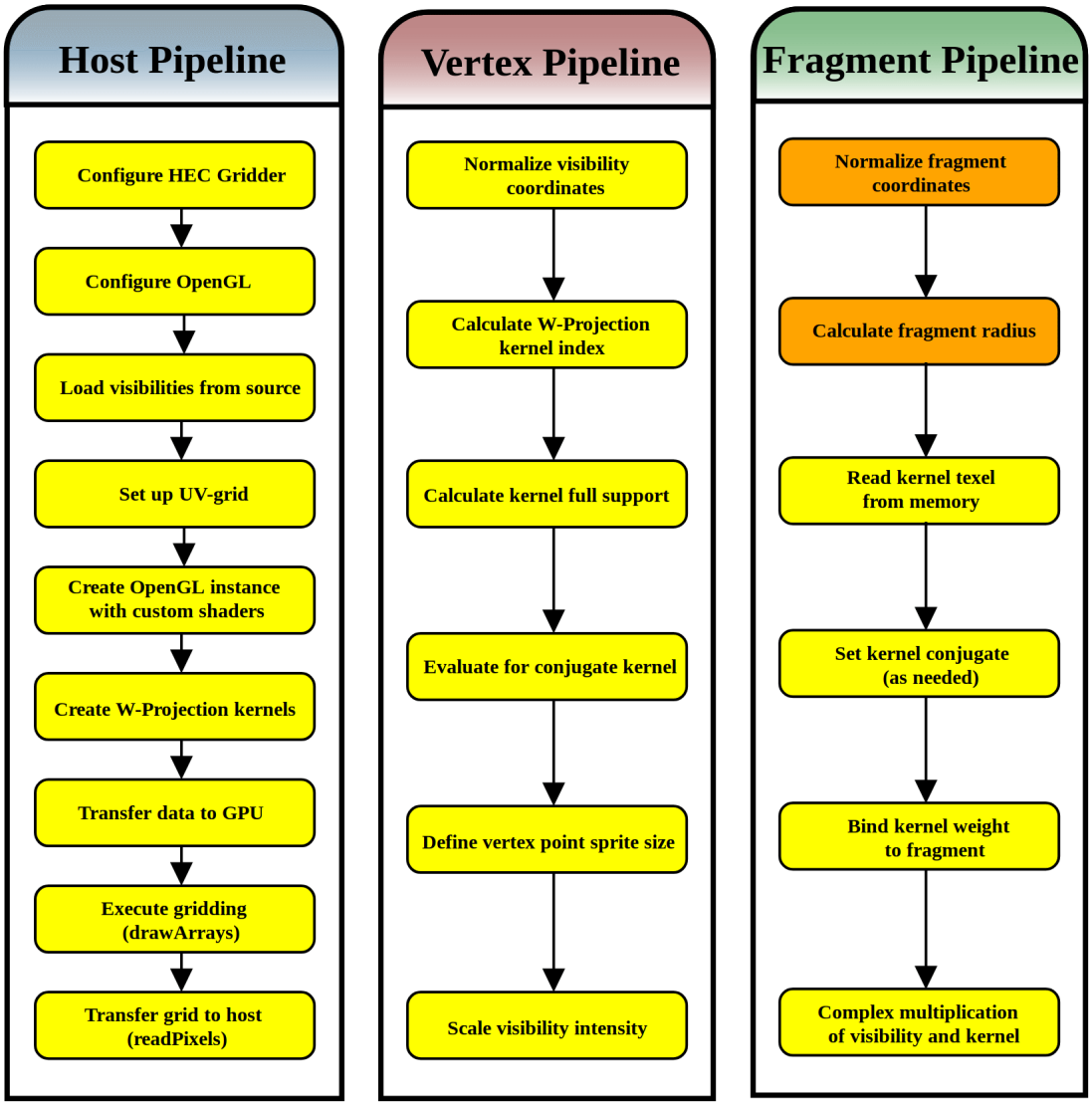


Figure 1.2: Overview of the Hall-Ensor-Campbell gridding algorithm

Chapter 2

Literature Review

This chapter will provide the necessary background information and review of relative literature with respect to convolutional gridding. The first section will cover background information for radio interferometry, which describes how and why interferometers are used, as well as how observational data are obtained and processed into a sky image. This will be followed by a dedicated gridding section, which describes the various elements used within gridding. This includes a brief history of gridding, convolution kernels, preparation, and application, and an overview of both general gridding and modern W-Projection gridding. An overview of graphics processing units (GPU) will follow, describing why GPUs are utilized to implement high-performance algorithms. An overview of Open Graphics Library (OpenGL) and CUDA will given, with a brief description of hardware accelerated general-purpose computing. Lastly, modern hardware accelerated gridding algorithms will be reviewed to provide context for the current state of the art.

2.1 Radio Interferometry

Radio interferometry is an advanced technique used in radio astronomy for the observation and study of astronomical bodies; such as star clusters, galaxies, and nebulae. It utilizes a complex imaging process, referred to as *aperture synthesis*, in which advanced signal processing techniques are used created high quality images from measured electromagnetic radiation.

Figure 2.1 is one such example of a synthesized image obtained with the use of aperture synthesis; presenting the distant galaxy Hercules A (3C 348). The use of radio interferometry has enabled the discovery of massive plasma jets on either side of Hercules A, thought to be upwards of one million light years in length.

Figure 2.1: The galaxy Hercules A (3C 348) (Thompson et al., 2017)

Two major components of radio interferometry are needed to perform aperture synthesis: the first being the data collection mechanism, and the second being the

data processing mechanism. Data collection in radio interferometry is performed with the use of the *radio interferometer*, which consists of many connected pairs of radio antennae (*"elements"*), distributed over some defined region. Each pair of elements is referred to as a *baseline*, for which the radio interferometer will consist of many baselines of various length (distance between the two elements). By utilizing many baselines of varying distance, the interferometer is capable of simulating one giant radio telescope of large aperture, capable of measuring a range of spatial frequencies. This synthetically large aperture provides a large collection area, increased sensitivity of electromagnetic radiation, and increased angular resolution for higher quality images.

Figure 2.2 demonstrates several existing radio interferometers in use around the world. Different forms of antennae are utilized, which is dependant on the frequency bandwidth the interferometer is expected to support.

(a) Low-Frequency Array (*LOFAR*, 2017)

(b) Very Large Array (*VLA*, 2017)

(c) MeerKAT (*meerKAT*, 2017)

(d) Australian SKA Pathfinder (*ASKAP*, n.d.)

Figure 2.2: Various radio interferometers

The second major component of the interferometer is the processor, responsible for the execution of various signal processing algorithms. This is often referred to as the *imager* or *imaging pipeline*, as it iteratively synthesizes a sky image from measured spatial-frequency data.

During an observation of the sky, each baseline pair of elements continuously feeds electromagnetic radiation into a *correlator*. The correlator digitizes the signals of both elements into a complex number which represents the amplitude and phase of the measured signal. This number is referred to as a *visibility* in radio astronomy, which describes the intensity (or brightness) of the observed source in the sky, in the frequency domain.

Additionally, each visibility is paired with a (u, v, w) coordinate vector, which is used to position the visibility during subsequent imaging pipeline procedures. The (u, v, w) coordinates for a visibility are dependant on the position of the elements relative to the source of the sky, which changes as the Earth rotates (Romein, 2012). The correlator will periodically sample (integrate) visibilities to ensure the rest of the imaging pipeline can manage the large volume of incoming data.

Figure 2.3 demonstrates the basic principal of single channel interferometry. Visibilities are sent from the correlator to the imager, which performs the complex synthesis imaging process. The output of the imaging process is a synthetic approximate image of the sky. The finer details of the image processor will be described shortly.

The ability to synthesize an image from spatial-frequency data is dependant on the use of Fourier transform procedures. Fourier transformations such as the discrete Fourier transform (DFT) allow for signals sampled over time to be separated into individual spectral components. This means a signal can be divided into singular sinusoidal oscillations of distinct frequencies, including the amplitude and phase per frequency component.

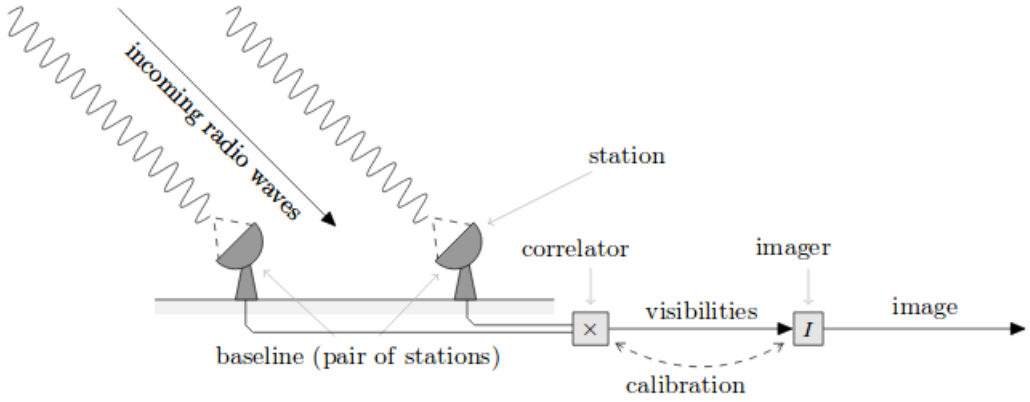


Figure 2.3: High-level overview of signal to image processing (Veenboer et al., 2017)

Figure 2.4 demonstrates this process of separation. Imaging pipelines utilize the fast Fourier Transform (FFT) as it can compute the DFT in $n \log n$ operations instead of n^2 (Cooley & Tukey, 1965).

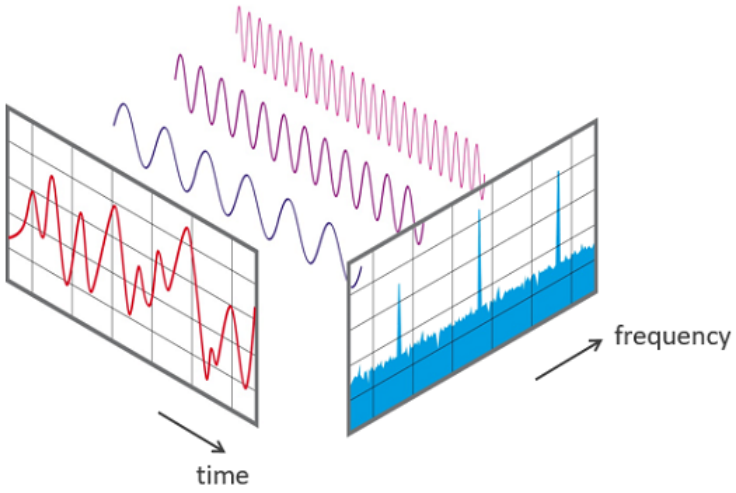
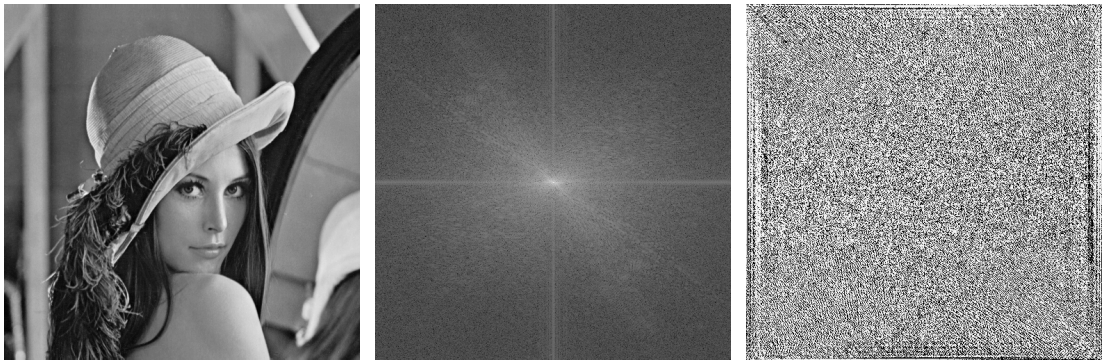


Figure 2.4: A signal in both the time and frequency domain (*FFT Basics*, n.d.)

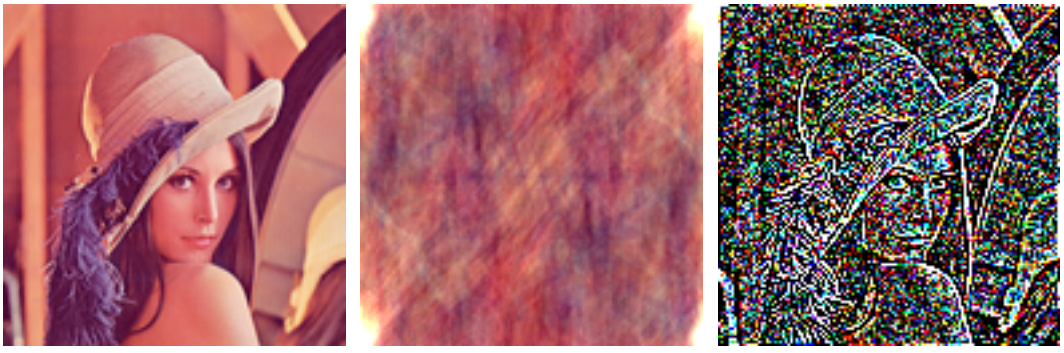
Figure 2.5 demonstrates the relationship between a typical image (a) and the Fourier domain representation of the image; the magnitude (b) and phase (c). Aperture synthesis focuses on the distribution of signal components into the Fourier domain using a technique known as *gridding*, and subsequently uses an inverse FFT to obtain the image.



(a) Image **(b)** Magnitude **(c)** Phase

Figure 2.5: Image and frequency domain components. (*Lena Söderberg*, n.d.)

Generally speaking, the magnitude is used to describe the frequencies of the image, and the phase describes the positioning of the frequency information. However, there does exist some overlap between the two components. Figure 2.6 demonstrates how a color image is reconstructed using a combination of magnitude and phase. It can be seen how the absence of either component negatively impacts the image, and how both components contributes to the quality of information.



(a) Magnitude and Phase **(b)** Magnitude Only **(c)** Phase Only

Figure 2.6: Image and frequency domain components. (*Lena Söderberg*, n.d.)

In aperture synthesis, images are synthetically coloured as radio waves lay beyond the spectrum of visible light. Multiple images sampled at different frequencies (a spectral cube), or one image of an averaged frequency are used.

2.1.1 The UVW Coordinate System

The UVW coordinate system is used when describing the position of a baseline pair of elements on the Earths surface, in relation to the fixed *phase center* at some given time during observation. The coordinate system represents a right handed Cartesian system, where the U and V axes represents East-West and North-South, with the W-axis representing the direction of the phase center.

The (u, v, w) coordinate vector bound to each visibility is measured in wavelengths for convenience, and is converted to meters prior to the gridding phase of the imaging pipeline. The components of the coordinate vector are described as a measurement of the baseline against the phase center, such that $(u, v, w) = \frac{\vec{B}}{\lambda}$ for some channel wavelength λ (Muscat, 2014).

Figure 2.7 demonstrates how the (u, v, w) coordinates for the baseline map to a point on the celestial sphere (*image plane*). This is represented by the directional cosines (l, m) where the direction vector is calculated as $(l, m, \sqrt{1 - l^2 - m^2})$ with respect to the UV axes.

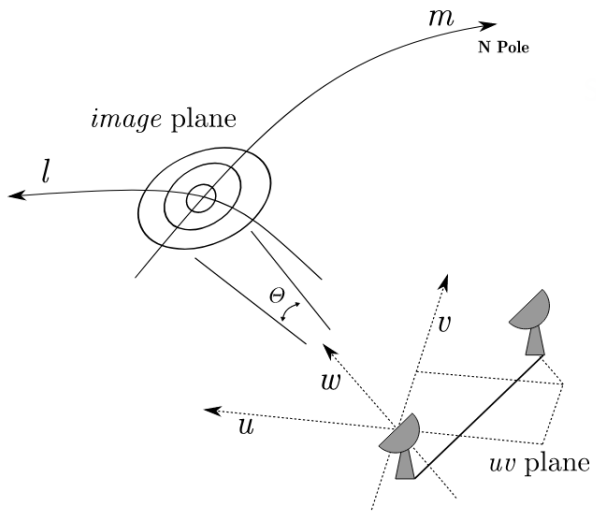


Figure 2.7: The uvw coordinate system (Veenboer et al., 2017)

2.1.2 Visibilities

Briefly introduced earlier, visibilities represent the measurement of the correlation of signals from each baseline pair of elements. Each element provides one X and Y polarization signal, which results in four correlated visibilities per baseline, per frequency. The relationship between visibilities $V(u, v, w)$ in three dimensions and the intensity distribution of the observed sky $I(l, m)$ for direction cosines (l, m) is presented in Equation 2.1 as the Van Cittert-Zernike theorem (van Cittert, 1934), (Zernike, 1938).

$$V(u,v,w) = \iint \frac{I(l,m)}{\sqrt{1-l^2-m^2}} e^{-i2\pi(ul+vm+w(\sqrt{1-l^2-m^2}-1))} dldm \tag{2.1}$$

Equation 2.2 describes the state of a visibility when observed using a small field of view, where the spheroid form of the sky under observation is close to planar (flat). If each baseline pair of receivers is also coplanar, then the effect of the w term becomes insignificant, such that the visibility can be simplified as Equation 2.2:

$$V(u,v) = \iint I(l,m) e^{-i2\pi(ul+vm)} dldm \tag{2.2}$$

This demonstrates that the sky image is a two-dimensional inverse Fourier transform of the visibilities. The imaging process shown in Figure 2.8 obtains the intensities $I(l, m)$ from the visibilities $V(u, v)$ by means of convolutional gridding and an inverse Fourier transform.

However, when performing wide-field imaging, the w term of a visibility cannot be ignored. Using a delta function, the visibility can be expressed by Equation 2.3. This shows that $I(l, m)$ can be obtained by an inverse three-dimensional Fourier transform, which would require a gridding algorithm that operates in three-dimensions.

$$V(u,v,w) = \iiint \frac{I(l,m)\ \delta(n-\sqrt{1-l^2-m^2})}{n} e^{-i2\pi(\text{ul+vm+wn})} dldmdn \tag{2.3}$$

This can be expressed using Equation 2.4, where $G(l,m,w) = e^{-i2\pi w(\sqrt{1-l^2-m^2}-1)} \cong e^{i\pi w(l^2+m^2)}$. This method is used in the W-Projection algorithm (Cornwell et al., 2008) to correct the w term, by deconvolving the visibilities $V(u,v,0)$ with the inverse Fourier transform $\frac{i}{w}e^{-i\pi(u^2+v^2)/w}$ of G.

$$V(u,v,w) = \iint \frac{I(l,m)}{\sqrt{1-l^2-m^2}} G(l,m,w) e^{-i2\pi(ul+vm)} dldm \tag{2.4}$$

2.1.3 Imaging Pipeline

The imaging pipeline is responsible for the synthesis of the image using the visibilities obtained through correlation. Several different processing steps are needed to synthesize the approximate sky image, with Figure 2.8 demonstrating a basic single channel imaging pipeline.

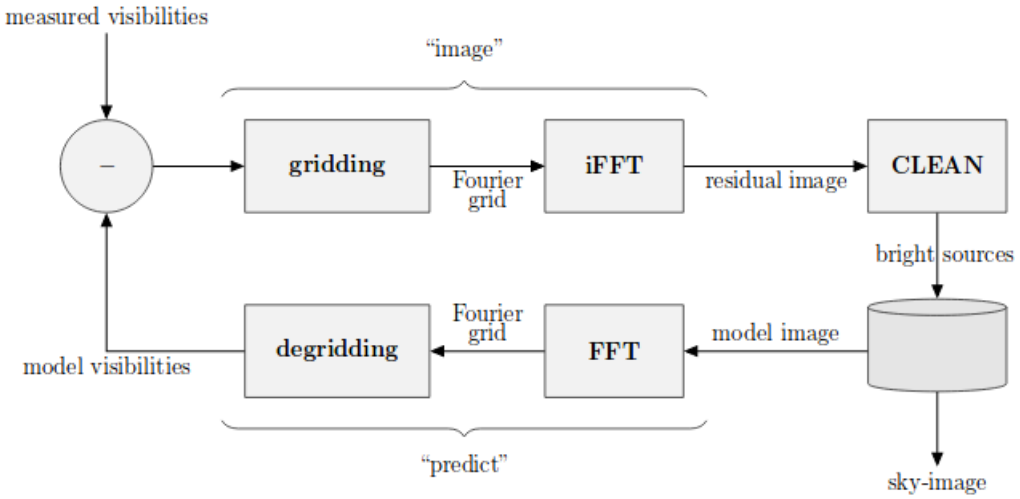


Figure 2.8: Imaging pipeline for a single channel image (Veenboer et al., 2017)

It is typical for 10 major cycles to be performed during imaging, as at this point no additional information can be extracted from the residual (dirty) image. A brief description of the imaging pipeline processes is described as follows:

Gridding Performs the convolutional gridding of visibilities into the UV-grid. This process is the focal point of this thesis.

Inverse Fourier Transform A standard inverse Fourier transform to convert the UV-grid into the residual (*"dirty"*) image for additional processing

Cleaning Performs the *cleaning* of the residual image to extract weak sources which are masked by brighter sources. The weaker sources are extracted using some variation of CLEAN algorithm, such as Högsboms CLEAN (Högbom, 1974) or Cotton-Schwab CLEAN (Schwab, 1984), and are added to the current sky model

Fourier Transform A standard forward Fourier transform to convert the clean model image into the UV-grid for further processing

Degridding Performs the *"prediction"* and extraction of visibilities from the UV-grid to identify bright model visibilities

Subtraction Performs the subtraction of *"predicted"* visibilities from the incoming measured visibilities, to ensure fainter sources are revealed through further iterations (Veenboer et al., 2017)

Perfect Imaging

Perfect imaging is a process in which an image is synthesized through a fully sampled UV-plane using non-uniformly sampled visibilities which are neither binned nor resampled across a regular grid (unlike gridding). This technique still produces a dirty image of the sky, albeit with the highest possible precision.

Several synthetic observational datasets are presented in Section 3.4 of Chapter 3, which are used during this development of an optimized OpenGL convolutional

gridding algorithm. At the time this research was conducted, an implementation of the perfect imaging algorithm was not readily available nor in the scope of the research, but was noted to be a critical step in confirming the usefulness of the implemented gridding solution. It has been found post-submission that the creation of a perfect image for each presented dataset would take approximately 19.25 days[1] to complete, which would reduce the amount of available research time by 2 months given only one GPU was available.

This imaging process requires tremendous amounts of computation to complete, even with the use of many-core accelerators. Convolutional gridding is also considered to be computationally expensive, but less so than that of the perfect imaging. Thus, gridding is used in radio interferometry as an approximation to perfect imaging.

Equation 2.5 describes how each pixel of the image is calculated as a summation of N visibilities V_n at (l, m). For the sake of performance, $(\sqrt{1 - l^2 - m^2} - 1)$ can be approximated by $-\frac{(l^2+m^2)}{2}$.

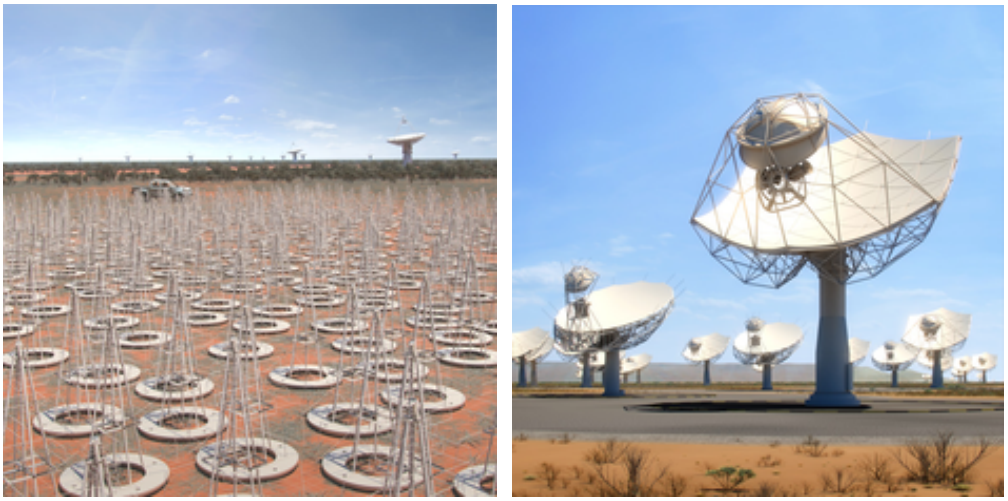
$$I(l,m) = \sqrt{1 - l^2 - m^2} \sum_{n=1}^{N} V_n \cdot e^{i2\pi(ul+vm-\frac{w(l^2+m^2)}{2})} \tag{2.5}$$

Operations are performed directly across the image domain, and not in the frequency domain as observed in convolutional gridding. Note that the sign of the w term for each visbility in the test datasets used in this thesis were required to be flipped.

[1] The reported time has been measured using an optimized CUDA based GPU accelerated perfect imaging algorithm (*Inverse Direct Fourier Transform*, 2019), which was implemented post-submission of this thesis. Calculating said time was performed by imaging a fraction of the full image obtained with the EL82-EL70 dataset. The dimension of the clipped image was 2048^2 (1.3% of the full $18,000^2$ image) using double precision, and took approximately 6 hours using the same GPU used to develop the algorithm presented in this thesis (covered in Table 3.4). Assuming $\approx$ 6 hours is needed for 1.3% of the full image, perfect imaging the total image yeilds an approximate execution time of 19.25 days.

2.1.4 Square Kilometre Array

The Square Kilometre Array (SKA) will eventually become the largest radio telescope ever constructed, featuring a total collection area of 1km^2. The SKA will be distributed over two continents as two separate instruments, and will be delivered over two separate phases. The first phase (SKA1) is expected to be operational by 2022, and will represent the foundations for both the low and mid frequency instruments, at only 10% of the total capacity of the telescope. Figure 2.9 demonstrates a rendition of the two instruments in their final form.



(a) SKA-1 Low Australia, (SKA, 2015) **(b)** SKA-1 Mid South Africa, (SKA, 2016)

Figure 2.9: A rendition of SKA1-low and SKA1-mid

South Africa will host the mid-frequency instrument of the SKA, referred to as *SKA1-mid* when describing the first phase. SKA1-mid is expected to have a total collection area of 33,000m^2, supporting a frequency range of 350 MHz to 14 GHz using approximately 200 dishes with baselines of up to 150 kilometres in distance (*SKA1 MID*, 2015).

The low-frequency instrument (SKA1-low) will be hosted in the Australian outback, supporting a range of frequencies between 50 MHz to 350 MHz. SKA1-low will consist of approximately 131,000 antennae distributed over 512 stations, with a maximum

baseline of approximately 65 kilometres (*SKA1 LOW*, 2015).

It is expected that SKA1 will produce approximately 720GB/s of visibility data. Thus, why it is clear to see why an optimized convolutional gridding algorithm is needed.

Many of the finer details for SKA2 remain indeterminate while SKA1 is under development. However, it is known that the collective capacity of the telescope will be substantially increased. SKA2-mid will be increased to several thousand receivers, and SKA2-low will increase to around one million antennae. The maximum baseline distance of the telescope will also be increased, providing support for higher angular resolution. This will drastically increase the volume of visibilities which must be gridded, and subsequently the size of the grid to be processed. It is expected that the imaging pipeline for the SKA will require operational computation speeds of approximately 250 petaFLOPS.

2.2 General Gridding

Early approaches to the gridding of radio astronomy data resorted to the naïve accumulation of visibilities $V(u, v)$ to its nearest available grid point (Mathur, 1969). An alternative approach saw visibilities being averaged to the nearest available grid points with respect to the (u, v) coordinates of the visibility (Hogg et al., 1969). The technique referred to as the *gridding* method was first demonstrated by Brouw (1975). The gridding method demonstrated the use of a weighted sum based on the distance between the coordinates of each visibility $V(u, v)$, and their corresponding grid point; such as a Gaussian average (Brouw, 1975). The use of sinc functions to achieve optimized gridding were also demonstrated (O'sullivan, 1985). However, this technique is not ideal as sinc functions are infinite in extent.

Practical gridding solutions require the use convolutional functions of finite extent

(Jackson et al., 1991). This technique is referred to as *convolutional gridding*, and is the standard gridding method for modern imaging pipelines. This approach to gridding will be discussed in detail shortly; for now, context will be given to the various components used in the convolutional gridding process.

2.2.1 The UV-Grid

The UV-grid is a two-dimensional regular Fourier domain representation of the image plane. Non-uniformly sampled visibilities are uniformly resampled into the UV-grid using some form of two-dimensional convolution kernel, contributing some distribution of intensity (or brightness) of the source under observation.

The (u, v) coordinates for each visibility trace out an ellipse on the UV-grid over time as the Earth rotates during an astronomical observation. Longer baselines (elements further apart) result in ellipses with larger axes in the UV-grid.

The elements of an interferometer are typically positioned to provide optimal coverage of the UV-grid. However, it is possible that the area of the sky under observation falls out of view during the Earths rotation. This results in incomplete ellipses in the UV-grid, as visibilities are not measurable when the source is out of view. Having an incomplete ellipse does not result in an unobtainable image, it just means the image may be of lesser quality due to the absence of complete information. Figure 2.10 demonstrates how visibilities trace an ellipse over time during observation.

2.2.2 Convolution Kernels

When performing convolutional gridding with a small field of view, the use of a general smoothing function is sufficient for resampling of visibilities onto the grid. The purpose of utilizing a smoothing function is to ensure visibilities are gradually integrated into

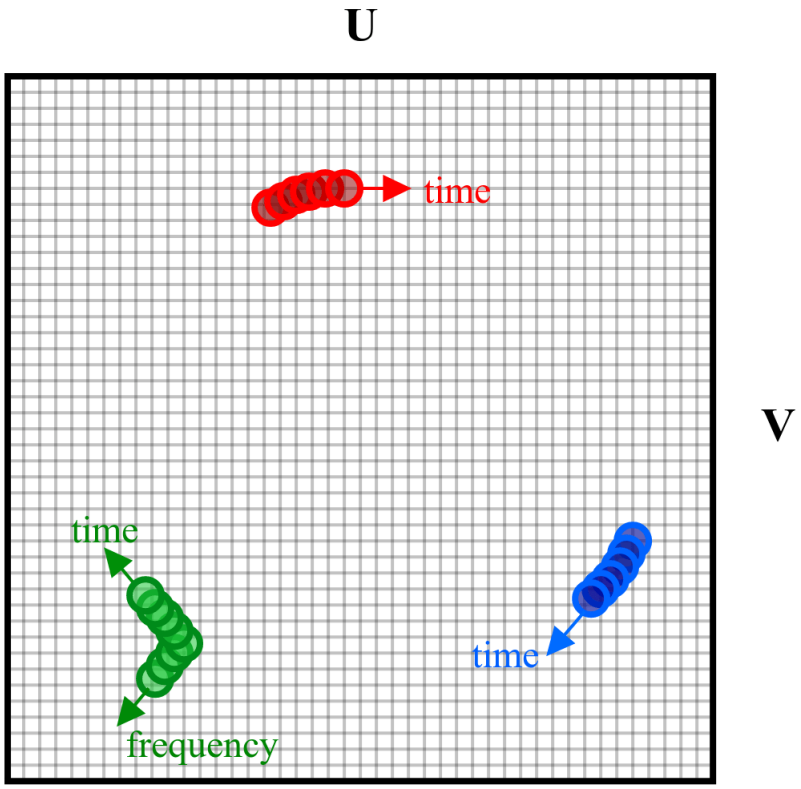


Figure 2.10: Visibility distribution in the UV-grid (Romein, 2012)

the UV-grid, which would otherwise result in the presence of artifacts in the final image.

As the resampling of visibilities is performed onto a two-dimensional plane, the use of a two-dimensional smoothing function is required; referred to as a *convolutional kernel*. Kernels are often calculated in one-dimension for cache efficiency, and a two-dimensional kernel sample is produced as the product of two samples during the gridding process. This is referred to as a *separable* convolution kernel, which does not demonstrate kernel isotropy (radial symmetry). The one exception being the Gaussian window function, which is both separable and isotropic (Awad & Baba, 2011).

The size (number of samples) of a convolution kernel is referred to as the kernel *support*, a common term used in gridding literature. Humphreys & Cornwell (2011) states that there is frequent misunderstanding within gridding literature when discussing kernel support. The use of the term has been used to describe both full-width and

half-width kernels (Humphreys & Cornwell, 2011). In the case of this thesis, the term support refers to full-width kernels, unless specified otherwise.

Modern radio astronomy gridding algorithms utilize prolate spheroidal wave functions (PSWF) for two-dimensional convolution kernels as they are band-limited[2], and are useful for recovering time-limited[3] functions from the Fourier transform. Additionally, the PSWF ensures the concentration of main lobe[4] energy is maximized, and the amount of noise in the spectrum is averaged out. Thus, reducing a loss of information at the edges of the window (Slepian & Pollak, 1961), (Landau & Pollak, 1961).

Medical imaging gridding algorithms ((Rosenfeld, 1998), (Beatty et al., 2005)) favour the Kaiser-Bessel window function. The window (Kaiser, 1966) represents an approximation of the PSFW. The Kaiser-Bessel window is favoured, as typical PSWF optimizations are not relevant for computer tomography or magnetic resonance imaging (Schomberg & Timmer, 1995).

Various convolution window functions have been examined and compared, including cosine, Gaussian, PSWF, and an alternative implementation of the Kaiser-Bessel window function. It was found the Kaiser-Bessel and PSWF are the most optimal of functions, demonstrating comparable results with respect to image quality (Jackson et al., 1991). Jackson et al. suggests that the difference in time to calculate either function is a factor to consider when selecting a function for use (Jackson et al., 1991). One would speculate this may be true for situations where kernel pre-sampling[5] is not performed.

When performing wide-field imaging, the baselines may be represented in a *non-coplanar* region. This requires an effective gridding method for resampling of visibilities

[2] Band-limiting refers the limiting of the frequency domain representation of a signal to zero above some finite frequency, such that a finite number of Fourier series terms can be calculated from said signal.

[3] Time-limiting refers to a signal which is non-zero for a finite length interval of time.

[4] The main lobe (or main beam) refers to lobe which exhibits the greatest power (field stength) in a radio antennaes radiation pattern.

[5] Suitable kernels are pre-calculated ahead of time and are cached for use at a later time. This is opposed to calculating the same kernel many times during the execution of the algorithm, which is evidently inefficient.

which accounts for the difference in coplanarity of baseline elements. The W-Projection gridding algorithm is commonly used to correct these effects (Cornwell et al., 2008). The use of the W-Projection gridding algorithm means that standard smoothing functions cannot be applied for all visibilities, and a set of specialized correction kernels of varying support must be used.

Figure 2.11 demonstrates a side on visualisation of a set of W-Projection kernels, where the w term is represented by the vertical axis, and the (u, v) terms are represented as the other two axis. Larger W-Projection kernels require greater kernel support, which is relative to the w term being corrected. The bottom of Figure 2.11 represents the smallest W-Projection kernel, which typically defaults to a general smoothing function for visibilities from co-planar baselines ($w = 0$). As the w term increases, so to does the support of the kernel. Samples for negative W-Projection kernels are also utilized, which are just the complex conjugate of a positive kernel, and are typically computed during the gridding process.

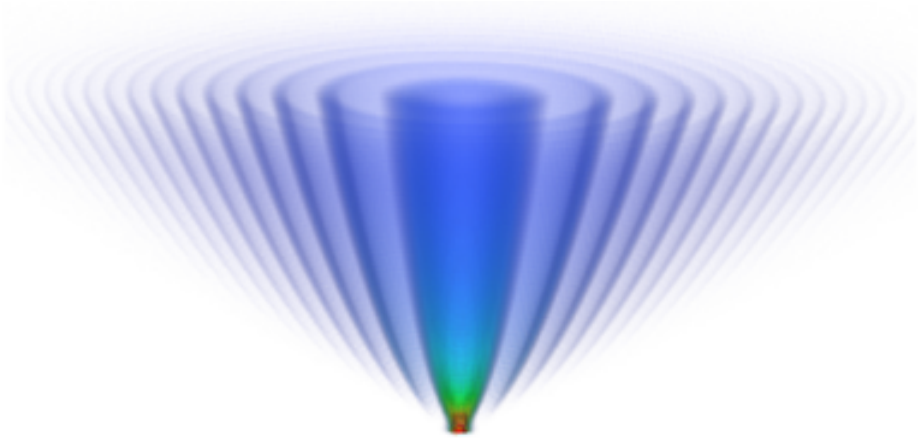


Figure 2.11: W-Projection convolution kernels (Humphreys & Cornwell, 2011)

One of the implications of using W-Projection kernels is that they are not separable by nature. Therefore, a W-Projection convolution sample cannot be produced from two one-dimensional samples. Merry (2016a) has demonstrated that W-Projection based

convolution kernels could be produced with separability, but only by means of a less precise approximation (Merry, 2016a). Further elaboration on W-Projection gridding will be presented in Subsection 2.2.3 of this chapter.

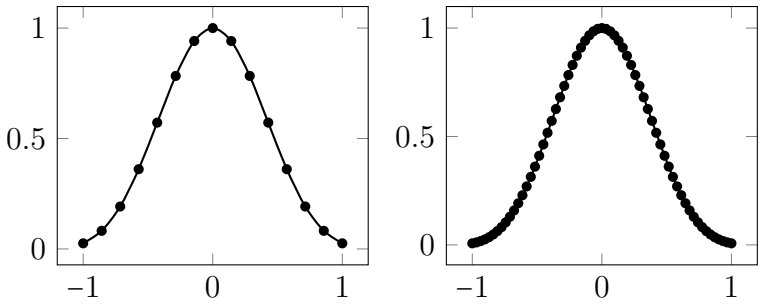
Kernel Pre-sampling

Pre-sampling involves the creation of convolution kernels which are cached and applied at a later time (during gridding). This is a standard practice for modern gridding algorithms, and is critical to the efficiency of the algorithm. Pre-sampling requires a set of kernels to be produced which are a satisfactory approximation of the various kernels needed during gridding; especially in the case of W-Projection gridding. In contrast, by not pre-sampling, a unique kernel must be calculated during the gridding of each visibility. This is of course too computationally expensive to be feasible, which is why it isn't a common practice.

Beatty et al. suggests that the creation of convolution kernels is relatively inexpensive as a preparation phase to gridding. However, if not performed, the majority of computation and processing time would be spent on the creation of kernels, and not on actual gridding of visibilities (Beatty et al., 2005).

Kernel Oversampling

Kernel oversampling is a precision enhancement technique in which a kernel with a discrete number of samples n is oversampled (multiplied) by some factor m; such that the resulting kernel contains nm sampling points. Figure 2.12 demonstrates how oversampling influences a prolate spheroidal with a support of 15 by a factor of four, thereby reducing the overall rounding/approximation error when sampled during convolutional gridding.



(a) $n = 15, m = 1$ **(b)** $n = 15, m = 4$

Figure 2.12: Oversampling a one-dimensional convolution kernel

Methods of Kernel Sampling

The method of sampling refers to what technique is used when selecting a specific sample from a kernel to convolve a visibility into the UV-grid. Two common sampling techniques are observed: *nearest-neighbour*, and *interpolation*.

Nearest-neighbour sampling describes the selection of a kernel sample by means of closest available approximation. This means a visibility will be convolved and accumulated to the grid as approximately precise as possible. Oversampling is useful in conjunction with nearest-neighbour sampling, as a finer sampled kernel will provide a closer approximation to a perfectly calculated kernel for some visibility. However, this does require more memory to cache higher oversampled kernels.

The use of interpolation based sampling means a kernel can be less oversampled than a nearest-neighbour counterpart, whilst providing comparable precision. This does require some additional computation to perform during gridding, but can provide a good compromise between kernel cache efficiency and necessary compute. The nearest suitable samples are interpolated to synthesize a new sample of approximate precision. Beatty et al. suggests that interpolation of lesser oversampled kernels offers comparable precision to that of highly oversampled kernels, whilst heavily reducing the amount of memory needed to cache a set of kernels (Beatty et al., 2005).

2.2.3 Convolutional Gridding

Convolutional gridding was described earlier as the modern approach to gridding, which uses convolution functions of finite extent to resample visibilities into the UV-grid. A naïve approach to convolutional gridding is presented below. In this example, a convolution kernel is chosen and applied for each visibility in each of its channels (*frequencies*). One basic efficiency which can be made is to use the same convolution sample for the visibilities polarization pairs XX, XY, YX, YY.

Initialise $grid_{xx}$, $grid_{xy}$, $grid_{yx}$, $grid_{yy}$ *accumulators to 0*
foreach *visibility V(u, v, w)* **do**
  foreach *channel f* **do**
    $x \leftarrow round(u), y \leftarrow round(v)$
    $C \leftarrow getConvolutionKernel(u, v, w)$
    foreach *kernel column i* **do**
      foreach *kernel row j* **do**
        $\Delta i \leftarrow x + i - u, \Delta j \leftarrow y + j - v$
        $grid_{xx}(x+i, y+j) \ += V_{xx}(u, v, w) \cdot C(\Delta i, \Delta j)$
        $grid_{xy}(x+i, y+j) \ += V_{xy}(u, v, w) \cdot C(\Delta i, \Delta j)$
        $grid_{yx}(x+i, y+j) \ += V_{yx}(u, v, w) \cdot C(\Delta i, \Delta j)$
        $grid_{yy}(x+i, y+j) \ += V_{yy}(u, v, w) \cdot C(\Delta i, \Delta j)$
      end
    end
  end
end

Algorithm 1: General Convolutional Gridding

Unlike usual image processing convolutions, the samples at (u, v) are not likely to be directly mapped to the grid points at (x, y). Thus, the convolution must be utilized at non-integer fractional positions $(\Delta i, \Delta j)$. This situation complicates the representation of a convolution in the form of a fixed matrix. To avoid calculating the convolution at arbitrary fractional values during gridding, each Δi and Δj can be rounded to the nearest multiple of 0.125 (assuming an arbitrary oversampling factor of 8). However, this would require 64 versions of the convolution kernel (C) to be pre-sampled prior to gridding.

The version of the kernel selected for each visibility would then depend on the number to which each $u-x$ and $v-y$ is closest $(-0.375, -0.25, -0.125, 0.0, 0.125, 0.25, 0.375, 0.5)$.

In the case of the W-Projection gridding algorithm, the convolution kernel needed varies with the w coordinate of each visibility. The W-Projection gridding algorithm is presented below, and assumes an oversampling factor of 64.

Initialise $grid_{xx}$, $grid_{xy}$, $grid_{yx}$, $grid_{yy}$ *accumulators to 0*
foreach *visibility V(u, v, w)* **do**
  foreach *channel* f **do**
    $x \leftarrow round(u)$, $y \leftarrow round(v)$
    $a \leftarrow round(8(u-x))$, $b \leftarrow round(8(v-y))$
    $C_{a,b} \leftarrow getConvolutionKernel(a, b, w)$
    foreach *kernel column i* **do**
      foreach *kernel row j* **do**
        $\Delta i \leftarrow x + i - u, \Delta j \leftarrow y + j - v$
        $grid_{xx}(x+i, y+j) \ += V_{xx}(u, v, w) \cdot C_{ab}(\Delta i, \Delta j)$
        $grid_{xy}(x+i, y+j) \ += V_{xy}(u, v, w) \cdot C_{ab}(\Delta i, \Delta j)$
        $grid_{yx}(x+i, y+j) \ += V_{yx}(u, v, w) \cdot C_{ab}(\Delta i, \Delta j)$
        $grid_{yy}(x+i, y+j) \ += V_{yy}(u, v, w) \cdot C_{ab}(\Delta i, \Delta j)$
      end
    end
  end
end

Algorithm 2: W-Projection Gridding

When using the W-Projection gridding algorithm, the size of the kernel scales with the value of the w term. The smallest kernels utilized are generally 7^2 or 9^2 (when $w = 0$), ranging up to 71^2 (for the maximum w), but can be as large as 129^2 (Cornwell et al., 2008). W-Projection kernels are calculated using regularly spaced $\sqrt{w}$ terms to reduce aliasing effects in the resulting image. Depending on the oversampling factor and supported number of W-Projection planes, memory requirements for storing a full set of kernels can reach gigabyte requirements.

2.3 Graphics Processing Units

Graphics processing units (GPUs) are a special-purpose many-core processor traditionally used for the optimal rendering of computer graphics. The use of dedicated graphics rendering hardware removes the computationally expensive rendering processes from the central processing unit (CPU). The GPU facilitates substantially higher performance rendering over traditional CPU rendering, as they feature higher memory bandwidth, a considerably greater number of cores (see Figure 2.13), and peak single-precision floating-point arithmetic performance.

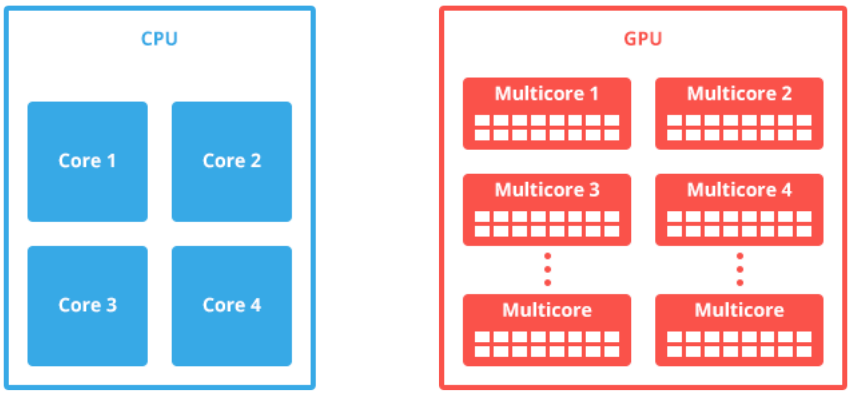


Figure 2.13: Core comparison between CPU and GPU (*CPU versus GPU*, 2014)

Modern GPUs are recognized not only for their powerful rendering capabilities, but also for supporting highly parallelized data processing for computationally expensive algorithms (Owens et al., 2008). The many-core architecture of the GPU means that a large number of data elements can be independently processed using high-performance parallelization. The use of *single instruction, multiple data* (SIMD) processing allows the GPU to achieve data level parallelization by executing the same operation on multiple points of data simultaneously. SIMD processing is used heavily in graphics rendering, as it reduces the number of memory reads and clock cycles needed to compute the same operation against a similar number of data points; i.e. modifying the brightness of an image via pixel manipulation.

2.3.1 Open Graphics Library

Open Graphics Library (OpenGL) is application programming interface (API) used for the rendering of two or three-dimensional vector graphics. OpenGL is a mature API, which has since developed into the most widely supported interface for cross-platform graphics rendering in industry. The OpenGL API is used to access the underlying hardware accelerated graphics rendering functionality of GPUs, through the use of asynchronous OpenGL commands (functions).

The process of *rendering* is one in which various transformations of geometric data (vertices) are sequentially performed to create two or three-dimensional graphical structures. These structures are eventually drawn to a display in the form of a two-dimensional raster graphic, regardless of the dimensionality of the graphical geometry by simulating depth. Figure 2.14 demonstrates the various transformations performed in the rendering pipeline, transforming a set of vertices into a graphical representation of a cow.

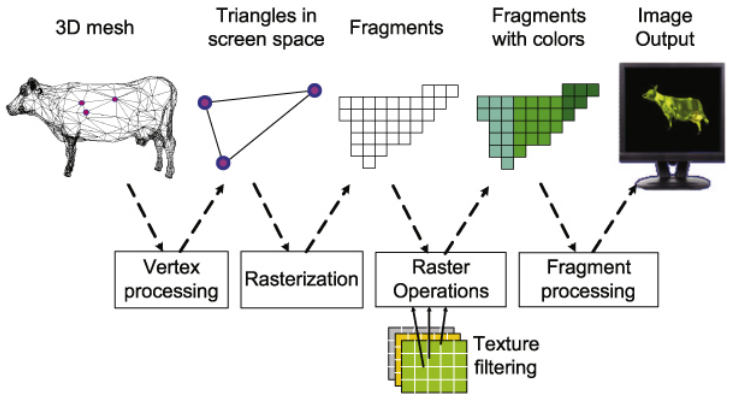


Figure 2.14: Rendering a cow with OpenGL (*Rendering of a cow*, 2009)

The steps in which graphical objects are rendered using OpenGL is referred to as the graphics rendering pipeline. Figure 2.15 demonstrates the sequence of steps performed during the rendering of computer graphics. Previously, the OpenGL rendering pipeline was a *fixed function* pipeline; meaning that customization of the rendering process

was limited to basic configurations. Lighting and texture operations were essentially hard-coded, with limited customizability.

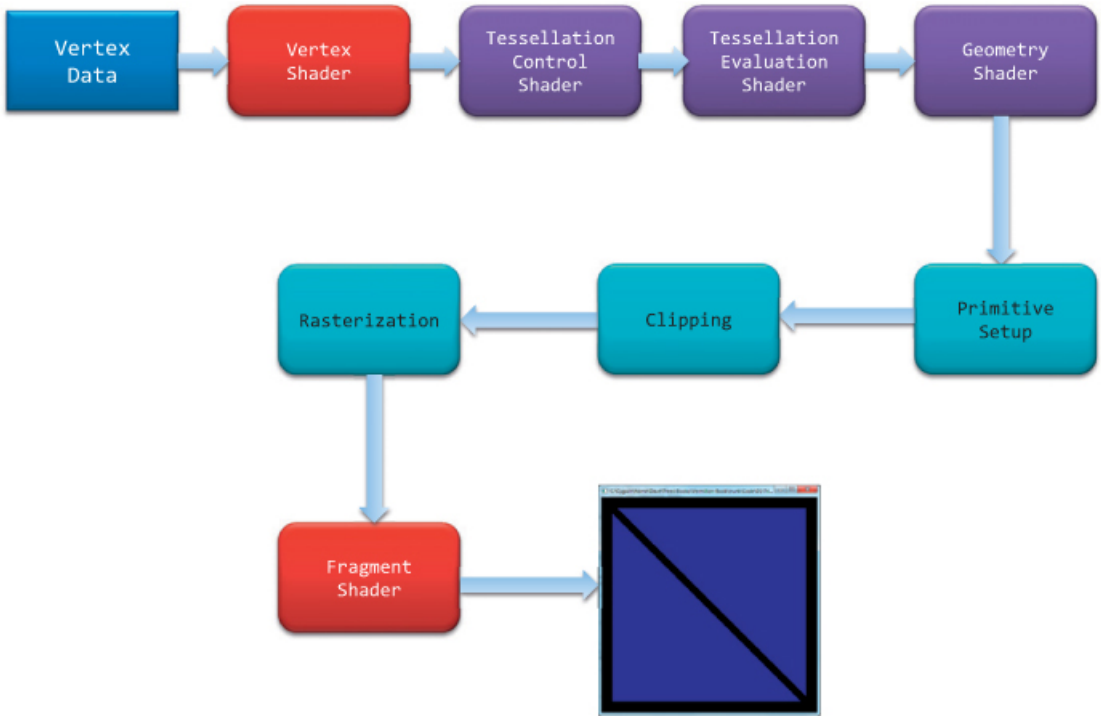


Figure 2.15: The OpenGL rendering pipeline (Kessenich et al., 2016)

Modern OpenGL supports the inclusion of custom rendering logic, in the form of programmable shaders. A C-like language referred to as Graphics Library Shader Language (GLSL) is used to create custom programmable shaders, which allows for custom vertex and pixel (fragment) logic. SIMD calculations are performed in GLSL with the use of *swizzling*, which vectorize various computations performed in each custom programmable shader. The use of custom shaders can also be optimized to achieve better performance than the fixed pipeline (Kessenich et al., 2016).

The basic premise for rendering a graphical scene using OpenGL and the rendering pipeline is as follows:

1. Custom programmable shaders (if any) are compiled at runtime on the CPU, and are bound to an instance of an OpenGL program on the GPU. Basic geometry data (vertices) are also bound prior to the execution of the rendering pipeline. Optional data can also be bound, including uniforms or graphical textures.
2. Vertex shading executes the translation of bound vertices, performing any necessary rotations, perspective projections, definition of colours, or lighting effects.
3. OpenGL primitives are then assembled from a number of vertices. This is dependant on the primitive being rendered, and includes points, lines, triangles, quads, and other polygons.
4. Primitive processing performs the clipping and culling of primitives which lay outside of the active viewport.
5. Hardware accelerated rasterization is then used to convert primitives into a set of pixels, or *fragments*, which cover the region defined by each OpenGL primitive.
6. Fragment shading is then used to define each fragments color. It is typical for textures to be applied during this step, performing texture lookups to map appropriate texels (*texture pixels*) to fragments. Additional lighting can also be applied at this stage.
7. A series of basic tests are then performed against each fragment. This includes fragment stencil/depth/scissor/blend/ownership testing to reduce redundancy when applying fragments to the bound frame buffer.
8. Lastly, fragments which have not been culled during the previous step are written to the frame buffer at their corresponding location. (Segal & Akeley, 2017)

2.3.2 General-purpose Computing on GPUs

To improve the performance of computationally intense algorithms, it is common for GPUs to be utilized for general-purpose computing (GPGPU). This involves shifting the more computationally expensive portion of an algorithm to the GPU in order to benefit from high data parallelization. Often this requires redesigning certain aspects of the algorithm to operate in a parallelized environment.

The scientific community has benefited greatly from the use of GPGPU, in a great variety of disciplines. Hall (2014) demonstrates how computer vision based feature detection algorithms achieve greatly improved performance over CPU counterparts (Hall, 2014). The Rivest–Shamir–Adleman cryptographic algorithm demonstrates improved encryption and decryption times using many-core GPUs (Harrison & Waldron, 2009). In the bioinformatics discipline, GPUs greatly improve the performance of molecular mechanics simulations, aiding in the study of biomolecule behaviour (Stone et al., 2007).

Such examples only touch on a small fraction of general-purpose computing for the scientific community. Many other disciplines demonstrate the use of GPUs for improved scientific research, including but not limited to: artificial intelligence, weather forecasting, signal processing, physics simulations, audio and image processing, computational finance, and astrophysics.

2.3.3 Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) is a general-purpose computing platform developed by NVIDIA. CUDA is designed to leverage the many-core architecture of GPUs to enable high performance for computationally expensive algorithms. It provides a layer of abstraction over the conventional rendering pipeline to present familiar coding environments for engineers (Luebke, 2008). The overall computation is distributed

over many GPU cores, leveraging the many thousands of threads which operate in parallel as *single instruction, multiple thread* processing. Figure 2.16 demonstrates how a three-dimensional computing problems is broken down using the CUDA model.

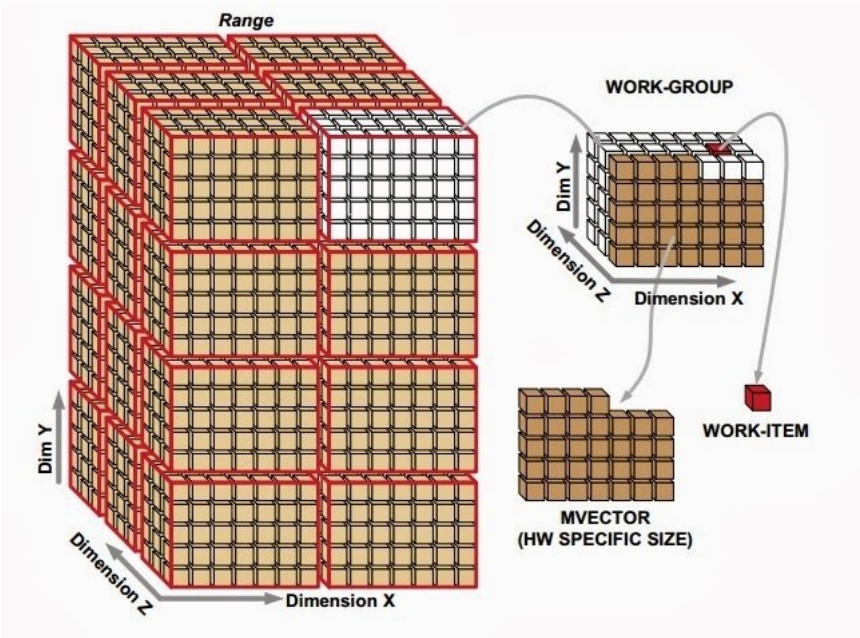


Figure 2.16: Parallelized distribution of computation (*Compute Work Distribution*, n.d.)

User defined functions for performing computation on the GPU are referred to as *kernels*, for which each individual work-item will execute. One, two, or three-dimensional computing problems are sectioned into a user defined number of work-groups, each of which has a user defined number of threads allocated. A number of available work-items for each work-group is executed in parallel over a streaming multiprocessor, with each work-item invoking the specified kernel. One main limitation of CUDA is that it is only supported by NVIDIA hardware. Thus, the breadth of GPUs which can be used for computation via CUDA is limited to NVIDIAs propriety hardware.

2.4 Hardware Accelerated Gridding

Modern convolutional gridding algorithms utilize hardware accelerators to achieve improved gridding performance. One of the biggest challenges in achieving optimized gridding performance is dependant on the effective utilization of available hardware resources. Concurrency is also one of the biggest problems to manage, as it is to be expected that numerous points on the UV-grid will require multiple visibilities to be accumulated during convolutional gridding. The use of atomic accumulators resolves the situation of race conditions, but does penalize the performance of the algorithm whilst threads wait their turn.

Specialized gridding algorithm designs are observed in the literature which remove the need for atomic accumulators. Romein (2012) presents one of the most influential convolutional gridding algorithms. His work demonstrates a W-Projection gridding solution which simplifies concurrency by partitioning a singular grid into a set of sub-grids. Each sub-grid is sized according to a fixed kernel support, with one thread allocated per grid point of the sub-grid. Concurrency is managed by using the same thread for the same relative grid point per sub-grid, which performs the sequential convolution and accumulation of visibilities which fall into its respective grid point (Romein, 2012). Merry (2016b) extends this solution by utilizing thread coarsening. This results in each thread supporting a 2x2 region of each sub-grid instead of one thread per grid point per sub-grid (Merry, 2016b).

To maximize gridding performance on hardware accelerators, several algorithms perform additional data processing such as sorting, searching, compression, or elimination (Humphreys & Cornwell, 2011), (Varbanescu, 2010). This additional processing requires unnecessary computation to suitably organize visibilities for efficient gridding.

Muscat (2014) demonstrates an extension of Romeins work, which utilizes a pre-gridding preparation step of compression or elimination of visibilities which overlap

on the UV-grid (Muscat, 2014). Edgar et al. (2010) demonstrates a gridding solution which relies on visibility searching, in which threads at each grid point search through a collection of visibilities to find visibilities to grid at, or near, its current grid point (Edgar et al., 2010). Du Toit (2017) also presents a gridding algorithm which utilizes grid tiling and visibility bucketing. The grid is sectioned into efficiently sized tiles suitable for storing in registers for quick access. Visibilities are bucket sorted according to which tile(s) they belong. Execution of the gridder sees each tile processed as a distributed workload and visibilities sequentially accumulated to the grid points of each tile (Du Toit, 2017).

An additional challenge for hardware accelerated convolutional gridding is how to improve gridding precision without impacting gridding throughput. Most gridding algorithms demonstrate the use of highly oversampled convolution kernels, typically with an oversampling factor of 8. Only one gridding solution has demonstrated exploration of texture based W-Projection kernels (Romein, 2012). It was suggested that textures are a viable solution for applying convolution kernels, and that the use of hardware accelerated interpolation may provide comparable precision without the need for excessive oversampling factors (Romein, 2012).

2.5 Summary

This chapter has presented the necessary background information needed for context of the research. An overview of radio interferometry has described the purpose of interferometry, and how data samples are obtained and processed using the imaging pipeline. Context for the Square Kilometre Array has been given, describing the need for efficient gridding due to large volumes of observational data. Convolutional gridding, and the various elements used in gridding have been described, which includes common convolution functions of choice and kernel enhancement techniques. Graphics

processing units have been introduced, and an overview of the Open Graphics Library and CUDA interfaces have been given. The challenges of hardware accelerated gridding have been discussed, demonstrating how modern gridding algorithms rely on additional data processing to achieve high performance. It was suggested that oversampling could be better performed with the use of texture based convolution kernels, and hardware accelerated interpolation.

Chapter 3

Research Design

The aim of this research is to create an optimized hardware accelerated convolutional gridding algorithm, suitable for the needs of the Square Kilometre Array (SKA). This research theorizes that gridding can be simplified, and performance improved with the use of the graphics rendering pipeline. To test this theory, an Open Graphics Library (OpenGL) based gridding algorithm ("*gridder*") has been designed, developed, and demonstrated for its effectiveness as an optimized gridding solution.

Both action research and design science were reviewed as the methodology of choice for conducting this research. Design science was conclusively selected as it closely relates to traditional software development methodologies (iterative design and development; frequent evaluation), with supporting practices necessary for conducting information systems research. Design science is intended to shape new realities via the construct of original and innovative solutions, whereas action research is most suited when researching existing complex realities (Iivari & Venable, 2009).

Design science emphasizes the development and delivery of an *artifact* to effectively address an outstanding critical business problem (Hevner et al., 2004). In contrast, when conducting action based research, actions (changes) are performed against an existing reality in order to study the propagated results (Baskerville, 1999).

The development of an OpenGL based gridding algorithm has been briefly examined in the past (Edgar et al., 2010). However, no performance results have been reported to demonstrate the efficacy of the solution, nor any *existing reality* (source code) for which to improve upon. Therefore, design science presents an effective framework for the design, development, and evaluation of a new solution; the hypothesized graphics based convolutional gridder.

3.1 The Design Science Approach

Hevner et al. (2004) presents seven practical guidelines for performing effective information systems research using design science. Creation of an artifact is emphasized as the most important outcome of the research, which must provide a solution to an unsolved problem. Application of the body of knowledge and existing theories for the problem domain must be applied during the design and development of the artifact. Rigorous evaluation of the artifact is necessary to demonstrate the quality, utility, and efficacy of the implemented solution. Finally, communication of the research must be effectively performed to suitable audiences (researchers, managers, stakeholders, et cetera) (Hevner et al., 2004).

Peffers et al. (2007) presents these seven guidelines as a six step framework for conducting high impact design science research:

Problem identification and motivation Definition of the specific research problem under investigation, with a justification of the value the solution will bring by solving the problem.

Define the objectives for a solution The inference of objectives of the solution from the problem definition. Use of rational thought to specify realistic objectives of what is feasible from the problem definition, and knowledge of the problem being investigated.

Design and development Designing and developing the research artifact. This phase includes determining the functionality of the artifact, its architecture, and the implementation of the artifact.

Demonstration Demonstrating use of the artifact as a viable solution to one or more of the identified problems.

Evaluation Evaluating the effectiveness and efficacy of the artifact as a solution by means of observation and measurement.

Communication Effectively communicating the problem to be solved, why it is important to solve, the artifact produced, and its effectiveness to solve the outlined problem (Peffers et al., 2007).

3.2 Methods of Evaluation

Evaluating the efficacy of the hypothesized gridding solution (*"the artifact gridder"*) requires accurate performance measuring techniques. The definition of performance in this thesis has been described earlier (Section 1.2 of Chapter 1), which has stated the primary aspects of interest are gridding speed, gridding precision, and effective utilization of memory.

3.2.1 Gridding Speed

Measurement of gridding speed was performed by utilizing graphics processing unit (GPU) based timing functions. This was achieved by timing scoped sections of host-side code containing desired OpenGL commands, and blocking the central processing unit (CPU) until the issued command(s) were completed. This technique is necessary as OpenGL operates mostly as an asynchronous process, exclusive from the host system.

Each demonstration of gridding speed presented in this thesis has been performed across one hundred iterations to ensure timing integrity. The times presented in Chapter

5 represent the mean result calculated over all one hundred iterations. Standard deviation has been recorded for each demonstration, but has not been presented in the relevent graphs in Chapter 5. The reasoning for this is because there was not a considerable amount of variance observed across each experiment performed, and so was deemed insignificant. However, for the sake of completeness, it was observed that mean standard deviation across all experiments performed was in the range of approximately 2-8 milliseconds.

3.2.2 Gridding Precision

Precision of the artifact gridder was measured by calculating the difference between two processed grids, using the relative error L2 norm described in Equation 3.1. The two grids evaluated consist of one gridded result produced by the artifact gridder, and one by the comparison gridder (described shortly). Comparable gridding configurations are used to ensure equality and fairness. Referring to Equation 3.1, G represents the grid produced by the artifact gridder, and G' represents the grid produced by the comparison gridder. The indices i and j simply refer to the current element for either two-dimensional grid.

$$error = \frac{\sqrt{\sum_{ij} \left| G_{ij} - G'_{ij} \right|^2}}{\sqrt{\sum_{ij} \left| G'_{ij} \right|^2}} \tag{3.1}$$

Measuring the precision of the artifact gridder this way is not the most ideal approach, but was the most suitable at the time. True measurement of gridding algorithm precision requires relative error to be measured against a high precision sky image obtained via perfect imaging (described in Subsection 2.1.3 of Chapter 2).

3.2.3 Memory Utilization

Memory utilization refers to the amount of memory needed to store a set of convolutional kernels within the GPU. The artifact gridder utilizes uniformly sized texture based convolution kernels, so calculating the memory requirements is relatively straightforward. Three variations of fragment shader are presented in this thesis (*"full"*, *"reflective"*, *"isotropic"*), which demonstrate improved usage of textures during fragment shading. Each of the shaders depends on a differing amount of kernel to perform convolution, so memory requirements do vary. Equation 3.2 presents the formulae for calculating the number of bytes needed for each set of textured kernels.

$$\text{Full Textures} = t^2 n8$$

$$\text{Reflective Textures} = \frac{t}{2}^2 n8 \tag{3.2}$$

$$\text{Isotropic Textures} = \frac{t}{2} n8$$

Referring to the terms of the formulae, t represents the uniform size of each texture in one dimension, and n is the number of W-Projection convolution kernels to produce. Each weighted sample (texel) in a convolution kernel texture is represented by a single precision complex number; hence the 8 byte constant.

3.3 The Numerical Algorithms Group Gridder

Evaluation of the artifact gridder requires an existing gridding solution for which performance characteristics can be measured and compared. Du Toit (2017) presents an advanced W-Projection gridding algorithm implemented and optimized for the NVIDIA Tesla P100 GPU. This algorithm was developed and optimized as a conjoint effort

between NVIDIA, Oxfords e-Research Centre, and Oxfords Numerical Algorithms Group (NAG), and will be referred to as the *"NAG gridder"* from the remainder of this thesis. Source code for the algorithm was obtained via an online repository (*GPU-gridding*, 2017).

The NAG gridder was implemented using the CUDA application programming interface, and is a heavily optimized extension of the work demonstrated by Romein (2012). The NAG gridder approach to convolutional gridding observes the standard UV-grid partitioned into small sub-grids (*"tiles"*), where each tile is sized effectively for optimal processing via registers. Visibilities are bucket sorted according to which tile they belong (likely several), and tiles are then batch processed on the GPU as a distributed workload. Use of the work-distribution threading technique (Romein, 2012) ensures visibilities are accumulated to each grid point sequentially, removing the need for atomic accumulators. The NAG gridder does support the use of W-Projection convolution kernels, but does not produce its own. Flexible Image Transport System (FITS) files supplied with the algorithm source code include several synthetic observation datasets, configuration parameters for W-Projection gridding, and sets of pre-calculated W-Projection kernels for the relevant datasets.

The NAG gridder supports both single and double precision gridding. However, only single precision gridding will be used during the evaluation of the artifact gridder. This is to ensure testing is fair between the NAG and artifact gridder, which is limited to single precision as a constraint of the graphics rendering pipeline.

3.4 Synthetic Observation Datasets

As previously mentioned, three synthetic datasets have been supplied with the NAG gridder (*Gridding Datasets*, n.d.). Each dataset represents a downscaled astronomical observation, similar to what is anticipated from use of the Square Kilometre Array. The

W-Projection algorithm is needed for the gridding of these datasets, as the visibilities are synthesized using software which simulates long baseline interferometry (*OSKAR*, 2013).

Through inspection of each dataset, it was found that approximately 32 million visibilities are present in each file. Each individual visibility is paired with six single precision attributes: the (u, v, w) coordinates, the complex intensity/brightness (amplitude and phase), and a unique density (scalar weight).

Table 3.1 provides an overview for each of the three datasets, describing their characteristics, and configurations needed to facilitate high precision W-Projection gridding.

Table 3.1: An overview of the synthesized observation datasets

	EL30-EL56	**EL56-EL82**	**EL82-EL70**
Number of Visibilities	$31,395,840$	$31,395,840$	$31,395,840$
Grid Dimension	$18,000^2$	$18,000^2$	$18,000^2$
Gridding Region	$3,243 \times 4,170$	$3,981 \times 4,426$	$4,001 \times 4,385$
Number of W Planes	922	601	339
Min Kernel Support	9	9	9
Max Kernel Support	191	145	89
Max W	$\approx 19,225$	$\approx 12,534$	$\approx 7,083$
Kernel Memory	280MB	88MB	16MB
Cell Size (radians)	$\approx 6.0\text{e}-6$	$\approx 6.0\text{e}-6$	$\approx 6.0\text{e}-6$
Observation Frequency	100MHz	100MHz	100MHz

Referring to Table 3.1, *kernel memory* describes the amount of memory needed for the NAG gridder to cache each set of pre-calculated kernels. The kernels are oversampled by a factor of four, and only one quarter of each kernel is retained as each kernel is mirrored during gridding. The *number of w planes* refers to how many positive w coordinate kernels are needed. Kernel samples for negative w coordinate kernels are obtained during convolutional gridding. *Gridding region* describes the subregion of the full grid which is sent to the GPU for processing. The NAG gridder does not transfer

the entire grid to the GPU as it has predetermined the region for which the visibilities will be gridded.

Figure 3.1 demonstrates the distribution of visibilities from the three datasets to their respective full kernel support needs. It is noted that there is a large amount of variation between the three datasets and their kernel support needs. Figure 3.1 suggests that the three datasets could be classified as small (EL82-EL70), medium (EL56-EL82), and large (EL30-EL56), with respect to kernel memory requirements and the overall computation needed to grid each set.

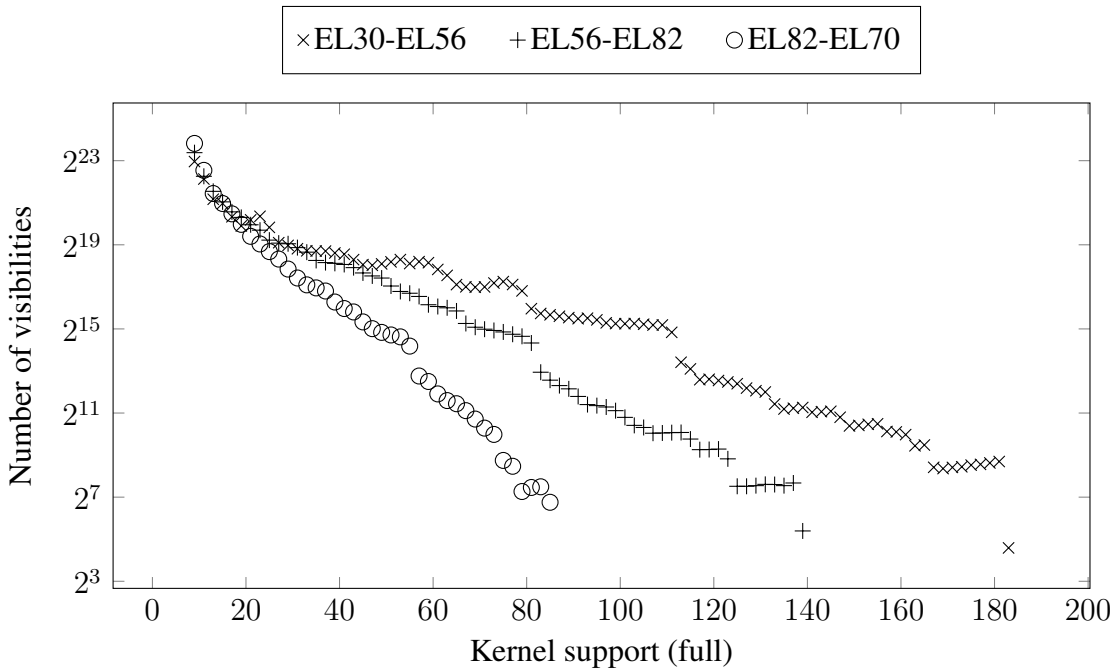


Figure 3.1: Mapping the distribution of visibilities to required kernel support sizes

Table 3.2 presents a sample of arbitrarily selected visibilities from the EL82-EL70 dataset. At this time, the (u, v, w) coordinates are not yet scaled from wavelengths to meters. However, this is performed prior to the commencement of convolutional gridding.

Table 3.2: An arbitrary sample of visibilities from the EL82-EL70 dataset

Intensity	Density	u	v	w
$0.107880 - 0.001004i$	0.007634	−4736.927236	4204.883453	−802.135032
$-45.289967 + 41.020031i$	0.011364	7200.512015	−3599.337088	1064.881877
$-0.262139 + 0.334296i$	0.012048	10022.234603	−3100.472484	1381.926582
$-0.274084 + 0.051271i$	0.055556	32211.273841	17121.246817	2964.302592
$-0.126185 - 0.104156i$	0.021277	−13842.320969	−5417.835614	−1437.271630

3.5 Application of Design Science

The research presented in this thesis was conducted using the design science framework presented by Peffers et al. (2007), based on the guidelines proposed by Hevner et al. (2004). The structure of this thesis follows the presentation suggestions for high impact design science research recommended by Gregor & Hevner (2013).

3.5.1 Problem Identification and Motivation

Chapter 1 introduces and discusses the problem under consideration in detail, including the motivation and justification for why the problem must be solved. Chapter 2 demonstrates how the problem has been approached in the past, and provides supporting discussion as to why it should approached from an alternative viewpoint.

3.5.2 Defining Solution Objectives

The objective of the solution is to effectively and efficiently perform GPU accelerated convolutional gridding by leveraging the graphics rendering pipeline. Effective and efficient gridding requires that the artifact need not rely on additional data processing mechanisms to achieve improved performance.

The artifact gridder must emphasize the maximization of the defined gridding performance metrics. It is difficult to specify definitive minimum performance thresholds for measuring success, as performance varies across different GPUs. Therefore, the NAG gridder is used to set the baseline performance for which the artifact gridder must be comparable using identical hardware. Additionally, wide-field imaging must also be supported by the artifact gridder in the form of the W-Projection algorithm.

3.5.3 Design and Development

Subsection 4.1.1 of Chapter 4 describes the design search process used to find an optimized solution. This includes a discussion of the artifact gridder and its operations in detail. Table 3.3 lists the hardware used during the design, development, and evaluation of the artifact gridder for reference.

Table 3.3: HEC gridder development hardware

Component	Model
Operating System	Ubuntu 17.10 64-bit
Motherboard	Gigabyte X399 AORUS Gaming 7
Central Processing Unit	AMD Ryzen Threadripper 1920X 12-core
Graphics Processing Unit	NVIDIA TITAN X 12GB (Pascal)
Random Access Memory	G.Skill 16GB DDR4 Trident Z (2x, total 32GB)

3.5.4 Demonstration and Evaluation

Demonstration and evaluation of the artifact gridder must be performed to ensure that one or more defined problems have been resolved by the artifact. Demonstration of the artifact gridder was performed by evaluating and measuring the performance of the artifact in comparison to the performance of the NAG gridder. The gridding configurations used, and evaluation results are presented in Chapter 5, which are discussed in depth later in Chapter 6.

It was mentioned earlier that the NAG gridder is optimized for use on the NVIDIA Tesla P100 GPU, which was not accessible during the conducting of this research. Therefore, the performance results presented in this thesis for the NAG gridder were obtained using the same hardware described in Table 3.3. Table 3.4 provides the specification for both the Titan X and Tesla P100 cards for reference.

Table 3.4: Specification of the NVIDIA Titan X and NVIDIA Tesla P100 GPUs (*TITAN X Graphics Card for VR Gaming from NVIDIA GeForce*, n.d.), (*NVIDIA Tesla P100*, 2016)

Specification	**Titan X**	**Tesla P100**
Architecture	Pascal	Pascal
Number of CUDA Cores	3584	3584
Stream Multiprocessors (SM)	28	56
Cores per SM	128	64
Base Clock (MHz)	1417	1328
Boost Clock (MHz)	1531	1480
Single-precision (TFLOPS)	10.97	9.3
L2 Cache Size (KB)	3072	4096
Standard Memory Config (GB)	12	16
Memory Interface Width	384-bit GDDR5	4096-bit HBM2
Memory Bandwidth (GB/sec)	480	732
Graphics Card Power (W)	250	250

3.5.5 Communication

Design science is useful for conducting information systems based research, typically the problems considered are critical to business operations. Therefore, communication of the research should assume the audience consists of both managers, and those whom of which are technologically savvy. As such, the communication of this research must be presented that both specialists and laymen can extract value from the findings. The research performed in this thesis serves as the platform of communication, which includes the produced software artifact (*HECGridder*, 2017).

Chapter 4

Algorithm Design and Implementation

This chapter provides an overview of the produced artifact, informally titled the Hall-Ensor-Campbell (HEC) convolutional gridding algorithm (*"HEC gridder"*). A brief overview will introduce the Open Graphics Library (OpenGL) based gridding solution, outlining the benefits of the algorithm. Exploration into the design search process will follow, informing on the design and development history of the algorithm. This will be followed by the configuration of the algorithm, describing the parameters which are customizable, and their application within the gridding solution. The arrangement of OpenGL to facilitate gridding will follow. A description of data mapping between the host and the graphics processing unit (GPU) will describe how the grid, visibilities, and kernels are maintained during gridding. Creation of W-Projection convolution kernels will be elaborated to inform how custom texture based kernels were implemented. Custom vertex and fragment shader stages responsible for gridding will be described in detail. Three generations of fragment shader will be explored, and will discuss the optimization of each generation. Finally, several optimizations attainable by using OpenGL and the rendering pipeline will conclude this chapter.

4.1 The Hall-Ensor-Campbell Gridding Algorithm

Development of the HEC convolutional gridding algorithm was performed as a conjoint effort between Dr. Seth Hall, Dr. Andrew Ensor, and myself. The naming of the algorithm is an informal reference to the authors of the solution, and the ordering of the surnames does not represent the level of contribution from each author.

Dr. Ensor is credited for conceptualizing the idea of gridding as a graphics rendering problem, and has been continuously supportive with problem solving during the development of the algorithm. Dr. Seth Hall is credited for supporting me with his extensive knowledge of the OpenGL programming interface, and for conjointly developing the custom vertex shader, and three generations of custom fragment shader. I have worked alongside Dr. Hall on the custom vertex and fragment shaders, and have developed the host-side code which includes the setup, configuration, operation, testing, and experimentation of the gridder. W-Projection kernels have also been implemented by myself as a C based implementation of the Cornwell et al. W-Projection algorithm (Cornwell et al., 2008). This was ported from the Python implementation listed in the *Algorithm Reference Library* software package (*algorithm-reference-library*, 2018). The C based implementation of W-Projection also includes bicubic interpolation and normalization techniques required for the use of texture based W-Projection kernels.

The HEC gridder is the result of twelve months worth of research invested in developing an efficient gridding algorithm suitable for the Square Kilometre Array (SKA). By exploiting the similarities between graphics rendering and gridding, the HEC gridder mitigates traditional hardware accelerated gridding complexities. The list below provides an overview of the advantages offered by the HEC gridder:

High Processing Speeds The asynchronous nature of OpenGL ensures visibilities are constantly being processed through differing stages of the rendering pipeline at any given time.

No Additional Processing Visibilities require no form of sorting, bucketing, searching, or compression for the algorithm to achieve optimized gridding performance.

Implicit Concurrency OpenGL manages threading and concurrency during the rendering process, removing the need for complicated algorithm designs or manual thread management.

High Resolution Kernels Utilization of textures, and hardware accelerated interpolation provide high resolution kernels for high precision gridding.

Hardware Compatibility Wide industry support for OpenGL ensures the gridder can operate on a breadth of GPUs, and is not limited to specific hardware vendors.

It is important to note that there are two caveats to the HEC gridder which result from the utilization of the graphics rendering pipeline. The first is that the gridder only supports single floating point precision, which stems from OpenGL having limited support for double precision. OpenGL is a mature programming interface, and historically GPUs were not intended to handle double precision graphics rendering. The overhead required to process double precision was not worth the trouble, as the human eye would not distinguish between single or double precision renderings. Secondly, OpenGL requires that a display be connected to the GPU in order for the program to run. OpenGL is not intended to be run as a background process, and assumes that rendered graphics will ultimately be displayed to an end user. This can be resolved by connecting a display to the GPU, or by enabling the use of display virtualization when using remote servers or headless GPUs (*Remote Off-Screen Rendering with OpenGL*, 2014).

4.1.1 Design Search Process

The design search process of the Hall-Ensor-Campbell gridding algorithm has been largely experimental during its development. This is partly due to the non-conventional approach towards convolutional gridding as a rendering problem, and the general

complexity of producing graphics rendering software. Initial development of the algorithm began as a simplified central processing unit (CPU) based serial gridding algorithm written in C. This implementation laid the foundations for the HEC gridder, providing simple gridding operations such as positioning and sequential accumulation of visibilities to a small regular grid. Single fixed sized prolate spheroidal kernels were implemented to perform convolution of visibilities.

The next phase in development was to produce a GPU accelerated implementation to make effective use of data parallelization. Development of an OpenGL compute shader gridder improved on the throughput attainable from the serial implementation. Concurrency was managed using the work-distribution threading technique presented by Romein (2012), which was shown to resolve the need for atomic accumulation to the grid, but presented a limitation to the adaptability of the compute shader gridder. This was due to the proposed *tiling* technique, which observes the grid being fragmented into fixed kernel support sized tiles (Romein, 2012).

In theory, the tiling technique would be able to facilitate variable kernel support sizes for W-Projection gridding. However, this would require visibilities to be sorted by w coordinate to maximize performance, and would require tiles and threads to be recalculated and allocated after each set of w related visibilities is processed. Additionally, it became apparent that compute shaders suffer from grid sizing limitations, and slow memory allocation times. It was found that grids with a size of approximately 1200^2 would take up to 10-15 minutes for the GPU to set up before gridding could commence. It was unclear as to why this occurred, as only approximately 11 megabytes of memory is needed to store a grid of this dimension; assuming each grid point stores one single precision complex number.

The slow allocation time, and limitation of grid dimensions resulted in the experimentation of an alternative compute shader approach. This was intended to support

greater grid dimensions at the cost of *some* additional visibility processing. The algorithm operated as a form of *stitch* based gridding, which involved the full grid being quartered into four quadrants. Visibilities would be bucketed on the host for the respective quadrant, and each quadrant would be processed on the GPU as each bucket became *full*. A processed quadrant would then be returned to the host. After some time, all four quadrants would be stitched together at the seams to reform the complete grid. Each of the quadrants was appropriately padded by the active kernel support to ensure that visibilities need not be bucketed more than once in case of overlap. Figure 4.1 demonstrates the general concept of stitch gridding via compute shaders. Note that Table 4.1 provides the definition of terms used in Figure 4.1.

Sectioning the grid into four quadrants was an arbitrary choice, as any number of divisions could have proven useful. However, it is worth noting that dividing the grid into a greater number of sections would result in more data transfer between the host and device (GPU). Data transfer is one of the primary bottlenecks in high performance computing, and mitigating as much data transfer as possible is critical to algorithmic performance.

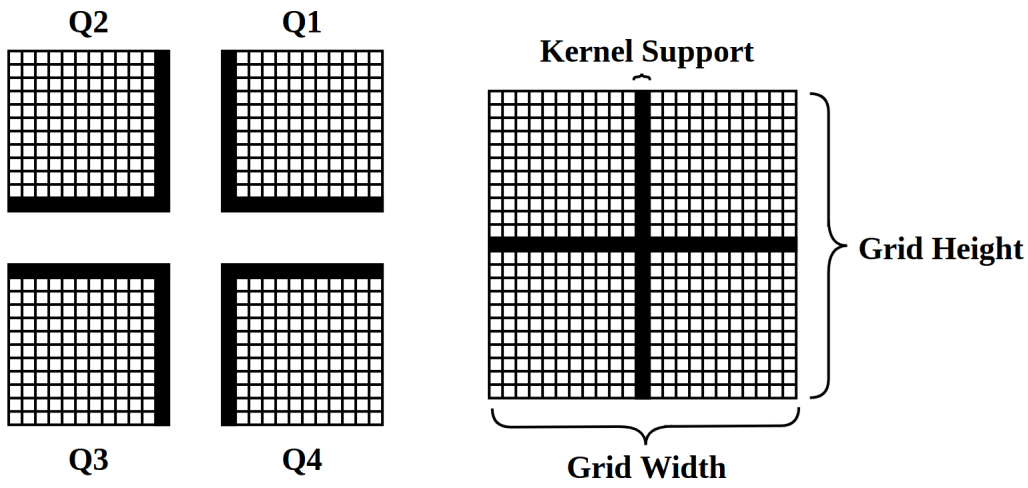


Figure 4.1: The OpenGL compute shader stitch gridding technique

This approach demonstrated an improvement to grid size limitations, but introduced the need for additional visibility processing, additional computation for stitching, and a large number memory transfers between the host and GPU. Grid allocation time on the GPU demonstrated a reduction of 75%, but the necessary processing and data transfers outweighed this improvement; further development was not considered. Instead, what followed was an investigation into a more traditional graphics rendering approach to convolutional gridding.

Conceptually, convolutional gridding shares a number of similarities with typical graphics rendering operations. This led to the investigation of gridding with the use of customized vertex and fragment shaders. A primitive vertex/fragment shader program was created, which demonstrated the application of fixed size two-dimensional prolate spheroidal sprites to arbitrary points on a grid. It was found that this approach demonstrated support for grid sizes much larger than that of the compute shader. Additionally, the vertex/fragment shader approach only required several seconds for the GPU to set up the necessary elements to commence gridding.

Experimentation with the rendering pipeline demonstrated that visibilities could be processed as point primitives using custom vertex shader logic. Use of dynamically sized points ensured that each point could define a custom size at runtime; determined by the kernel support for the underlying visibility. The implicit rasterization phase of the rendering pipeline results in fragmentation of each point, for which each fragment would subsequently be processed using custom fragment shader logic. The use of texture based W-Projection convolution kernels also ensured each fragment could be convolved and accumulated using appropriately sampled kernel textures. This design was deemed functional as the foundations of the OpenGL based convolutional gridding algorithm, and was used going forward to create various fragment shading techniques for optimized gridding.

4.1.2 Algorithm Configuration

The HEC gridder does require an observation dependant configuration to be defined before gridding commences. The majority of the parameters listed are typical for gridding algorithms which facilitate W-Projection gridding. The others are unique to the HEC gridder, which are needed for the configuration of texture based convolution kernels. Table 4.1 defines the list of parameters which can be configured, and a brief description of each parameters purpose within the HEC gridder.

Table 4.1: Configurable parameters of the Hall-Ensor-Campbell gridding algorithm

Parameter	**Purpose within the HEC gridder**
Grid Dimension	The size of the grid (one dimension)
Render Width	The width of the rendering subregion
Render Height	The height of the rendering subregion
Texture Size	The size of each kernel texture (one dimension)
Resolution Size	The size of each calculated kernel (one dimension)
Kernel Min Support	The smallest kernel full support
Kernel Max Support	The largest kernel full support
Visibility Count	The number of visibilities to be processed
Visibility Parameters	The number of attributes per visibility
Visibility Source File	The file containing visibilities
Frequency Hz	The frequency of the observation
Fragment Shader Type	The version of HEC fragment shader to be used
Max W	The maximum w term of the observation (absolute)
Cell Size Radians	The angle the sky for each grid cell
Number of Kernels	The number of W-Projection kernels required

Referring to Table 4.1, Grid Dimension specifies the width and height of the regular two-dimensional UV-grid. Render Width and Render Height allow for non-square gridding to be facilitated, much like the NAG gridder. This is optional, and allows for gridding to be performed on a subregion of the full grid, whilst ensuring visibilities are scaled

under the assumption of full gridding conditions. Kernel Min Support refers to the minimum convolution kernel support size for the observation, and Kernel Max Support refers to the maximum support size, respectively. These min/max support sizes assist in the production of accurate W-Projection kernels, and associated sizing calculations during the gridding process.

Visibility Count defines the number of visibilities to be batch processed, and Visibility Parameters defines the number of attributes for each bound visibility. The HEC gridder is configured with a default of six attributes, where each visibility consists of three coordinates (u, v, w), one single precision complex number (intensity/brightness), and one scalar (density). At present, the algorithm processes visibilities from file, which is specified using the Visibility Source File parameter. Frequency Hz defines the frequency at which visibilities have been measured, and is used to scale visibility (u, v, w) coordinates from wave lengths to meters prior to gridding.

Max W defines the maximum absolute w coordinate to be supported during the production of W-Projection kernels. Cell Size Radians defines the size of each grid point as an angle of the sky under observation, and is used to ensure W-Projection kernels are accurate for the angular resolution of the observation. Number of Kernels defines how many W-Projection kernels are to be created prior to gridding. The number of kernels specified only represents the number of positive w coordinate kernels, as the HEC gridder dynamically facilitates negative kernels during convolutional gridding.

Fragment Shader Type defines which generation of fragment shader will be used to conduct the gridding of visibilities. Three variations are supported by the HEC gridder, which are referred to as the *"full", "reflective"*, or *"isotropic"* fragment shaders; these will be presented in detail shortly. Resolution Size specifies the resolution of convolution kernels during their creation. Texture Size defines the size of textured kernels after interpolation is performed during kernel creation. Further details regarding the creation of texture based W-Projection kernels, and the use of Resolution Size and

Texture Size will follow in Subsection 4.1.5 of this chapter.

4.1.3 OpenGL Configuration

OpenGL must be configured to ensure the rendering operations utilized are optimal for gridding. Unless specified otherwise, OpenGL will make default assumptions about the processing of data through the rendering pipeline. The code sample featured in Snippet 1 describes which defaults are overridden to support gridding. Note that the use of `GL_TEXTURE_2D` or `GL_TEXTURE_3D` is dependant on which generation of fragment shader is used to perform gridding.

```
// Disable clamping of RGBA colors
glClampColorARB(GL_CLAMP_VERTEX_COLOR_ARB, GL_FALSE);
glClampColorARB(GL_CLAMP_READ_COLOR_ARB, GL_FALSE);
glClampColorARB(GL_CLAMP_FRAGMENT_COLOR_ARB, GL_FALSE);

// Enable accumulative blending of fragments
glBlendEquationSeparate(GL_FUNC_ADD, GL_FUNC_ADD);
glBlendFuncSeparate(GL_ONE, GL_ONE, GL_ONE, GL_ONE);

// Enabling Vertex Point Size to be defined during gridding
glEnable(GL_POINT_SPRITE);
glEnable(GL_VERTEX_PROGRAM_POINT_SIZE);

// Configuring the texture cube/plane for the minification
// and magnification of texels, and clamping texture bounds
GLenum dim = GL_TEXTURE_2D; // or GL_TEXTURE_3D
GLfloat interp = GL_NEAREST; // or GL_LINEAR
glTexParameterf(dim, GL_TEXTURE_MIN_FILTER, interp);
glTexParameterf(dim, GL_TEXTURE_MAG_FILTER, interp);
glTexParameteri(dim, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(dim, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
// Only setting third dimension when using texture cube
glTexParameteri(dim, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
```

Code Snippet 1: Overriding default OpenGL settings

The first configuration affects the numerical capacity of single precision data between the host and GPU, and when passed between each stage of the rendering

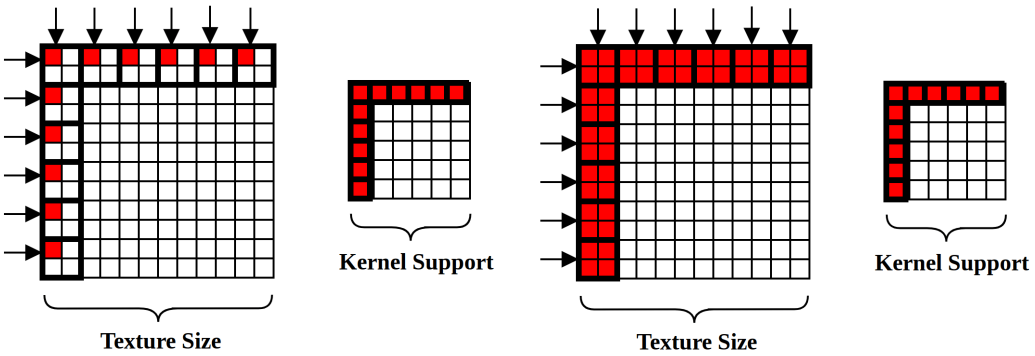
pipeline. Each grid point in the HEC gridder is represented by a four element color vector (RGBA). As such, OpenGL assumes that each grid point will store normalized color data which is limited to the range of $[0.0, 1.0]$. This is suitable for storing colors, but does not facilitate suitable storage of convolved visibilities. This assumption prevents the use of negative values, as well as the accumulation of positive values which exceed 1.0. Removal of this constraint is performed by disabling *color clamping*, effectively increasing the capacity of each element to that of a 32-bit float.

The second configuration affects the behaviour of new fragments. By default, new fragments will naturally substitute existing gridded data at the position of the new fragment within the UV-grid. This results in a loss of data, and prevents any possibility of accumulative gridding of visibilities. However, by specifying the use of `GL_FUNC_ADD` for the `glBlendEquationSeparate` function, OpenGL is forced to accumulate new fragments with existing grid points in the form of blending. Specifying `GL_ONE` for the `glBlendFuncSeparate` function ensures that the full value of each fragments color vector (RGBA) is accumulated.

The third configuration is the enabling of vertex point sprites, and dynamic point sizing. The HEC gridder processes each visibility using custom vertex shader logic, which results in each visibility being treated as a point within the Fourier domain. To convolve a point with zero dimensions into a two-dimensional grid, each point must define its size (diameter) to ensure renderability. This is achieved by defining the `gl_PointSize` attribute for each vertex during vertex shading. A sprite can then be rendered over the rasterized fragments at a later stage of the pipeline. Enabling the use of `GL_POINT_SPRITE` ensures each point of zero dimensions results in some graphically rendered component. Additionally, by enabling `GL_VERTEX_PROGRAM_POINT_SIZE`, each point can define its size from the underlying visibilities kernel support.

Finally, the last configuration specifies how kernel textures will be sampled during gridding. Two texture parameters required are the `GL_TEXTURE_MIN_FILTER` and `GL_TEXTURE_MAG_FILTER`, which define how textures will be scaled and sampled during fragment shading. The HEC gridder supports the use of two filters, `GL_NEAREST` and `GL_LINEAR`. Specifying which filter to use determines the attainable precision by textured kernels during fragment shading, with `GL_NEAREST` performing nearest neighbour sampling, and `GL_LINEAR` performing linear interpolation sampling. The dimensionality of the interpolation performed is dependant on the dimensionality of the kernel texture in use. Thus, three-dimensional textures will perform trilinear interpolation.

Figure 4.2 demonstrates a simplified application of nearest and interpolation based sampling with textures. Nearest sampling selects the nearest available texels (red) relative to the sampling points (black arrows). Whereas bilinear interpolation produces a calculated intermediate texel which is composed from the four (2x2) nearest pre-sampled texels.



(a) Nearest **(b)** Bilinear Interpolation

Figure 4.2: Nearest and bilinear sampling

4.1.4 Data Mapping

Mapping of gridding related data to the GPU requires several OpenGL buffers. The grid is allocated on the host with the necessary dimensions, and is mapped to the GPUs frame buffer as a two-dimensional texture with four element color vectors (RGBA) as each grid point. Binding to the frame buffer ensures the grid can be written to during gridding, and eventually read back to the host once gridding is complete. Use of the `glReadPixels` function transfers the grid back to the host from the frame buffer. Additionally, the `glFinish` function is used to ensure all gridding of visibilities has been completed prior to transfer of the grid.

Visibilities and visibility attributes are bound to the GPU in the form of vertices, using a `GL_ARRAY_BUFFER` object. Two vertex attribute vectors are defined, with each vector storing three visibility attributes. The `location` attribute vector stores the (u, v, w) coordinates, and the `complex` attribute vector stores the complex intensity/brightness and density of each visibility. Figure 4.3 demonstrates how this is arranged in memory as a flat `GL_ARRAY_BUFFER` object.

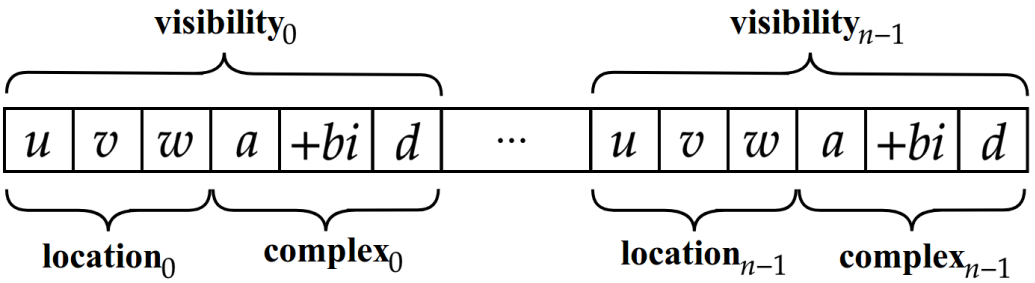


Figure 4.3: Mapping of visibilities and visibility atributes

Lastly, convolution kernels are arranged in memory as a two-dimensional texture plane, or three-dimensional texture cube. Ultimately, this is dependant on which fragment shader is used for gridding. Kernels are of course pre-sampled on the host using the specified gridding configuration prior to the binding of kernels to the device.

4.1.5 Creating Convolution Kernels

The HEC gridder produces high resolution W-Projection kernels to support wide-field imaging, by utilizing the configurable parameters described earlier in Table 4.1. Two-dimensional kernels are produced on the host with the fixed size of each kernel defined as some power of two (2^n); which is defined by the *Resolution Size* parameter. Kernels are then interpolated down to a smaller size (also a power of two) specified by *Texture Size*. This ensures kernels are calculated with high resolution, and are scaled to a suitable texture size whilst retaining resolution. This is discussed further in Subsection 4.3.1.

For the HEC gridder to operate efficiently, the following sizing guideline should be respected: *Resolution Size* ≥ *Texture Size* ≥ *Kernel Max Support*. Following this guideline ensure kernels are created with an appropriate level of precision, and is capable of providing full support coverage for all kernel support sizes defined in the range [*Kernel Min Support*, *Kernel Max Support*]. Violating this recommendation will result in kernels being stretched (magnified) during gridding, which will ultimately result in poor precision. The decision to use texture sizes which are some 2^n was made to ensure effective use of coalesced memory access for quicker loading of texture samples. However, any arbitrary texture size could theoretically be used.

4.2 Gridding Shaders

Once the algorithm and OpenGL context is configured, kernels are pre-sampled, and gridding data are bound to the GPU, the rendering pipeline executes the asynchronous gridding of visibilities. This is performed with the use of two programmable shader stages plugged into the partially fixed function rendering pipeline. The first stage of gridding uses custom vertex shader logic to partially process each visibility in the form

of an OpenGL point primitive. The vertex shader evaluates each visibility to extract additional attributes, which simplifies the workload for the subsequent fragment shader. Implicit rasterization of each point primitive occurs between this stage and the fragment shader. The rasterizer deconstructs the point into a set of n^2 unique fragments, where n represents the kernel support for the underlying visibility. This is followed by the second custom stage of the HEC gridder, which performs the convolution and accumulation of visibility fragments into the UV-grid using custom fragment shader logic.

4.2.1 Shader Uniforms

To ensure convolutional gridding is performed accurately, both custom shaders depend on knowledge of the gridding configuration in use. OpenGL uniforms provide a means for primitive values to be pre-calculated on the host, and bound to both shaders during the creation of the OpenGL program context. Use of pre-calculated uniforms reduces the total amount of work performed by each shader; thus, improving performance. Table 4.2 describes the uniforms used in the HEC gridder, and how they are calculated on the host.

Table 4.2: Common uniforms used within the HEC gridder

Uniform	**Formula**
Grid Center Row	Render Height / 2
Grid Center Col	Render Width / 2
Grid Center Offset Row	1 / Render Height
Grid Center Offset Col	1 / Render Width
Minimum Support Offset	Kernel Min Support
Number of W Planes	Number of Kernels
W Step	1 / Number of Kernels
W Scale	Number of Kernels^2 / Max W
UV Scale	Grid Dimension × Cell Size Radians
W To Max Support Ratio	(Kernel Max Support - Kernel Min Support) / Max W

The first two uniforms, Grid Center Row and Grid Center Col refer to the element of the grid which is considered the center. In theory, the center of the grid should be between the four central texels of the grid texture bound to the frame buffer; OpenGL would support such an operation. However, to make use of the inverse fast Fourier transform to recover the image from the Fourier domain, the grid center must be one specific texel. There is a differentiation between the row and column center as the HEC gridder supports gridding on a non-square regular grid, if desired. The Grid Center Offset Row and Grid Center Offset Col further add to this, ensuring that the middle of each grid texel is considered to be the center of a grid point.

The Minimum Support Offset uniform specifies what the minimum convolution kernel support is for the current observation, and is used to offset the calculation when determining each visibilities kernel support at runtime. The Number of W Planes uniform simply defines how many convolution kernels are bound to the current kernel texture cube or plane. W Step specifies the *distance* between each kernel with respect to normalized kernel indices. Use of the W Scale uniform aids in the calculation of a kernel index for each visibility. W To Max Support Ratio is used in conjunction with the w coordinate of each visibility and the minimum support offset to determine the necessary kernel support. Finally, the UV Scale uniform is utilized in the normalization of each visibilities (u, v) coordinates into the normalized (s, t) coordinate system of the grid.

4.2.2 Vertex Shading

The vertex shader is the entry point of the HEC gridder which begins the convolutional gridding process. This shader is responsible for the partial processing of each visibility, which involves normalizing its position into the grid domain and the extraction of additional attributes to simplify fragment shading. Each visibility passes through this

stage as an individual vertex, and is processed in the form of an OpenGL point primitive. Snippet 2 describes the operations performed by this shader stage.

```
#version 430
precision highp float;
uniform float minFullSupport;
uniform float wToMaxSupportRatio;
uniform float gridCenterRow;
uniform float gridCenterCol;
uniform float gridCenterOffsetRow;
uniform float gridCenterOffsetCol
uniform float wScale;
uniform float wStep;
uniform float uvScale;
uniform float numPlanes;
in vec3 position;
in vec3 complex;
out vec2 fComplex;
out float wIndex;
out float conjugate;
void main() {

   // normalize vis UV coords to ST grid coords
   gl_Position.st = ((position.xy * uvScale)
                   / vec2(gridCenterCol, gridCenterRow))
                   + vec2(gridCenterOffsetCol,
                       gridCenterOffsetRow);

   // calculate the required w-proj kernel index [0.0, 1.0]
   wIndex = sqrt(abs(position.z * wScale))
            * wStep + (0.5 * wStep);

   // calculate the required kernel full support
   float wSupport = abs(wToMaxSupportRatio * position.z)
                   + minFullSupport;

   // determine if vis requires negative w-proj kernel
   conjugate = -sign(position.z);

   // set the kernel full support for visibility
   gl_PointSize = wSupport + (1.0 - mod(wSupport, 2.0));

   // scale the visibility intensity by its density
   fComplex = complex.xy * complex.z;
```

Code Snippet 2: GLSL shader logic for the vertex shader

The first step in processing the vertex is to normalize the position of the visibility into the gridding domain. Coordinates for the visibility are bound into the `position` attribute vector, such that `position(x, y, z)` represents the visibility $V(u, v, w)$.

Snippet 3 this normalization process. The (u, v) coordinates are scaled using the UV Scale uniform and grid render dimension uniforms, defining the position of the vertex in world space coordinates (s, t). The center of each grid point is assumed to be half-way across each grid texel, which is why the calculation utilizes an offset in both directions. The resulting vertex position is normalized to the range of $[-1.0, 1.0]$ on both axes, as required for world space mapping. The coordinates for each visibility $V(u, v, w)$ are pre-scaled from wavelengths to meters prior to vertex shading, which is why it is not necessary to perform during vertex shading.

Unlike the layout of traditional Cartesian coordinate systems, the world coordinate system in the HEC gridder has an inverse s (or v) axis. This means that the four quadrant coordinate system has positive axes in the bottom-right quadrant. This is a result of the default world coordinate system used in OpenGL, and is automatically corrected when reading the grid back from the device using the `glReadPixels` function.

```
21    gl_Position.st = ((position.xy * uvScale)
22                     / vec2(gridCenterCol, gridCenterRow))
23                     + vec2(gridCenterOffsetCol,
24                         gridCenterOffsetRow);
```

Code Snippet 3: Normalizing and scaling the visibility to a position within the grid

The next step is determining which W-Projection convolution kernel is required during fragment shading for the current visibility. Snippet 4 demonstrates the evaluation of the visibilities w coordinate to determine the necessary kernel index. Indices for texture based kernels are normalized to the range of $[0.0, 1.0]$ (referred to as texture coordinate space). Unlike conventional gridding algorithms, rounding is not performed,

as OpenGL will identify appropriate kernel samples based on the texture filtering technique in use.

Indices are *stepped in* slightly; hence the inclusion of the 0.5 constant in Snippet 4. This is needed to ensure textures are evenly sampled across the point primitive, as OpenGL is slightly biased towards the edges of the texture. This *stepping in* of kernel samples on each plane is performed during kernel creation, which is why only the third dimension is affected during this computation.

```
27    wIndex = sqrt(abs(position.z * wScale))
28             * wStep + (0.5 * wStep);
```

Code Snippet 4: Calculating the index for a W-Projection convolution kernel

The visibility is then evaluated for the necessary kernel support. Snippet 5 demonstrates the identification of the support as a relationship between the visibilities w coordinate, the minimum and maximum kernel supports, and the maximum w coordinate supported by the current observation. The resulting kernel support could be either odd or even in size, and is refined at a later stage.

```
31    float wSupport = abs(wToMaxSupportRatio * position.z)
32                   + minFullSupport;
```

Code Snippet 5: Calculating the full support for the convolution kernel

The w coordinate is evaluated once again to determine whether a negative W-Projection convolution kernel is needed. Snippet 6 demonstrates this evaluation, using the Graphics Library Shader Language (GLSL) `sign` function to determine the signage of the w coordinate.

```
35    conjugate = -sign(position.z);
```

Code Snippet 6: Evaluating the visibility for negative W-Projection kernels

Equation 4.1 demonstrates the logic behind this calculation in the form of a piecewise function. In the case that `conjugate` is 0.0, the imaginary component of the texel will be set to zero. This does not introduce error during convolution, as visibilities with a `conjugate` of 0.0 evidently have a w coordinate of 0.0. This indicates that the visibility was obtained with a small field of view, and does not require non-coplanar baseline correction. Therefore, a basic smoothing function is applied as the convolution function.

$$\text{conjugate} = \begin{cases} +1.0, & \text{if } position.z < 0.0 \\ 0.0, & \text{if } position.z = 0.0 \\ -1.0, & \text{if } position.z > 0.0 \end{cases} \tag{4.1}$$

By reusing the support size calculated earlier, the vertex point size can be defined. Snippet 7 demonstrates how the support size is refined further prior to the setting of the point size. Use of the GLSL `mod` function ensures that even support sizes are incremented to the next odd support. This is performed to ensure consistency with texture based convolution kernels, all of which are calculated using odd kernel supports. Setting the size of the point as the required kernel support ensures that OpenGL will produce enough fragments during rasterization to facilitate proper convolution of the visibility.

```
38    gl_PointSize = wSupport + (1.0 - mod(wSupport, 2.0));
```

Code Snippet 7: Setting the vertex point size

Figure 4.4 demonstrates the conceptual idea of gridding via vertex shading. An arbitrary visibility is positioned into the grid by its (u, v) coordinates as a point primitive, and has a defined point size of 5^2.

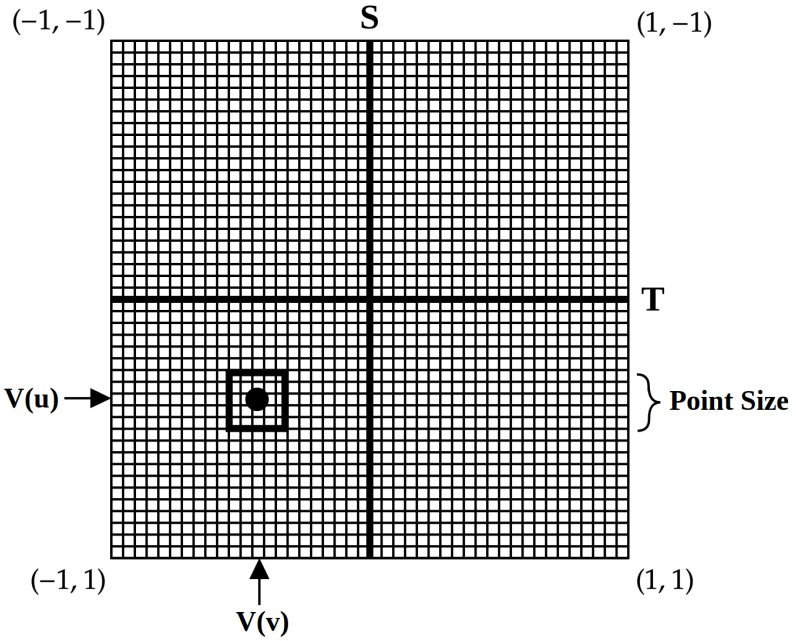


Figure 4.4: Positioning a vertex with a defined point sprite size

Finally, the last step is to scale the intensity (brightness) of the visibility by its bound density. The intensity and density of the visibility is bound into the `complex` attribute vector, such that `complex(x, y, z)` represents the complex intensity and density. Snippet 8 performs the simple scaling operation.

```
41    fComplex = complex.xy * complex.z;
```

Code Snippet 8: Scaling the visibility intensity by its density

At this stage of the gridding process, the visibility is now partially processed. An implicit rasterization stage of the rendering pipeline will follow, performing the segmentation of the point into a number of individual fragments. Each fragment

will be individually processed by the next HEC gridder shader stage. The computed values stored in `wIndex`, `conjugate`, and `fComplex` from the vertex shader will be forwarded on to each invocation of the fragment shading stage to assist in the convolution process. Figure 4.5 demonstrates the conceptual idea of rasterization, where the vertex point from Figure 4.4 has been fragmented into 5^2 unique fragments.

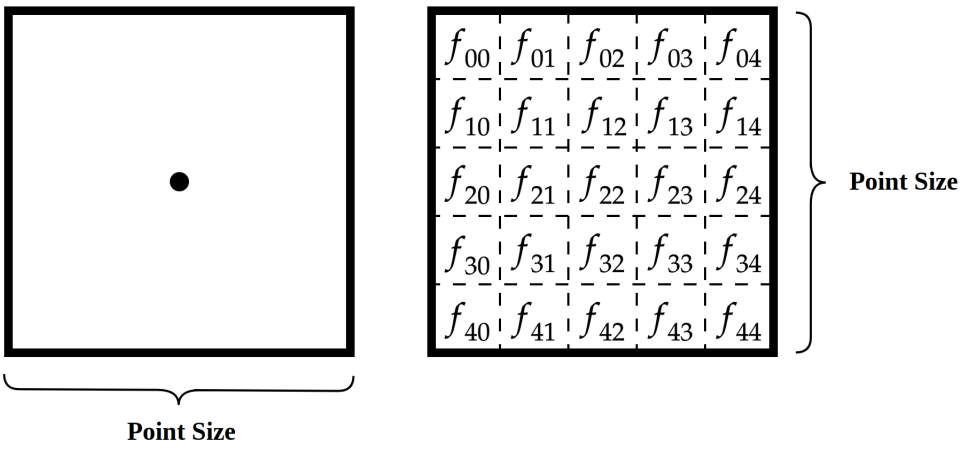


Figure 4.5: Rasterization of a vertex point sprite

4.2.3 Full Texture Fragment Shading

The use of custom fragment shading logic completes the remaining work needed to convolve the visibility into the UV-grid. This stage is invoked for each new fragment created, and utilizes the `wIndex`, `conjugate`, and `fComplex` values calculated earlier for its corresponding point primitive. The fragment shader presented in this section is the first of three fragment shaders implemented for the HEC gridder. It performs convolutional gridding using full sized kernel textures; hence, the informal name of the shader. Snippet 9 describes an overview of the operations performed.

```
#version 430
precision highp float;
uniform sampler3D kernelTex;
in vec2 fComplex;
in float wIndex;
in float conjugate;
void main() {

  // read convolution kernel texel from texture cube
  vec2 kernelLookup = texture(kernelTex,
                      vec3(gl_PointCoord.st, wIndex)).rg;

  // set conjugate of texel imaginary component
  kernelLookup.g = kernelLookup.g * conjugate;

  // bind convolution texel weight to fragment
  gl_FragColor.ra = kernelLookup.rg;

  // complex multiplication of convolution texel
  // and visibility intensity, bind to fragment
  gl_FragColor.gb = vec2(kernelLookup.r * fComplex.x
                    - kernelLookup.g * fComplex.y,
                    kernelLookup.g * fComplex.x
                    + kernelLookup.r * fComplex.y);
}
```

Code Snippet 9: GLSL shader logic for the full texture fragment shader

The first step in processing a visibility fragment is to perform a texture lookup to read a W-Projection kernel sample (texel) from memory. Using the GLSL `texture` function, an appropriate sample can be read from the texture cube bound to the `kernelTex` sampler3D uniform.

The localized coordinates of each fragment (`gl_PointCoord`) are relative to the position of the origin of the point sprite. In the case of the HEC gridder, the origin is the bottom-left corner of the point sprite. As such, this means that the bottom-left fragment of the sprite has an (s, t) coordinate of $(0.0, 0.0)$, and the top-right fragment has the coordinates of $(1.0, 1.0)$. This system simplifies the mapping of texels to fragments, regardless of the actual dimensions of the texture or point.

The pre-calculated kernel index (`wIndex`) is used to locate the required texture, serving as the normalized third dimension index of the cube. OpenGL will use these coordinates to obtain a suitable texel from the cube. Dependant on the texture filter in

use, OpenGL will either select the nearest suitable sample (GL_NEAREST), or perform trilinear interpolation on the nearest eight samples (GL_LINEAR).

```
10   vec2 kernelLookup = texture(kernelTex,
11                               vec3(gl_PointCoord.st, wIndex)).rg;
```

Code Snippet 10: Locating a complex texel from the kernel texture cube

After the texel is stored in kernelLookup, it is manipulated using the pre-calculated conjugate flag. Snippet 11 demonstrates this simple operation. The logic behind this flag has been described earlier in Equation 4.1.

```
14   kernelLookup.g = kernelLookup.g * conjugate;
```

Code Snippet 11: Manipulating the imaginary component of the complex texel

The sample stored in kernelLookup is then assigned to the red and alpha color components of the fragment color vector. This information will eventually be accumulated into the grid during subsequent blending operations, along with the convolved visibility. The purpose of accumulating kernel samples into the grid is to provide supporting information for normalization of the grid, if desired. Snippet 12 demonstrates the simple assignment of the kernel sample.

```
17   gl_FragColor.ra = kernelLookup.rg;
```

Code Snippet 12: Accumulating the convolution kernel texel weight to the grid

The final step of the fragment shading process is to perform the complex multiplication of the visibility (fComplex) and the sample (kernelLookup). This is

performed in Snippet 13, with the result being stored in the remaining green and blue color components of the fragment color vector.

```
21   gl_FragColor.gb = vec2(kernelLookup.r * fComplex.x
22                     - kernelLookup.g * fComplex.y,
23                     kernelLookup.g * fComplex.x
24                     + kernelLookup.r * fComplex.y);
```

Code Snippet 13: Accumulating the convoluted visibility to the grid point

At this stage of the HEC gridder, fragment processing is complete. The fragment which contains the fraction of convolved visibility will be accumulated into the UV-grid using additive blending functions. After this has performed for all fragments of a vertex point, the convolution of the full visibility is effectively complete. Figure 4.6 demonstrates the assignment of fragment vector elements, with the green and blue colors implying complex multiplication.

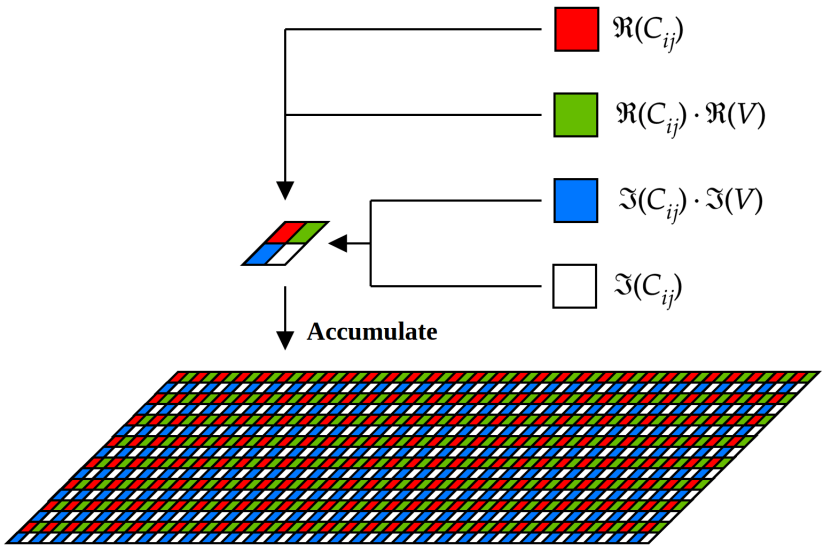


Figure 4.6: Fragment assignment and blending

Figure 4.7 demonstrates the layout of W-Projection kernels in the texture cube (left). The fraction of the convolution kernel obtained from the texture cube is represented

by the red border (right). In this case, the full convolution kernel is obtained entirely from texture lookups and is therefore an unoptimized texturing technique. Further improvements to textured gridding can be implemented, with this shader serving as a template.

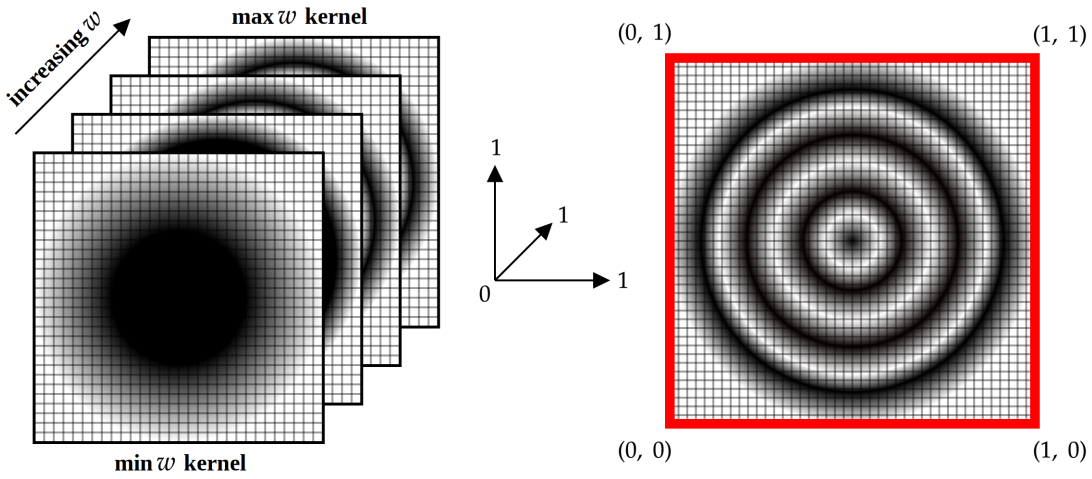


Figure 4.7: The three-dimensional full sized texture cube and its application

4.2.4 Reflective Texture Fragment Shading

The reflective approach to textured fragment shading improves on the previous technique by reducing the texture cube by 75% of its original size. Reflective texture shading achieves the same result, whilst taking advantage of partially symmetric properties observed in W-Projection kernels. Snippet 14 presents a complete overview of the operations performed in the reflective texturing shader. This shader does perform several computations which have been previously discussed in the previous shading technique, and will not be elaborated again for brevity.

```
#version 430
precision highp float;
uniform sampler3D kernelTex;
in vec2 fComplex;
in float wIndex;
in float conjugate;
void main() {

  // calculate position of current fragment relative
  // to center of vertex point
  vec2 coord = abs(2.0 * gl_PointCoord.st - 1.0);

  // read convolution kernel texel from texture cube
  vec2 kernelLookup = texture(kernelTex,
                      vec3(coord.st, wIndex)).rg;

  // set conjugate of texel imaginary component
  kernelLookup.g = kernelLookup.g * conjugate;

  // bind convolution texel weight to fragment
  gl_FragColor.ra = kernelLookup.rg;

  // complex multiplication of convolution texel
  // and visibility intensity, bind to fragment
  gl_FragColor.gb = vec2(kernelLookup.r * fComplex.x
                    - kernelLookup.g * fComplex.y,
                    kernelLookup.g * fComplex.x
                    + kernelLookup.r * fComplex.y);
}
```

Code Snippet 14: Shader code for the HEC gridder reflective texturing fragment shader

As the reflective texture cube only contains one quarter of each kernel, the localized coordinate of the fragment must be reflected, so that it can be mapped to a corresponding texel coordinate. Snippet 15 demonstrates how this transformation is performed.

```
  vec2 coord = abs(2.0 * gl_PointCoord.st - 1.0);
```

Code Snippet 15: Calculating the reflected coordinate of a fragment

After the reflected coordinate is obtained, a texture lookup is performed to read a suitable texel from the quartered texture cube. Snippet 16 demonstrates the lookup of a convolution texel using the reflected coordinate and the pre-calculated `wIndex`.

```
14    vec2 kernelLookup = texture(kernelTex,
15                                vec3(coord.st, wIndex)).rg;
```

Code Snippet 16: Performing the texture lookup to locate a suitable convolution texel

The remaining steps performed are very much the same as observed in the first generation fragment shader; i.e. conjugate manipulation, weight assignment, and the complex multiplication. Figure 4.8 demonstrates the layout of W-Projection kernels in the quartered texture cube (left). The fraction of the kernel obtained from the texture cube is represented by the red border (right). The reflective method presents a more optimized texturing approach, synthesizing a complete kernel with only 25% of the original convolution using a simple coordinate transformation. However, further improvements can still be made.

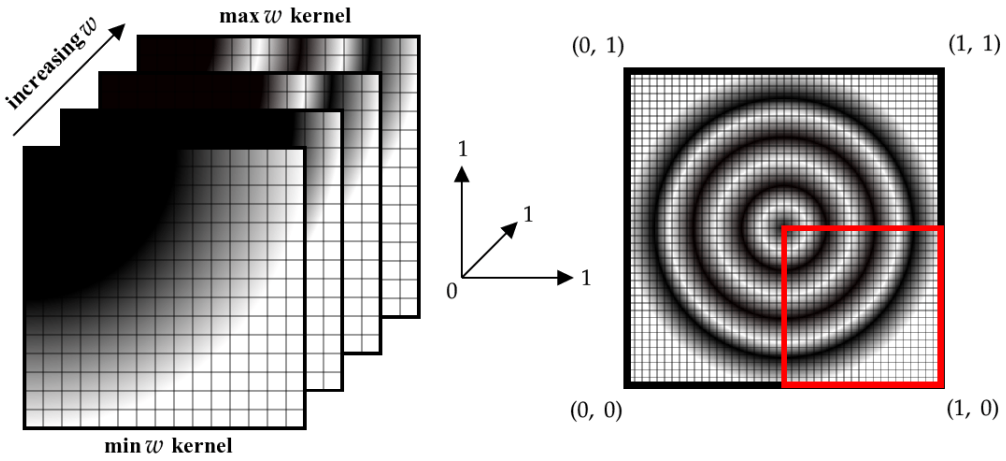


Figure 4.8: The three-dimensional quarter texture cube and its application

4.2.5 Isotropic Texture Fragment Shading

The third and final fragment shading technique presented in this thesis demonstrates convolutional gridding with the use of isotropic (radially symmetric) W-Projection kernels. This approach drastically reduces the amount of texture needed to complete full convolution of a visibility.

It was noted during the literature review that W-Projection kernels are not naturally isotropic. This is because a separable prolate spheroidal is used during the creation of W-Projection kernels, which does not demonstrate isotropic properties. Once the spheroidal is applied during the creation of each W-Projection kernel, the resulting plane loses its separability. This is why one quarter of each kernel is typically used to perform mirrored convolutional of visibilities in modern gridding algorithms. However, a simple modification to the calculation of the prolate spheroidal can be performed to support isotropy. For now, the fragment shading technique will be presented, and the implementation details of isotropic W-Projection kernels will be discussed in Subsection 4.3.2 of this chapter.

By utilizing isotropic kernels, only one half row from the middle of each texture is needed to synthesize a full convolution kernel. The drastic reduction in texture means that a set of kernels can be effectively cached as two-dimensional texture plane. Once again, some of the operations performed during full texture fragment shading are needed, and will be excluded from the discussion. Snippet 17 presents the custom shader logic for the isotropic fragment shading technique.

```
#version 430
precision highp float;
uniform sampler2D kernelTex;
in vec2 fComplex;
in float wIndex;
in float conjugate;
void main() {

  // calculate position of current fragment relative
  // to origin of vertex point
  vec2 coord = vec2(2.0 * gl_PointCoord.s - 1.0,
                    2.0 * gl_PointCoord.t - 1.0);
  vec2 coordSqr = vec2(coord.s * coord.s,
                       coord.t * coord.t);

  // calculate the radius of current position
  float radialLookup = clamp(sqrt(coordSqr.s + coordSqr.t),
                       0.0, 1.0);

  // read convolution kernel texel from texture plane
  vec2 kernelLookup = texture(kernelTex,
                      vec2(radialLookup, wIndex)).rg;

  // set conjugate of texel imaginary component
  kernelLookup.g = kernelLookup.g * conjugate;

  // bind convolution texel weight to fragment
  gl_FragColor.ra = kernelLookup.rg;

  // complex multiplication of convolution texel
  // and visibility intensity, bind to fragment
  gl_FragColor.gb = vec2(kernelLookup.r * fComplex.x
                    - kernelLookup.g * fComplex.y,
                    kernelLookup.g * fComplex.x
                    + kernelLookup.r * fComplex.y);
}
```

Code Snippet 17: GLSL shader logic for the isotropic texture fragment shader

To effectively perform isotropic fragment shading, each fragment must have its local coordinate transformed to ensure accurate texel mapping is performed (see Snippet 18). This transformation of fragment coordinate simplifies the next step, in which the radius of the fragment is determined.

```
11    vec2 coord = vec2(2.0 * gl_PointCoord.s - 1.0,
12                      2.0 * gl_PointCoord.t - 1.0);
13    vec2 coordSqr = vec2(coord.s * coord.s,
14                         coord.t * coord.t);
```

Code Snippet 18: Calculating the position of the fragment

From here, the radius of the fragment is calculated (see Snippet 19). Use of the GLSL `clamp` function ensures that the radius cannot exceed 1.0. The last texel of each kernel has its sample (weight) zeroed out during kernel creation. This ensures that fragments with a radius of 1.0 do not contribute any value into the UV-grid. Thus, ensuring true isotropic convolution is achieved.

```
17    float radialLookup = clamp(sqrt(coordSqr.s + coordSqr.t),
18                         0.0, 1.0);
```

Code Snippet 19: Calculating the clamped radius of the fragment

The radius and pre-calculated `wIndex` is used to perform a texture lookup with the GLSL `texture` function. In this case, only performing a two-dimensional texture lookup in the `kernelTex` sampler2D uniform (see Snippet 20). Use of `GL_LINEAR` texture filtering for this shader results in bilinear interpolation instead of trilinear interpolation, as determined by the dimensionality of the texture.

```
21    vec2 kernelLookup = texture(kernelTex,
22                        vec2(radialLookup, wIndex)).rg;
```

Code Snippet 20: Performing a two-dimensional texture lookup

The remaining steps for isotropic fragment shading are much the same as the full and reflective shader techniques. It is speculated that this fragment shading technique will execute the fastest, as textures are small enough to be efficiently cached. The precision

offered by isotropic textures remains questionable, as this approach to convolutional gridding has not been observed in gridding literature. However, the implications of this technique will become evident during the evaluation of the gridder in the following chapter. Figure 4.9 demonstrates how kernels are stored in the two-dimensional texture plane, with each row corresponding to each unique W-Projection kernel. One can observe just how significantly small the isotropic kernel textures are (red border), relative to the size of a full convolution kernel.

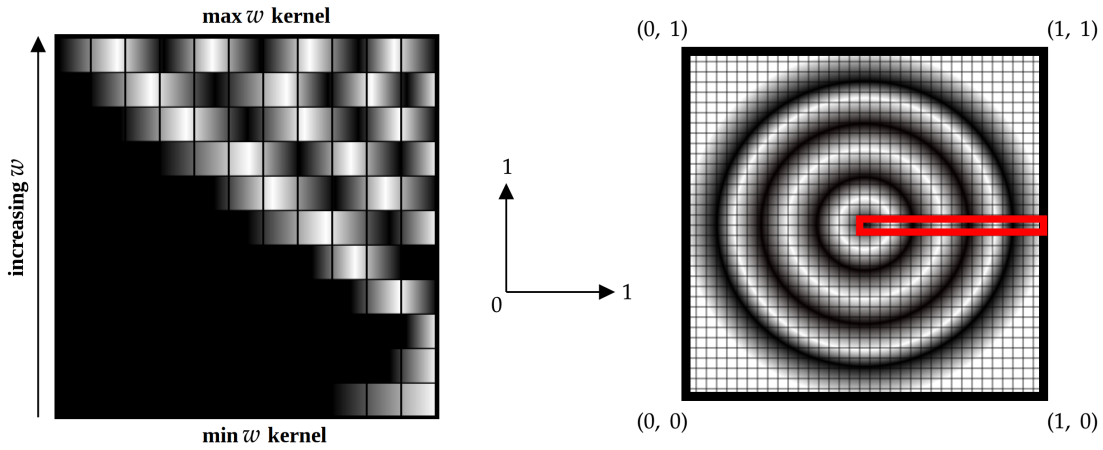


Figure 4.9: Two-dimensional isotropic texture plane and application

4.3 Optimizations

Several optimizations have been identified and implemented which improve the efficiency and effectiveness of convolutional gridding via the graphics rendering pipeline. The use of OpenGL in itself could be argued as an optimization over conventional gridding algorithms, due to the simplicity in which gridding can be performed as a graphics rendering problem. However, the focus of this section is to describe optimizations which boost the effectiveness of graphics based gridding.

4.3.1 Texture Based Convolution Kernels

The HEC gridder supports wide-field imaging through the implementation of textured W-Projection convolution kernels. In typical implementations of the W-Projection algorithm, a set of kernels is produced using a fixed two-dimensional resolution plane, fixed oversampling factor, and an increased w term for each subsequent plane. The intensity of phase correction in each plane is determined by the w term being corrected. This also determines the support size for the underlying W-Projection kernel obtained after the Fourier transform is applied.

The transform typically results in excessive *padding* of the kernel, which must be trimmed away to expose the desired W-Projection convolution. The edges of the oversampled kernel are identified by measuring the two-dimensional resolution from the outside-in. Evaluation of the sample at each point against some minimum threshold determines where the padding ends, and the kernel begins. This is followed by the extraction of the oversampled kernel, which is then normalized and stored in a buffer for use at a later stage. This general process is demonstrated in Figure 4.10, where the oversampled kernel is identified by the red square after the Fourier transform is performed.

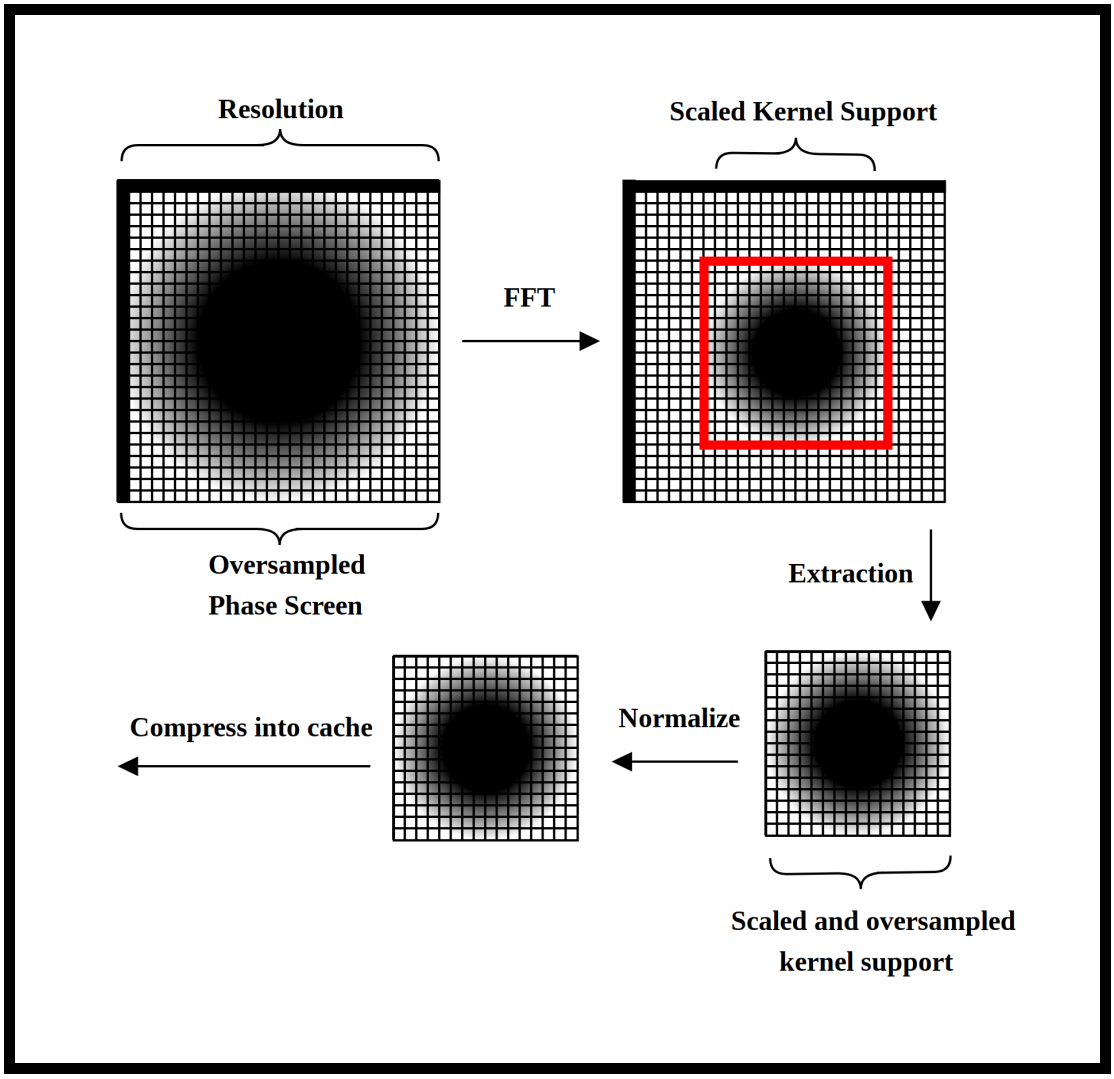


Figure 4.10: Typical creation of W-Projection convolution kernels

The dimensions of each W-Projection kernel are not uniform using this process, and are dependant on the oversampling factor and w term being corrected. As such, the full set of kernels cannot be stored as a uniformly sized cube. This is typically resolved by compressing the set of kernels into a flat block of memory, and using a secondary array to map indices for the first element of each subsequent kernel.

Creating uniformly sized texture based W-Projection kernels for the HEC gridder has required the development of an alternative approach to kernel production. Figure 4.11 demonstrates this difference in creation process.

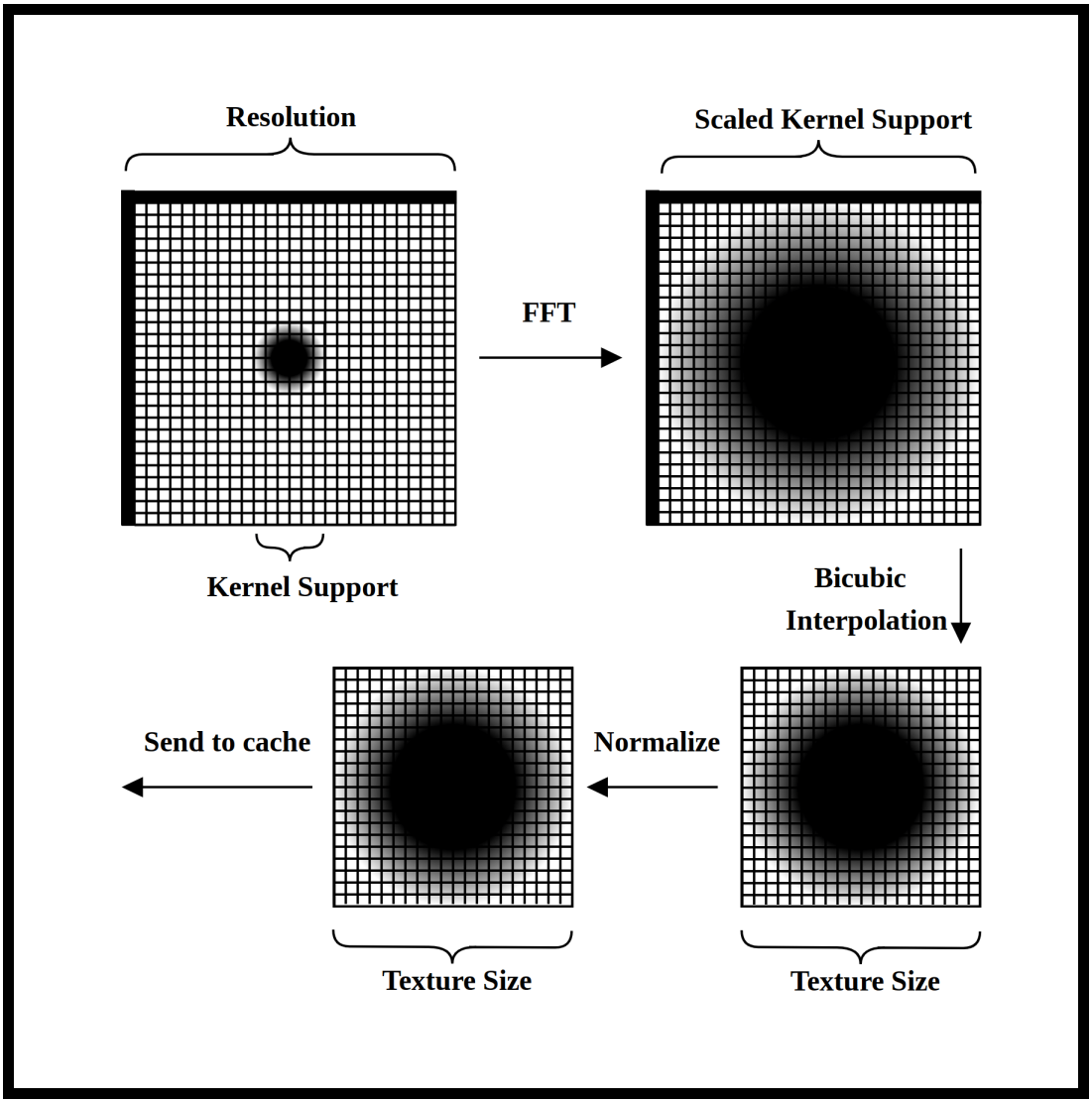


Figure 4.11: Custom process for texture based W-Projection convolution kernels

Each kernel is calculated using a fixed two-dimensional resolution plane, with a defined phase screen sized to match the kernel support of each w term. The time/frequency scaling property of the Fourier transform is used to produce a high resolution W-Projection kernel, which is then scaled down to an appropriate texture size defined during configuration. Bicubic interpolation is used to maintain the integrity of each W-Projection kernel during scaling, as bilinear interpolation would result in jagged approximations of the original W-Projection kernel. This is followed by normalization,

which scales the values of the kernel, such that when the kernel is applied, the W-Projection samples represent a close approximation of a normal unscaled W-Projection kernel. Use of full or reflective texture fragment shading requires the normalization of kernels to be performed using the formula presented in Equation 4.2.

$$\frac{t^2 \ / \ s^2}{\sum_{ij} \mathfrak{R}(K_{ij})} \tag{4.2}$$

Referring to Equation 4.2, t specifies the dimension of the texture for which the kernel will be stored. The s term defines the full support of the current two-dimensional kernel, which varies by the w term being processed. K refers to the current kernel being normalized, and Re refers to the real component of the complex value located at each K_{ij}

Isotropic textures must be normalized using a different formulation, due to the nature in which the texture is applied as determined by fragment radius. Still requiring the use of the t, s and K terms, this normalization factor accounts for the fact that the kernel will be applied radially. Equation 4.3 expresses the normalization factor for isotropic textures, and is only calculated against the half row extracted from the middle of each two-dimensional texture.

$$\frac{t^2 \ / \ s^2}{\sum_{i} 2\pi i \mathfrak{R}(K_i)^2} \tag{4.3}$$

Lastly, each kernel is stored into a flat memory buffer prior to being bound as a two-dimensional texture plane or three-dimensional texture cube in the GPU. The amount of each kernel buffered ultimately depends on the fragment shader in use.

4.3.2 Isotropic W-Projection Kernels

The development of isotropic W-Projection convolution kernels has been performed out of curiosity for its impact towards gridding performance. W-Projection kernels have been previously described as non-separable and are at best partially symmetric. Additionally, W-Projection kernels are not naturally isotropic (*radially symmetric*).

For isotropic W-Projection kernels, and isotropic fragment shading to be feasible, an alternative approach to calculating W-Projection kernels was needed. There is a reliance on a general smoothing function during the creation of W-Projection convolution kernels, with the prolate spheroidal being the function of choice in radio astronomy.

When calculating a one-dimensional prolate spheroidal with s number of samples, or *weights*, an equal distribution of input terms must be defined. An example of this distribution is performed in Equation 4.4, where each input x for a desired prolate spheroidal is calculated over s number of samples. This equation assumes that i represents the current index of each sample s in the range $[0, s-1]$, and that s represents the full support of the spheroidal.

$$x = \left| -1 + i \left(\frac{2}{s-1} \right) \right| \tag{4.4}$$

Each x can then be used to calculate one sample of the prolate spheroidal. Equation 4.5 utilizes each input x to generate one prolate spheroidal weight y. Equation 4.5 assumes that f is a function capable of producing prolate spheroidal samples, such as the prolate spheroidal approximation function implemented by Fred Schwab in the *Casacore* radio astronomy software package (*casacore*, 2015).

$$y = f(x) * (1 - (x)^2) \tag{4.5}$$

Figure 4.12 demonstrates the mapping of inputs (x) to outputs (y) for a one-dimensional prolate spheroidal with a full support of 17. Using this approach, a typical separable, but not symmetric two-dimensional prolate spheroidal can be calculated from the product of two one-dimensional spheroidal weights.

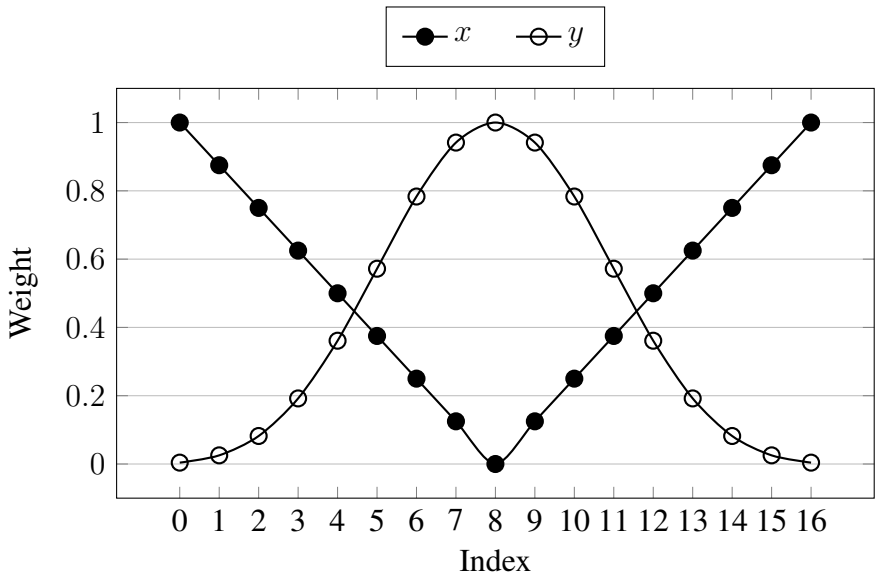


Figure 4.12: Calculating a one-dimensional prolate spheroidal

To implement isotropic W-Projection kernels, prolate spheroidal samples must be calculated using the radius of x across each row (i) and column (j) of the two-dimensional kernel. This is demonstrated in Equation 4.6, where r_{ij} represents the radius of x at row (i) and column (j). The prolate spheroidal sample can be obtained using the same generating function f.

$$\begin{aligned} r_{ij} &= \sqrt{x_i + x_j} \\ y_{ij} &= f(r_{ij}) * (1 - (r_{ij})^2) \end{aligned} \tag{4.6}$$

Performing this calculation for each point in a two-dimensional W-Projection kernel results in a W-Projection kernel with radially symmetric properties. This allows for one half row to be extracted from the middle of each two-dimensional kernel; thus, heavily reducing the memory requirements to store a set of W-Projection kernels. During isotropic based convolutional gridding, the positional radius of each fragment can be determined to map to an isotropic convolution kernel texel.

4.4 Summary

This chapter has introduced, and described the Hall-Ensor-Campbell convolutional gridding algorithm. A overview of the design search process has discussed the idea of using OpenGL compute shaders to produce a *stitch* based gridder. It was discussed as to why this approach was not effective enough for further development, and instead a traditional graphics rendering approach was implemented. A custom vertex shader was demonstrated to be ideal, processing each visibility as a vertex point primitive. Three custom fragment shaders were also shown to complete effective convolution of visibilities. Each fragment shader demonstrated an improved utilization of the graphics rendering pipeline, reducing the amount of memory needed to store sets of kernels, whilst maintaining full kernel integrity. The use of textures and various texture filtering techniques has been presented as a suitable solution for gridding in the graphics rendering pipeline, and will be evaluated in the following chapter. Use of Fourier transform scaling and suitable normalization techniques has shown that variably oversampled kernels can be created and stored into uniformly sized texture cubes or planes. The development of an isotropic fragment shading solution has shown that convolution gridding can be facilitated with only a fraction of each W-Projection kernel. Figures 4.13, 4.14, and 4.15 represent the three gridded datasets using the

Hall-Ensor-Campbell convolutional gridding algorithm. The intensity of red within the images is a result of accumulated kernel samples, and does not indicate intensity of accumulated visibilities. Additionally, the images are not to scale, and are only to demonstrate the gridded result of the three tested datasets.

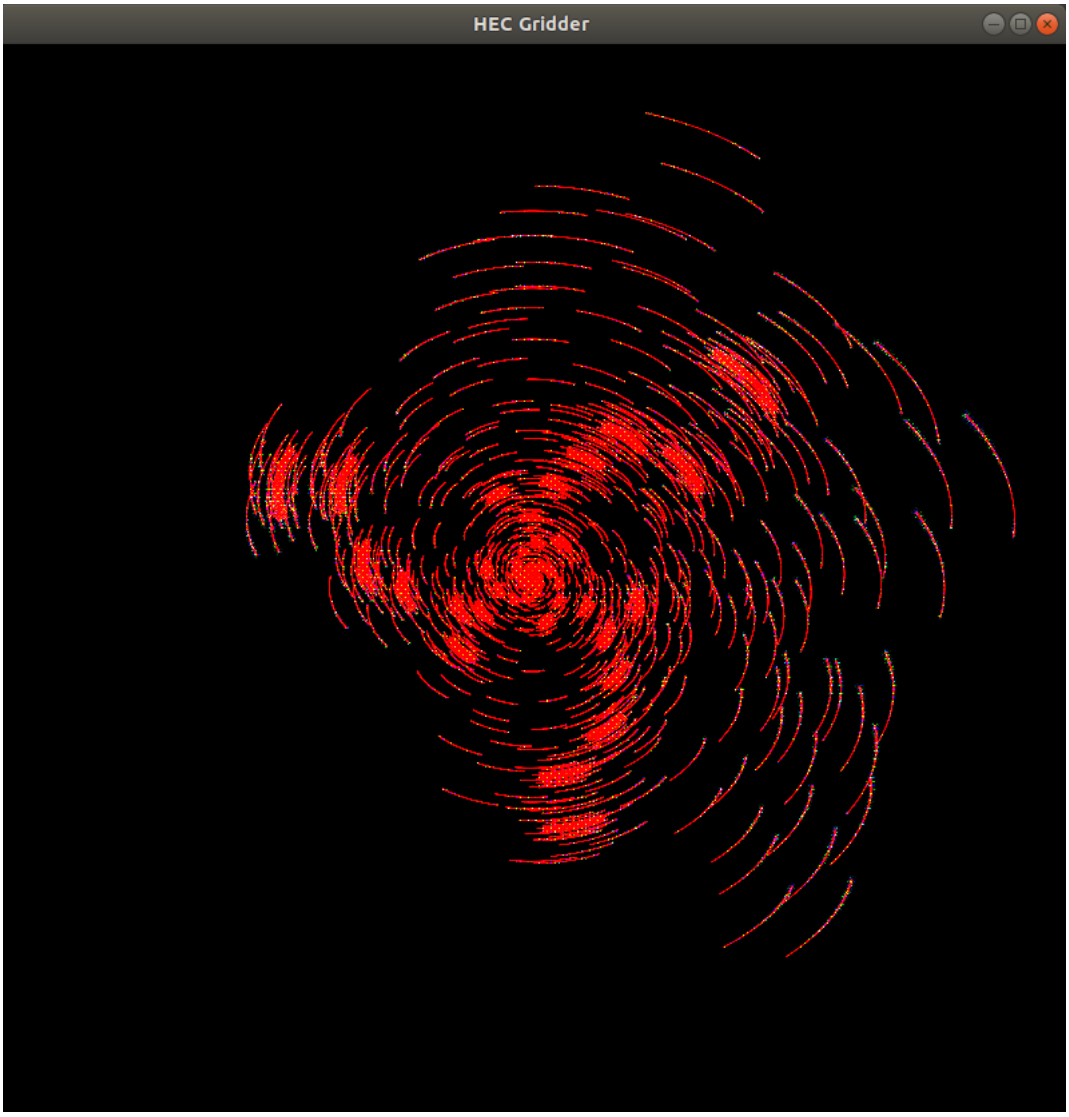


Figure 4.13: Graphically rendered results of the EL82-EL70 dataset

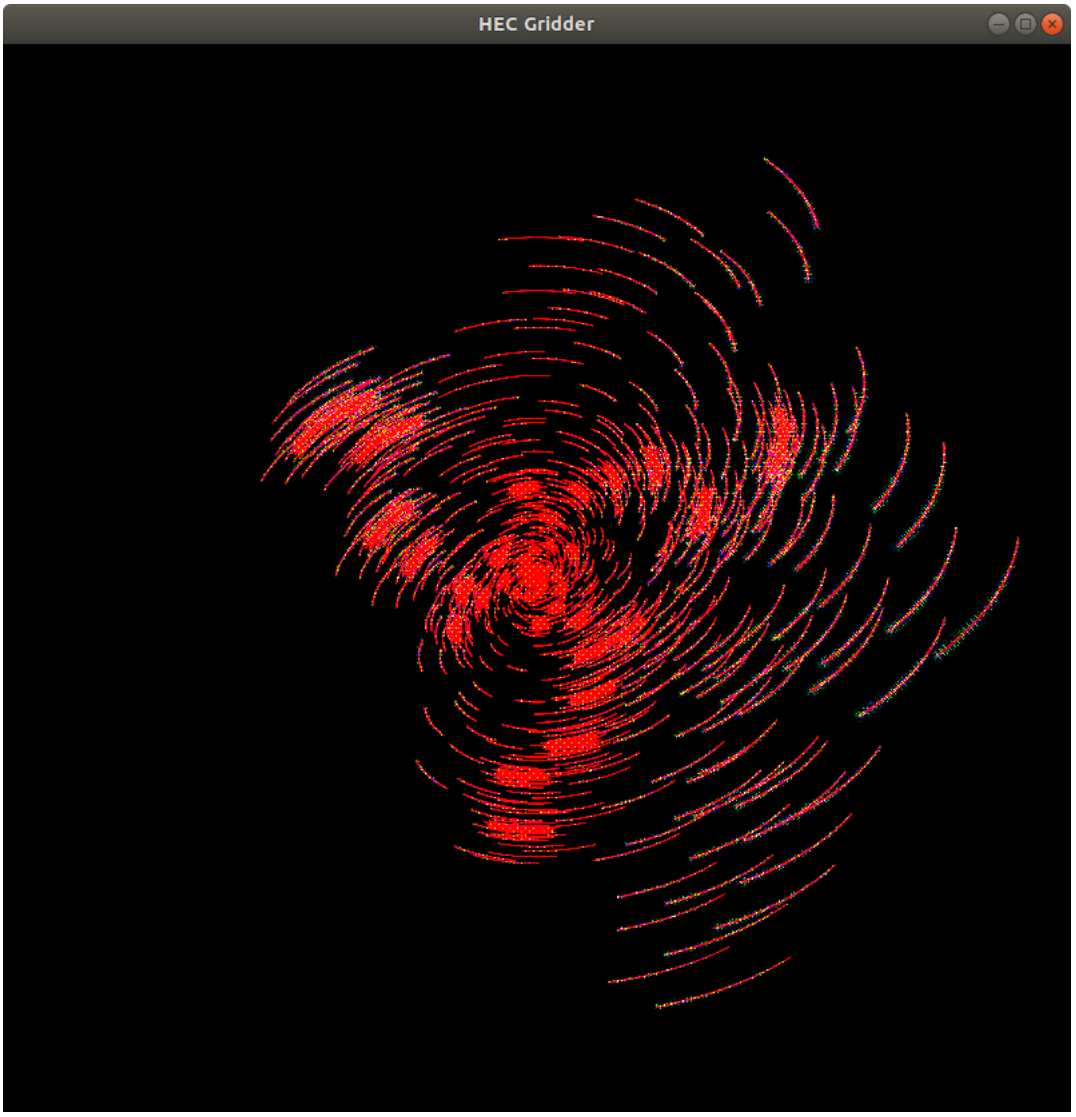


Figure 4.14: Graphically rendered results of the EL56-EL82 dataset

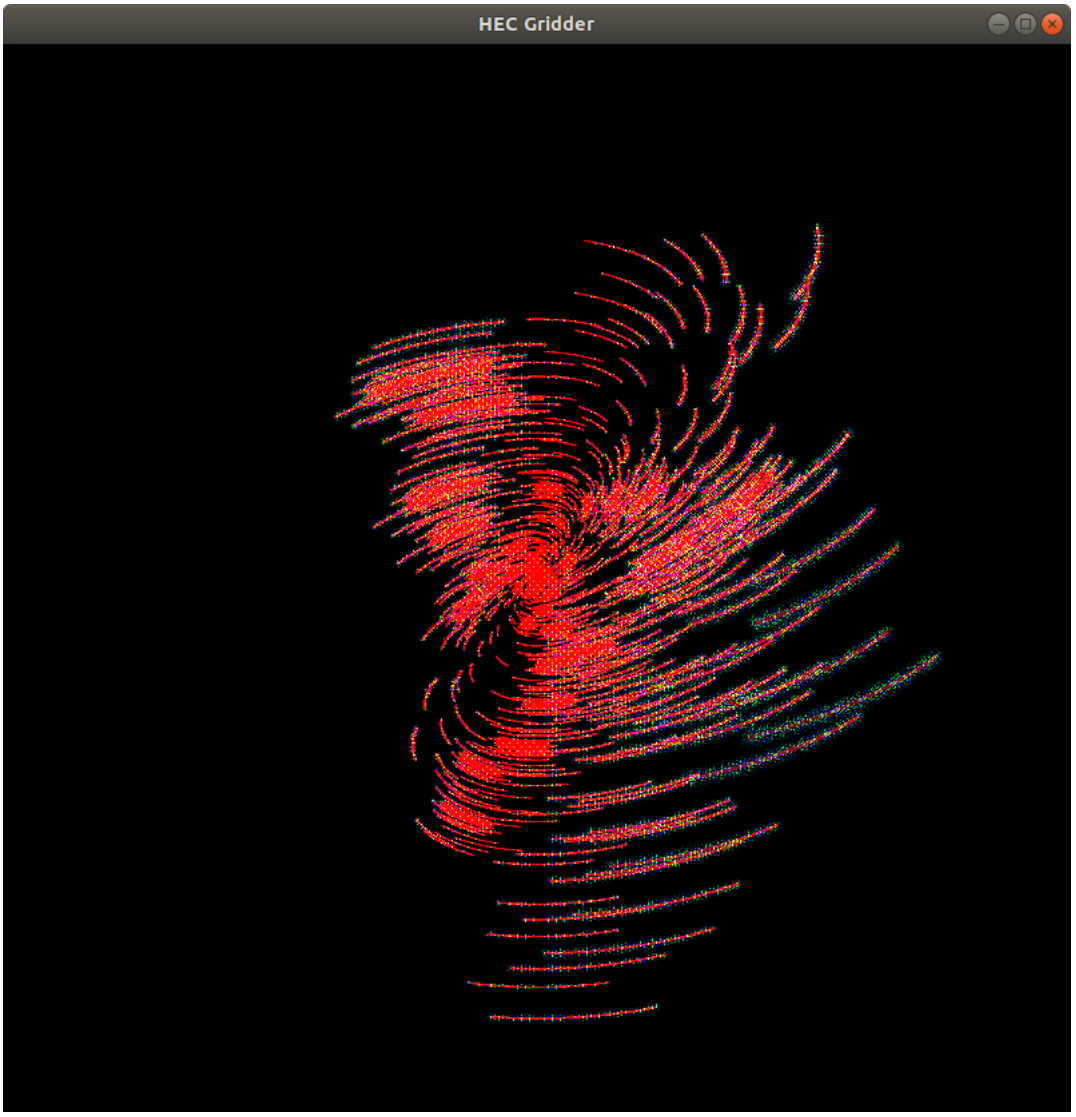


Figure 4.15: Graphically rendered results of the EL30-EL56 dataset

Chapter 5

Algorithm Analysis and Evaluation

This chapter demonstrates and evaluates the Hall-Ensor-Campbell (HEC) convolutional gridding algorithm. The results are used for comparative performance evaluation of the HEC gridder against the Numerical Algorithms Group (NAG) gridding algorithm. This chapter will begin by defining the baseline performance of the NAG gridder. This will be followed by a description of various data transfer rates, and the algorithm configurations used for the demonstration and evaluation of the HEC gridder.

Evaluation of texture based convolution kernels will follow, demonstrating how variable oversampling is achieved using several differing texture sizes. The purpose of processing visibilities as vertices will be briefly covered, and will be followed by a detailed evaluation of the reflective and isotropic fragment shaders. The full textured fragment shader is excluded from evaluation as it is evidently less optimal, and has served its purpose as a functional template for subsequent optimized fragment shaders.

Evaluation of the reflective and isotropic fragment shaders focuses on the performance aspects of gridding described earlier in Section 1.2 of Chapter 1: gridding speed, gridding precision, and memory utilization.

5.1 NAG Gridding Performance

Effective comparison of the HEC gridder against the NAG gridder requires that baseline performance be defined. This was performed by executing the NAG gridder on the same hardware used to develop the HEC gridder (presented earlier in Table 3.3 of Chapter 3).

Table 5.1 presents a number of timings reported by the NAG gridder, which were recorded using standard CUDA timing functions. Each execution of the NAG gridder reports several resulting times (in milliseconds): data transfer to the device, tile preparation, one gridding cycle, and transfer of the grid back to the host.

Table 5.1: CUDA timings for various NAG gridder operations

	EL30-EL56	**EL56-EL82**	**EL82-EL70**
Inputs to GPU (ms)	89	78	66
Tile Preparation (ms)	192	89	58
Gridding (ms)	1178	719	533
Outputs to CPU (ms)	109	110	106
Total Processing Time (ms)	1370	805	591

The *total processing time* represents the sum of the tile preparation phase, and one gridding cycle for each dataset. This time will be used as the baseline for comparison of one gridding cycle using the HEC gridder.

It is important to note that the tile preparation phase is required for the NAG gridder to achieve optimized performance, which includes an unoptimized bucket sorting of visibilities. The NAG gridder does not include or report bucket sorting in their timing operations. Ideally, it should be included as it does contribute additional computational overhead needed to maximize algorithm performance.

Taking into consideration the visibility data rates for the Square Kilometre Array, one can speculate that this would add considerable overhead to the gridding process. However, to keep algorithm comparison relatively fair, the time for bucket sorting will be excluded, and the evaluation will focus primarily on the total processing time.

5.1.1 HEC Data Transfer Rates

Although not the primary focus of gridding performance evaluation, it is useful to know the data transfer rates for various elements of the HEC gridding algorithm, presented in Table 5.2. By having defined data transfer rates, one can calculate how the algorithm will approximately scale for gridding of larger observations or alternative gridding configurations.

The reported times were obtained using an OpenGL Query object to begin and end the timing of scoped blocks of code, with continuous OpenGL polling to determine the completion of issued OpenGL commands. Each reported time was obtained by requesting the measured `GL_TIME_ELAPSED` value from the GPU for each scoped block of commands, and are once again reported in milliseconds.

Table 5.2: Data transfer timings for the HEC gridder

Data to GPU	**EL30-EL56**	**EL56-EL82**	**EL82-EL70**
Visibilities (ms)	79	78	77
Reflect Kernels - 32^2 (ms)	3	2	2
Reflect Kernels - 64^2 (ms)	7	4	2
Reflect Kernels - 128^2 (ms)	14	5	4
Reflect Kernels - 256^2 (ms)	53	35	20
Isotropic Kernels - 32^2 (ms)	3	2	2
Isotropic Kernels - 64^2 (ms)	3	2	2
Isotropic Kernels - 128^2 (ms)	3	2	2
Isotropic Kernels - 256^2 (ms)	3	2	2
Subregion Grid (ms)	21	38	33
Full Grid (ms)	473	462	475
Data to Host			
Subregion Grid (ms)	110	100	130
Full Grid (ms)	3536	3552	3530

It can be seen that transfer rates for the three visibility datasets to the graphics processing unit (GPU) are comparable; this is not unexpected. Reflective kernels have some variation in transfer time, due to the difference in memory required to store larger textures and the number of kernels. Isotropic kernels are relatively consistent regardless of dataset, no doubt due to the miniscule amount of memory needed to store kernels using this technique.

It is important to note that the dimensions of the kernel textures (32^2, 64^2, 128^2, 256^2) do not represent the true dimensions of the bound textures. The specified size describes the upper dimension of each texture plane; that is, the full size if reflected or applied radially with no minification or magnification. This refers to the dimensionality of a singular texture plane in each block of textures, so the true dimensionality of the texture block is actually some defined dimension, squared, and multiplied by the number of W-Projection kernels specific to each dataset. This texture plane notation

is used throughout this chapter when describing various texture sizes used during experimentation.

The *subregion* grids represent a portion of the total grid which is used during convolutional gridding. The point of using subregion grids over the full sized grid is to ensure the evaluation of the HEC gridder is consistent with the NAG gridder. This also means that the subregion grid in the HEC gridder will only store two color components (RG) per grid point instead of the usual four (RGBA). Thus, only convolved complex visibilities will be stored in the grid, and raw convolution kernel samples will be ignored.

Transfer of the subregion grid takes approximately 5% of the time needed to transfer the full grid, which makes sense as it only represents approximately 5% of the full grid ($18,000^2$). Unfortunately, transferring the grid back to the host takes several times longer. In the case of the full grid, this takes approximately 3.5 seconds. This is quite a substantial difference, especially if the Square Kilometre Array grid dimensions are approximately ($65,536^2$). However, this may be considered negligible as the grid is only read back to the host once after the full imaging process is complete (≈ 10 major cycles). It is speculated that the reduction in transfer speed is a result of the `glReadPixels` function being unoptimized, as it would likely be an irregular function to call during traditional graphics rendering; i.e. video games.

It can be seen that the transfer back to the host of the HEC gridder takes considerably longer than that of the NAG gridder, which takes approximately $20 - 40\%$ longer than transfer to the GPU. Data transfer functions in CUDA are of course optimized, as transfers between the host and device are critical to the effectiveness of the CUDA parallel computing model.

5.1.2 Gridding Configurations

To ensure a fair evaluation and comparison against the NAG gridder, the HEC gridder has been configured to be as identical as possible. Table 5.3 describes the configurations used for the three datasets. Note that the HEC gridder will be operating under the assumption of full grid dimensions ($18,000^2$) for the sake of precision, but will only be accumulating visibilities onto a subregion of the full grid. The subregion grid is defined by the render width and render height of each dataset. Several texture sizes will be tested, specifically 32^2, 64^2, 128^2, and 256^2.

Unless otherwise specified, all experiments performed will utilize a full set of positive w convolution kernels specific to each datasets needs. Finally, the use of two OpenGL texture filters will be employed during the evaluation process: GL_NEAREST (nearest-neighbour), and GL_LINEAR (trilinear/bilinear interpolation). Any deviation from the prescribed configuration will be specifically stated.

Table 5.3: Gridding configurations used for the evaluation of the HEC gridder

Parameter	**EL30-EL56**	**EL56-EL82**	**EL82-EL70**
Number of Visibilities	$31,395,840$	$31,395,840$	$31,395,840$
Grid Dimension	$18,000^2$	$18,000^2$	$18,000^2$
Render Width	$3,243$	$3,981$	$4,001$
Render Height	$4,170$	$4,426$	$4,385$
Elements per grid point	2	2	2
Number of Kernels	922	601	339
Min Kernel Support (full)	9	9	9
Max Kernel Support (full)	191	145	89
Resolution Size	512	512	512
Max W	$\approx 19,225$	$\approx 12,534$	$\approx 7,083$
Cell Size (radians)	$\approx 6.0e-6$	$\approx 6.0e-6$	$\approx 6.0e-6$

5.2 Texture based W-Projection Kernels

By utilizing texture based convolution kernels, it is possible to achieve a balance between precision and memory utilization. Referring to Figure 5.1, it can be seen that the use of a fixed texture size for a set of W-Projection kernels results in a variable distribution of oversampling across all kernels. Evidently, the smallest of kernel support sizes benefit the most from this technique.

The majority of visibilities for each dataset tested require a small kernel support (Figure 3.1 of Chapter 3), so their precision is amplified by this technique. However, this does put the small fraction of visibilities with large kernel support needs at a disadvantage. Experimentation with larger texture sizes ultimately improves the number of kernels which benefit from amplified oversampling, and subsequently the number of visibilities which receive higher precision convolution. However, this comes at the cost of additional memory. One might consider the use of interpolation based sampling to supplement the precision for lesser oversampled kernels.

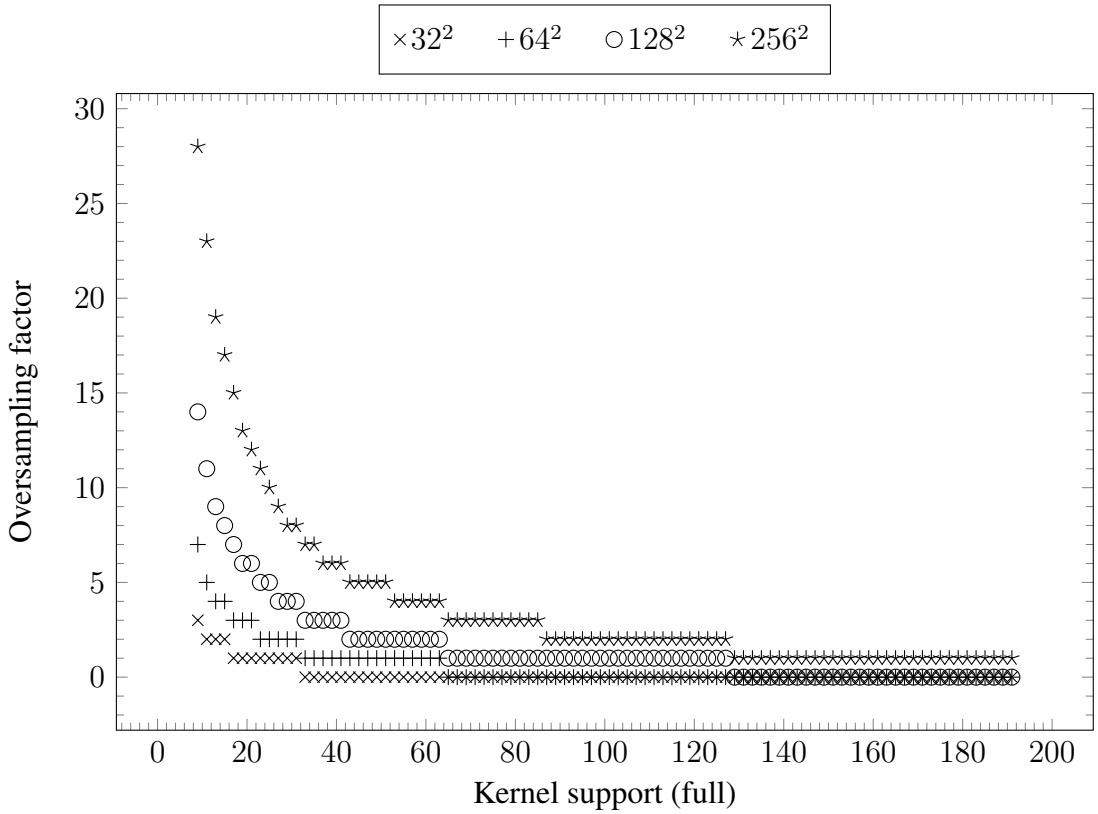


Figure 5.1: Distribution of oversampling for various kernel supports

Table 5.4 describes the percentage of visibilities which receive minimum kernel support when using various texture sizes. Astronomical observations which utilize longer baselines will ultimately require greater breadth of kernel support for W-Projection kernels. This is evident when observing the EL30-EL56 dataset, which requires W-Projection kernel support sizes in the range of [9^2, 191^2]. A texture size of 128^2 is necessary to achieve accurate convolution of at least 99.87% of this dataset. It may not be worth the additional overhead to double the texture size to 256^2 to support the remaining 0.13% of visibilities. However, by utilizing textured based convolution kernels, compromises can be made when balancing the precision and memory utilization during observation.

Table 5.4: Percentage of visibilities achieving full kernel support

Texture Size (per plane)	EL30-EL56	EL56-EL82	EL82-EL70
32^2	77.67%	89.19%	97.49%
64^2	93.98%	98.88%	99.96%
128^2	99.87%	99.99%	100.00%
256^2	100.00%	100.00%	100.00%

5.3 Vertex Shader Processing

Introduced in Chapter 4, a custom vertex shader was described as the first half of the convolutional gridding process of the HEC gridder. Evaluation of each visibility is performed in this stage, resulting in the extraction of additional attributes used to simplify the subsequent fragment shading procedure.

For the uninitiated, it may not be obvious as to why these attributes aren't extracted during fragment shading to simplify the workload of the vertex shader. However, referring to Table 5.5, it is observed that each visibility will invoke the vertex shader once, but will ultimately invoke the fragment shader many times. For the datasets tested, the number of fragment shader invocations is upwards of tens of billions.

Table 5.5: Approximate number of vertex and fragment shader invocations per dataset

	EL30-EL56	EL56-EL82	EL82-EL70
Vertex Invocations	$31,395,840$	$31,395,840$	$31,395,840$
Fragment Invocations	$\approx 30,862,043,928$	$\approx 13,897,957,232$	$\approx 6,459,013,560$

The exact number of fragment shader invocations is dependant on the kernel support of each visibility. Evidently, shifting the evaluation of the visibility into the fragment shader would result in the degradation of performance for the HEC gridder. The increased computational overhead results in more time being spent processing each visibility. Thus, increasing the time in which a complete set of visibilities can be

processed, and increasing the amount of energy needed to achieve the same result. Ultimately, the end goal of the HEC gridder is to process each visibility with as little overhead as possible in order to maximize gridding capabilities for the SKA.

5.4 Reflective Texture Fragment Shading

The reflective texture fragment shader was also introduced in Chapter 4, as a second generation fragment shading technique. This shader has been described as the final stage of the HEC gridder, responsible for the convolution of visibility fragments. This fragment shader is capable of performing full convolution of visibilities with the use of only one quarter of a full convolution kernel. Thus, improving on memory utilization by reducing the total amount of kernel texture by 75%. This was made possible due to the partial symmetric properties of W-Projection kernels. In this section, the reflective texture fragment shader will be demonstrated and evaluated for its efficacy.

5.4.1 Gridding Speed

Gridding speed focuses on the GPU processing time spent during the convolutional gridding of visibilities. Figure 5.2 demonstrates how rapidly each dataset can be processed using reflective texture fragment shading in comparison to the NAG gridder. Note that as mentioned in Subsection 3.2.1 of Chapter 3, standard deviation is not presented due to the insignificant measurement of variance, which approximated between 2-8 milliseconds across the reported results.

It is observed that the two smallest datasets (EL56-EL82, EL82-EL70) are processed in comparable time to that of the NAG gridder, when utilizing the three smallest texture sizes (32^2, 64^2, and 128^2). The largest dataset (EL30-EL56) takes approximately 30% longer to grid than NAG when considering the same texture sizes. This is likely due to

the processing of a greater number of overlapping fragments (observed in Figure 4.15 of Chapter 4) and inefficient texture caching.

Evidently, the use of 256^2 textures results in less adequate performance; taking an additional 30-50% longer to complete the gridding process. Again, this observed penalty will likely be a result of inefficient texture caching due to its larger dimensions.

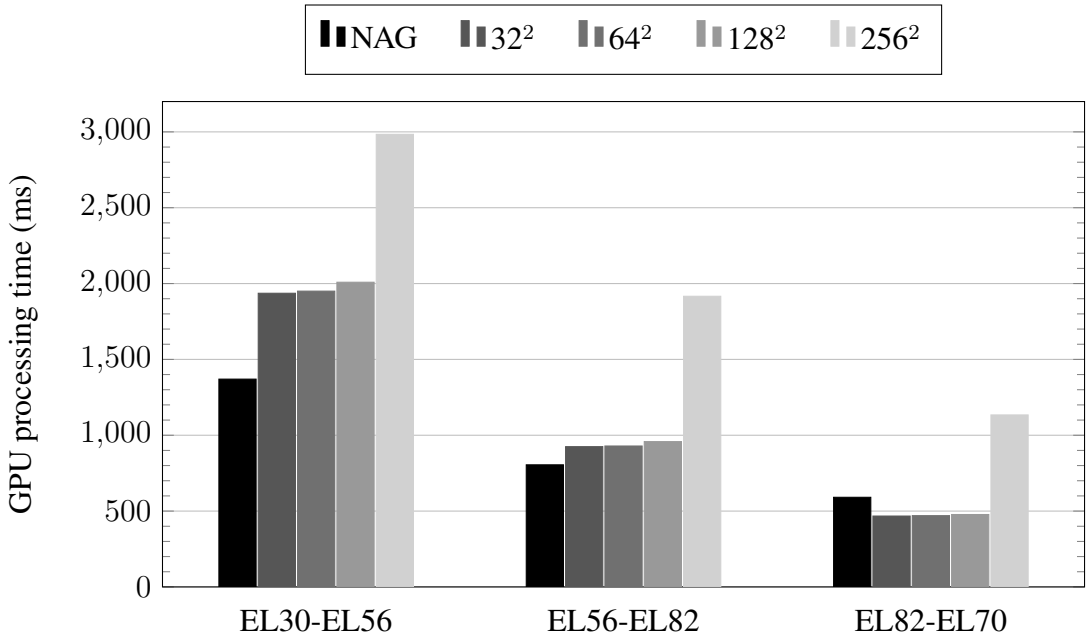


Figure 5.2: Reflective texture gridding time (full set of kernels, nearest-neighbour)

Table 5.6 demonstrates how the reflective fragment shader performs when comparing nearest-neighbour and trilinear interpolation texture filters. Use of trilinear interpolation for 32^2 and 64^2 sized textures demonstrates a negligible penalty of several milliseconds. This is likely a result of effective texture caching due to the memory footprint for small textures. Some reduction in gridding speed is observed for 128^2 textures, taking approximately 20-35% longer to complete. The use of 256^2 textures results in heavily penalization by an average of 56%, which again stems from inefficient caching of larger textures.

The three smaller texture sizes present comparable gridding times to one another,

when utilizing the nearest-neighbour filter. Much like before, the use of 256^2 textures penalizes the throughput of the algorithm. However, not to the extent that interpolation does, as data transfer is needed to obtain a kernel sample from GPU memory.

Utilization of interpolation texture filtering does demonstrate suitability for smaller textures, but heavily penalizes the gridding speed when larger textures are needed. Furthermore, the achieved precision of large interpolated kernel samples may not justify the reduction in gridding speed; this will be evaluated shortly.

Table 5.6: Gridding time using reflective texturing (full set of kernels)

	EL30-EL56		**EL56-EL82**		**EL82-EL70**	
Texture Size (per plane)	Nearest	Trilinear	Nearest	Trilinear	Nearest	Trilinear
32^2	1936ms	1938ms	924ms	927ms	467ms	471ms
64^2	1950ms	1974ms	929ms	932ms	469ms	474ms
128^2	2008ms	2529ms	957ms	1481ms	477ms	709ms
256^2	2983ms	6637ms	1916ms	4380ms	1133ms	2629ms

It is evident that the memory footprint for storing textured kernels has a direct impact on the gridding speed of the algorithm. Therefore, efficient caching of textures is desirable, and can be improved by reducing the number of kernels by some arbitrary amount. The following demonstration reduces the number of produced kernels by 50%, whilst ensuring an equal distribution of w coordinate support over the range of $[0.0, MaxW]$.

Figure 5.3 demonstrates the time in which it takes to perform gridding with 50% less kernels for each dataset. Comparing these results to those of Figure 5.2 (all kernels used), there is a negligible improvement for the three smallest textures sizes. The benefit of using less kernels is only obvious when utilizing the largest texture size. Gridding speed improvements of between 11-16% are achieved by having fewer textures for which to sample.

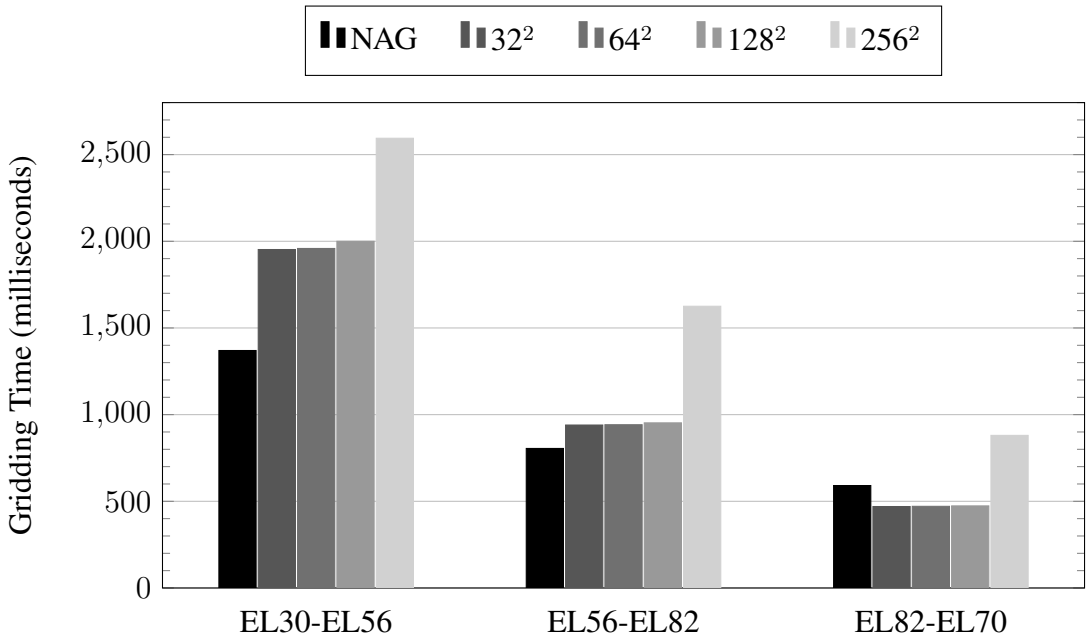


Figure 5.3: Reflective texture gridding time (half set of kernels, nearest-neighbour)

Table 5.7 provides precise measurements of achieved processing times when using 50% less kernels. Gridding speeds are mostly unaffected for smaller texture sizes for either filtering technique. Some penalty is noted for 128^2 textures, which observes a reduction in gridding time by approximately 10-20% when using interpolation sampling. 256^2 textures demonstrate significantly slower gridding time using interpolation compared to nearest-neighbour, which is to be expected with additional computation.

Comparing the results for the 128^2 and 256^2 textures to the figures in Table 5.6 (all kernels), significant improvements to gridding speed are observed. A reduction in gridding time of approximately 300-400 milliseconds for nearest-neighbour, and between 400-900 milliseconds for trilinear interpolation.

Table 5.7: Gridding time using reflective texturing (half set of kernels)

	EL30-EL56		**EL56-EL82**		**EL82-EL70**	
Texture Size (per plane)	Nearest	Trilinear	Nearest	Trilinear	Nearest	Trilinear
32^2	1953ms	1955ms	940ms	941ms	470ms	471ms
64^2	1959ms	1972ms	942ms	946ms	471ms	477ms
128^2	2000ms	2216ms	953ms	1151ms	474ms	588ms
256^2	2594ms	5852ms	1625ms	3902ms	880ms	2215ms

5.4.2 Memory Utilization

The next performance consideration is the effective utilization of memory. This focuses on the amount of memory needed to efficiently store a set of texture based convolution kernels within the GPU. Evidently, this is dependant on the size of the textures in use, and the number of kernels required per dataset.

Table 5.8 describes the kernel memory footprint for each of the three datasets when supporting various texture sizes. Considerable reductions of this footprint are achieved using textures in most cases, when compared to the footprint for the oversampled kernels used by the NAG gridder. The measurements presented in Table 5.8 are calculated under the assumption that a full set of kernels is used. However, one can easily scale these measurements accordingly to the desired fraction of kernels.

Table 5.8: Memory requirements for a full set of reflective convolution kernels

Texture Size (per plane)	**EL30-EL56**	**EL56-EL82**	**EL82-EL70**
32^2	1.80MB	1.17MB	0.66MB
64^2	7.20MB	4.70MB	2.65MB
128^2	28.81MB	18.78MB	10.59MB
256^2	115.25MB	75.13MB	42.38MB
NAG Kernels	280.00MB	88.00MB	16.00MB

5.4.3 Gridding Precision

The last performance consideration to demonstrate and evaluate is the precision offered by reflective texture fragment shading. The level of precision achieved by a gridding solution is critical to the synthesis of high quality images, and could be considered the most important performance aspect. Evaluation of HEC gridder precision is limited to comparison of gridding results between HEC and NAG algorithms under similar conditions. Ideally, perfect imaging is needed to obtain an accurate measurement of gridding precision. However, this was not feasible given the allocation of time to conduct this research, and is planned for future work.

Precision is obtained by measuring the relative error between the HEC and NAG gridding algorithms using the relative error L2 norm described in Chapter 3. One gridding cycle is performed by both algorithms before the measurement of error is obtained. Table 5.9 demonstrates the relative error measured for the three datasets.

Table 5.9: Relative error using reflective texturing (full set of kernels)

	EL30-EL56		**EL56-EL82**		**EL82-EL70**	
Texture Size (per plane)	Nearest	Trilinear	Nearest	Trilinear	Nearest	Trilinear
32^2	5.0712	3.1709	0.1727	0.1830	0.0982	0.1030
64^2	0.1468	0.1576	0.1021	0.1058	0.0728	0.0728
128^2	0.1134	0.1149	0.0861	0.0924	0.0660	0.0654
256^2	0.1044	0.1047	0.0819	0.0821	0.0640	0.0636

As one might expect, the use of larger textures provides the lowest relative error. Nearest-neighbour and trilinear interpolation texture filtering provides a comparable level of error for larger kernel sizes; likely due to the greater number of kernel samples available for nearest-neighbour sampling. The smallest dataset (EL82-EL70) benefits the most from a variety of texture sizes; largely due to the majority of visibilities in

the set needing small kernel support sizes. It is observed that in most cases, trilinear interpolation produces a higher relative error than nearest-neighbour filtering. It is speculated that this is a result of an increased support range for visibility w terms by interpolating between pre-calculated W-Projection convolution kernels, which is not performed by the NAG gridder.

The NAG gridder is limited to using a fixed configuration, which includes a fixed kernel oversampling factor of four. This constraint makes precision evaluation of the HEC gridder somewhat challenging, as the relative error reported may demonstrate that the HEC gridder is worse; when in truth it is only *different*. Therefore, one should interpret the precision results *cum grano salis*; that is, with a grain of salt, until perfect imaging confirms these speculations.

Referring back to Table 5.9, it can be seen that smaller textures hinder the gridding precision of visibilities measured using long baselines. The use of long baselines results in the need for broader kernel support, and it is clear to see that small textures are not ideal for such cases. A prime example is observed when attempting to use a 32^2 sized texture for the EL30-EL56 dataset.

Reducing the number of kernels by 50% has been demonstrated as an effective technique for reducing gridding time. Table 5.10 demonstrates that all datasets, using all texture sizes, and texture filtering techniques incur some minor penality in gridding precision when using half the number of kernels; approximately 5% across the board.

This is an implication of reducing the number of accurately pre-calculated kernels by some degree. Less kernels means a smaller number of precise samples to use during nearest-neighbour sampling. Therefore, trilinear interpolation would be desirable under these conditions to supplement the reduction of available samples.

Table 5.10: Relative error using reflective texturing (half set of kernels)

	EL30-EL56		**EL56-EL82**		**EL82-EL70**	
Texture Size (per plane)	Nearest	Trilinear	Nearest	Trilinear	Nearest	Trilinear
32^2	7.7527	5.3440	0.1745	0.1846	0.1013	0.1056
64^2	0.1507	0.1611	0.1047	0.1084	0.0768	0.0763
128^2	0.1185	0.1197	0.0892	0.0897	0.0703	0.0691
256^2	0.1100	0.1100	0.0851	0.0853	0.0685	0.0674

It could be argued that a greater reduction in the number of pre-calculated kernels could improve gridding throughput, given the small loss in observed gridding precision. However, it is important to note that the underlying correction provided by each unique W-Projection kernel in a set does differ over the range of supported w terms ($[0.0, MaxW]$). Substantially reducing the number of planes means that the difference in correction between consecutive kernels would be greater; thus less likely to provide accurate convolution per given visibility.

5.5 Isotropic Texture Fragment Shading

The isotropic texture fragment shader was the third generation fragment shader introduced in Chapter 4. This fragment shading technique is based on convolutional gridding via radial based texturing, which utilizes isotropic (radially symmetric) W-Projection kernels to achieve convolution. This section will demonstrate and evaluate the effectiveness of the isotropic fragment shading approach towards convolutional gridding.

5.5.1 Gridding Speed

Previously, the reflective fragment shader technique demonstrated a relationship between optimized gridding speed and the memory footprint for a set of W-Projection kernels.

The use of larger textures resulted in improved gridding precision, at the cost of decreased gridding speed.

The isotropic approach to convolutional gridding provides a drastically reduced memory footprint for storing sets of W-Projection kernels (see Table 5.13). Figure 5.4 demonstrates how gridding speed is improved using this technique. Significant improvements are observed, with the smallest dataset (EL82-EL70) being completely gridded in approximately 50% of the time taken by the NAG gridder.

The two larger datasets offer comparable gridding speeds to that of the NAG gridder, with the largest set (EL30-EL56) demonstrating significant improvements when compared to reflective fragment shading results.

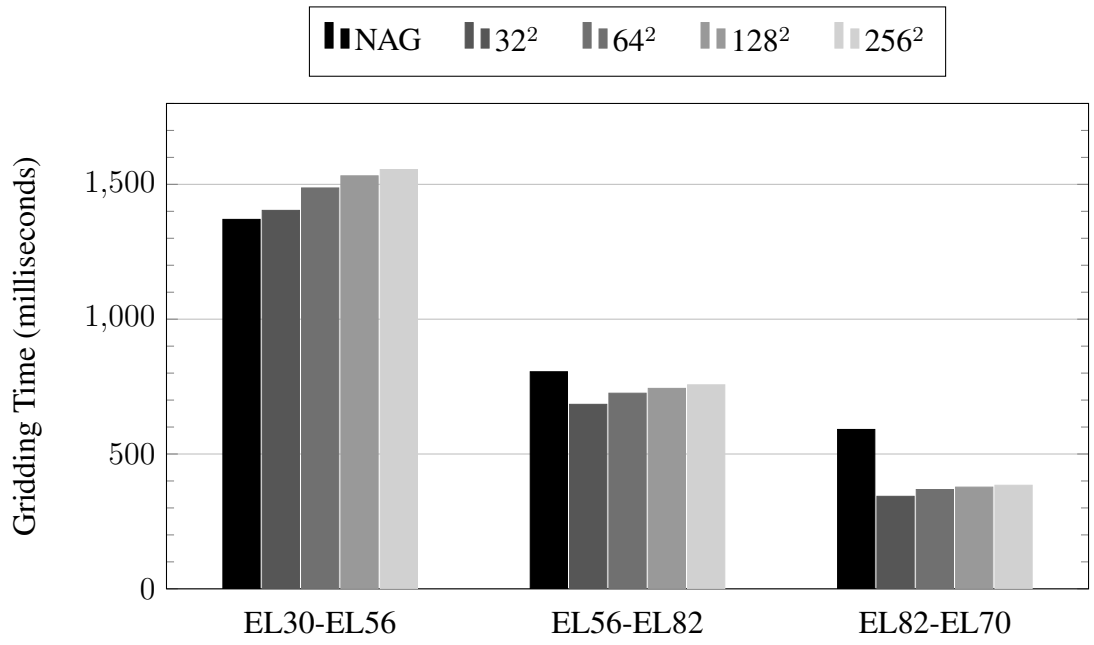


Figure 5.4: Isotropic texture gridding time (full set of kernels, nearest-neighbour)

Table 5.11 demonstrates that that all three datasets are completely gridded in comparable times regardless of the supported texture size, or texture filtering method. Comparing these results to those obtained with the reflective fragment shader, this technique does not suffer from penalized processing speeds when employing 256^2 sized

textures. This approach to fragment shading evidently makes use of efficient caching of textured convolutional kernels.

Table 5.11: Gridding time using isotropic texturing (full set of kernels)

	EL30-EL56		**EL56-EL82**		**EL82-EL70**	
Texture Size (per plane)	Nearest	Bilinear	Nearest	Bilinear	Nearest	Bilinear
32^2	1403ms	1490ms	684ms	724ms	343ms	371ms
64^2	1487ms	1533ms	725ms	742ms	368ms	380ms
128^2	1532ms	1557ms	743ms	754ms	377ms	386ms
256^2	1555ms	1572ms	757ms	764ms	384ms	388ms

However, a reduction in the number of pre-calculated kernels does demonstrate a small penalty towards gridding speeds. Arbitrarily reducing the number of kernels by 50% observes an approximate 1% reduction in speed for large textures, and between 5-7% for smaller textures. Based on the isotropic fragment shading evidence so far, one could speculate that that reducing the number of kernels is an unnecessary operation for this shader. No obvious benefits are obtained, whilst precision is slightly impacted.

Table 5.12: Gridding time using isotropic texturing (half set of kernels)

	EL30-EL56		**EL56-EL82**		**EL82-EL70**	
Texture Size (per plane)	Nearest	Bilinear	Nearest	Bilinear	Nearest	Bilinear
32^2	1420ms	1502ms	695ms	734ms	344ms	369ms
64^2	1503ms	1546ms	733ms	754ms	368ms	382ms
128^2	1548ms	1570ms	755ms	765ms	379ms	386ms
256^2	1571ms	1588ms	767ms	772ms	384ms	389ms

5.5.2 Memory Utilization

Isotropic kernels provide an optimized utilization of memory, as only one half row of each kernel is needed to synthesize a full texture during fragment shading. This

heavy reduction in memory means that a set of kernels can effectively be cached as a two-dimensional texture plane, instead of a three-dimensional texture cube.

Table 5.13 demonstrates how much memory is needed to store a full set of isotropic kernels for each dataset. The hardware used during the development and evaluation of the HEC gridder features a 3 megabyte L2 cache. Thus, any set of isotropic kernels which can be efficiently cached will provide better overall gridding performance.

Table 5.13: Memory requirements for a full set of isotropic convolution kernels

Texture Size (per plane)	**EL30-EL56**	**EL56-EL82**	**EL82-EL70**
32^2	0.11MB	0.07MB	0.04MB
64^2	0.23MB	0.15MB	0.08MB
128^2	0.45MB	0.29MB	0.17MB
256^2	0.90MB	0.59MB	0.33MB
NAG Kernels	280.00MB	88.00MB	16.00MB

It was found that reducing the number of kernels by an arbitrary amount does not supplement isotropic fragment gridding speeds. However, assuming the 3 megabyte limit is respected, a reduction in the number of kernels does free up some memory. This available memory could then be used to support larger texture sizes than those demonstrated, which ultimately improves the oversampling factor per kernel and general gridding precision.

Table 5.14 presents some examples of this, demonstrating a handful of potentially supported texture sizes for the 3 megabyte cache limit. These examples assume a half set of kernels are pre-calculated. Italicized memory sizes represent texture sizes which are feasible without overstepping the upper limits of the cache.

Table 5.14: Upper limits for a half set of efficiently cached isotropic kernels

Texture Size (per plane)	EL30-EL56	EL56-EL82	EL82-EL70
512^2	*0.90MB*	*0.58MB*	*0.33MB*
1024^2	*1.80MB*	*1.17MB*	*0.66MB*
2048^2	3.60MB	*2.35MB*	*1.32MB*

5.5.3 Gridding Precision

Evaluation of isotropic gridding precision was performed using the same technique as the reflection evaluation; relative error is measured between one gridding cycle for both the HEC and NAG gridders under similar conditions. Table 5.15 demonstrates the measured relative error for the three datasets.

Table 5.15: Relative error using isotropic texturing (full set of kernels)

	EL30-EL56		EL56-EL82		EL82-EL70	
Texture Size (per plane)	Nearest	Bilinear	Nearest	Bilinear	Nearest	Bilinear
32^2	4.1951	3.3359	**7.8591**	**6.0533**	0.1039	0.1083
64^2	0.6345	0.5405	0.1080	0.1121	0.0787	0.0788
128^2	0.1199	0.1214	0.0925	0.0932	0.0719	0.0717
256^2	0.1111	0.1113	0.0884	0.0887	0.0700	0.0700

Consistent patterns are noted, such as lower relative error when utilizing larger texture sizes, and interpolation texture sampling demonstrating higher relative error than nearest-neighbour sampling. Comparing this approach to reflective fragment shading (see Table 5.9), it can be seen that isotropic fragment shading demonstrates comparable relative error; averaging approximately 7% worse for the larger two texture sizes.

This increase in relative error is likely a result of applied isotropic kernels when compared to the conventional W-Projection kernels used by the NAG gridder. Being radially symmetric, isotropic kernels provide a smooth convolution across all angles. This is not the case for traditional W-Projection kernels, which are only partially symmetric across the x and y axes. It is to be expected that some accumulative difference will be observed when 32 million visibilities have been isotropically convolved.

There is one obvious discrepancy when using 32^2 isotropic convolution kernels for the two largest datasets (EL30-EL56, EL56-EL82). The relative error reported suggests that the grids produced using these texture sizes are drastically different. This abnormality was noted using reflective textures of a similar size, but only for the largest dataset (EL30-EL56). It is speculated that this tremendous amount of error is due to the poor choice of texture size for a dataset with large kernel support requirements. Such a small texture will negatively impact gridding precision, as the texture will be stretched (magnified) during fragment shading. Thus, resulting in low resolution gridding of visibilities. However, it is unclear as to why this specific dataset, and specific texture size is so heavily penalized.

Ultimately this demonstrates an interesting point. Texture based kernels are useful, but careful analysis of gridding requirements is crucial to ensure an appropriate texture size is chosen. Further analysis is needed to understand exactly what is occurring in this specific situation, and is considered for future work.

Table 5.16 demonstrates the measured relative error when reducing the number of pre-calculated kernels by 50%. Once again, precision is ever so slightly impacted by the reduction in available samples suited for nearest-neighbour texture filtering, and interpolation of samples at runtime. Oddly enough, the precision issue experience by the EL56-EL82 is corrected by the reduction of kernels, adding to the need for further investigation of this specific circumstance.

Table 5.16: Relative error using isotropic texturing (half set of kernels)

	EL30-EL56		**EL56-EL82**		**EL82-EL70**	
Texture Size (per plane)	Nearest	Bilinear	Nearest	Bilinear	Nearest	Bilinear
32^2	6.0707	4.5046	0.2473	0.2322	0.1067	0.1106
64^2	0.8555	0.6619	0.1104	0.1144	0.0824	0.0819
128^2	0.1246	0.1260	0.0954	0.0960	0.0758	0.0751
256^2	0.1162	0.1162	0.0914	0.0916	0.0741	0.0735

5.6 Full Gridding Potential

Evaluation and demonstration of HEC gridder performance has used gridding configurations comparable with those of the NAG gridder; this was performed to ensure a fair evaluation of the HEC gridder takes place. The experiments performed so far have only taken place on a subregion of the full grid, and have only accumulated convolved visibilities at each grid point. During a real astronomical observation, use of a clipped subregion grid would be indicative of a poor gridding configuration. This section will demonstrate the implications towards gridding speed when utilizing a full grid $(18,000^2)$.

Additionally, the HEC gridder supports the accumulation of raw convolution kernel samples during the processing of visibilities. This is achieved by utilizing a four element color vector at each grid point (RGBA); previous evaluations have used two element color vectors to be consistent with the NAG gridder. Storage of convolution kernel samples is useful for normalization of the grid at a later stage. For the sake of brevity, only one dataset (EL82-EL70) will be demonstrated under these conditions, with table 5.17 presenting the results of this demonstration.

Table 5.17: Utilizing the full potential of the HEC gridder - EL82-EL70 dataset

	Reflective		**Isotropic**	
Texture Size (per plane)	Nearest	Linear	Nearest	Linear
32^2	884ms	888ms	643ms	688ms
64^2	887ms	889ms	684ms	707ms
128^2	892ms	937ms	707ms	718ms
256^2	1215ms	2679ms	719ms	723ms

It can be seen that the processing time for gridding of the full dataset roughly doubles when accumulating four values instead of two. This is to be expected, as the amount of memory being transfer per fragment has effectively doubled. The size of the grid does not appear to influence the speed at which gridding can be performed. However, experimentation with much larger grid sizes, such as those needed by the SKA ($65,536^2$) would confirm these speculations.

5.7 Summary

This chapter has presented the evaluation and demonstration of Hall-Ensor-Campbell convolutional gridding performance. Results were presented for gridding speed, precision, and memory utilization of the HEC gridder, which were used to compare against the Numerical Algorithms Group gridder. Use of texture based convolution kernels has demonstrated how variable oversampling can be achieved using various texture sizes.

It was found that partial processing of visibilities via vertex shading drastically reduces the amount of computation which would otherwise be necessary. Use of reflective texture fragment shading has demonstrated comparable gridding performance to that of the NAG gridder in most cases. Isotropic fragment shading has been demonstrated as a feasible solution for maximizing overall gridding performance, by utilizing heavily optimized sets of convolution kernels. Interpolation based sampling of textures has

been shown to improve gridding precision for smaller textures, but is redundant for larger, oversampled textures. Nearest-neighbour sampling is effective, so long as the texture is sufficiently sized to ensure precision is not impacted. Reducing the number of pre-calculated kernels was also found to benefit gridding speeds under some circumstances.

The full potential of the HEC gridder was demonstrated, and was shown to require approximately double the amount of processing time to complete. Various data transfer times have also been presented. It was noted that transferring the grid back to the host using `glReadPixels` was heavily penalized for full grid dimensions; presumably due to the function being unoptimized. However, it was noted that this procedure is only executed once at the end of the imaging process, and may be negligible overhead whilst benefiting from optimized convolutional gridding.

Additionally, it was noted that gridding precision requires further analysis by means of perfect imaging, and is planned for future work.

Chapter 6

Discussion

Demonstration and evaluation of the Hall-Ensor-Campbell (HEC) convolutional gridding algorithm was performed in Chapter 5; the results of the evaluation present evidence to the effectiveness and efficacy of the gridding solution. In this chapter, an interpretation of the findings will discuss the implications of using Open Graphics Library (OpenGL) and the graphics rendering pipeline to facilitate gridding. The two research questions proposed in Chapter 1 will be answered, and the practical significance of the HEC gridder will be presented. This chapter will conclude with an overview of the limitations of the research.

6.1 Graphics based Convolutional Gridding

This thesis has demonstrated how OpenGL and the graphics rendering pipeline can effectively facilitate W-Projection convolutional gridding. Edgar et al. (2010) has suggested that OpenGL based convolutional gridding is effective, but was not able to provide any substantial evidence to its efficacy. Therefore, a custom vertex shader, and several custom fragment shaders have been implemented to demonstrate and support

these claims. The presented solution was shown to be entirely devoid from data pre-processing mechanisms which plague other gridding solutions.

6.1.1 Vertex Processing

The application of custom vertex shading has been demonstrated as a critical step in achieving optimized convolutional gridding via the rendering pipeline. This was made possible by conforming each measured visibility into the form of an OpenGL point primitive. The extraction of attributes during this stage is critical to the precise convolution of the visibility at a later stage of the pipeline, and ultimately reduces the amount of computation needed to fulfil gridding. Additionally, use of a custom vertex shader has demonstrated that visibilities do not require any form of data processing (searching, sorting, compression, or elimination) to achieve high gridding performance. Evidently, gridding via the rendering pipeline would not be possible without the use of vertex shading as a preliminary rendering stage. Its presence contributes to the overall performance of the HEC gridding algorithm.

6.1.2 Reflective Texture Fragment Shading

The reflective texture fragment shader demonstrates a conventional approach to the convolution of each visibility by utilizing partially symmetric W-Projection kernels. This approach to fragment shading presents a reduction in convolution kernel memory by 75%, and synthesizes full kernels by means of reflected fragment coordinates. The findings suggest comparable gridding performance is achieved with this technique, but degrades when utilizing larger texture sizes. This appears to be a result of inefficiently cached kernel textures due to the large amount of memory needed to store each set.

It was found that gridding speed could be improved by reducing the number of pre-calculated kernels by some desired amount, at the cost of some precision. Gridding is by definition an approximation algorithm, so reducing the number of kernels to improve throughput may not be too detrimental to the quality of the final image. Further analysis would determine the implications of this suggestion.

The findings suggest that larger textures benefit most from nearest-neighbour sampling. Gridding precision was reported as comparable, with a considerably shorter gridding time in comparison to interpolation based sampling. It is suggested that smaller textures are utilized with interpolation based sampling, and larger textures are utilized with nearest-neighbour sampling. This ensures precision is maximized, without inhibiting the overall gridding speed of the algorithm when performing reflective fragment shading.

Overall, this shading technique presents a reasonable solution for which gridding can be performed in considerable time. The additional cost of gridding via reflective fragment shading may be worthwhile to save on data processing costs. However, further improvements to this shading technique appear to be somewhat limited. Ultimately, this is because texture lookups are expensive, and this technique relies on such a large amount of texture per convolution kernel which cannot be efficiently cached.

6.1.3 Isotropic Texture Fragment Shading

The implementation of isotropic texture fragment shading has demonstrated a non-conventional approach to convolutional gridding, one which has not yet been observed in gridding literature. The findings suggest that isotropic shading achieves comparable precision to conventional reflective convolution of W-Projection kernels. Memory utilization is heavily optimized by this technique; only requiring one half row of each kernel to synthesize a full convolution. Gridding speeds have demonstrated

drastic improvement over conventional gridding methods, with overall speeds remaining relatively comparable regardless of texture size.

An overall increase in the relative error of this technique was reported. It is speculated that this increase is an indication of difference when using a non-standard technique, and does not necessarily indicate error. The findings suggest that an efficiently cached set of textures provides the best gridding performance, by respecting the capacity of the GPU cache. The findings do suggest that a reduction in the number of pre-calculated kernels would support the use of broader texture sizes. Thus, increasing oversampling of each kernel, and subsequent gridding precision.

Efficient caching of isotropic textures has shown that both nearest-neighbour and interpolation based sampling can be performed in comparable time, and obtain comparable precision. This suggests that either sampling technique could be a valid option, but if one is to reduce the number of kernels by some fraction, interpolation would be ideal to supplement the loss of pre-calculated samples. This is true for smaller texture sizes, but does require further investigation to validate the impact this has on larger texture sizes. Overall, this fragment shading technique appears to be promising. However, further precision analysis is needed to determine if image quality is acceptable when gridded with isotropic form.

6.2 Textured Convolution Kernels

Texture based convolution kernels have been demonstrated as an effective method for the storage and application of kernel samples during convolutional gridding. Romein (2012) had suggested that textures would be a viable alternative to traditional W-Projection kernels, which at present appears to hold true. For the HEC gridder, a custom solution was implemented to create W-Projection kernels suitable for storage as a set of uniformly sized textures. These custom W-Projection textures expose the HEC

gridder to various precision enhancements for smaller kernel sizes; such as variable oversampling and hardware accelerated interpolation. However, further investigation is needed to determine the implications this has towards larger kernels. Romein (2012) also believed that hardware accelerated interpolation could contribute to an improved image quality, especially with the use of oversampled textures (Romein, 2012). This research suggests that whilst the use of texture based convolutional kernels does appear to be promising, further analysis is needed to determine the effects of texture based kernels with respect to gridding precision, and subsequently image quality.

6.2.1 Oversampling

The use of a single fixed size texture for each kernel ensures uniformity when storing kernels in a texture plane or cube. It has been demonstrated that this technique results in a variable oversampling factor for each kernel, which is dependant on the support of each kernel. Evidently, the smaller the kernel support, the higher the oversampling achieved. Figure 5.1 in Chapter 5 demonstrates that kernel textures with larger support sizes do not benefit as much from this approach.

Table 5.4 in Chapter 5 demonstrated that each of the three datasets can sufficiently support the convolution of most visibilities (99%) using different texture sizes per dataset. It was suggested that increasing the texture size to account for the remaining visibilities may not be worth the additional computational overhead. Therefore, careful analysis should be performed when choosing a suitable texture size. The sacrifice of precision for some visibilities may be worth the overall improved gridding speeds.

It is recommended that when selecting a texture size, one should utilize a size which is at least some power of two greater than the largest kernel full support for the observation. This ensures coalesced memory access is achieved, and that all visibilities will receive accurate convolution via minified texture filtering. Failing to adhere to

this recommendation will evidently result in stretched (magnified) convolution kernels, which are likely to reduce gridding precision and introduce artifacts into the synthesized image.

6.2.2 Nearest-neighbour Sampling

Nearest-neighbour sampling has been demonstrated as an effective sampling method when utilizing large textures during convolutional gridding. This is evidently due to the factor of oversampling present in large textures. The findings suggest that nearest-neighbour sampling offers a comparable level of precision to that of interpolation sampling, without the additional computational overhead. It is suggested that nearest-neighbour sampling is less suitable for smaller texture sizes, and should be substituted for interpolation based sampling under these conditions. Additionally, nearest-neighbour sampling may be less suitable when the number of kernels in use is reduced by some degree.

6.2.3 Interpolation Sampling

The ability to supplement pre-calculated convolution kernels using interpolation is one of the major benefits of texture based kernels. By utilizing interpolation based sampling, a finer level of precision is achieved at runtime, at the expense of additional computation.

The findings suggest this sampling method is most effective under two gridding conditions. The first is when small texture sizes are in use, which are effectively under-sampled. This of course depends on the kernel requirements on a per observation basis. The second condition is when the number of kernels in a set has been reduced by some user defined amount (e.g. 50%). When this is performed, the *distance* of the supported w coordinate for each W-Projection kernel is effectively increased. Under these conditions,

interpolation supports the precise convolution of visibilities by interpolating samples from between the two nearest W-Projection planes. It is recommended that interpolation not be used when utilizing larger textures, as comparable precision is achieved using nearest-neighbour sampling without the additional computational overhead.

6.3 Research Questions

Two research questions were proposed the beginning of this thesis (Section 1.2 of Chapter 1). The first question relates to the feasibility of OpenGL and the graphics rendering pipeline to facilitate convolutional gridding; questioning if and how it can be achieved. The second question considers what aspects of graphics based gridding can be optimized to improve algorithm performance.

6.3.1 Convolutional Gridding via Rendering Pipeline

Formally, the first research question proposed was *"how can OpenGL and the graphics rendering pipeline be used to perform convolutional gridding?"*. Evidently, a convolutional gridding algorithm - the HEC gridder - has been designed, developed, and successfully demonstrated using OpenGL and the graphics rendering pipeline to facilitate W-Projection convolutional gridding.

The HEC gridder effectively performs convolutional gridding with the use of custom vertex shader logic to partially process visibilities as individual vertices. Each visibility is processed through the vertex shader, which performs the extraction of additional attributes relative to the visibility. The vertex defines its point size based on the kernel support needs of each visibility, which subsequently determines how many fragments are produced. The additional attributes are transferred to each invocation of a custom fragment shader, which performs the convolution of each fragment using a suitable

W-Projection kernel texel. The resulting fragments are then accumulatively blended into the UV-grid, completing the gridding process.

6.3.2 Optimized Convolutional Gridding

The second research question proposed was *"what optimizations can be implemented to improve the performance of graphics based convolutional gridding?"*. This research has since demonstrated several optimizations which improve graphics based convolutional gridding performance.

Texture based convolution kernels were implemented, which present an effective approach for storage and application of high precision W-Projection kernels. Inclusion of hardware accelerated interpolation supplements pre-sampled kernels by interpolating finer samples at runtime.

Implementation of isotropic W-Projection kernels, and subsequent isotropic fragment shading has shown to improve general gridding performance. Isotropic kernels are efficiently small enough to be cached, which greatly improves gridding speed.

6.4 Practical Significance

Use of the Hall-Ensor-Campbell convolutional gridding algorithm in a practical real-world setting appears feasible. The claims of effective OpenGL based gridding by Edgar et al. (2010) can now be supported. The algorithm has been demonstrated to operate *as advertised*, and appears suitable for use as a convolution gridding solution. The use of OpenGL and the graphics rendering pipeline ensures the algorithm is supported on a broad range of graphics processing hardware, and is not constraint to a specific hardware vendor. Additional processing of visibilities (sorting, searching, compression, et cetera) is non-existent, ideal for achieving efficient and effective gridding with minimal overhead. Use of texture based kernels allows for high resolution kernels to be

applied during convolutional gridding, which scale effectively as needed. The algorithm is highly customizable, and can be configured as desired to achieve a balance between high performance, high precision, or a ratio of the two.

Researchers can benefit from the development of the HEC gridder. It serves as an example of how to solve a complex high performance computing problem using traditional graphics rendering technology. The custom approach to creating W-Projection texture based convolution kernels is also useful should any existing or future gridding implementation wish to implement textured kernels. The texture sampling recommendations also provide guidance for suitable use of various texture filtering techniques. Additionally, the concepts and ideas of the HEC gridder are transferable to other imaging domains. W-Projection was implemented in this thesis, as it is required by the Square Kilometre Array (SKA). However, the general approach of gridding via the rendering pipeline could be extended to others disciplines to simplify, improve, or inspire the development of new gridding solutions.

6.5 Limitations

Although the presented research has fulfilled the aims set out to be achieved and has answered the defined research questions, several limitations of the study have been identified which may impact the presented findings. Some of the limitations identified are simply relative to the complexity of the research topic, and others are due to a lack of available resources, as gridding is a niche research area.

6.5.1 Availability of Gridding Algorithms

Interest in developing highly efficient W-Projection convolutional gridding algorithms has increased in recent years as the requirements for the SKA become apparent. This has lead to the development of various gridding techniques which attempt to maximize

various aspects of gridding performance. Unfortunately the number of optimized W-Projection based gridding algorithms with available source code is extremely scarce. There was not enough time allocated for this research to manually implement several gridding algorithms from the ground up. Thus, only one optimized W-Projection gridding algorithm has been available for comparative analysis of HEC gridder performance.

6.5.2 Availability of Data

There is also a sparse availability of observational datasets suitable for the evaluation of W-Projection based convolutional gridding algorithms. Several published algorithms provide results using datasets which are limited to a fixed kernel size. These datasets are not suitable for the evaluation of W-Projection gridding, which relies on a set of variable sized kernels. Interferometry software packages are available to synthesize data, but this requires specialized radio astronomy background knowledge to operate effectively. Therefore, the three datasets provided by the Oxford e-Research Centre have been the only datasets tested with the HEC gridding algorithm.

6.5.3 Ordering of Visibilities

Adding to the sparsity of available datasets, the three sets used during the testing of the HEC gridder have only been processed using a constant fixed ordering of visibilities. One of the main benefits of the HEC gridder is that visibilities need not be sorted to achieve high gridding throughput. However, this fixed ordering means the HEC gridder has not been evaluated by processing visibilities in different, or *randomized* ordering. The order of the observational data could have some implications on gridding performance.

6.5.4 Availability of Hardware

A limitation of available graphics processing units means the HEC gridder has not been evaluated on a breadth of hardware. The algorithm has only been evaluated using the NVIDIA Titan X (Pascal) GPU. Advanced Research Computing (ARC) of Oxford University was in the process of providing access to two NVIDIA Tesla P100 (Pascal) cards, but unfortunately time ran out for the duration of the research before access was granted. Additionally, the HEC gridder has only been evaluated using NVIDIA hardware of a specific architecture. Advanced Micro Devices (AMD) also supports OpenGL, and Intel are planning to release dedicated GPUs over the next few years. It would be desirable to evaluate the performance of the HEC gridder using AMD graphics cards, as well as other variations of NVIDIA GPU architecture.

6.5.5 Measuring Precision

Reporting on the precision of the HEC gridder has been limited by the measurement of relative error between the HEC gridder and the NAG gridder. True measurement of gridding precision requires the HEC gridder to be evaluated against a perfect sky image, which was not possible in the time allotted for this research. Gridding precision is critical for determining the usefulness of a convolutional gridding algorithm, and this critical step will be conducted as future work.

6.5.6 SKA Comparative Testing

At the time in which this research was performed, there exists no such hardware capable of running a convolutional gridding algorithm at the requirements needed for the SKA. The SKA is expected to require a supercomputer capable of operating at over several hundred petaFLOPS, with supported grid dimensions of approximately $65,536^2$. As such, the HEC gridder has only been evaluated as a downscaled problem of the SKA.

As new hardware becomes available over time, further testing will reveal the scalability of the HEC gridding solution.

6.5.7 Consistent Oversampling Ratio

Use of a single sized texture for a set of W-Projection convolution kernels results in each kernel having a variable factor of oversampling. This demonstrates a larger factor of oversampling for smaller kernels, and subsequently less oversampling for the largest of kernels. This approach hinders the available oversampling factor for larger kernel supports. It is speculated that the use of several variably sized texture planes or cubes could improve the distribution of oversampling and potentially gridding speed. However, this would require considerable modifications to the existing HEC gridding solution. Implementation of multiple textures is considered for future work to provide a fair distribution of oversampling across all W-Projection kernels.

Chapter 7

Conclusion

This research investigated how optimized convolutional gridding could be achieved with the use of Open Graphics Library (OpenGL) and the graphics rendering pipeline. It was hypothesized that convolutional gridding could be simplified, and performance improved by treating gridding as a graphics rendering problem. Furthermore, it was thought that efficient and effective gridding could be achieved by removing any dependency on unnecessary data processing mechanisms, which are typically required by other gridding algorithms to achieve high performance. The aim of this research was to develop a suitable W-Projection convolutional gridding solution for the Square Kilometre Array, in order to help reduce the overall financial costs needed to operate and maintain the upcoming radio telescope. Therefore, the design science research methodology was used to design, develop, and demonstrate an efficient gridding solution in the form of an *artifact*.

Two research questions were proposed in Section 1.2 of Chapter 1:

1. *How can OpenGL and the graphics rendering pipeline be used to facilitate convolutional gridding?*
2. *What optimizations can be implemented to improve the performance of graphics*

based convolutional gridding?

It was speculated that OpenGL compute shaders may be suitable for parallelized work-distribution for the convolution of visibilities. It was also speculated that textures could be useful for optimized kernel sampling via hardware accelerated interpolation. The design search process quickly revealed that compute shaders are inadequate for high performance convolutional gridding. This was partly due to a lack of support for large grid sizes, and excessive loading times. Therefore, no further investigation into a compute shader gridding algorithm was performed, and a traditional graphics rendering approach was implemented.

It was found that convolutional gridding could be achieved using a custom vertex shader for processing of visibilities, and custom fragment shader for convolution and accumulation. It was also found that implementation of isotropic W-Projection kernels meant that gridding could be achieved using radius based fragment shading. This vertex and fragment shader approach to gridding was informally titled the Hall-Ensor-Campbell (HEC) convolutional gridding algorithm.

The performance of the HEC gridder was comparatively measured against a leading convolutional gridding algorithm developed by NVIDIA, Oxfords e-Research Centre, and Numerical Algorithms Group. Select performance criteria were used to evaluate the effectiveness of the HEC gridder, targeting the gridding speed, precision, and utilization of memory for textured kernels.

Two optimized fragment shading techniques were presented in Chapter 4: one utilizing reflective textured fragment shading, and one performing isotropic textured fragment shading. In Chapter 5, it was shown that reflective fragment shading can achieve comparable gridding speeds for two of the datasets tested. The isotropic fragment shader demonstrated an improved gridding speed for all datasets, and appears to be a result of efficiently cached textures.

Implementation of two-dimensional texture planes, and three-dimensional texture cubes demonstrated an effective approach for the application of convolution kernels during gridding. Precision of the HEC gridder was measured using the relative error L2 norm against the comparison algorithm, and demonstrated reduced error with the use of larger texture sizes. Comparable levels of precision were observed using both reflective and isotropic shading. Isotropic fragment shading appears promising but requires further analysis.

Several optimizations were achieved by using OpenGL and the graphics rendering pipeline for convolutional gridding. It was found that smaller texture based kernels benefit from hardware accelerated interpolation, as the dynamic range of the kernels is increased for more precise gridding. It was also found that the number of kernels required for gridding could be reduced by 50% without heavily impacting precision when using interpolation sampling. The implementation of isotropic fragment shading has demonstrated to be an effective gridding technique which heavily reduces the kernel memory footprint, which in turn benefits the processing speed of the gridding algorithm.

The Hall-Ensor-Campbell convolutional gridding algorithm has been demonstrated as an advantageous approach to hardware accelerated W-Projection gridding. It has been shown that gridding is feasible using the graphics rendering pipeline, but does require further analysis to ensure gridding performance meets the standards of the Square Kilometre Array.

7.1 Future Research

The results presented in this thesis demonstrate promise for effective graphics based convolutional gridding. However, additional research is needed to ensure this approach to gridding meets the needs of the Square Kilometre Array. This includes additional performance testing, and enhancements which may further improve on the presented gridding performance results.

7.1.1 Analysis of Scaled W-Projection Kernels

To use W-Projection kernels in the form of uniformly sized textures within the HEC gridder, a non-standard approach to the creation of W-Projection kernels was implemented. These non-standard W-Projection kernel textures appear to be suitable for graphics based convolutional gridding. However, further analysis is needed to understand the impact this technique has towards accurate W-Projection gridding.

7.1.2 Perfect Imaging

To obtain a definitive measurement of convolutional gridding precision, comparison must be performed against a high precision image of the sky. Generating such an image was described as being extremely computationally expensive by means of the perfect imaging algorithm. Perfect imaging has been noted as a critical tool for evaluating gridding precision, but an implementation was not readily available or within scope during the undertaking of this research. A GPU accelerated implementation of the perfect imaging algorithm is to be created[1] for more precise analysis of the HEC gridder as additional research is conducted, and hopefully published in a future research paper.

[1] An implementation of the perfect imaging algorithm was produced post-submission, and revealed approximately 19.25 days worth of continuous computation for each dataset used in this thesis. Finer details have described in Subsection 2.1.3 in Chapter 2

7.1.3 Analysis of Energy

A reduction of maintenance and running costs for the Square Kilometre Array is highly desirable. An effective convolutional gridding algorithm should leverage as much of the hardware as possible to ensure minimal energy wastage. This is an additional factor for consideration when measuring algorithmic performance. Power consumption for the HEC gridder was not included in the performance characteristics reported in this thesis, but is necessary to determine the overall optimality of the solution.

7.1.4 Small Texture Sampling

It was noted that the use of small textures result in poor gridding precision when wide field of view imaging is needed. The reported relative error indicated that the resulting grids were drastically different between the NAG and HEC gridding implementations. Some relative error is to be expected due to poor choice of texture size with respect to kernel support requirements, but not to the excessive levels observed. Further investigation into the use of small textures is needed to understand the limitations and implications of magnified textures.

7.1.5 Multiple Kernel Textures

Use of single sized textures for convolutional kernels has been demonstrated as feasible. Variable oversampling is achieved with this approach, providing a higher factor of oversampling for kernels which require small support. It is speculated that the use of multiple texture planes and cubes of various sizes would support a more equalized distribution of oversampling. Therefore, future enhancements to the HEC gridder will see multiple textures being implemented for a better distribution of oversampling.

7.1.6 Vulkan Implementation

Khronos Group have released a new graphics rendering and 3D compute application programming interface (API) as of 2016, referred to as the *Vulkan* API. Vulkan has been advertised as the *"next generation OpenGL for modern GPUs"*, providing much lower level control for all aspects of GPU accelerated applications over the current OpenGL API. Several benefits of Vulkan are already of interest, including low energy usage and the ability to perform background computation; thus, removing the dependency on connected displays for the HEC gridder. It is also of great interest to know whether HEC gridding performance can be improved with the use of Vulkan, and is worth investigating further.

References

algorithm-reference-library. (2018, Jun). Retrieved from https://github.com/SKA-ScienceDataProcessor/algorithm-reference-library

Altgelt, C. A. (2005). The world's largest "radio" station. *accessed from internet*.

Askap. (n.d.). Retrieved from http://www.scienceimage.csiro.au/images/cache/detail/976_0_BU2161.jpg

Awad, A. I. & Baba, K. (2011). An application for singular point location in fingerprint classification. In V. Snasel, J. Platos & E. El-Qawasmeh (Eds.), *Digital information processing and communications* (pp. 262–276). Berlin, Heidelberg: Springer Berlin Heidelberg.

Baskerville, R. L. (1999). Investigating information systems with action research. *Communications of the AIS*, *2*(3es), 4.

Beatty, P. J., Nishimura, D. G. & Pauly, J. M. (2005, June). Rapid gridding reconstruction with a minimal oversampling ratio. *IEEE Transactions on Medical Imaging*, *24*(6), 799-808. doi: 10.1109/TMI.2005.848376

Brouw, W. (1975). Aperture synthesis. In *Image processing techniques in astronomy* (pp. 301–307). Springer.

casacore. (2015, Mar). Retrieved from https://github.com/casacore/casacore

Compute work distribution. (n.d.). Retrieved from https://qph.fs.quoracdn.net/main-qimg-4b8b0a8dd36c3b14de0d6d800f3644cd-c

Cooley, J. W. & Tukey, J. W. (1965). An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, *19*(90), 297–301.

Cornwell, T. J., Golap, K. & Bhatnagar, S. (2008, Oct). The noncoplanar baselines effect in radio interferometry: The w-projection algorithm. *IEEE Journal of Selected Topics in Signal Processing*, *2*(5), 647-657. doi: 10.1109/JSTSP.2008.2005290

Cpu versus gpu. (2014, August). Retrieved from http://www.scalescale.com/wp-content/uploads/2014/08/cpu-vs-gpu11.png

Du Toit, J. (2017). *Convolution gridding on cpu, gpu, and knl*. Retrieved from http://ska-sdp.org/sites/default/files/attachments/sdp_memo_36.pdf

Edgar, R., Clark, M. A., Dale, K., Mitchell, D. A., Ord, S. M., Wayth, R. B., ... Greenhill, L. J. (2010). Enabling a high throughput real time data pipeline for a large radio telescope array with gpus. *Computer Physics Communications*, *181*(10), 1707–1714.

Fft basics. (n.d.). Retrieved from https://www.nti-audio.com/portals/0/pic/news/FFT-Time-Frequency-View-540.png

Gpu-gridding. (2017, Sep). Retrieved from https://github.com/OxfordSKA/GPU-gridding

Gregor, S. & Hevner, A. R. (2013, June). Positioning and presenting design science research for maximum impact. *MIS Q.*, *37*(2), 337–356. Retrieved from https://doi.org/10.25300/MISQ/2013/37.2.01 doi: 10.25300/MISQ/2013/37.2.01

Gridding datasets. (n.d.). Oxford e-Research Centre. Retrieved from http://oskar.oerc.ox.ac.uk/gridding_challenge/gridding_data.zip

Hall, S. (2014). *Gpu accelerated feature algorithms for mobile devices* (Unpublished doctoral dissertation). Auckland University of Technology.

Harrison, O. & Waldron, J. (2009). Efficient acceleration of asymmetric cryptography on graphics hardware. In *International conference on cryptology in africa* (pp. 350–367).

Hecgridder. (2017, July). Retrieved from https://github.com/sorgenskammer/HECGridder

Hevner, A. R., March, S. T., Park, J. & Ram, S. (2004, March). Design science in information systems research. *MIS Q.*, *28*(1), 75–105. Retrieved from http://dl.acm.org/citation.cfm?id=2017212.2017217

Högbom, J. (1974). Aperture synthesis with a non-regular distribution of interferometer baselines. *Astronomy and Astrophysics Supplement Series*, *15*, 417.

Hogg, D., Macdonald, G., Conway, R. & Wade, C. (1969). Synthesis of brightness distribution in radio sources. *The Astronomical Journal*, *74*, 1206–1213.

Humphreys, B. & Cornwell, T. (2011). Analysis of convolutional resampling algorithm performance. *SKA Memo*, *132*.

Iivari, J. & Venable, J. (2009). Action research and design science research-seemingly similar but decisively dissimilar. In *Ecis* (pp. 1642–1653).

Inverse direct fourier transform. (2019, Apr). Retrieved from https://github.com/ska-telescope/CUDA_InverseDFT

Jackson, J. I., Meyer, C. H., Nishimura, D. G. & Macovski, A. (1991, Sep). Selection of a convolution function for fourier inversion using gridding [computerised tomography application]. *IEEE Transactions on Medical Imaging*, *10*(3), 473-478. doi: 10.1109/42.97598

Kaiser, J. F. (1966). Digital filters. *Ch*, *7*, 218–285.

Kessenich, J., Sellers, G. & Shreiner, D. (2016). *Opengl programming guide: The official guide to learning opengl, version 4.5 with spir-v*. Pearson Education. Retrieved from https://books.google.co.nz/books?id=vUK1DAAAQBAJ

Landau, H. J. & Pollak, H. O. (1961). Prolate spheroidal wave functions, fourier analysis and uncertainty—ii. *Bell System Technical Journal*, *40*(1), 65–84.

Lena söderberg. (n.d.). Retrieved from https://upload.wikimedia.org/wikipedia/en/7/7d/Lenna_%28test_image%29.png

Lofar. (2017, April). Retrieved from lofar.ie/wp-content/uploads/2017/04/superterp.jpg

Luebke, D. (2008, May). Cuda: Scalable parallel programming for high-performance scientific computing. In *2008 5th ieee international symposium on biomedical imaging: From nano to macro* (p. 836-838). doi: 10.1109/ISBI.2008.4541126

Mathur, N. (1969). A pseudodynamic programming technique for the design of correlator supersynthesis arrays. *Radio Science*, *4*(3), 235–244.

McKay, J., Gelb, A., Monga, V. & Raj, R. (2017). Using frame theoretic convolutional gridding for robust synthetic aperture sonar imaging. *arXiv preprint arXiv:1706.08575*.

meerkat. (2017, October). Retrieved from https://www.ska.ac.za/wp-content/uploads/2017/10/2017_meerkat_01-1030x578.jpg

Merry, B. (2016a). Approximating w projection as a separable kernel. *Monthly Notices of the Royal Astronomical Society*, *456*(2), 1761-1766. Retrieved from http://dx.doi.org/10.1093/mnras/stv2761 doi: 10.1093/mnras/stv2761

Merry, B. (2016b). Faster gpu-based convolutional gridding via thread coarsening. *Astronomy and Computing*, *16*(Supplement C), 140 - 145. Retrieved from http://www.sciencedirect.com/science/article/pii/S2213133716300476 doi: https://doi.org/10.1016/j.ascom.2016.05.004

Muscat, D. (2014). High-performance image synthesis for radio interferometry. *arXiv preprint arXiv:1403.4209*.

NASA. (2007). *Electromagnetic spectrum properties.* Retrieved from https://upload.wikimedia.org/wikipedia/commons/thumb/c/cf/EM_Spectrum_Properties_edit.svg/800px-EM_Spectrum_Properties_edit.svg.png

Nvidia tesla p100. (2016). Retrieved from https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

Oskar. (2013, Apr). Retrieved from https://github.com/OxfordSKA/OSKAR

O'sullivan, J. (1985). A fast sinc function gridding algorithm for fourier inversion in computer tomography. *IEEE Transactions on Medical Imaging*, *4*(4), 200–207.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E. & Phillips, J. C. (2008, May). Gpu computing. *Proceedings of the IEEE*, *96*(5), 879-899. doi: 10.1109/JPROC.2008.917757

Peffers, K., Tuunanen, T., Rothenberger, M. & Chatterjee, S. (2007, December). A design science research methodology for information systems research. *J. Manage. Inf. Syst.*, *24*(3), 45–77. Retrieved from http://dx.doi.org/10.2753/MIS0742-1222240302 doi: 10.2753/MIS0742-1222240302

Remote off-screen rendering with opengl. (2014, Jul). Retrieved from https://arrayfire.com/remote-off-screen-rendering-with-opengl

Rendering of a cow. (2009, June). Retrieved from http://cdn.iopscience.com/images/1742-5468/2009/06/P06016/Full/1354202.jpg

Romein, J. W. (2012). An efficient work-distribution strategy for gridding radio-telescope data on gpus. In *Proceedings of the 26th acm international conference on supercomputing* (pp. 321–330). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2304576.2304620 doi: 10.1145/2304576.2304620

Rosenfeld, D. (1998). An optimal and efficient new gridding algorithm using singular value decomposition. *Magnetic Resonance in Medicine*, *40*(1), 14–23.

Schomberg, H. & Timmer, J. (1995, Sep). The gridding method for image reconstruction by fourier transformation. *IEEE Transactions on Medical Imaging*, *14*(3), 596-607. doi: 10.1109/42.414625

Schwab, F. (1984). Relaxing the isoplanatism assumption in self-calibration; applications to low-frequency radio interferometry. *The Astronomical Journal*, *89*, 1076–1081.

Segal, M. & Akeley, K. (2017, Jun). *The opengl graphics system: A specification (version 4.5).* Retrieved from https://www.khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf

SKA. (2015). *Ska low frequency aperture array.* Retrieved from https://www.skatelescope.org/wp-content/uploads/2015/04/SKA1_AU_closeup_midres.screen.jpg

SKA. (2016). *Ska-mid africa.* Retrieved from https://www.skatelescope.org/wp-content/uploads/2015/07/SKA1_SA_closeup_highres.screen.jpg

Ska1 low. (2015, April). Retrieved from https://www.skatelescope.org/wp-content/uploads/2015/04/SKA-infographics_OP-2-rgb.screen.jpg

Ska1 mid. (2015, April). Retrieved from https://www.skatelescope.org/wp-content/uploads/2015/04/SKA1-mid-Infographic.screen.jpg

Slepian, D. & Pollak, H. O. (1961). Prolate spheroidal wave functions, fourier analysis and uncertainty—i. *Bell System Technical Journal*, *40*(1), 43–63.

Stone, J. E., Phillips, J. C., Freddolino, P. L., Hardy, D. J., Trabuco, L. G. & Schulten, K. (2007). Accelerating molecular modeling applications with graphics processors. *Journal of computational chemistry*, *28*(16), 2618–2640.

Thompson, A. R., Moran, J. M. et al. (2017). *Interferometry and synthesis in radio astronomy*. Springer.

Titan x graphics card for vr gaming from nvidia geforce. (n.d.). Retrieved from https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/

van Cittert, P. H. (1934). Die wahrscheinliche schwingungsverteilung in einer von einer lichtquelle direkt oder mittels einer linse beleuchteten ebene. *Physica*, *1*(1-6), 201–210.

Varbanescu, A. L. (2010). On the effective parallel programming of multi-core processors.

Veenboer, B., Petschow, M. & Romein, J. W. (2017, May). Image-domain gridding on graphics processors. In *2017 ieee international parallel and distributed processing symposium (ipdps)* (p. 545-554). doi: 10.1109/IPDPS.2017.68

Vla. (2017, May). Retrieved from https://wxe5pvbgxi-flywheel.netdna-ssl.com/wp-content/uploads/2017/05/VLA_New_Mexico_5.jpg

Wise, M. (2013). *The techniques of radio interferometry iii: Imaging*. Retrieved from https://www.astron.nl/astrowiki/lib/exe/fetch.php?media=ra_uva:ra_uva_lecture8.pdf

Zernike, F. (1938). The concept of degree of coherence and its application to optical problems. *Physica*, *5*(8), 785–795.