Creating and Optimizing a Sky Tessellation Algorithm for Direction-Dependent Effects

Submitted in partial fulfilment
of the requirements of the degree of

Bachelor of Science (Honours)

of Rhodes University

Antonio Bradley Peters

Grahamstown, South Africa

October 26, 2016

Abstract

ACM Computing Classification System Classification

Thesis classification under the ACM Computing Classification System[1] (2012 version valid through 2016):

- [500] **Theory of computation** → ***Computational geometry***
- [500] **Computing methodologies** → ***Parallel computing methodologies***
- [300] **Theory of computation** → ***Divide and conquer***
- [300] **Mathematics of computing** → ***Combinatoric problems***
- [100] **Theory of computation** → ***Generating random combinatorial structures***
- [100] **Applied computing** → ***Astronomy***

[1] http://www.acm.org/about/class/2012/

Acknowledgments

This work was undertaken in the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA, Tellabs/CORIANT, Easttel, Bright Ideas 39, THRIP and NRF SA (UID 90243). The authors acknowledge that opinions, findings and conclusions or recommendations expressed here are those of the author(s) and that none of the above mentioned sponsors accept liability whatsoever in this regard.

Contents

List of Figures

Chapter 1

Introduction

When the bid to co-host the Square Kilometre Array (SKA) in Southern Africa was won in 2012, the SKA Africa team celebrated a great achievement. Seven years of hard work had rewarded them with the opportunity to build the largest scientific instrument in human history. But with this came the challenges of achieving the goals set out by the SKA; engineering, data capture and data processing on a scale which had never before been thought of, let alone attempted, would be needed to see these goals come to fruition.

Galaxy formation and evolution, life elsewhere in the universe and a deeper understanding of dark matter and dark energy[1]are just some of the areas the SKA could shed light on. But answering questions like these can be difficult unless the data obtained from the array of antennae making up the SKA is clean and accurate.

Part of the solution to providing an accurate data set to operate on, arises from the need to correct DD effects (Smirnov, 2011). DD effects stem from interference radio waves experience while passing through the upper layers of the atmosphere and imperfections in the rotating mounts the antennae are mounted on. These distortions manifest themselves as blurring in images generated by the data.

While means to correct these errors exist, they can be slow, work independently of the data they operate on or both. The best current model uses a Voronoi tessellation (Okabe *et al.*, 2009) to break the image into processable units, which do not overlap, and which are then independently processed for cleaning to produce an overall improved image. While

[1]Taken from `http://www.ska.ac.za/about/faqs/`

this method works well, more can be done to ensure the is broken up optimaly which, in turn, makes the cleaning more effective and the data as accurate as possible.

1.1 Research Statement

This thesis will seek to imporve the current model for breaking up the image. It will do so by exploring geometric concepts to improve the structure of the pieces and parallel computation to improve the execution time of the process.

1.2 Research Objectives

Given the research statement above, the objectives of this research are the following:

1. Improved Voronoi tessellation algorithm to achieve an easily scalable algorithm which completes tessellation in a faster computational time while also being parallisable for future improvements.
2. Implementing a means to abstractly measure the effectiveness of a given tessellation structure on a set of sources with a fixed number of facets.
3. Overall improvement to the faceting process to increase the effect that individual data sources within the space have on the final structure of the tessellation without increasing the total number of facets.
4. Using data parallelism on a Graphical Processing Unit (GPU) to further increase the performance and decrease the computation time of the faceting process at key points.

1.3 Proposed Approach

To better understand the problem, research on the SKA and radio astronomy in general, and specifically the image capturing process was first needed. In order to create an improved tessellation algorithm that tessellates the plane efficiently and effectively, an understanding of the different available tessellation algorithms was required. A metric

was used to analyse the effectiveness of the tessellation; this metric should be sensitive to the unique structure of the data sources, the means through which the data are captured and the distortion which is being corrected for. Once the tessellation algorithm and metric have been obtained, a means of clustering the data in an effective manner, in order to improve the tessellation, was sought. As stated in the objectives, this clustering method should allow the overall structure of the tessellation to be proportionally affected by each data source, no matter how minor. Once these steps have been completed, a means to improve the performance of the algorithm was investigated by studying the NVIDIA GPU architecture and the CUDA GPU programming language as well as other data parallelism paradigms.

1.4 Structure of Thesis

The remainder of this thesis is designed to address the issues stated above and lead the reader through the investigation process to achieve the stated objectives. It is structured as follows:

Chapter 2 includes literature used to build knowledge on the means to solve the problems at hand. It is divided into three main sections, and begins by discussing radio astronomy, the structure of the telescope, the process of capturing and processing images and naive methods for correcting DD effects. It then proceeds to discuss Voronoi tessellations, extensions and algorithms for generating Voronoi tessellations and several clustering algorithms. The chapter concludes by discussing data parallelism, NVIDIA GPUs, CUDA and several optimizations for GPUs.

Chapter 3 discusses the design of the proposed solution to the problem including some pitfalls and problems in the process. It begins by discussing the means by which data sources are selected to generate the tessellation. It then discusses the algorithm of the improved Voronoi tessellation, its structure, creation and problems. The metric for computing the effectiveness of the algorithm is then evaluated and discussed as well as how it can be used to improve the tessellation overall. The chapter goes on to discuss the merging operation, which is used to improve the overall structure of the tessellation. It discusses how the operation finds and executes the optimal merge for the tessellated structure.

Chapter 4 discusses how the GPU was used to optimise key points of the algorithm

through concurrent data processing. It discusses the libraries that were used as well as how the algorithm was converted from executing only on a CPU to being executed on both a CPU and a GPU.

Chapter 5 discusses the results obtained from comparisons between the naive tessellation, the simple Voronoi tessellation and the facet merging tessellation created in Chapters 3 and 4. It compares their performance for a given number of facets using the performance metric created in Chapter 3. It also compares the efficiency and speed of the GPU implementation to that of the standard CPU implementation.

Chapter 6 concludes the thesis by analysing and discussing the results obtained in Chapter 5. It discusses problems with the solution obtained as well as its successes. It concludes by noting future improvements and approaches to solve the problems that have arisen and further improve the results.

Chapter 2

Literature Review

In this chapter we discuss the literature and resources required to create the sky tessellation algorithm and the techniques that could be used to optimize it. We begin by looking at the background of the problem in radio astronomy. We discuss how radio telescopes work, how the image is created and how direction-dependent effects occur and how they can be corrected.

We then look at Voronoi tessellations, what they are and variations in how they work. We focus especially on Voronoi tessellations of weighted points. Voronoi tessellation algorithms are also explained as well as algorithms for clustering data. Their efficiencies and complexities are also discussed.

Lastly we look at parallelism, the GPU architecture and the technicalities of programming on a GPU. We discuss the hardware to be used, the NVIDIA GTX 750 Ti, and the GPU programming language CUDA. Various optimizations using CUDA are also covered.

2.1 Radio Astronomy

Radio astronomy is the study of inter- and extragalactic objects by collecting and studying the electromagnetic signals they emit. In 1928, the physicist Karl Guthe Jansky, was searching for possible sources of radio interference for transatlantic communication. What he discovered was a large amount of noise coming from the center of our galaxy and from this the field of radio astronomy was born. Unlike optical telescopes, radio telescopes are

able to see through the dust of our galaxy to give us better insight into what lies in its center. Radio frequency radiation is also emitted from cold sources, allowing us to view extragalactic bodies with greater quality and better precision(NRAO, 2016).

This section will look at how an array of radio antenna can be used for radio interferometry to detect correlations in radio frequency radiation from extragalactic bodies. It will analyze how the two main types of telescope mounts, altazimuth and equatorial, work and why the Altazimuth is preferred even though the Equatorial produces a clearer image. The aperture synthesis of telescopes was discussed and how it uses Fourier transforms to produces the image; the section will also discuss how the primary beam of the antenna affects the image generated. The section will then see how DD-effects are generated from this, how they are corrected for, and how an algorithm which finds a good compromise between computationally feasibility and sufficient error reduction is needed.

2.1.1 Radio Telescopes

Radio Telescope Design

The most common design for radio telescopes is that of the parabolic reflector antenna. The design is a large parabolic dish with a sub-reflector at the parabola's focal point channeling the input into the feed horn at the center of the dish; a diagram of this can be seen in Figure 2.1. While it is possible to have a single antenna as a telescope for radio astronomy, in order for them to produce meaningful results, the antennas need to be extremely large (diameter of +70 m) which in most cases can be structurally infeasible especially if the antenna is made to be steerable. Instead, a series of smaller ($8 \sim 30$ m) antenna are used collectively in an array to produce more accurate signal detection. These arrays do so through radio interferometry (Cheng, 2009).

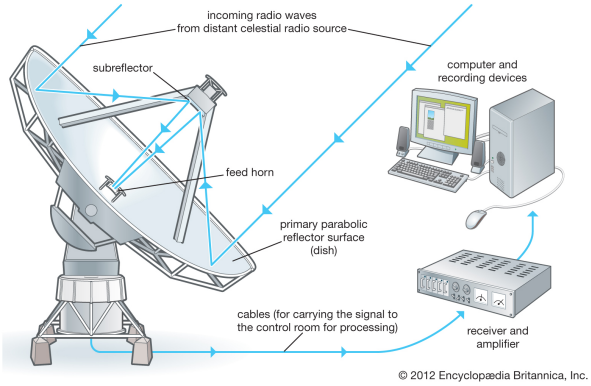


Figure 2.1: Parabolic reflector antenna design[1].

Radio Interferometry

Radio interferometry uses an array of antennas to detect and measure objects emitting radiation in the radio-wave frequencies. Radio waves are defined as electromagnetic radiation with wavelengths of the order of 10^{-3} to 10^{5} m (Cheng, 2009). The interferometer finds the source of these waves by detecting correlations in the parallel ray signal transmitted by the radiating source (Tasse, 2016) and collected by multiple antennas to determine the delay as well as the amplitude and frequency of the source to calculate the position, size and intensity of the source (Thompson *et al.*, 2008).

Radio Telescope Mounts

The choice of mount used for a radio telescope plays a large role in how well the telescope is able to track an object. The two main models used are the altazimuth and equatorial mounts. An altazimuth mount rotates on two independent axes, giving it a wide range of motion. The equatorial mount has one axis which is fixed to be parallel to the equator. This allows the antenna to simply move across the sky in one direction to track an object. The equatorial mount also follows the natural rotation of the sky as it passes to obtain less distortion (Section 2.1.2) than an altazimuth mount. Altazimuth mounts are still more common as they are relatively cheaper and easier to build than an equatorial mount (Thompson *et al.*, 2008).

[1]Taken from `http://kids.britannica.com/comptons/art-145514`

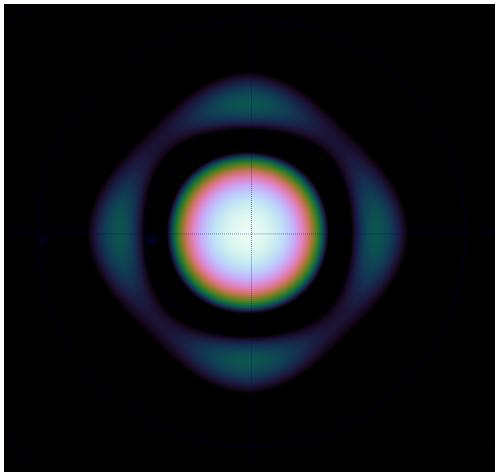
2.1.2 Image Capturing and Processing

Aperture Synthesis

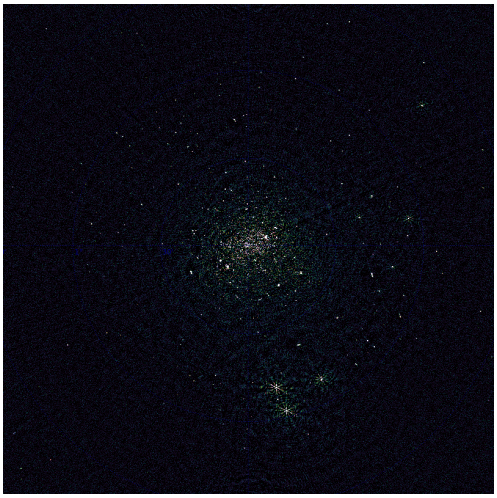
The electromagnetic radiation collected by the antenna is correlated into voltage differences. The data are collected and stored over some hours and the resulting correlations in the data taken in by each antenna in the telescope are Fourier transformed from the frequency domain to the spatial domain, to give a two dimensional image (Sault & Wieringa, 1994).

The Primary Beam

The primary beam is a mathematical function that describes the sensitivity pattern of an antenna. Naturally the beam is most sensitive in the center of the direction in which the antenna is facing, with fringes of sensitivity radiating out as shown in Figure 2.2a. The circular sensitivity present in Figure 2.2a can be seen affecting the uncorrected image present in Figure 2.2b (Thompson *et al.*, 2008).



(a) Sensitivity pattern.



(b) Resulting uncorrected image produced through radio interferometry.

Figure 2.2: Primary beam and its effects.

Errors and Error Correction

As with any real-world data input, the image capturing process of radio interferometry is subject to errors. These errors can be classified as arising from two main groups,

namely Direction-Independent (DI) and DD effects (Smirnov, 2011). DI effects are due to differences in the top layer of the atmosphere distorting the signal. This is also known as the complex gain and can be easily corrected for. The DD effects in particular arise from distortions due to interference from the ionosphere and deviations of the primary beam from the sky rotation model (due to altazimuth mounts discussed in Section 2.1.1). This distortion, D, can be corrected, but only relative to a chosen point, ξ. The correction at ξ is almost perfect, but as the correction drifts further from ξ, it introduces an error which propagates away from ξ. This error, E_i at x_i is dependent on the intensity at point x_i, I_i, and the distortion at the point relative to ξ, $D(x_i, \xi)$. Therefore, to minimize this error, every point can be made a correction seed and the image can be broken up according to these points and reassembled to form an image with little to no error. However, this is very computationally intensive as there are hundreds of sources per image and the image is sparely populated. We therefore seek a method that optimises both computational feasibility and error reduction (Smirnov & Tasse, 2015).

Figure 2.3: Figure 2.2b corrected on a 23×23 grid.

2.1.3 Naive Method for Error Correction

The most basic compromise is dividing the image evenly into a grid of smaller images and correcting for these from either the center of the sub-image, the point with the strongest source, or the "center of mass" (average location of points) of all the points in the sub-image, either weighted by intensity or not. The problem with this method is that if the

sub-image is void and has no definite points or if ξ is set at the center, it could be far from every other point and would have no substantial effect on reducing the overall error. Alternatively, if ξ is set at the strongest source or the center of mass, it could lie too close to the boundary of the sub-image and, again, have no overall impact on error reduction (Tasse, 2016). An example can be seen in Figure 2.3.

2.2 Models and Algorithms

In this section the Voronoi algorithms, namely the incremental, divide and conquer, and Fortune's algorithm will be looked at. It will be seen that the incremental is easier to implement but the most time consuming computationally. The divide and conquer uses a recursive method which is faster than the incremental, but harder to properly implement. Fortune's algorithm uses a spatial transform to generate the tessellation in the same amount of time as the divide and conquer, but with the programming ease of the incremental. Clustering methods for grouping points in a space will then be discussed. It will be seen that the k-means is effective, but unpredictable in the time it takes to complete and varied in its result depending on where the initial guesses are made. The agglomerative is the most stable but takes the longest to calculate, but this can be corrected by first generating a minimum spanning tree of the points. The bisecting k-means stabilized the run time of the k-means with its fixed recursive runs and also produces more stable clusterings.

2.2.1 Voronoi Tessellations

A Voronoi diagram is a partitioning of a space S by a set of points. Given n points (seed points) the space, $P = \{p_0, p_2, ..., p_{n-1}\}, P \subset S$, is partitioned into n regions, known as Voronoi regions or Voronoi cells, where every point, $s \in S_i, 0 \leq i \leq n-1$ in a region, $S_i \subset S$, is closest to a single seed point, $p_i \in P$, in terms of the space's distance measurement operation, d (Okabe *et al.*, 2009). An example of a Voronoi diagram is illustrated in Figure 2.4.

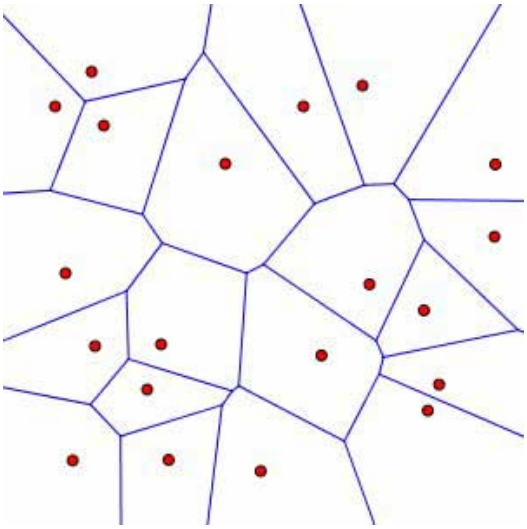


Figure 2.4: Voronoi diagram[2]

Weighted Voronoi Tessellations

The basic form of the Voronoi tessellation has the seed points as being indistinguishable from one another, other than having a different position in the space. An extension of the tessellation is to break this rule and to have the seeds have some bias or weighting associated with them. These weightings can represent a property of the data, for example, in terms of radio interferometry, they can represent the intensity of each source detected. This weighting can affect d in different ways, depending on how the weighting is accounted for; this is know as the "weighted distance". Some of the methods, discussed in (Okabe *et al.*, 2009) include multiplicative, additive, compound and power Voronoi diagrams. These diagrams have distance operators described below as d_M, d_A, d_C, and d_P respectively.

$$d_M(s, p_i) = \frac{1}{w_i} d$$
$$d_A(s, p_i) = d - w_i$$
$$d_C(s, p_i) = \frac{1}{w_{i1}} d - w_{i2}$$
$$d_P(s, p_i) = d^2 - w_i$$

Problems arise in attempting to compute the tessellations for the multiplicative, additive and compound Voronoi diagrams as the edges of these diagrams, could potentially be curved by a circular arc (d_M,d_C), a hyperbolic arc (d_A,d_C) or a fourth order polynomial

[2]Taken from `http://www.ams.org/samplings/feature-column/fcarc-voronoi`

arc (d_C) (Okabe *et al.*, 2009). This leaves the power diagram as the only Voronoi diagram that enforces that the edges are straight lines and the resulting tessellation is a convex polygon, similar to the standard Euclidean Voronoi diagram. For the power diagram, if the weighting is equal for all points, the resulting diagram is the same as that of a standard Euclidean Voronoi diagram. It is possible in the power diagram, that a seed point will not be contained within its own associated Voronoi polygon. This occurs when two seed points ($p_i, p_j \in P$, $w_i < w_j$, $i \neq j$) are close enough together such that the weighted bisector, defined by

$$b(p_i, p_j) = \frac{1}{2}(||\mathbf{x}_i||^2 - ||\mathbf{x}_j||^2 + w_i - w_j) \qquad \mathbf{x}_i = (x_i, y_i), \tag{2.1}$$

does not lie on the line segment $\bar{p_i p_j}$. When this occurs, p_i lies in the region of V_j. If the difference in weighting between p_i and p_j is large enough and the distance between them small enough, the points in V_i may be an empty set. It is worth noting that Power Diagrams are also referred to as General Voronoi Diagrams (Aurenhammer, 1987).

2.2.2 Voronoi Tessellation Generation Algorithms

Although Voronoi tessellations extend to multiple dimensions, for simplicity we only discuss those in a two dimensional plane.

Incremental Algorithm

The most simplistic of the generation algorithms, the Incremental is an iterative algorithm as described below (Green & Sibson, 1978) (Okabe *et al.*, 2009):

1. Start from $i = 0$ with an empty plane.
2. A seed point, p_i is placed into the plane.
3. The nearest neighboring seed point $p_f = p_{nn}$ is found.
4. A perpendicular bisector is drawn between p_i and p_f (if it exists).
5. The bisecting line is followed in both directions until it intercepts an existing edge or the plane's boundary on both ends.
6. A new edge is defined by this segment of the bisector as part of both p_i and p_f.

7. The seed point of the polygon that shares the found edge clockwise to p_f (anticlockwise to p_i) is then set to p_f.

8. Continue from step 4 until $p_f = p_{nn}$.

9. Set $i = i + 1$ and repeat from step 2 until $i = n$.

In its most naive form, this algorithm achieves an efficiency of $O(n^2)$.

Divide and Conquer Algorithm

The Divide and Conquer algorithm was first proposed by Shamos & Hoey (1975), but also described in (Okabe *et al.*, 2009). It is a recursive algorithm that improves on the Incremental algorithm by having an execution time of $O(n \log n)$.

1. If the space contains only one point, return it with the entire plane as its Voronoi region.

2. Divide the space, S containing the set of n seed points, P, into two subspaces, S_L and S_R, such that S_L and S_R contain $n/2$ seed points and every seed point of P_L lies to the left of every seed point of P_R (this is made easier if P is ordered).

3. Recursively compute the Voronoi tessellations for P_L in S_L and P_R in S_R; called V_L and V_R, respectively.

4. A polygonal line, Q, must now be found such that Q merges V_L and V_R into a single Voronoi tessellation, V:

   (a) Start with the polygon of V_R which contains the top-left corner of S_R, p_R and the polygon of V_L which contains the top-right corner of S_L, p_L. Since p_L must lie to the left of p_R, they must overlap when V_L and V_R are extended into S.

   (b) A perpendicular bisector is drawn between p_L and p_R and segmented between its two closest edge intercepts from the shortest distance between p_L and p_R. This segment is added to Q.

   (c) If the lower edge, intercepted by the bisector is in V_R then p_R is set to the seed point polygon which shares this edge and similarly if the edge is in V_L.

   (d) Continue from step 4b until the bottom of S is reached.

5. Remove all line segments of V_L to the right of Q and all those of V_R to the left of Q to form V.

6. Return V recursively until the full Voronoi tessellation is complete.

Part of achieving this efficiency is assuming P is co-lexicographically ordered, meaning for all $p_i, p_j \in P$, $0 \leq i < j < n$; $x_i > x_j$ or ($x_i = x_j$ and $y_i > y_j$). This speeds up the partitioning of P into P_R and P_L at each level of recursion.

Fortune's Algorithm (Sweep-Line Method)

Fortune (1987) describes an algorithm where the tessellations are found by a line "sweeping" over the space and solving the problem at each step of the sweep. This can be problematic for Voronoi tessellations as the line may intercept the Voronoi region of a seed point before it intercepts the point. Therefore the Voronoi tessellation is not computed directly, but through a geometric transform. The transform $\phi(x(s), y(s))$ works such that for any point, $s \in S$ with coordinates $(x(s), y(s))$,

$$\phi(x(s), y(s)) = (x(s) + r(s), y(s)), \tag{2.2}$$

where $r_i(s)$ is defined as the distance to the seed point $p_i \in P$ and $r(s) = min\{r_i(s)|1 \leq i \leq n-1\}$, is the distance to the closest seed point to s. This transform can easily be reversed to re-obtain S and its set of Voronoi tessellations. Now, for the transform of S, $\phi(S)$ denoted by Φ, the left-most point of each Voronoi Region is its seed point (except the left-most seed point); this is essential for the algorithm. It is important to note that the perpendicular bisectors of seed points in S, through the transform, become hyperbolas in Φ (provided they are not horizontal in S). For $p_i, p_j \in P$, the hyperbola is denoted as h_{ij} which can be split into h_{ij}^+ and h_{ij}^- as the upper and lower half-hyperbolas about the left-most point, respectively. Set Q is denoted as the set of all event points in the algorithm. Q is initially populated with the seed points (in co-lexicographical order) but the edge interceptions are added as they are found. The algorithm, as described by Okabe *et al.* (2009) is as follows:

1. Add P to Q.

2. Choose and delete the leftmost seed point, p_i from Q.

3. Create a list, L containing the transformed Voronoi region of p_i, $\phi(V_i)$.

4. While Q is not empty do the following:
   (a) Choose and delete the leftmost element, w of Q.
   (b) If w is a seed point:
      i. Set $p_i = w$.
      ii. Find the region, $\phi(V_j)$, containing p_i.
      iii. Replace $\phi(V_j)$ in L with $(\phi(V_j), h_{ij}^-, \phi(V_i), h_{ij}^+, \phi(V_j))$.
      iv. Find the half-hyperbola intercept(s) with any other hyperbolas, if they exist, and append these to the front of Q.
      v. Repeat from step 4.
   (c) If w is a half-edge:
      i. Set $\phi(q_t) = w$ where $\phi(q_t)$ is the intercept of $h_{ij}^{\pm}$ and $h_{jk}^{\pm}$.
      ii. Replace all sequences of the form $(h_{ij}^{\pm}, \phi V_j, h_{jk}^{\pm})$ on L with $h = h_{ik}^+$ or $h = h_{ik}^-$ appropriately.
      iii. Remove from Q any intersections of $h_{ij}^{\pm}$ and $h_{jk}^{\pm}$ with other half-hyperbolas.
      iv. Move any intersections of h in L to Q.
      v. Mark $\phi(q_t)$ as a Voronoi vertex incident to $h_{ij}^{\pm}$, $h_{jk}^{\pm}$ and h.
      vi. Repeat from step 4.
5. Return the half-hyperbolas on L, the set of marked intersections from step 4(c)v and the relations among them.

2.2.3 Clustering Algorithms

It may be the case that the number of potential seed points in a space, N_p, is much larger than the optimal number of facets, N_v. In these cases it would reduce the overall computation time drastically if the N_p points were grouped into N_v clusters. From each of these clusters, a point is then chosen as a seed point to be used to find the corresponding Voronoi tessellation. Some key examples of such clustering algorithms are described in this subsection.

K-Means Algorithm

K-means clustering is an iterative process where an initial guess at the center of a cluster, c, is made and then improved with each iteration. It is named as such because it seeks

to separate n objects into k clusters where, for each object in a cluster ($o^i \in C_i$, $i \in \mathbb{R}$, $0 \leq i \leq k-1$) the mean point of that cluster, c_i, is closer to it than any other mean point and the c_i is representative of the average values of all points, $\frac{1}{m}\sum_{j=1}^{m} o_j^i$, in C_i. Way *et al.* (2012) describe the algorithm as follows:

1. Randomly choose k mean points $(c_0, \ldots, c_{k-1})$.
2. Assign each c_i an empty object set, C_i.
3. Iterate through all the objects in the space $(o_0, \ldots, o_{n-1})$ and assign the object to the object set of the mean point closest to it.
4. Set all c_i to be the average of all points in their respective C_i.
5. If the sum of the changes in c_i, $\sum_0^{k-1} \Delta c_i$, is greater than some given tolerance, ϵ, then repeat from step 2, else return the set of means (and their object sets if necessary).

One obvious problem with this algorithm is that the number of iterations can be unpredictable; this is addressed by having the sum of changes only converge to ϵ, instead of complete convergence. With large data sets and large k-values where the mean points converge in smaller steps with every iteration, this can drastically reduce the runtime of the algorithm. The clusters produced are also dependent on the initial placement of the mean points (Way *et al.*, 2012). Other methods of improving the runtime include probabilistic choices of starting mean points (Arthur & Vassilvitskii, 2007) and constraining the distance and using the triangle inequality (Hamerly, n.d.).

Bisecting K-Means Algorithm

A variation on the k-means algorithm is to embed it into another iterative method, which, by design, reduces the computation time and also improves the quality of the clusters produced. The algorithm works by branching large clusters into smaller ones. The algorithm, described by Steinbach *et al.* (2000), is as follows:

1. Start with the entire set of objects in the space as part of a single cluster.
2. Choose the largest cluster in the space.

3. Split the objects into two sub-clusters and refine iteratively by way of the k-means algorithm.

4. Repeat from step 2 until k clusters are produced.

Agglomerative Clustering Algorithm

Contrary to the k-means algorithm (and more specifically the bisecting k-means) is the agglomerative clustering algorithm. Instead of starting with a single cluster containing all points, this algorithm instead places every point in its own cluster and merges them until the required number of clusters are produced. Way *et al.* (2012) describe the algorithm as:

1. Begin with each object in its own cluster.

2. Merge the two closest clusters.

3. Repeat step 2 until k clusters remain.

Although this algorithm will always yield the same result for a given data set, it is far more expensive than the k-means. Improvements can be made on this, however by instead building a minimum spanning tree, weighted by the distance between the data and iteratively removing the links with the highest weights until the number of required clusters is produced.

2.2.4 Related Work

Standard Voronoi Faceting

In (Tasse, 2014), (Smirnov & Tasse, 2015) and (van Weeren *et al.*, 2016), a series of observed or simulated extragalactic points are clustered into facets using a Voronoi tessellation algorithm with the seed points for these facets set as the brightest points in each facet. An example of this can be seen in Figure 2.5 where the facets are superimposed over the source image from which they are derived.

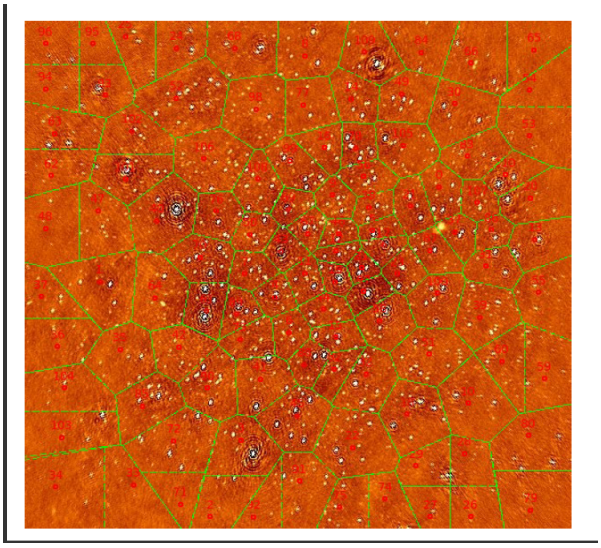


Figure 2.5: Example of Voronoi faceting to group extragalactic points for DD calibration

.

2.3 GPU Architecture and Concepts

In this section the history of GPUs from their initial use for pixel generation in gaming to the parallel data processing titans they are today is looked at. More specifically at the NVIDIA GTX 750 Ti, and how its GM107 architecture has improved to give high performance with low power consumption is discussed. How the warps are used in the streaming multiprocessor to allow more processes to run concurrently which in turn improves the efficiency of the GPU is investigated. The GPU programming language, CUDA, will be discussed as well as how it uses threads and blocks in a grid for parallel data processing, and how it incorporates special types memory on the GPU in order to further improve the efficiency of the GPU. Finally we discuss CUDA best practices for optimizing the run time of the code, using asynchronous data transfers, running multiple kernels and making use of the CUDA libraries.

2.3.1 Parallelism

One of the main means of reducing processing time is through parallelism. The two main forms of parallelism are task and data parallelism. Task parallelism can be seen as running multiple processes concurrently where communication between the processes is explicitly

defined to avoid race conditions (Subhlok *et al.*, 1993). Data parallelism is the distribution of a data set over a number of identical processes each of which performs operations on a unique subset of the data. Race conditions occur when parallel processing streams access data or perform operations out of the intended order, leading to errors or incorrect output being produced. A combination of task and data parallelism can lead to an ideal speed-up, but both have their limits depending on the task and the data being operated on (Subhlok *et al.*, 1993).

The increased need for parallelism came about in 2005, when Central Processing Unit (CPU) frequency peaked at 4 GHz due to heat dissipation issues. However, Moore's Law (Moore, 2006), still holds, and is still expected to hold until 2025 (that is, that the number of transistors for a computer will double every two years). This leads to a problem where the speed at which an operation is done cannot be increased (due to the frequency limit), but the number of concurrent operations can still increase. This means that the only way to speed up an operation is to change it from a sequential to a parallel process (Rajan, 2013).

2.3.2 GPU Execution

GPUs were originally designed for rendering pixels and vectors in games. They were especially designed for this since CPUs are optimized to run sequential instructions as fast as possible, whereas pixel and vector calculations are inherently parallel. With NVIDIA's release of CUDA in 2006, General Purpose GPU (GPGPU) programming became feasible as a way to accelerate data processing through data parallelism and task parallelism through the simultaneous execution of similar tasks (NVIDIA, 2016).

The power of a GPU comes from its architecture which is optimized for a special case of Single Instruction Multiple Data (SIMD) processing known as Single Instruction Multiple Threads (SIMT). SIMD allows a central processor to distribute a set of instructions to multiple simple processors which then act on the data simultaneously. SIMT is more generalized as each warp (Section 2.3.3) of the GPU can perform different tasks given the same set of instructions. This is due to the way in which the GPU handles branching at the thread level. By exploiting these processes, and this instructional architecture, some instructions can be computed faster than is possible on a a CPU (Vuduc & Choi, 2013).

2.3.3 The NVIDIA GeForce GTX 750 Ti

GM107 Maxwell Architecture

The NVIDIA GeForce GTX 750 Ti GPU was released on the 18th of February 2014. It boasts 640 CUDA cores, 1020 MHz base clock speed, 1305.6 GFLOPs and a memory bandwidth of 86.4 GB/sec. It is NVIDIA's first-generation Maxwell architecture, designed for high performance at relatively low power consumption (60 W) and has the codename 'GM107'. The GPU uses PCI Express 3.0 to interface with the host machine through the GigaThread engine. The first-generation Maxwell (from now simply referred to as Maxwell) is made up of one Graphics Processing Cluster (GPC) on which the processing occurs. It also contains a large L2 cache at 2048 KB and two 64-bit memory controllers to access the 2048 MB global memory. This design can be seen in Figure 2.6 (NVIDIA, 2016a), (NVIDIA, 2014).

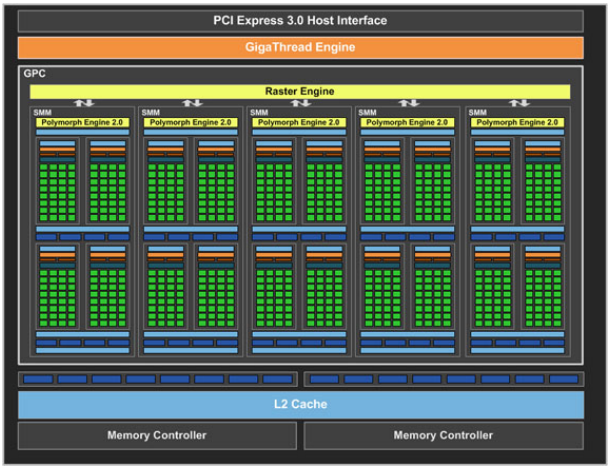


Figure 2.6: NVIDIA Maxwell Architecture.[3]

Streaming Multiprocessors

The GPC is further broken down into five Streaming Multiprocessor (SM)s which are divided into four processing blocks. The processing blocks (or warps) each contain an instruction buffer, a scheduler and 32 CUDA cores as seen in Figure 2.7. These warps are set in a lock step, meaning each core in a warp executes the same set of commands at the same time, with different valued variables (NVIDIA, 2015).

[3]Taken from `http://www.hitechlegion.com/reviews/graphics/38752-msi-gtx-750-ti-gaming-video-card-start=2`

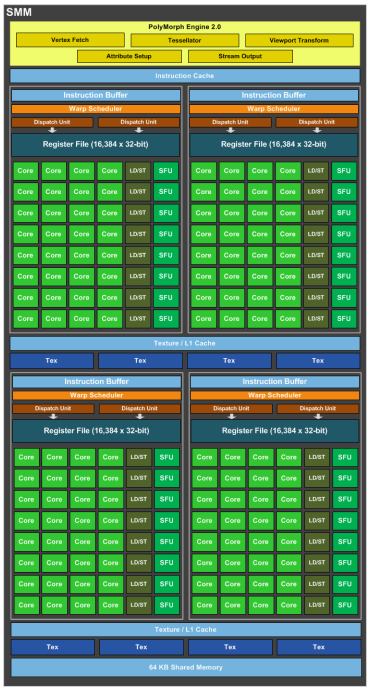


Figure 2.7: Maxwell Streaming Multiprocessor.[4]

2.3.4 CUDA

CUDA is a parallel programming language created by NVIDIA for the purpose of running on their brand of GPUs. CUDA was modeled as a C-like language with some C++ features. Its main feature is the way in which it separates CPU and GPU code. The CPU code is labeled as "host" code and the GPU's as "device" code. Device code is called by the host through a special case of a method, known as a kernel. The basic structure of a kernel is as follows:

```
kernel0<<<grid, block>>>(params);
```

In this instance `kernel0` would be the name of the kernel being called, `grid` is the three dimensional value of the number of blocks to be assigned, `block` being similar to `grid` is a three dimensional value of the number of threads needed and `params` is simply the parameters needed by the kernel to execute (similar to those of a method) (NVIDIA, 2015).

[4]Taken from `http://www.legitreviews.com/nvidia-geforce-gtx-750-ti-2gb-video-card-review_135752/2`

Threads

The thread is the smallest processing unit of the GPU. GPU threads are designed to be cheap and lightweight compared to those of a CPU so that it can easily be created, run its small task and be destroyed to make place for the next thread. Threads are arranged into Three Dimensional (3D) blocks with each thread having a unique 3D ID within that block, namely an x, y and z ID. Generally the thread ID is used as the means of determining the difference in the task process of each thread (NVIDIA, 2015).

Blocks

Each block may have a maximum of 2056 threads in total and 1024 for any single dimension; this is the reason why they are bundled into a larger, 3D grid structure. Similarly to threads, blocks have a unique 3D ID in the grid. While warps execute each step of a process simultainously, this is not necessarily true blocks when spread over mutiple warps. More often than not, blocks exchange data within their threads and if precaution such as thread synchronisation are not taken, race conditions could ensue to break the code. Each block, when executing, must occupy a whole number of warps (rounded up). This is done as warps are constantly in lock step. Threads within a block share a fast memory, located in the L1 cache of the streaming multiprocessor. This shared memory must be preallocated when the kernel is called as a third parameter within the kernel launch (parameters within the triple angle brackets) (NVIDIA, 2015).

GPU Memory Hierarchy

In order to maximize concurrency on the card, the GPU has a structured memory hierarchy. The largest, slowest and most generally accessible of these is the global memory which resides on the device memory. This memory is visible to every thread and also to every kernel called in one application.

The constant memory also resides on the device memory; as the name suggests, values stored here cannot be altered and are read-only until the space is deallocated. Variables stored in constant memory also have the ability to broadcast their values to multiple threads simultaneously.

Similar to constant memory, texture memory also lies on the device memory and is also read-only; it is optimized for storing 2D arrays where multiple neighboring values of the array can be read concurrently.

The shared memory lies on the SM and is visible only to a block as a means for threads within a block to exchange data. Shared memory must be declared with the size of the memory needed (up to the maximum 64 Kb) when the kernel is executed.

Each processor is assigned its own memory space to be used by each thread, in the form of registers. Registers are visible only by the thread currently on that processor and reset with each change of thread. Registers hold the variables created in and passed to the thread. The register is by far the smallest memory on the card at 32 bits per register, but 255 registers per thread. Should the thread call too many variables or variables too large to fit in the registers, the variables spill over into local memory, which lies in the L1 and L2 caches for active threads. Should the thread need to be temporarily halted for another thread to use the processor, then the threads variables are stored in local memory on the device memory (NVIDIA, 2015). A visual description of the memory hierarchy can be seen in Figure 2.8.

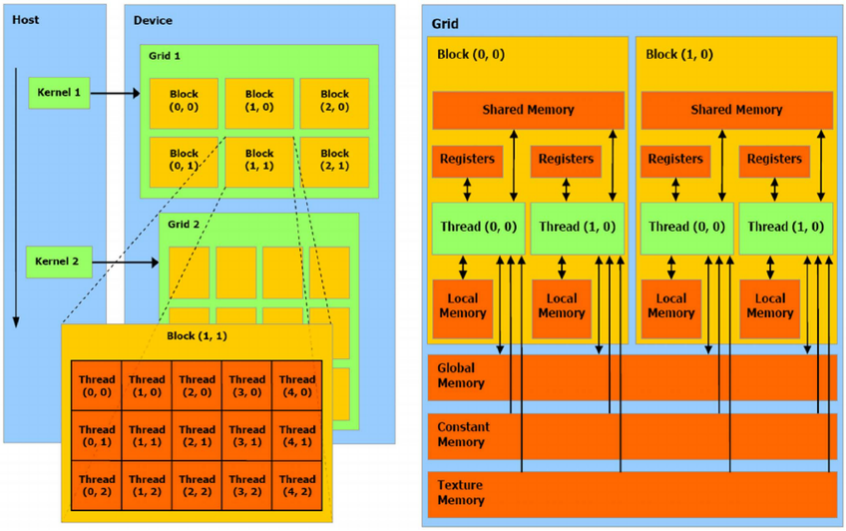


Figure 2.8: CUDA memory hierarchy.[5]

[5]Taken from `https://www.researchgate.net`

2.3.5 CUDA Optimization

As stated in Section 2.3.2, GPUs are designed for parallel computing to speed up the execution of a process when compared with running it on a CPU. To ensure the greatest speedup a variety of optimizations are typically necessary to improve naive parallel code. These optimizations can be categorized into three groups, memory, execution configuration and instruction optimizations. The NVIDIA Visual Profiler (NVVP) is invaluable assisting in users to identify areas in their CUDA code that require optimization.

Memory Optimization

Memory optimization seeks to maximize the bandwidth of the GPU so that more time is spent using the faster memory (e.g. registers, L1 and L2 caches) and less time on the slower memory (e.g. device memory, host memory). An example of a memory optimization is that CUDA has the ability to asynchronously transfer data between the host and device by breaking a kernel into streams, thereby allowing the device to process one section of the data while another is still being transferred to it and a third is being transferred back to the host. Alternately, memory on the host can simply be mapped to the device. This memory is accessible by both the host and device and is known as zero-copy memory. This can only be done on pinned memory, which is memory set aside by CUDA to be used by the GPU, and is also optimized to have a higher transfer rate to the GPU than any other host memory. This can be taken a step further through unified virtual addressing, where the host and device share a single virtual memory space, this address space lies on the host, but through predictions by CUDA on the need for certain sections of the memory, parts of the memory are transfered to the device as they are needed. The different types of memory found in the GPU as discussed in Section 2.3.4 show examples of this as well. Constant and texture memory, for example, improve latency by having the value broadcast to all threads and 2D spatial locality read abilities, respectively. The use of sequential reads and stride accessing can also improve bandwidth usage as the memory required is aligned on the device (NVIDIA, 2015).

Execution Configuration Optimization

Execution configuration optimizations seek to improve the overall usage of cores on the GPU, keeping as much of the hardware as possible occupied to improve the overall execution time. One way of achieving this is through concurrent kernel execution, that is,

multiple kernels running concurrently to reduce the idle time of each warp. Another way is by setting the number of threads in a block to always be a multiple of 32 so that all the cores in a warp are occupied. Other examples include having multiple small blocks instead of a single large block, especially when using thread synchronizations and also having a minimum block size of 32 threads (NVIDIA, 2015).

Instruction Optimization

Instruction optimization uses the knowledge of how certain instructions are executed to speed up code in critical areas. This form of optimization is most prevalent in arithmetic operations. Most notably, single precision is encouraged as well as using the multiple CUDA maths libraries that interface directly with the hardware. These libraries include cuBLAS for linear algebra, cuSparse for sparse matrix operations, cuRAND for random number generation, nvGRAPH for graph analytics and cuFFT for fast fourier transforms. These and more libraries are discussed in (NVIDIA, 2016b). Another way of reducing latency is by minimizing the use of global memory, as it is generally the slowest to access. Making use of constant and shared memory for read-only values and shared memory for block specific values can drastically improve memory access times (NVIDIA, 2015).

2.3.6 Related Work

Only one study relating to GPU implimentation of a Voronoi tessellation has, to the best of our knowledge, been published. This algorithm is called "Jump Flooding" and is explained below.

Rong & Tan (2006) describe a method of approximating a Voronoi transform using a method known as jump flooding. The algorithm works by seeing the space as a discrete $n \times n$ grid. It works by having grid cells with an identified closest seed point (or in this case, seed cell) and projecting this seed cell to surrounding cells without an identified seed cell in incrementally smaller steps, starting from a step size of $n/2$.

While the Jump Flood does outperform standard CPU algorithms, it is sensitive to the resolution of the output grid. Therefore the algorithm could potentially be slower than a CPU implimentation where the CPU implimentation is grid independant, such as divide and conquer or Fortune's algorithm.

2.4 Summary

In this chapter we have discussed many of the technical and theoretical aspects of the algorithm which needs to be developed. We began by looking at radio astronomy and how radio telescopes are designed as parabolic arcs. We then reviewed Voronoi diagrams and discussed how power diagrams are their natural extension when weights are incorporated. We looked at three algorithms again, namely the k-means, the agglomerative, and the bisecting k-means algorithms. Lastly we looked at GPUs, their architecture, and some of the concepts involved. We discussed parallelism and how it has arisen as a computational norm from the issue of frequency CPUs are unable to overcome.

At each stage we also looked at existing models which do work similar to what will be done in the later paper. We looked at the naive grid method for DD-effect error correction and its shortcomings due to empty blocks and off center optimal points. We then discussed a standard Voronoi model for use in the correction of DD effects, we saw how it improved on the naive method but also how it fell short with it only regarding the brightest points which may lie too close to one another and warp the entire polygon. We lastly looked at the jump flood algorithm as a GPU based voronoi tessellation algorithm, we saw that it approximated a tessellation by changing the space to a grid of finite cells and used decremental steps to generate a tessellation in a reasonable time.

Chapter 3

Tessellation Algorithm Design

This chapter will discuss the design of the tessellation algorithm. It begins by describing the improvements made by implementing the divide and conquer Voronoi algorithm as well as the details of the algorithm itself. It then discusses the error metric which will be used to test the performance of our algorithm and concludes by detailing the cell merge algorithm used to group Voronoi cells to form larger cells.

3.1 Overview

Given a large set (~ 10000) of extragalactic sources obtained by aperture synthesis (Section 2.1.2), an optimal means to tessellate the space containing the sources into facets must be found so that the correction of DD-effects is maximal.

A metric is needed to calculate the effectiveness of the algorithm. Since each tessellated facet or cell contains exactly one correction centre and, potentially, multiple sources, an error metric is used. The error for each cell, ϵ_i, is defined as the sum of the distance from each source to the correction centre, and the sum of all the errors per cell given the error over the entire tessellation, $E = \sum_i^n \epsilon_i$ where n is the number of cells. The algorithm must therefore minimise E by choosing cell boundaries and centres such that each ϵ_i is as small as possible while simultaneously keeping n as low as possible.

A Voronoi tessellation (Section 2.2.1) was used to tessellate the space into cells. However, given the large number of sources, using each source as a centre is suboptimal and defeats

the purpose of finding an algorithm which minimises both the error and the number of cells needed, especially considering the large intensity differences in the sources, which can differ in the order of 10^5 in magnitude. A smaller subset of the sources is therefore used to generate the Voronoi tessellation.

In cases where two large sources are close neighbours of one another, a single larger cell would be preferred over two smaller cells, this would however affect the overall error, E, of the tessellation. A merge algorithm was designed and used to find centres with relatively close centres leading to the lowest overall increase in the error. For the sake of the merge algorithm, it is preferable to start with the entire set of sources as centres and merge centres together until the maximum allowed error is reached or the number of cells is minimised.

In the following sections, each of the steps taken in implementing the sequential tesellation algorithm is explained.

3.2 Source Selection

We begin the algorithm with a list of n sources in the plane. Each source has three parameters: x, y and z. The x and y parameters define the spatial coordinates of the source while the z parameter is used to reflect the intensity of the source.

The sources are read in and are used to generate the Voronoi centres. Each centre has the following attributes:

1. An x, y, and z value inherited from its source.
2. A Boolean value to determine if it is the circumcenter which defaults to false.
3. A centre clockwise to it if it lies on the convex hull which defaults to none.
4. A centre counter-clockwise to it if it lies on the convex hull which defaults to none.
5. A list of all its neighbouring points and the line that bisects them set into a tuple, the list is set to none by default.
6. A list of sources in the cell created by the centre which is set to none by default.

7. The cell's error which is set to zero by default as there are no sources at initialisation.

8. A list of all the cells that have been merged with this cell; by default, this is also set to none.

Sources are chosen as centres for a Voronoi cell if they have an intensity greater than some predetermined threshold. Once the set of centres, $C = \{c_0, c_1, ..., c_n\}$, is created, it is sorted in lexical order i.e. by their x values and then by the y values if the x values are equal. Once the centres have been created, generation of the Voronoi tessellation can begin.

For the sake of testing the system, the x and y coordinates of the sources are randomly generated on a 600×600 plane with the intensities, z, randomly generated as the absolute value of a random normal distribution, centred at zero with a standard deviation of 3000. Using this, 10000 sources are randomly generated. From these sources, we generate the centres to be used by the Voronoi, where these centres are sources with an intensity greater than one standard deviation of the mean. Since the absolute value of the source intensity is used, this includes all sources with an intensity greater than 3000, thus accounting for approximately 32% of the sources.

Figure 3.1 shows sources in green with their intensities represented by the size of the circle on the plane; for simplicity, only 200 points are generated. The crosses in red represent the centres that are used to generate the Voronoi tessellation.

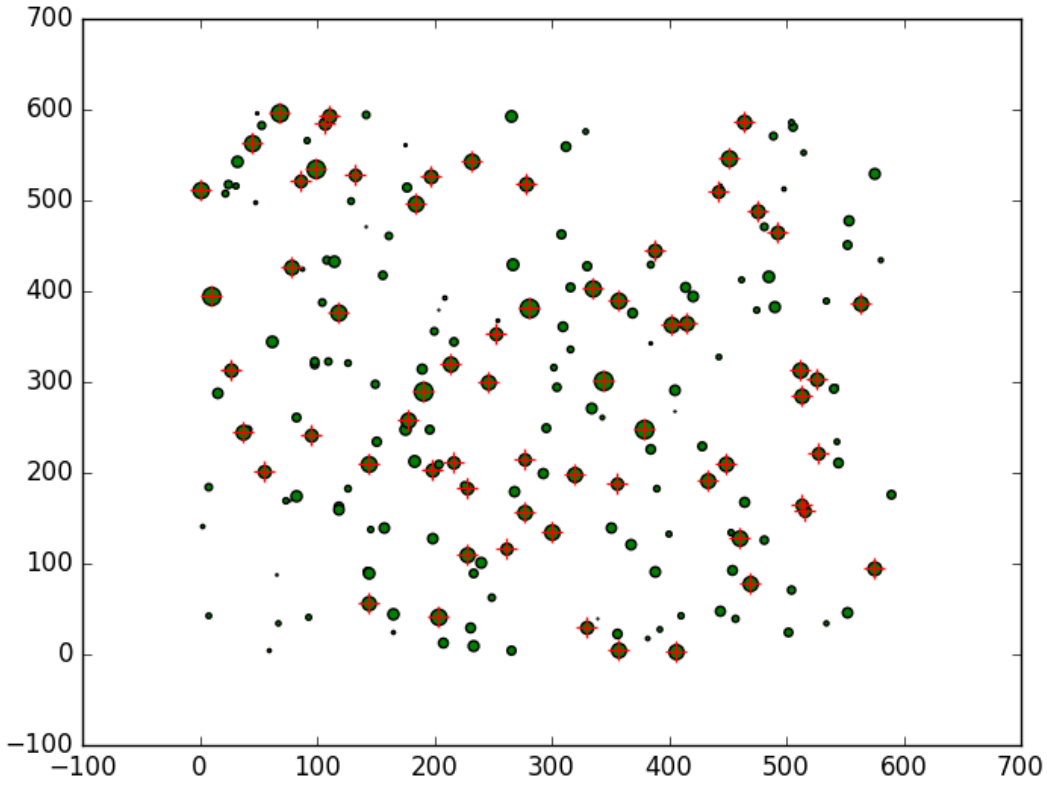


Figure 3.1: All sources (green circles) with those selected as centres (red crosses).

3.3 Voronoi Tessellation Generation

The current best model for creating a Voronoi tessellation uses k-means clustering on all the points in a discrete space to the centres. For every point (the plane is seen as a finite number of pixels) in the plane, p, it finds the centre, c_i, which is the shortest distance from it, i.e. such that $\sqrt{\big(p(x) - c_i(x)\big)^2 + \big(p(y) - c_i(y)\big)^2}$ is minimised. This is suboptimal as the time taken to create the tessellation relies on the dimensions of the plane and the number of centres therein. Instead we seek a method which is invariant of the plane size and relies solely on the centres. For the sake of decreasing the computation time, this method must also be parallelizable.

To satisfy these constraints, the divide and conquer algorithm (Section 2.2.2) was chosen. The algorithm works by ordering the centres lexicographically and then dividing the centres into two subsets, left, C_L, and right, C_R. The Voronoi tessellations are generated

for the left and right subsets separately using the divide and conquer method. The convex hulls of the left and right Voronoi tessellations are then found. The lowest common support line between the hulls is calculated and from this a dividing polygonal chain is generated until it intercepts the upper bounds of the plane. The intersecting edges with the polygonal chain are determined, and lines in the polygonal chain are cut to become part of the Voronoi cells edges.

Code for the Divide and Conquer algorithm was adapted from that in the git repository pyVoronoi[1]. pyVoronoi is an implementation for the Divide and Conquer Voronoi algorithm written in Python 2.7. It uses a simple GUI to generate a Voronoi tessellation in a fixed plane using sources read in from a text file. This interface, while simple to use, is constraining as it does not allow users to specify their own spacial constraints or generate the Voronoi as part of a pipelined process. The visualiser and interface were therefore removed and the remaining code re-factored so that the process is a function that can be called by the user or used in a larger process.

3.3.1 Structures Used by the Voronoi

The Voronoi structure consists of two main structures, lines and centres. Centres (Section 3.2) are mainly used to define the sources, but are also used to generate lines.

A line is made up of the following attributes:

1. Two points, defined as centres which make up the end points of the line. These are defined when the line is created.
2. Two centres which the line bisects, which is set to be empty by default.
3. A centre defining the circumcentre of the line which is empty when the line is created.
4. A list of all lines connected to this one which is used during the merge of the divide and conquer and is empty by default.
5. A Boolean value determining whether the line is active or depricated. This is set to true by default, but this is updated when two cells are no longer neighbours, generally due to a merge.

[1] https://github.com/twmht/pyVoronoi

3.3.2 Voronoi Function

The Voronoi function takes in the set of centres and a range of centres it will operate on (initially the entire set, C). Depending on the number of points in the subset range, one of three operations occurs.

If the range is made up of more than three points, the range is divided into two equal subranges: one from the starting point of the range to the median, $C_L = \{0, ..., \frac{n}{2}\}$ where n is the range, and another from the point after the median to the end of the range,$C_R = \{\frac{n}{2} + 1, ..., n\}$. The Voronoi function is then called again for each subrange, denoted as C_L and C_R for left and right subranges respectively, with the full set of points. Once the Voronoi structure for these two have been calculated, the merge function is called with V_L and V_R, the Voronoi structures of C_l and C_R respectively, and the function ends as it is not value returning.

If the range is three, the function generates three bisecting lines, $b_{1,2}$, $b_{1,3}$ and $b_{2,3}$ one for each pair of the three points, c_1, c_2 and c_3. The intercept of $b_{1,2}$ and $b_{1,3}$ is determined. If it does not exist, the centres are collinear and an invalid line exists, and the line, $l_{1,3}$ is removed as, lexicographically, c_2 must lie between c_1 and c_3 and $l_{1,3}$ must therefore lie in the cell of c_2. If the intercept does exist, it lies on the circumcentre, cc, of the triangle formed by the centres and the lines $l_{1,2}$, $l_{1,3}$ and $l_{2,3}$ with the centres as endpoints. The lines, $l_{1,2}$, $l_{1,3}$ and $l_{2,3}$, are ordered by the square of their lengths and the triangle they form is determined as either acute, obtuse or right-angled. The centres and corresponding lines are appended to each centre's list of related centres, e.g. $(c_2, b_{1,2})$ and $(c_3, b_{1,3})$ are appended to c_1's list of related centres. The bisecting lines, $b_{1,2}$, $b_{1,3}$ and $b_{2,3}$, are then clipped so that they all intercept at cc. Figure 3.2 illustrates this process.

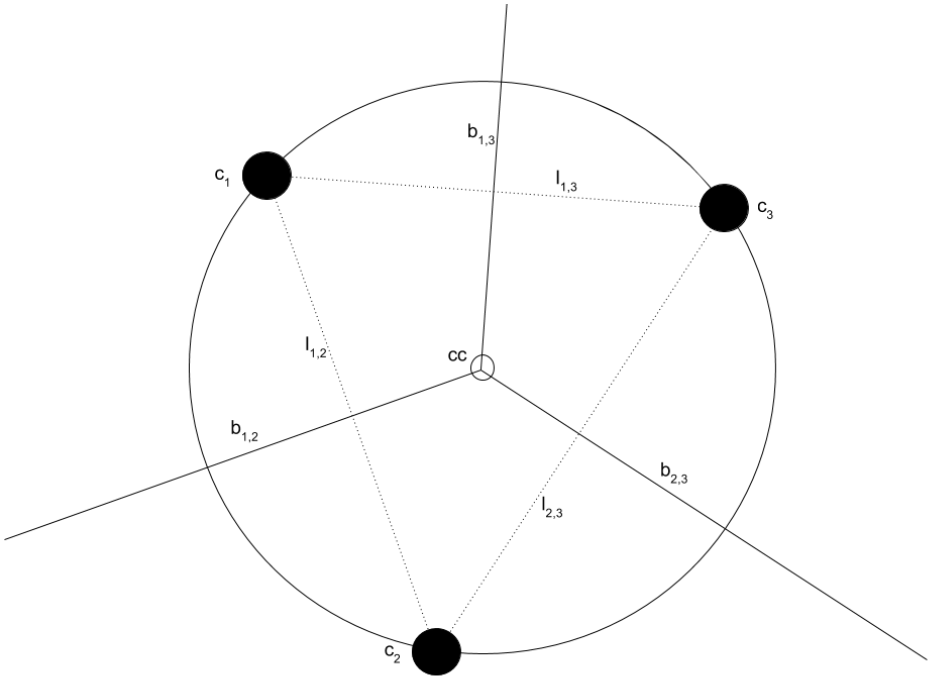


Figure 3.2: A Voronoi tessellation of three points (c_1, c_2 and c_3) with circumcentre (cc).

A list of the bisecting lines, the range of the points and the convex hull (which is yet to be determined) are returned.

For a range of two points, the bisecting line, $b_{1,2}$, is determined as the only line in the structure. The centres and $b_{1,2}$ are placed into one another's related centre list and the line is placed in a list as the only element. The line list, the range of the centres and their convex hull are returned.

The case of a single centre is excluded as the case of multiple centres divides the set of centres either into two equal subsets or two subsets differing in size by one. The second and third cases will catch sets of three or two centres and therefore the case of a single centre cannot be determined unless a set of a single is passed through initially. This is, however, meaningless as the Voronoi tessellation for this will be a single cell containing the entire space.

3.3.3 Convex Hull

The convex hull of a set of n points, $P = p_0, p_1, ..., p_n$, is defined as the smallest convex set that contains all the points of P. The convex set can be seen as a polygon where each vertex of the polygon is convex (has an internal angle less than π radians). For the convex set to be minimal, the vertices of the convex set must lie on a subset of the points of P. A convex hull of P can be determined in $O(n \log(n))$ time (Eddy, 1977). Figure 3.3 shows an example of a convex hull.

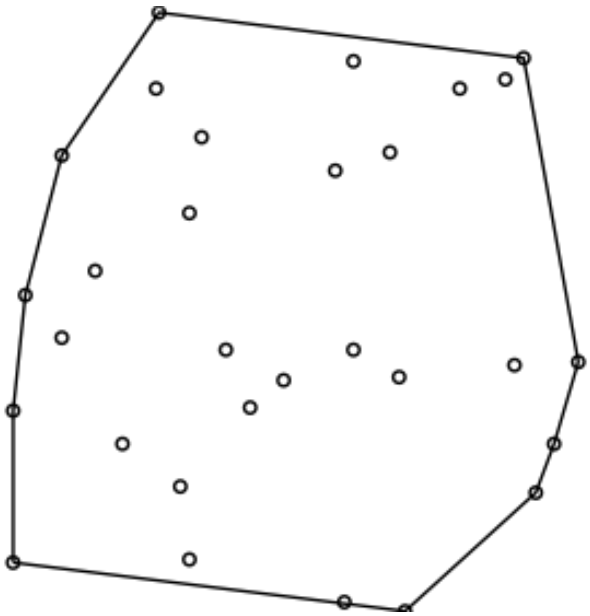


Figure 3.3: A convex hull of a set of points[2].

Andrew's monotone chain convex hull algorithm (Andrew, 1979) was used to determine the convex hull of each subset of centres. The function is first passed a range of centres on which to operate. The centres must be ordered lexicographically, which was done before the Voronoi creation process was started. A list of zeros, twice the size of the number of centres in the range is created; this list holds the elements of the chain. The complete convex hull is calculated in two steps, with the first finding the upper hull and the second finding the lower hull.

The first centre, c_0, which is also the leftmost centre and the second centre in the range, c_1, are both added to the list. We then iterate over the rest of the range, adding the next centre to our list, if the angle created by the three points at the end of the list c_{j-1}, c_j

[2]Taken from `http://www.evanjones.ca/convexhulls.html`

and c_{j+1}, is less than π radians. The three centres remain in the list and the next centre, c_{j+2} is added. If the angle is greater than π radians, the two centres at the end of the list, c_j and c_{j+1}, are removed and the process continues by adding the next point, c_{j+2}. This continues until the rightmost centre, c_n, is reached, at which time the upper hull of the convex hull has been created.

To generate the lower hull, the same process is used, but the range is iterated over in reverse, starting at the rightmost centre, c_n, and ending at the leftmost, c_0.

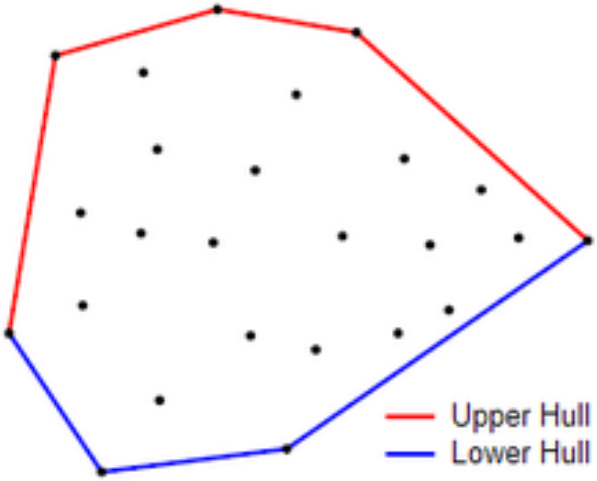


Figure 3.4: A convex hull created using Andrew's monotone convex hull algorithm with the upper hull in red and the lower hull in blue[3].

Once the list has been populated with the centres of the lower and upper hulls, the monotone chain is formed. The chain is then iterated over with each centre in the chain adding the preceding centre to its counter-clockwise parameter and the next point to its clockwise one.

3.3.4 Voronoi Merge Function

The merge begins by finding the upper and lower tangents, t_u and t_l respectively, of the sets C_L and C_R. These tangent lines are defined as connections between the convex hulls of C_L and C_R such that they form part of the convex hull of the union of C_L and C_R. Starting from the rightmost centre in C_L and the leftmost centre in C_R, the algorithm finds the upper tangent by creating two sets of three centres, $(c_i^L, c_{i-1}^L, c_j^R)$ and $(c_i^L, c_j^R, C_{j+1}^R)$

[3]Taken from `https://en.wikibooks.org/wiki/Algorithm_Implementation/Geometry/Convex_hull/Monotone_chain`

where c^L and c^R indicate centres in C_L and C_R, respectively. These overlapping sets must both satisfy the convex condition, i.e. the angle between these three centres must be less than π radians. Once these conditions are met, t_u is created between C_i^L and C_j^R. The operation moves counter-clockwise on C_L and clockwise on C_R. To find the lower tangent, t_l, the order is reversed with C_L stepping clockwise and C_R counter-clockwise. An example of this can be seen in Figure 3.5.

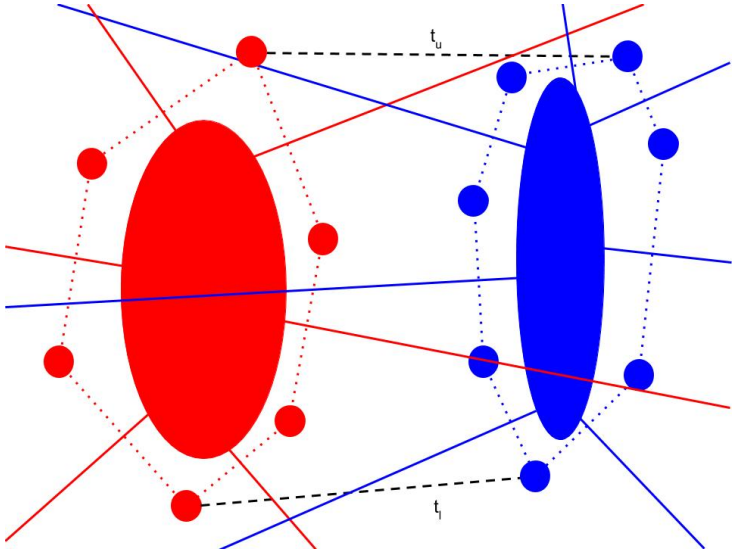


Figure 3.5: Two neighbouring tessellations, C_L(red) and C_R(blue). Only centres affected by the merge on the convex hull and their bisecting lines are shown. The upper and lower tangents, t_u and t_l are shown in black.

The upper tangent, t_u, is used to find the starting point of the polygonal chain used to merge the Voronoi. We start by creating an empty list, HP, which stores the line segments of the polygonal chain. The bisector of t_u, b_0, is used as the first line segment in the polygonal chain and is appended to HP. The endpoint of t_u with the greatest y value is determined, c_0^u, and a new tangent is generated with the lower centre, c_1^u, and a neighbour of c_0^u in the same subset as c_0^u, not necessarily in the convex hull, so long as the related line to c_0^u is both available and intercepts b_0. The bisecting line of c_0^u and its chosen neighbour is appended to a list of lines which intercepts the polygonal chain, known as *clip_lines*. The bisector of c_1^u and the chosen neighbour of c_0^u is determined and appended to HP and the process begins anew. An illustration of a step in the process can be seen in the transition from Figure 3.6 to Figure 3.7.

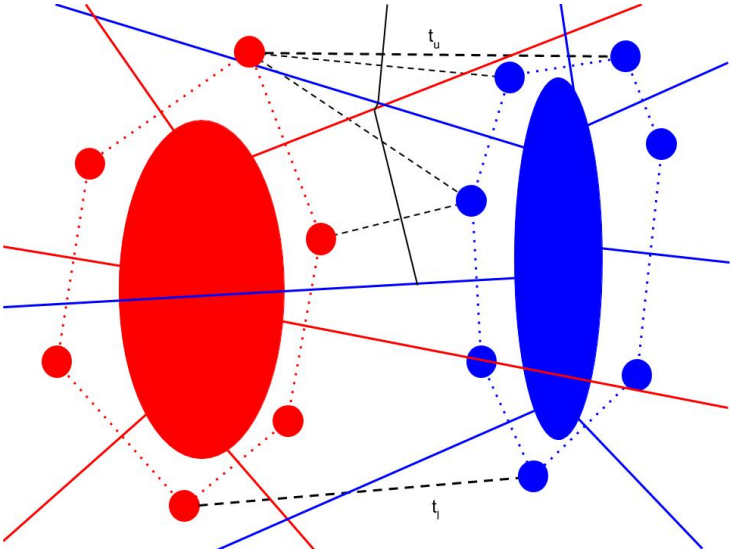


Figure 3.6: Two neighbouring Voronoi tessellations merging: the dashed black lines depicting the tangent lines and the solid black line segments depict the growing polygonal chain.

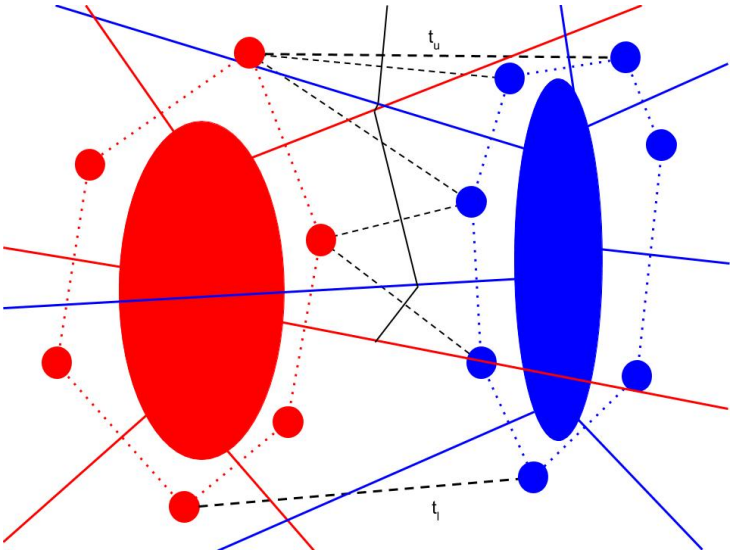


Figure 3.7: Two neighbouring Voronoi tessellations continuing to merge.

This process is repeated until the bisecting line of t_l is determined,as shown in Figure 3.8.

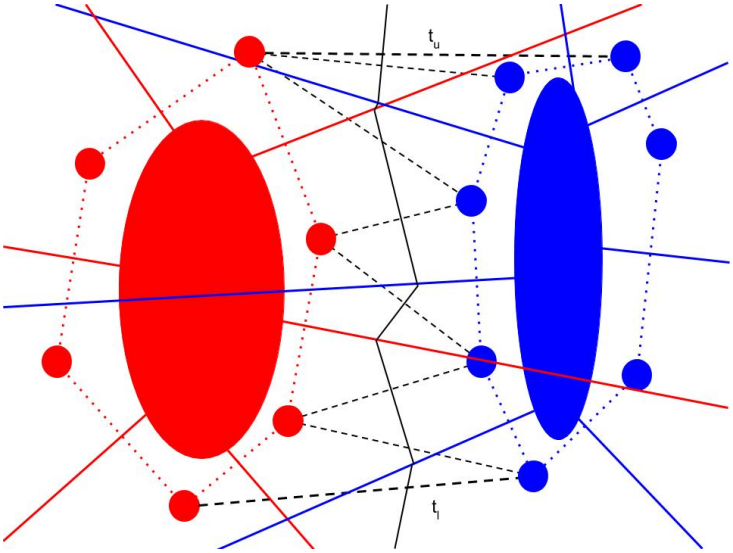


Figure 3.8: A near complete Voronoi merge: the lower tangent t_l has been reached by the polygonal chain depicted as solid black line segments.

Once complete, we are left with a list, HP, which contains the bisecting lines between points in C_L and C_R and a list of bisecting lines from C_L and C_R, *clip_lines*. The lines in HP and the centres they bisect are added to the related lists of each of the centres they bisect. The lines in *clip_lines* are cut at their intercepts and all are appended to a list of lines to be passed back along with the range of points and the new convex hull of the combined Voronoi structure. A depiction of the two Voronoi tessellations with clipped lines can be seen in Figure 3.9.

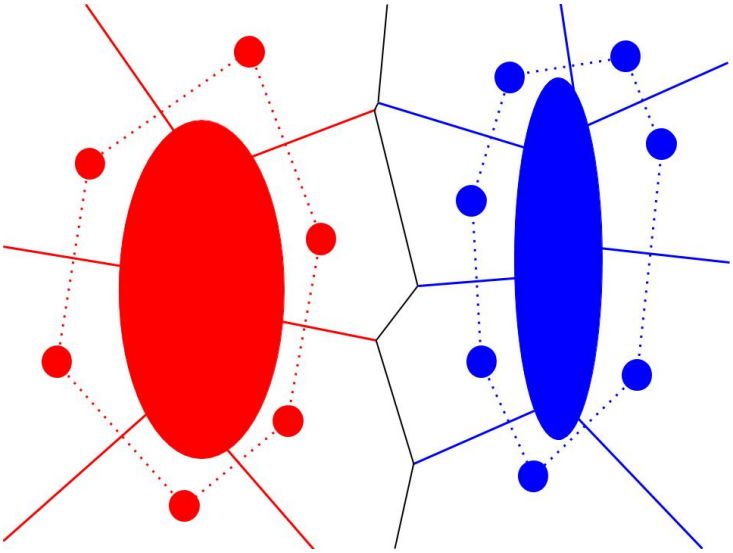


Figure 3.9: A completed Voronoi merge, with clipped lines denoting the new cell dimensions.

An example of a complete Voronoi Tessellation is illustrated in Figure 3.10.

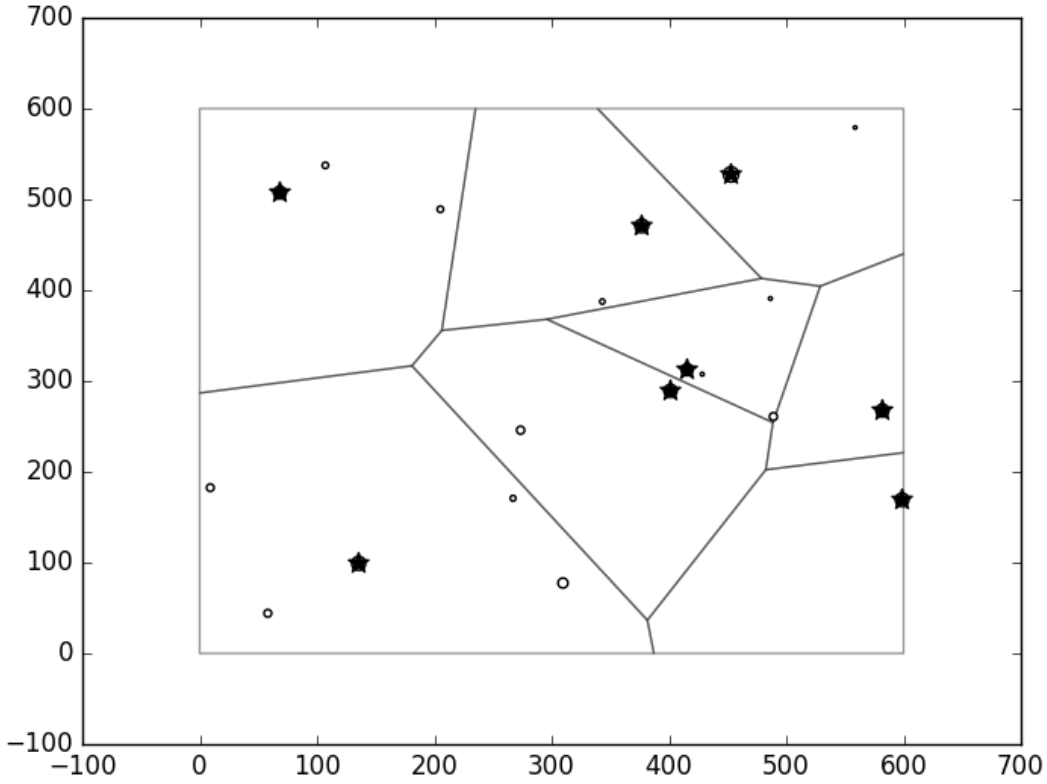


Figure 3.10: A Voronoi tessellation generated by the divide and conquer algorithm with sources as circles and stars to represent the Voronoi centres.

3.3.5 Weighted Voronoi Tessellation

An attempt was made to generate a weighted Voronoi tessellation using a distance transform which uses the intensities of the centres to redetermine the coordinates of the midpoint of the centres. This transform is defined in Equation (3.1) where z_1 and z_2, and $\vec{x_1}$ and $\vec{x_2}$ are the intensities and coordinates of c_1 and c_2, respectively.

$$d'(c_1, c_2) = \frac{z_1}{z_1 + z_2}\vec{x_1} + \frac{z_2}{z_1 + z_2}\vec{x_2} \tag{3.1}$$

This is an extension of the standard distance equation since, if $z_1 = z_2$ we obtain

$$d'(c_1, c_2) = \frac{z_1}{z_1 + z_2}\vec{x_1} + \frac{z_2}{z_1 + z_2}\vec{x_2}$$
$$d'(c_1, c_2) = \frac{1}{2}\vec{x_1} + \frac{1}{2}\vec{x_2}$$
$$d'(c_1, c_2) = \frac{\vec{x_1} + \vec{x_2}}{2}$$

Some complications were found in this during the Voronoi merge process; weighted centres that are not close enough to the convex hull but due to their larger weighting still have cells that dominate areas of the hull were not included in the merging process and their line segments were not clipped or deactivated. Another problem with this is that it generates undefined regions in the space, that is, regions where domains overlap due conflicting weightings, an example of which can be seen in Figure 3.11.

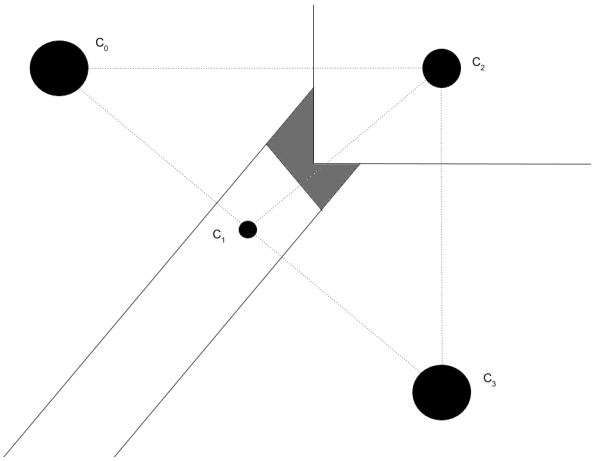


Figure 3.11: An unclassified area (grey) generated by a weighting conflict in c_1 and c_2.

The figure shows a conflicting area between c_1 and c_2. It would be expected that c_2, with its higher intensity, would claim the area, but in doing so it would cross into the area that should be the domain of c_0 or c_3. It was for this reason that the choice to use the intensities to weight the Voronoi tessellation creation process was abandoned. Instead the intensities would be used to calculate the error and determine and generate cell merges. An example of a failed weighted Voronoi tessellation can be seen in Figure 3.12.

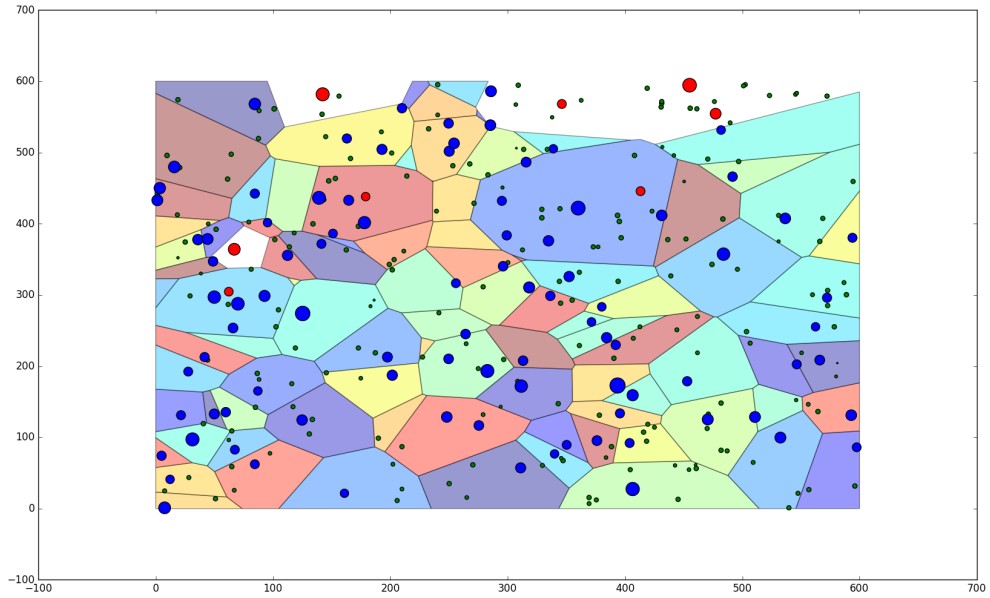


Figure 3.12: A failed visualisation of a weighted Voronoi tessellation with centres in blue, sources in green and centres without cells in red.

3.4 Cell Error

Once the Voronoi tessellation has been determined, it is re-centred based on the weighted average of the points in the cell. Sources are added to cells by determining the cell centre which is closest to it using the standard distance equation:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}, \tag{3.2}$$

where (x_i, y_i) is the location of a source in the plane and (x_j, y_j) is the location of a centre. The closest centre is such that d is minimum. For each source, s_i we therefore seek its minimum distance, d_i, such that for a set of n centres:

$$d_i = \min_j^n \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}. \tag{3.3}$$

This is done to add the influence of weaker sources in the overall correction and is especially necessary when the cell is generated by a source slightly above the intensity threshold and contains a source slightly below the threshold. We seek a new weighted centre such that

the error for a cell is minimum. The error for a cell containing N sources is defined as

$$\epsilon_j = \sum_{i=1}^{N} z_i |\vec{x_i} - \vec{x_j}|^2, \tag{3.4}$$

where $\vec{x_i} = (x_i, y_i)$ is the location, z_i is the intensity of some source in the cell, and $\vec{x_j}$ is the location of the new centre.

This error function has a local minimum at the point where its derivative with regard to $\vec{x_j}$ is zero, or

$$\begin{aligned} \frac{d\epsilon}{d\vec{x_j}} &= \sum_{i=1}^{N} \frac{d}{d\vec{x_j}} z_i (|\vec{x_i}|^2 - 2\vec{x_i} \cdot \vec{x_j} + |\vec{x_j}|^2) \\ &= \sum_{i=1}^{N} z_i (2\vec{x_i} - 2\vec{x_j}) = 0 \end{aligned}$$

or

$$2\sum_{i=1}^{N} z_i \vec{x_i} = 2\sum_{i=1}^{N} z_i \vec{x_j}.$$

Since $\vec{x_j}$ is not dependent on the sum, it can be removed and the equation reordered to give

$$\vec{x_j} = \frac{\sum_{i=1}^{N} z_i \vec{x_i}}{\sum_{i=1}^{N} z_i}. \tag{3.5}$$

From this, the new centre is determined. The intensity of the centre is determined as the sum of the intensities in the cell, or:

$$z_j = \sum_{i=1}^{N} z_i. \tag{3.6}$$

Once the cell's new centre has been obtained, its error is calculated using Equation (3.4). An example of centre correction can be seen in the transition from Figure 3.13 to Figure 3.14.

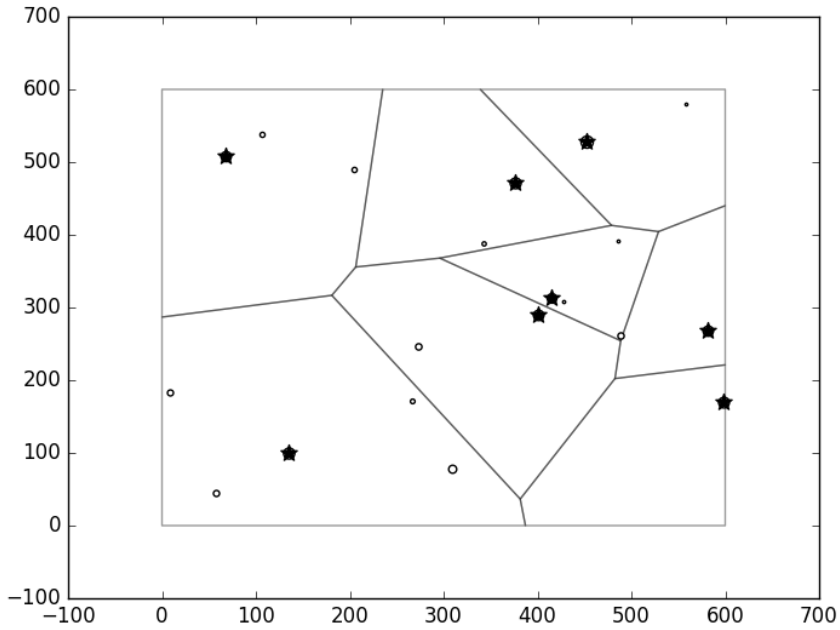


Figure 3.13: Tessellation with high intensity sources as centres.

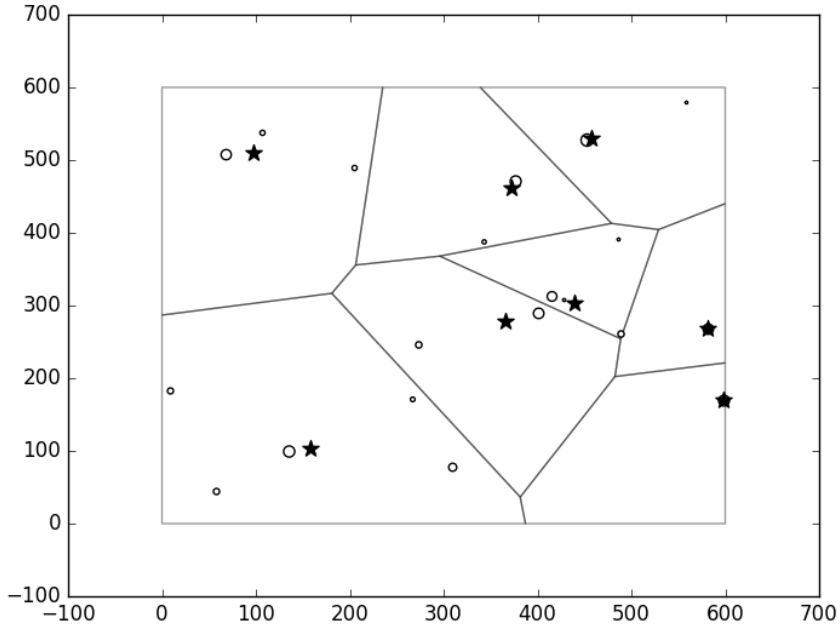


Figure 3.14: Tessellation with the weighted average of the sources in the cells as their centres.

3.5 Cell Merge

The merge process is iterative and is dependent on the total sum of the error of the cells, E. Initially, this error is relatively low as multiple cells are generated with one or very few sources contained in each cell. To keep the maximum error threshold relative, unless it is given as an input by the user, it is calculated as the product of the set standard deviation (σ), the size of the plane (x_{plane}, y_{plane}) and the number of sources in the plane ($|S|$) or

$$E_{max} = \sigma x_{plane} y_{plane} |S|. \tag{3.7}$$

The process begins by summing the errors of the cells and then goes through an iterative process of finding the best merge, checking if implementing the best merge exceeds the maximum error threshold and, if not, implementing the best merge.

3.5.1 Obtaining the best merge

The best merge is obtained by iterating over the list of centres and, for each centre, testing it with its active neighbouring centres.

The merge test works by calculating a new centre, $c_{new} = (\vec{X}, Z)$, determined by two existing centres, c_1 and c_2, with intensities z_1 and z_2, and positions $\vec{x_1}$ and $\vec{x_2}$, respectively, as

$$\vec{X} = t\vec{x_1} + (1-t)\vec{x_2} \text{ with } t = \frac{z_1}{z_1 + z_2}. \tag{3.8}$$

The new weight for the merged centre is defined as

$$Z = z_1 + z_2. \tag{3.9}$$

Since x_1 and x_2 are centred sums of the positions of the sources in their cells, expanding them to their original forms yields

$$\vec{x_1} = \frac{\sum_{i=1}^{N} z_{1i}\vec{x_{1i}}}{\sum_{i=1}^{N} z_{1i}} \text{ and } \vec{z_2} = \frac{\sum_{i=1}^{M} z_{2i}\vec{x_{2i}}}{\sum_{i=1}^{M} z_{2i}}.$$

Note that:

$$z_j = \sum_{i=1}^{N} z_{ji}.$$

Substituting these into Equation (3.8), we obtain

$$\begin{aligned}
\vec{X} &= t\vec{x_1} + (1-t)\vec{x_2} \\
&= \frac{z_1}{z_1+z_2}\frac{\sum_{i=1}^{N} z_{1i}\vec{x_{1i}}}{z_1} + (\frac{z_1+z_2}{z_1+z_2} - \frac{z_1}{z_1+z_2})\frac{\sum_{i=1}^{M} z_{2i}\vec{x_{2i}}}{z_2} \\
&= \frac{\sum_{i=1}^{N} z_{1i}\vec{x_{1i}}}{z_1+z_2} + \frac{\sum_{i=1}^{M} z_{2i}\vec{x_{2i}}}{z_1+z_2} \\
&= \frac{\sum_{i=1}^{N} z_{1i}\vec{x_{1i}} + \sum_{i=1}^{M} z_{2i}\vec{x_{2i}}}{z_1+z_2} \\
&= \frac{\sum_{i=1}^{N} z_{1i}\vec{x_{1i}} + \sum_{i=1}^{M} z_{2i}\vec{x_{2i}}}{\sum_{i=1}^{N} z_{1i} + \sum_{i=1}^{M} z_{2i}}.
\end{aligned}$$

This shows that Equation (3.8) is equivalent to finding the centre of all the sources in both x_1 and x_2.

Once the new centre has been found, the error must be determined by summing the square of the weighted distances to the new centre from each source. Once determined, the new centres coordinates and intensity, as well as the new error are returned. The error itself is not compared to find the best merge, but rather the change in error, that is, the merge that produces the lowest increase in the error. This is done by taking the tested merge error and subtracting from it the errors of the centres used to generate it; i.e., we seek $\Delta_{i,j}$ such that for some cells c_i and $c_2 j$, the comparison with the error of the merged cell, $c_{i,j}$ is:

$$\Delta_{i,j} = \min_{i=1}^{n} \min_{j=1, i\neq j}^{n} c_{i,j} - (c_i + c_j). \tag{3.10}$$

Once found, the result is stored along with the new coordinates of the centre, $c_{i,j}$ and the centres used to generate it.

Once the best merge has been found, it must be determined whether the new sum of errors exceeds the threshold, i.e. $E + \Delta_{i,j} \geq E_{max}$. If it does, the merge process is halted and the Voronoi structure is returned. If the threshold is set too high, it may occur that all the cells merge into a single cell. In this case, the process is again halted as no best merge could be found as the threshold was set too high for the system. The Voronoi structure is returned as only the set of sources and a single centre with no bisecting lines remaining and so no merged Voronoi can be generated. If the system finds a valid merge which is still less than the threshold, it adds the difference in the merge error to the total error and executes the merge.

3.5.2 Executing the merge

The merge execution algorithm takes in the new coordinates and intensity, $\vec{X}$, the new error, Z, as well as the centre, c_1, and the related centre with which it is to be merged, c_2. The algorithm starts by setting the coordinates and intensity of c_1 to those of $c_{1,2} = (\vec{X}, Z)$. It then deprecates the line relating c_1 to c_2 and appends the list of sources in c_2 to that of c_1. The error of c_1, z_1, is set to that of the new error, Z. Centre c_2 and its list of consumed centres are added to the list of consumed centres of c_1. Finally, the list of centres is iterated over and any centre which references c_2 is changed to reference c_1 and all lines which relate other centres to c_2 are changed to relate to c_1 instead. Once this is complete, the process of finding the best merge restarts until the error threshold is reached.

The transition from Figure 3.15 to Figure 3.16 shows the effects of a single merge. The weighted distance between the merged centres in Figure 3.15 is less than those of all other neighbouring centre pairs.

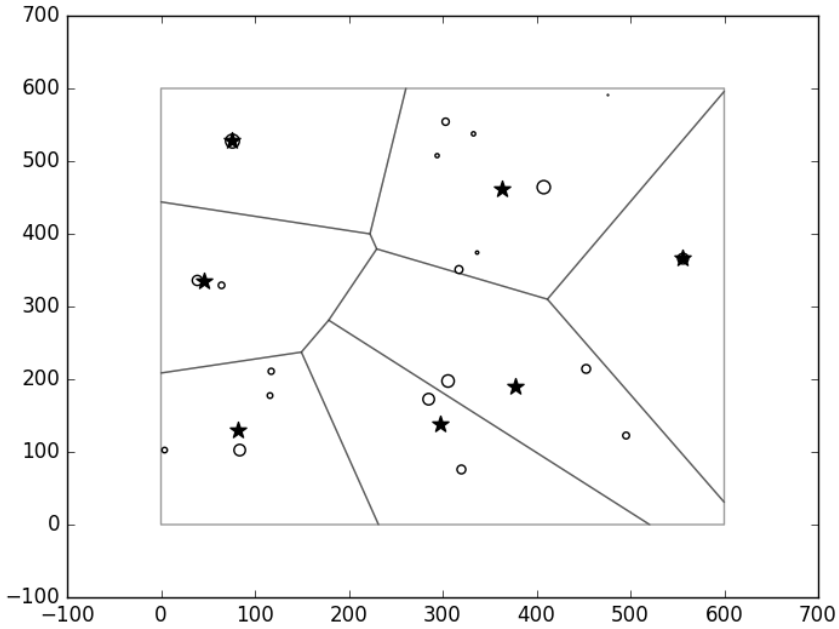


Figure 3.15: Re-centred Voronoi before merge.

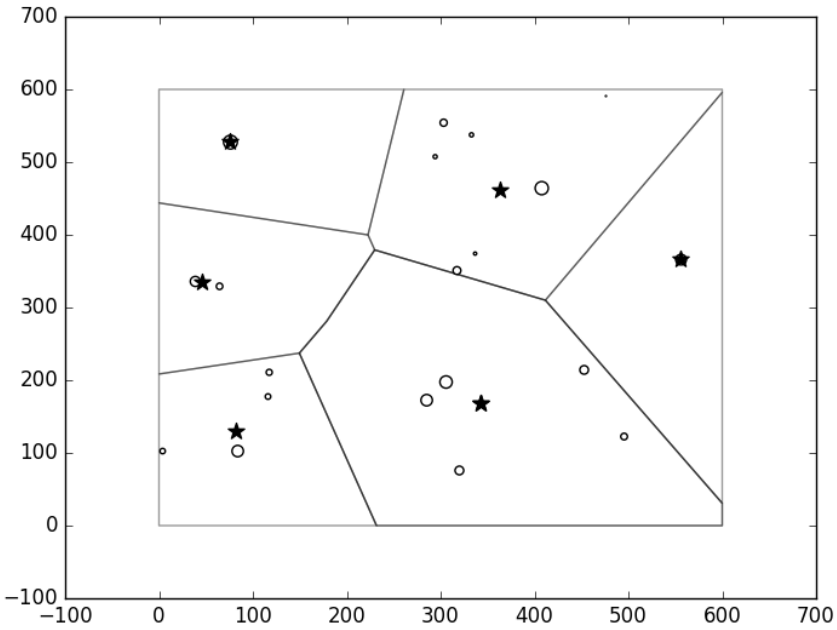


Figure 3.16: Re-centred Voronoi after single merge.

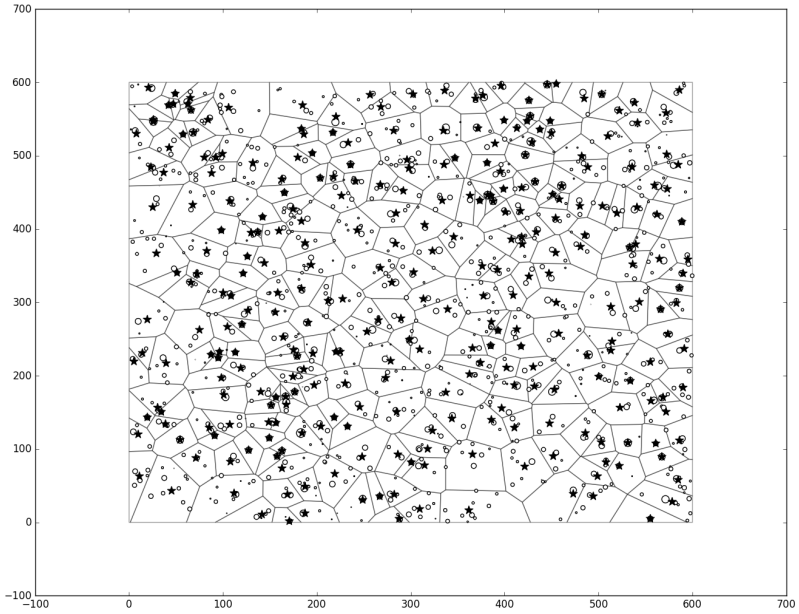


Figure 3.17: Re-centred Voronoi with 1000 sources and 301 centres before merge.

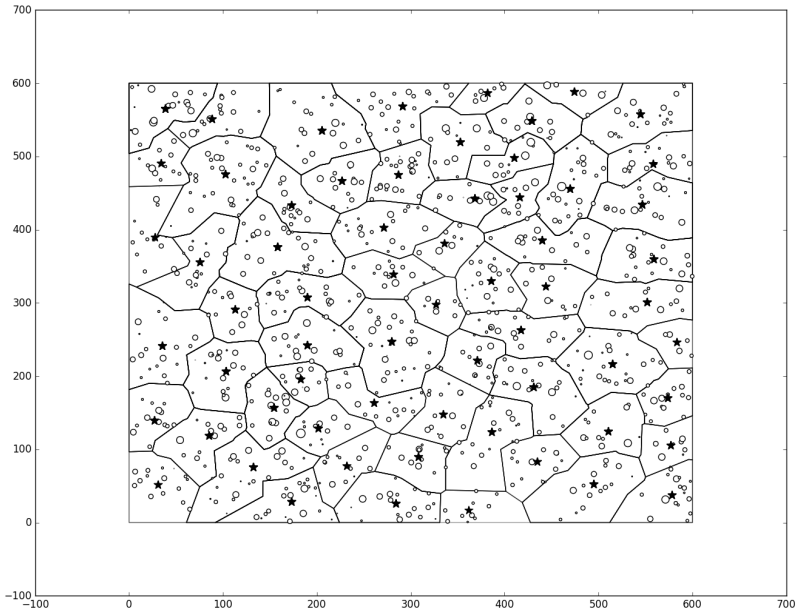


Figure 3.18: Re-centred Voronoi with 1000 sources and 64 centres after completing the merge process.

Figure 3.17 shows an initial Voronoi tessellation while Figure 3.18 shows its merged structure once the error threshold has been reached. The latter figure has larger structures with cells generally being more convex depending on the layout of sources in the cell.

Chapter 4

Parallel Implementation of the Tessellation Algorithm

It is evident that areas of the process can be optimised by using a parallel processing unit such as a GPU. Areas that can be optimised include the generation of the Voronoi, the divide and conquer can spawn two threads to compute the left and right diagrams, it would then simply take the diagrams returned by the threads it spawned, perform the merge operation and pass it back to it's parent thread. CUDA is limited to a nested depth of 24 calls (NVIDIA, 2015), which therefore allows for up to 2^{25} or 33554432 centres, this exceeds the expected number of centres since the number of sources is generally 10000 which is far below this. It should be noted that the GPU, with 2GB of memory, will likely run out of memory before the maximum recursive depth is reached. The maximum number of centres is set to 2^{25} instead of 2^{24} since the minimum number of points needed to generate a Voronoi tessellation for a given thread at the base case is two.

The other possible case for GPU parallelisation is determining the best possible merge to be executed. The merge operation itself cannot occur in parallel this may lead to conflicting merges if a single cell can optimally merge with multiple neighbours. For the case of the merge, since there is very little nested function calling, the only restriction on our data is the memory capacity of the GPU. For this research, we will only look at parallel computation of finding the best merge.

4.1 Numba

Numba is an optimized compiler for python. It's development project is sponsored by Continuum Analytics and is available either through Continuum's high performance python distribution platform, Anaconda or by directly downloading the compiler from the site[1].

Numba is optimized for array-based, mathematically intensive code and uses a just-in-time (JIT) compiler to compile code to native machine instructions to produce performance similar to that of C, C++ and Fortran.

GPU support is also included into Numba, as well as an API to interface with NVIDIA's GPU programming language, CUDA.

Numba is able to convert a strict subset of python code into CUDA. It allows users to write python code in a CUDA-like style to create kernels which are executable on the GPU through a CUDA JIT compiler by using the handle `@cuda.jit`. Numba CUDA is able to interpret simple data structures and numpy arrays, which are arrays who's structure is localised to an area of memory for faster access than standard python lists or tuples. These data structures are passed to and from the GPU dynamically with each kernel call, or can be implicitly transferred to the GPU by the user.

4.2 Data Restructuring

The Voronoi data structure, as stated in the previous chapter, consists of centres (Section 3.2) and lines (Section 3.3.1) linked together by relational centre-line tuples stored in the centre's list of related centres. While this structure is relatively simple for a sequential CPU system to traverse, GPU's are designed to make use of sequential data readings and therefore traversing a large structure of pointers would be difficult for the GPU to traverse. Increasing the need for a data restructuring is the fact that a structure of values with pointers pointing to different locations in memory can be difficult to transfer to a GPU which is designed to operate on basic data types.

[1] http://numba.pydata.org/

4.2.1 Data Selection

It was noted that not all of the data from the Voronoi structure would be needed. In order to obtain the best merge, the data that would be needed would be:

1. The location of the centre of the cell.
2. The intensity of the cell.
3. The error of the cell.
4. The neighbouring cells of the cell.
5. The sources located in the cell.

It was therefore determined that three multidimensional arrays would be created to store the necessary data to obtain the best possible merge. Given n centres, the arrays are defined as:

1. An $n \times 4$ array, named textttcentres, of 32 bit floats to store the position, intensity and error of the cell.
2. An $n \times n$ array of integers storing the indexes of the cells to which the cell is related to. The size of the array is large to allow space for cells with a many neighbours and allow for additions of neighbours through merges. The array is named `related`.
3. Given s initial sources, an $n \times s \times 3$ array of 32 bit floats is generated to hold the positions and intensities of the sources in a cell, this array is also set to be large enough to allow for a single cell to hold all the sources should the error be set too high and the merge continue to a single cell. The array is named `sources`.

The creation of these arrays can be seen in Listing 4.1

Listing 4.1: Array reconstruction for transfer to the GPU.

```
import numpy as np

centre = np.zeros((len(points),4),dtype='float32')
related = np.zeros((len(points),len(points)),dtype='int32')
sources = np.zeros((len(points),numobj,3),dtype='float32')
```

4.2.2 Data Transfer

The three arrays are defined and populated on the CPU but processed on the GPU. While it is possible to leave the data on the CPU and have Numba dynamically transfer the data between the CPU and GPU with each iteration of the merge test kernel call, since the data is minimally altered with each iteration, it would be more feasible to transfer the data to the GPU once and alter the data on the GPU after each iteration of the merge process.

Data is transferred from the host (CPU) to the device (GPU) using Numba's `cuda.to_device()` function and an example of this can be seen in Listing 4.2 below.

Listing 4.2: Transferring arrays to GPU.

```
from numba import cuda
...
d_centre = cuda.to_device(centre)
d_related = cuda.to_device(related)
d_sources = cuda.to_device(sources)
```

4.3 Parallel Merge Search

Once the data arrays have been transferred to the GPU, the iterative merge process can now begin, while similar to the merge described in Section 3.5, the GPU merge works by calculating the potential merge of each points with its neighbours simultaneously instead of sequentially as previously described.

It begins by generating a grid $\frac{n}{32} + 1$ one dimensional blocks each containing 32 threads. It then calls the `d_get_best` kernel to find the best merge. The kernel first uses the `cuda.grid()` function to obtain the processing threads global ID. It then checks to see if the thread is within the range of the number of cells.

If the thread is within the range it continues by creating two arrays local to the thread, `best` and `test`. If so, it iterates through the corresponding column of the `related` array and tests the merge using the same testing method as described in Section 3.5 with output data for each neighbouring point stored in `test`. If the error of `test` is found to be lower

than that stored in `best`, which is initialised to contain the maximum threshold error, its values are copued into `best`, if not, it is overwritten in the next call of the merge test with the next neighbour. The `test` (and `best`) arrays are structured as a seven-element array with the follows attributes:

1. The x coordinate of the merged cell centre.
2. The y coordinate of the merged cell centre.
3. The intensity, z, coordinate of the merged cell centre.
4. The total error of the merged cell.
5. The position of the related cell being tested to merge in the `centre` array.
6. The position of the current cell being tested to merge, also the thread ID.
7. The change in error δ, between the merge tested cell and its parent cells

Once the best merge for a given cell has been found it is stored in an $n \times 7$ array, `d_results` which was created on the GPU before the merge iteration began using the Numba command `cuda.device_array()`. The values are stored in the column corresponding to the thread ID. Once all threads have stopped executing by filling the `d_result` array with the best merges, `d_results` is copied back to the CPU with Numba's `copy_to_host()` function into an array called `results`.

The array `results` is then iterated over to find the column with the lowest change in error. Once found, the change in error is checked to see that it does not exceed the maximum error threshold, if it does, the execution is halted and the resulting tessellation structure is returned. If the error threshold is not reached, the change in error is added to the value of the tessellations total error and the merge is executed.

4.4 Executing the Merge

Once the best merge is found and the error deemed to be lower than the maximum error threshold, the merge can begin to be executed. While the merge execution on the CPU is nearly identical to that of the sequential merge algorithm, the changes must also be reflected in the GPU's representation of the data, should the merge process continue, so that the correct new merge can be obtained in the next iteration.

4.4.1 CPU Merge

As stated the CPU merge is nearly identical to that of the sequential merge with the exception being that the parallel CPU merge function takes in a list containing the positions of the cells to be merged instead of being passed pointers to their location in memory.

4.4.2 GPU Merge

The best merge in the `results` array is copied back to the GPU and the kernel `d_do_merge` is called to execute the merge on the GPU with the same grid and block parameters as the `d_get_best` kernel. The kernel begins by using the `cuda.grid` function to get the global thread ID of the processing thread and checks to see if the thread is within the range of the number of cells. The `related` array is iterated over and any element containing the value of the related merging cell, `best[4]`, is changed to the value of the main merging cell, `best[5]`.

Next, the thread ID is checked to be equal to that of the main merging cell. This thread then change the values of the coordinates, intensity and error of the main merging cell to that of the merged cell. The related cells of the related merging cell are appended to those of the main merging cell's list in the `related` array. The sources of the related merging cell are appended to that of the main merging cells in `sources`.

The function finally checks the thread ID to see if it is that of the related merging cell. This thread then changes the values of the position and intensity of the cell to that of the maximum error threshold, to ensure it will be unable to merge with any other cells and sets its error to zero in the `centre` array.

Once these changes have been made, the kernel ends its execution and the process begins again until the maximum error is reached.

Chapter 5

Results and Discussion

5.1 Cell Merge Performance

In order to compare the performance of the cell merge to that of the Voronoi tessellation a set of 1000 unique sources was randomly generated. Using these sources, 936 tessellations were generated using each algorithm giving 64 to 1000 cells per generation. For the Voronoi, the brightest sources were chosen as centres for tessellation for each iteration, the Voronoi's were then re-centred to minimise the error as much as possible. For each iteration of the cell merge, the space was tessellated using all 1000 sources as centres and the merge process was iterated until the desired number of cells was reached. For each iteration of both algorithms, the total error for the space was calculated and stored. While both algorithms place their centres optimally within their cells, the structure of the cells generated in the cell merge are dependent on the intensities and positions of all sources in the space while those in the Voronoi are generated in a way that is dependent on only the positions of the sources with the highest intensities. Figure 5.1 shows the error for a set number of cells for a fixed set of sources and Figure 5.2 shows the logarithmic scaled version of the resulting error data.

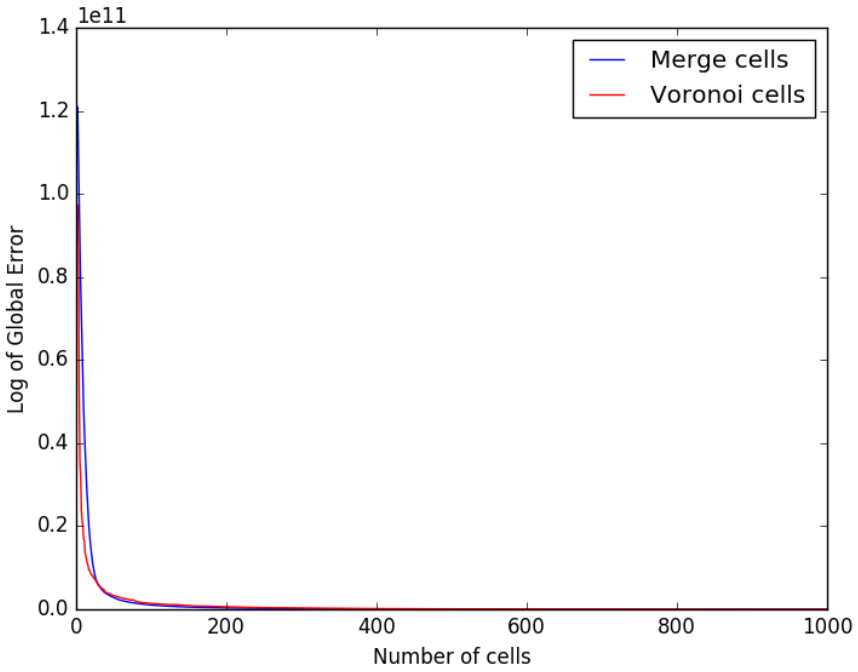


Figure 5.1: A comparison of the total error of the Voronoi tessellation (red) and the cell merge (blue) algorithms.

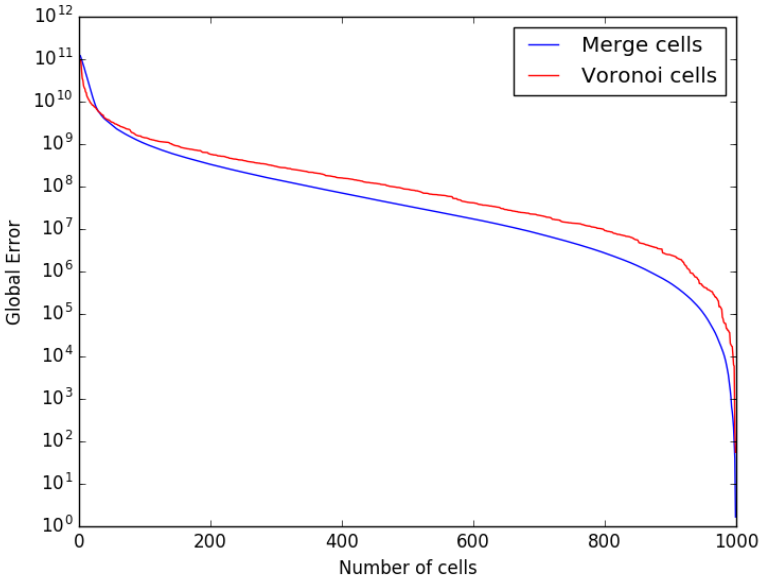


Figure 5.2: A logarithmic scale of the data presented in Figure 5.1.

As shown, both algorithms have zero error when generating 1000 cells but once the number of cells decrease, the error begins increasing exponentially. Figure 5.1 therefore shows that

the cell merge has a lower error compared to that of the Voronoi for a high number of centres. Figure 5.3 shows that for a very low number of cell, the Voronoi becomes more feasible. While both still have extremely large errors, the Voronoi's is lower than that of the cell merge until they once again meet when a single cell is used to generate the Voronoi.

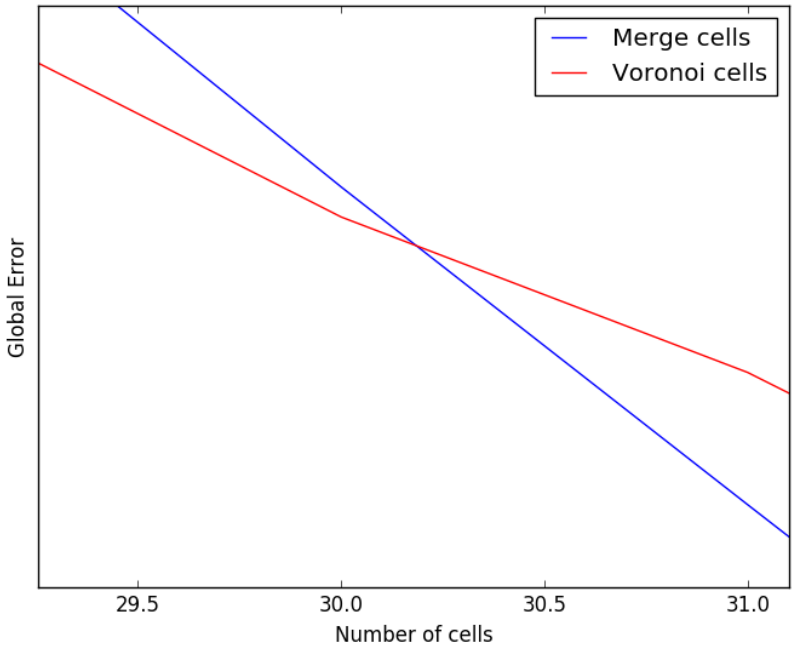


Figure 5.3: A close up of the logarithmic scale data showing the point at which the cell merge and Voronoi intercept.

5.2 Parallel Merge Performance

In order to calculate the efficiency of the parallelised section of the merge algorithm, the cell merge algorithm was timed from the moment the parallel and sequential code diverged to the moment at which they become the same again to output their results. The cell merge for both sequential and parallel merges was tested using the same set of data sources, the size of the data source varied with each iteration having 50, 100, 300, 500, 800, and 1000 sources being used to tessellate and then merge. The merge halting criteria is the error threshold defined by Equation 3.7 in Section 3.5. The resulting times for the algorithm to generate the merges can be seen in Figure 5.4 with a base 10 logarithmic graph of the results in Figure 5.5.

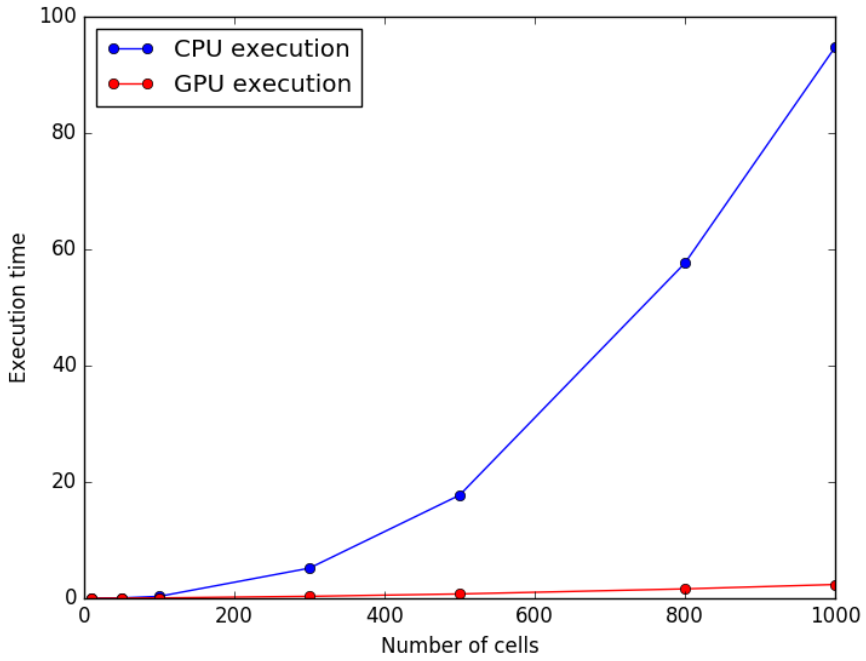


Figure 5.4: A comparison of the computation time of a sequential and parallel execution of the cell merge algorithm.

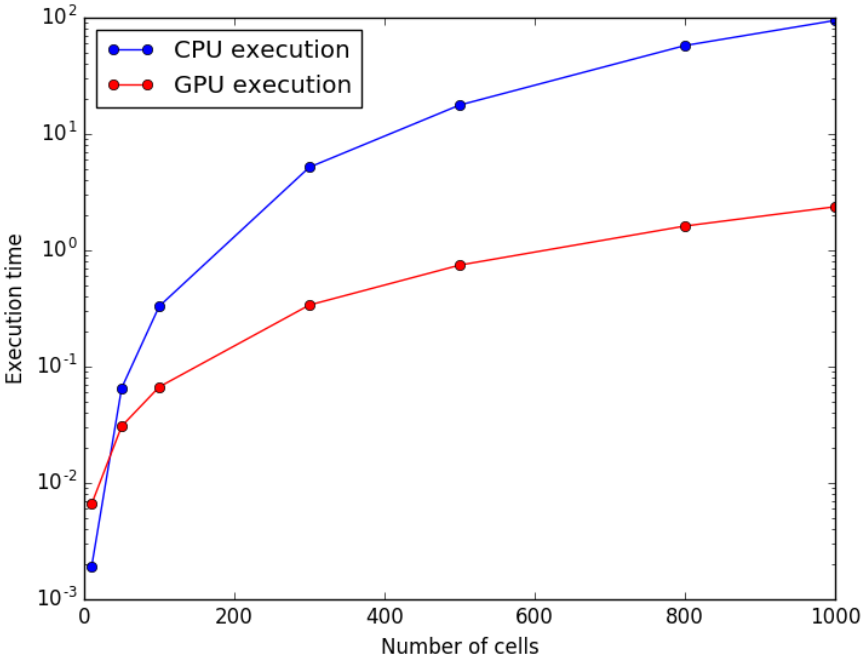


Figure 5.5: A logarithmic scale graph of the values depicted in Figure 5.4.

Observing these results it is clear that the GPU outperforms the CPU with its execution time for a higher number of sources. However, for a low number of sources, the overhead costs associated with the set up process before merging can begin on the GPU make it slower than the CPU. Figure 5.6 shows the speed-up of the parallel merge as compared to that of the sequential. The speed-up is calculated as $speedup = \frac{t_{sequential}}{t_{parallel}}$.

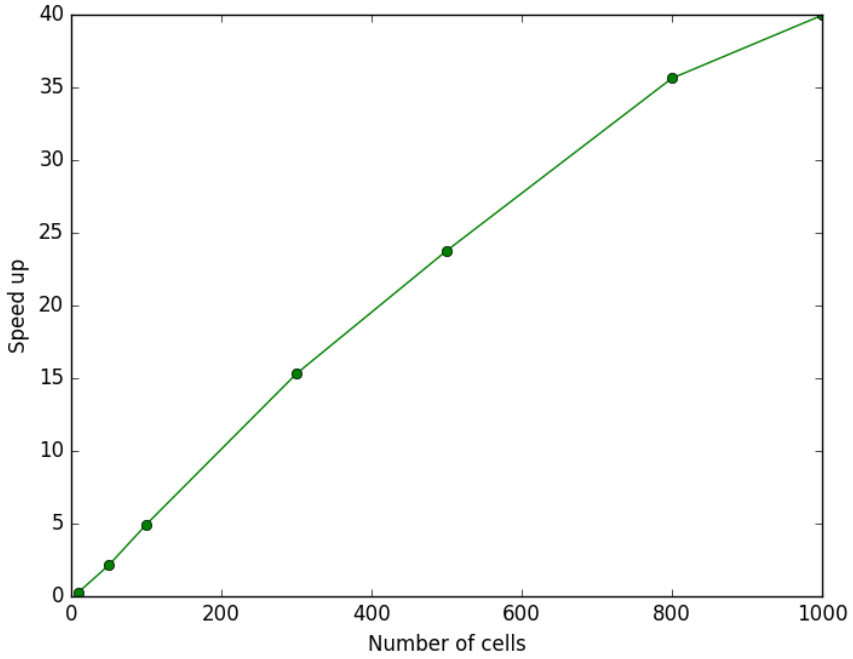


Figure 5.6: Speed-up of the parallel process using the times generated by Figure 5.4.

From this, it can be seen that the speed-up increases as the number of sources increases. With 1000 sources, a speed-up of 39.96x is obtained. Given that the average speed-up of a parallel implementation of a process, as stated by (Lee *et al.*, 2010), is within an order of magnitude of the sequential process, a speed-up of 39.96x can be seen as a great improvement in computation time.

In order to better analyse the speed-up and provide better insight as to how the parallel implementation operates and what makes up the speeds obtained, the times for each step of the parallel process must be analysed. Figure 5.7 shows the time taken for each step of the process to execute for a given number of sources.

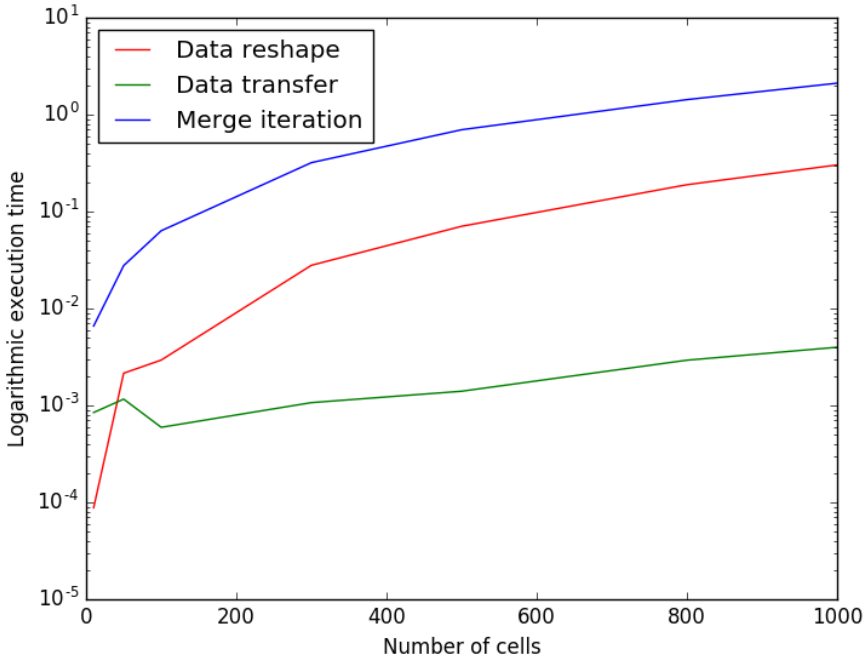


Figure 5.7: Breakdown of the time taken for each subprocess in the parallel merge process.

Figure 5.7 shows that, for merges with a fewer number sources, the transfer time to the GPU is relatively constant, while the merge iteration time and data reshape time scale up appropriately, this constant initial transfer cost is therefore what leads to the sequential process outperforming the parallel for smaller sets of sources. For larger sets of sources, the merge iteration and data reshaping time increases at a much higher rate than that of the data transfer. This again reinforcing the parallel implementation as the preferred operation for larger sets of sources.

While the speed-up of the parallel implementation of the cell merge algorithm is noteworthy, it should also be stated that, even with further optimisation, it will only increase to a theoretical limit. Summarising what was stated by (Amdahl, 1967), we find that the overall speed-up can be calculated as:

$$speedup = \frac{1}{(1-p) + \frac{p}{s}}, \tag{5.1}$$

where p is the percentage of the overall process which can be parallelised and s the speed up of the parallelised portion alone. As with all parallel systems, the cell merge is therefore limited to a maximum speed-up.

The leading limitation of the GPU is one of memory. Large arrays are created to store the relational data on the GPU. For n sources, it can be seen that the memory requirements to compute the merge is on the order of $O(n^2)$ since the `d_related` and `d_sources` arrays are $n \times n$ and $n \times n \times 3$, respectively where each source is initially used as a centre. Therefore, given the the NVIDIA GTX 750 Ti has an internal memory capacity of 2 GB, it was found that the maximum number of sources which could be merged on the GPU was restricted to around 7500. This is below the average number of sources for an image generated by a radio telescope which is generally on the order of 10000 sources.

Chapter 6

Conclusions

6.1 Thesis Summary

In this thesis an improvement to the current best model for generating tessellations for correcting direction dependent effects. We began by first researching what direction dependant effects are and why they need to be corrected for. Tessellation models, mainly the Voronoi tessellation, were researched as well as different methods of implementing Voronoi tessellations and clustering algorithms. Lastly parallel paradigms, GPUs, GPU architecture and CUDA, the NVIDIA GPU programming language were analysed for their potential contribution to this thesis.

The algorithm that would be used to generate the tessellation was then defined. It was decided that, optimally, the tessellation would occur in two stages. The first stage would be a Voronoi tessellation which was generated using most or all of the data sources as cell centres. For this the divide and conquer algorithm was chosen for its efficient $O(n \log n)$ computation time and also for its potential to be converted to a parallel algorithm. The second part of the tessellation used the error of the cells to find and execute optimal cell merges to lower the number of cells while increasing the overall error in a minimal way. The cell merged worked by iteratively cycling through finding the best merge to execute by testing the merge of each cell with its neighbour and, once found, executing the best merge until the maximum error threshold is reached.

The cell merge process within the main algorithm was ported to execute as a parallel process on the GPU. Numba's CUDA JIT compiler was used to convert specific python

commands and functions to CUDA code for execution of the GPU. The conversion of the data, from a large relational network of pointers to a set of multidimensional arrays of a fixed size was discussed. The parallelisation of the merge testing algorithm was discussed as well as executing the merge on both the CPUs host memory and on the GPU. Problems with the GPU execution due to errors in float calculations were also noted.

Results from testing the algorithm were then analysed. It was found that for most tessellations for a given data set, the cell merge algorithm had a lower error than that of using just the Voronoi. For a very low, $\leq 3\%$, cells per source, the Voronoi then becomes more effective with lower errors. For the GPU implementation of the cell merge, it was found that the speed-up of the algorithm varied with the number of sources. For a tessellation with 10 sources, it was found that the GPU execution was slower than that of the sequential algorithm while, for 1000 sources, the speed-up was 39.96x. The cause of the change in the speed-up was found to be the overhead associated with the GPU algorithm. For a smaller set of sources, the overhead cost is relatively high but increases at a much lower rate than that of the merge execution itself, which is the main process being sped-up.

6.2 Contribution of Research

Referring to the research objectives stated in Section 1.2, the contributions of the research done in this thesis are:

1. The divide and conquer algorithm was implemented to generate the Voronoi tessellation faster than that of the current best model which used a naive k-means clustering of all points in the space to generate the tessellation. The neighbouring structure generated by the divide and conquer was also beneficial for the cell merge algorithm implemented later.
2.
3.
4.

6.3 Future Work

References

Amdahl, Gene M. 1967. Validity of the single processor approach to achieving large scale computing capabilities. *Pages 483–485 of: Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM.

Andrew, Alex M. 1979. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, **9**(5), 216–219.

Arthur, David, & Vassilvitskii, Sergei. 2007. k-means++: The advantages of careful seeding. *Pages 1027–1035 of: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics.

Aurenhammer, Franz. 1987. Power diagrams: properties, algorithms and applications. *SIAM Journal on Computing*, **16**(1), 78–96.

Cheng, Jingquan. 2009. Radio Telescope Design. *The Principles of Astronomical Telescope Design*.

Eddy, William F. 1977. A new convex hull algorithm for planar sets. *ACM Transactions on Mathematical Software (TOMS)*, **3**(4), 398–403.

Fortune, Steven. 1987. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, **2**(1-4), 153–174.

Green, Peter J, & Sibson, Robin. 1978. Computing Dirichlet tessellations in the plane. *The Computer Journal*, **21**(2), 168–173.

Hamerly, Greg. Making k-means even faster. SIAM.

Lee, Victor W, Kim, Changkyu, Chhugani, Jatin, Deisher, Michael, Kim, Daehyun, Nguyen, Anthony D, Satish, Nadathur, Smelyanskiy, Mikhail, Chennupaty, Srinivas, Hammarlund, Per, *et al.* 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *ACM SIGARCH Computer Architecture News*, **38**(3), 451–460.

Moore, Gordon E. 2006. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp. 114 ff. *IEEE Solid-State Circuits Newsletter*, **3**(20), 33–35.

NRAO. 2016. *What is Radio Astronomy?* `https://public.nrao.edu/radioastronomy/what-is-radio-astronomy`. Accessed on: 23/10/2016.

NVIDIA. 2014. *GeForce GTX 750 Ti Whitepaper.* `http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf`. Accessed on: 26/04/2016.

NVIDIA. 2015. *CUDA C Programming Guide.* `http://docs.nvidia.com/cuda/`. Accessed on: 03/05/2016.

NVIDIA. 2016. *CUDA.* `http://www.nvidia.com/object/cuda_home_new.html`. Accessed on 26/05/2016.

NVIDIA. 2016a. *GeForce GTX 750 Ti.* `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-750-ti`. Accessed on: 26/04/2016.

NVIDIA. 2016b. *GPU-Accelerated Libraries.* `https://developer.nvidia.com/gpu-accelerated-libraries`. Accessed on: 22/05/2016.

Okabe, Atsuyuki, Boots, Barry, Sugihara, Kokichi, & Chiu, Sung Nok. 2009. *Spatial tessellations: concepts and applications of Voronoi diagrams.* Vol. 501. John Wiley & Sons.

Rajan, Krishna. 2013. *Informatics for materials science and engineering: data-driven discovery for accelerated experimentation and application.* Butterworth-Heinemann.

Rong, Guodong, & Tan, Tiow-Seng. 2006. Jump flooding in GPU with applications to Voronoi diagram and distance transform. *Pages 109–116 of: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games.* ACM.

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

Shamos, Michael Ian, & Hoey, Dan. 1975. Closest-point problems. *Pages 151–162 of: Foundations of Computer Science, 1975., 16th Annual Symposium on.* IEEE.

Smirnov, Oleg M. 2011. Revisiting the radio interferometer measurement equation-I. A full-sky Jones formalism. *Astronomy & Astrophysics*, **527**, A106.

Smirnov, OM, & Tasse, Cyril. 2015. Radio interferometric gain calibration as a complex optimization problem. *Monthly Notices of the Royal Astronomical Society*, **449**(3), 2668–2684.

Steinbach, Michael, Karypis, George, Kumar, Vipin, *et al.* 2000. A comparison of document clustering techniques. *Pages 525–526 of: KDD workshop on text mining*, vol. 400. Boston.

Subhlok, Jaspal, Stichnoth, James M, O'hallaron, David R, & Gross, Thomas. 1993. Exploiting task and data parallelism on a multicomputer. *Pages 13–22 of: ACM SIGPLAN Notices*, vol. 28. ACM.

Tasse, Cyril. 2014. Applying Wirtinger derivatives to the radio interferometry calibration problem. *arXiv preprint arXiv:1410.8706*.

Tasse, Cyril. 2016. *DDFacet imager*. TBD. Draft in preparation.

Thompson, A Richard, Moran, James M, & Swenson Jr, George W. 2008. *Interferometry and synthesis in radio astronomy*. John Wiley & Sons.

van Weeren, RJ, Williams, WL, Hardcastle, MJ, Shimwell, TW, Rafferty, DA, Sabater, J, Heald, G, Sridhar, SS, Dijkema, TJ, Brunetti, G, *et al.* 2016. LOFAR facet calibration. *arXiv preprint arXiv:1601.05422*.

Vuduc, Richard, & Choi, Jee. 2013. A brief history and introduction to GPGPU. *Pages 9–23 of: Modern Accelerator Technologies for Geographic Information Science*. Springer.

Way, Michael J, Scargle, Jeffrey D, Ali, Kamal M, & Srivastava, Ashok N. 2012. *Advances in machine learning and data mining for astronomy*. CRC Press.