Rhodes University

Submitted in partial fulfilment
of the requirements of the degree of

Bachelor of Science (Honours)

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

Antonio Bradley Peters

supervised by

Prof. Karen Bradshaw

Prof. Denis Pollney

project originated by

Dr. Cyril Tasse & Prof. Oleg Smirnov

October 17, 2016

Contents

Chapter 1

Background Information

1.1 Introduction

In this paper we discuss the literature and resources required to create the sky tessellation algorithm and the techniques which 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. Tessellation algorithms are also be explained as well as algorithms for clustering data. Their efficiencies and complexities are also discussed.
Lastly we look at parallelism, Graphical Processing Unit (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. The optimizations of GPUs and CUDA are also discussed.

1.2 Radio Astronomy

Radio astronomy is the study of inter- and extragalactic objects by collecting and studying the electromagnetic signals they emit. In 1928, a 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[1].

1.2.1 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 1.1. 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).

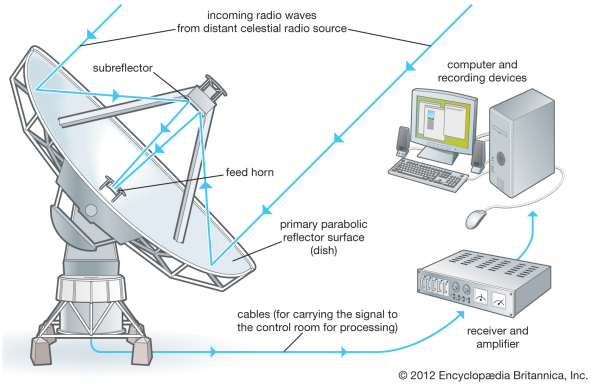


Figure 1.1: Parabolic reflector antenna design[2].

[1]Taken from https://public.nrao.edu/radioastronomy/what-is-radio-astronomy
[2]Taken from `http://kids.britannica.com/comptons/art-145514`

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 large 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 (discussed in Section 1.2.2) 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).

1.2.2 Image Capturing and Processing

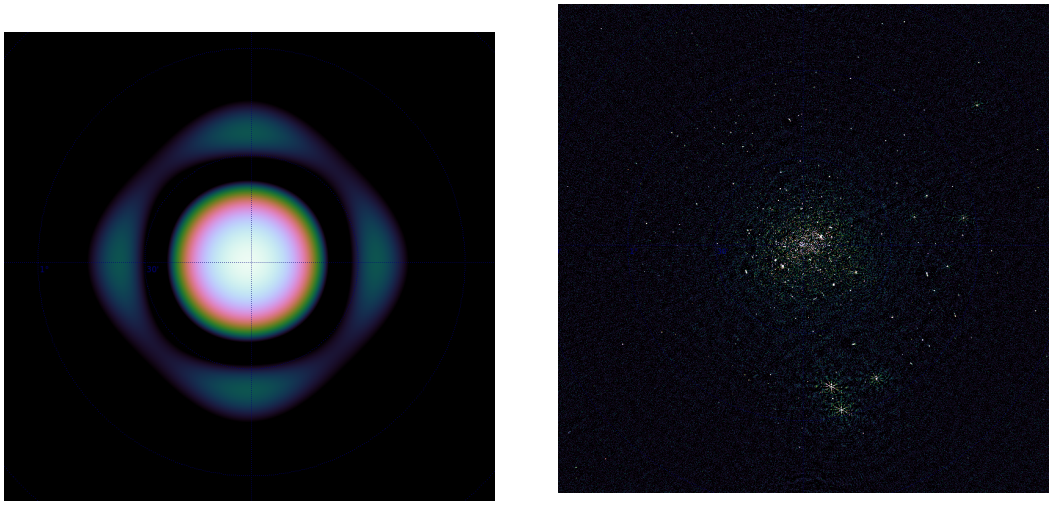
Aperture Synthesis

The electromagnetic radiation collected by the antenna is correlated into voltage differences. The data is 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 shown in Figure 1.2a. The circular sensitivity present in Figure 1.2a can be seen affecting the uncorrected image present in Figure 1.2b (Thompson *et al.*, 2008).



(a) Sensitivity pattern.

(b) Resulting uncorrected image produced through radio interferometry.

Figure 1.2: 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 Direction-Dependent (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 1.2.1). 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).

1.2.3 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 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 or 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 1.3.

Figure 1.3: Figure 1.2b corrected on a 23×23 grid.

1.3 Models and Algorithms

1.3.1 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 1.4.

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 these 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 arc (d_C) (Okabe *et al.*, 2009). This leaves the power diagram as the only Voronoi diagram which 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

$$b(p_i, p_j) = \frac{1}{2}(||\mathbf{x}_i||^2 - ||\mathbf{x}_j||^2 + w_i - w_j) \qquad \mathbf{x}_i = (x_i, y_i), \tag{1.1}$$

does not lie on the line segment $p_i\bar{p}_j$. 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).

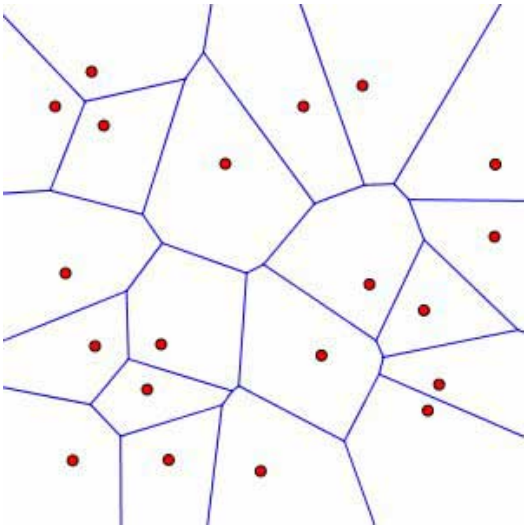


Figure 1.4: Voronoi diagram[3]

1.3.2 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):

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

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) and 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; 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) Starting 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 and add this segment 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 $(x(s), y(s))$,

$$\phi(x(s), y(s)) = (x(s) + r(s), y(s)), \tag{1.2}$$

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 then 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 tessential 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 will be 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 appended 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.

1.3.3 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, are 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 as 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.

1.3.4 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 1.5 where the facets are superimposed over the source image from which they are derived.

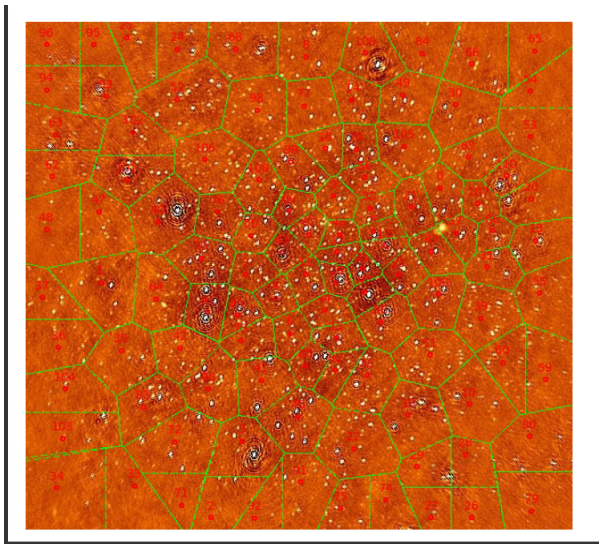


Figure 1.5: Example of Voronoi faceting to group extragalactic points for DD calibration

1.4 GPU Architecture and Concepts

1.4.1 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).

1.4.2 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 1.4.3) 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).

1.4.3 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 1.6 (NVIDIA, 2016a), (NVIDIA, 2014).

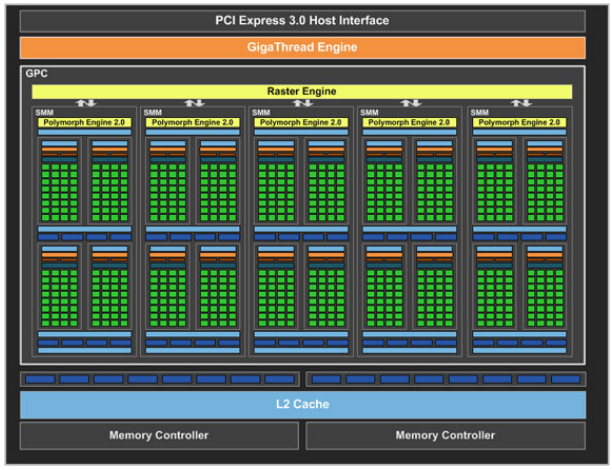


Figure 1.6: NVIDIA Maxwell Architecture.[4]

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

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 1.7. 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).

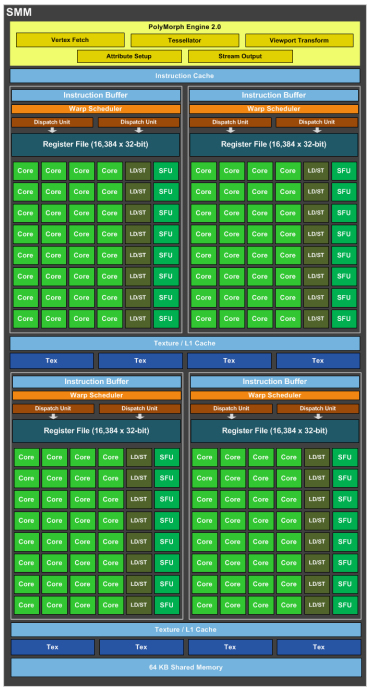


Figure 1.7: Maxwell Streaming Multiprocessor.[5]

1.4.4 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:

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

```
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).

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, hence why they are bundled into a larger, 3D grid structure. Similarly to threads, blocks have a unique 3D ID in the grid. Blocks exist such that each step of the processes execute simultaneously. This is done as, more often than not, blocks exchange data within their threads and if this precaution is 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 64Kb) when the kernel is executed.
Each processor is assigned its own memory to be used by each thread, these are called 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, then 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).

1.4.5 CUDA Optimization

As stated in Section 1.4.2, GPUs are designed for parallel computing to speed up the execution of a process when compared with running it on a CPU. NVIDIA has gone one step further to design ways to improve the efficiency of a GPU even further. These optimizations can be categorized into three groups, memory, execution configuration and instruction optimizations. The NVIDIA Visual Profiler (NVVP) can be key for assisting users in locating 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 1.4.4 show examples of this as well. Constant and texture memory, for example, improve latency by having the value broadcast 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 the overall usage of cores on the GPU and keeping as much of the hardware as occupied as possible to improve the overall execution time. One way of achieving this is through concurrent kernel execution, 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 that single precision is encouraged and that CUDA has multiple 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 can be found at 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 will drastically improve memory access times (NVIDIA, 2015).

1.4.6 Related Work

Jump Flooding

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 which have an identified closest seed point (or in this case, seed cell) and project this seed cell to surrounding cells without an identified seed cell in incrementally smaller steps, starting from a step size of $n/2$.

1.5 Summary

In this review 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 looked at how an array of radio antenna can be used for radio interferometry to detect correlations in radio frequency radiation from extragalactic bodies. We analyzed 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; we also discussed how the primary beam of the antenna affects the image generated. We then saw how DD-effects are generated from this and how they are corrected for and how the need for an algorithm which finds a good compromise between computationally feasibility and sufficient error reduction.
We then looked over Voronoi diagrams and discussed how power diagrams are their natural extension when weights are incorporated. We discussed Voronoi algorithms, namely the incremental, divide and conquer, and Fortune's algorithm. We saw that the incremental was easy to implement but the most time consuming computationally, the divide and conquer used a recursive method which is faster than the incremental, but harder to properly implement, and 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. We then discussed clustering methods for grouping points in a space. We looked at three algorithms again, namely the k-means, the agglomerative, and the bisecting k-means algorithms. We saw that the k-means was 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 was the most stable but took the longest to calculate, but we see that 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.
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. We then looked to the history of GPUs from their initial use for pixel generation in gaming to the parallel data processing titans they are today. We then looked, more specifically at the NVIDIA GTX 750 Ti, and how its GM107 architecture has improved to give high performance with low power consumption. We looked at how the warps are used in the streaming multiprocessor to allow more pro-

cesses to run concurrently which in turn improves the efficiency of the GPU. The GPU programming language, CUDA, was discussed and 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 discussed 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 were some of the improvements discussed.

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 2

WIP

Given a large set ($\sim$ 10000) of extragalactic sources obtained by aperture synthesis (Chapter 1.2.2), an optimal means to tessellate the space containing the sources must be found so that the correction of DD-effects is maximal.

A Voronoi tessellation (Chapter 1.3.1) will therefore be used to tessellate the space into polygonal subspaces (hereafter referred to as cells). However, the 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 on the order of 10^5 in magnitude.A smaller subset of the sources will therefore be used to generate the Voronoi tessellation.

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

2.1 Source Selection

The system is designed to take in three parameters, a list of all source points in the space, the dimensions of the space the points lie in and the intensity threshold. Each element of the list of sources has 3 parameters: x, y and z. The x and y parameters define the spatial coordinates of the source while the z parameter is seen as the intensity of the source, the list of n sources are passed into the function as an $n \times 3$ numpy array of doubles. The spatial dimension parameter takes in a tuple of two values, the width and height of the space the sources are in. The intensity threshold is a single double value which defines the minimum intensity a source needs to be a voronoi cell centre.

The sources are read in and are used to generate the voronoi centres as a points. Each point inherits the x, y and z values of the source and the rest are kept as default values. Once the list of points is created, it is sorted by the x then the y values of the sources. Once the centre points are generated, generation the voronoi diagram can begin.

For the sake of testing the system the coordinates are randomly generated on a 600×600 plane with the intensity 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 centres to be used by the Voronoi, centres are sources with an intensity greater than one standard deviation of the original mean, since the absolute value of the source intensity is used, this is all points with an intensity greater than 3000, this accounts for approximately 32% of the sources.

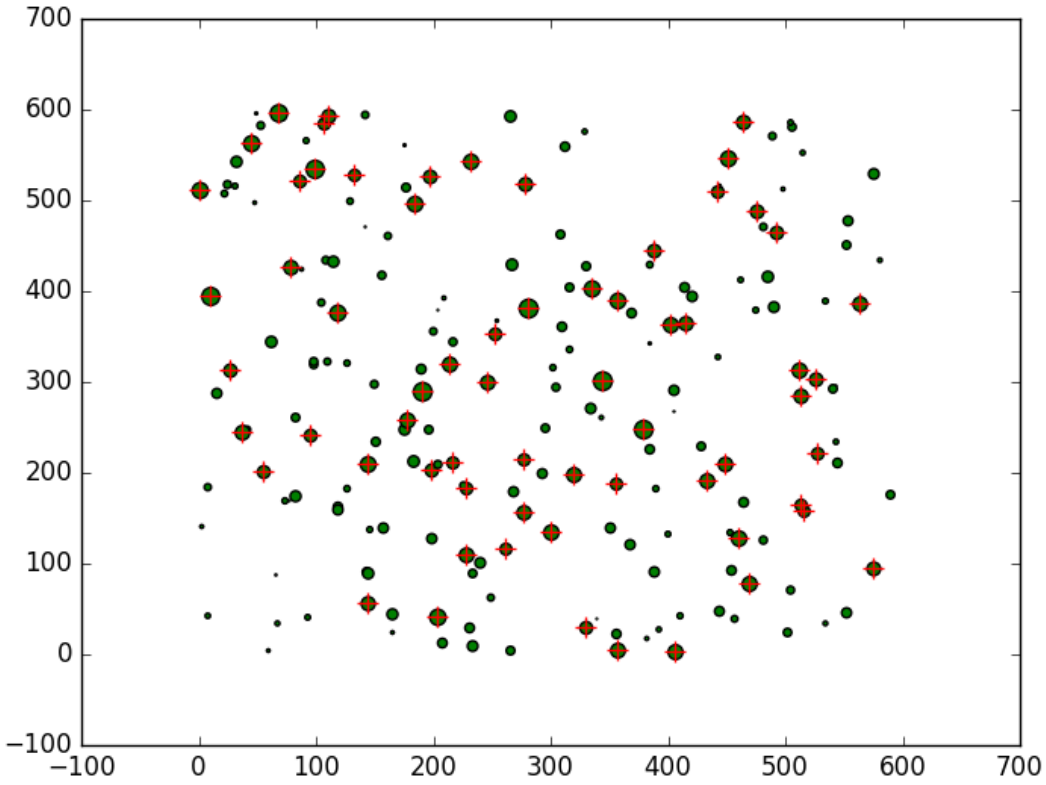


Figure 2.1: All sources with those selected as centres having a red cross.

Figure 2.1 shows sources in green with their intensities representing the radius of the on the plane, to simplify, only 200 points are generated. The crosses in red represent the centres that will be used to generate the Voronoi tessellation.

2.2 Voronoi

The current best model uses k-means clustering of of the points to the center points. For every point (the plane is seen as a finite number of pixels) in the plane, p, it finds the seed point, s_i, which is the shortest distance to it, i.e. such that $\sqrt{\big(p(x) - s_i(x)\big)^2 + \big(p(y) - s_i(y)\big)^2}$ is minimum. This is suboptimal as the time taken to create the diagram relies on the dimensions of the plane and the number of seed points in it. Instead we seek a method which is invariant of the plane size and relies solely on the seed points, for the sake of increasing the computation time, this method must also be parallelizable.

It was therefore decided that the divide and conquer method (Chapter 1.3.2) would be used. The algorithm works by ordering the points, first by their x then their y values. The points are then divided into two subsets, a left and right set. The Voronoi diagrams are then generated for the left and right subsets using the divide and conquer method. The convex hulls of the left and right Voronoi diagrams are then found. The lowest common support line between the hulls is then found 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 then determined, and cut so that part of the chain is now part of the Voronoi cells edges.

Code for the Divide and Conquer 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. pyVoronoi uses a simple GUI to generate a voronoi diagram 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 the user to specify their own spacial constrains 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 which may be called by the user or used in a larger process.

2.2.1 Structures of the Voronoi

The voronoi structure is mainly made up of two structures, lines and points. Points are mainly used to define the sources, but are also used to generate line. Points are made up of an x, y and z value, which determines the position and intensity of the point. Each point also has a circumcentre attribute to determine whether it is the point of intercept of three lines (this is most commonly used in the points at the end of a line), two point parameters to indicate the points on either side of the point if the point lies on the convex hull, these are initialised as None, a list of related structures containing another point and their corresponding bisecting line, this list is set as empty. The point also has a list of all the sources contained in the cell and an error value associated to it, these values are set to an empty list and zero respectively and will only come into use after the voronoi diagram has been generated.

A line is made up of two points which define its endpoints, if the line is a bisector it

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

contains two more points which define the source points used to generate it, if not these values are set to None. The line also contains a list of all the other lines the line is connected to as well. The line finally has an availability boolean parameter which defines whether or not the line is actively available to intercept with other lines or if it has been discarded from the overall voronoi structure.

2.2.2 Voronoi Function

The Voronoi function takes in the list of points and a range of points it will operate on (initially the entire set of points). Depending on the number of points in the subset range, one of four operations will occur. If the range is made up of more than three points, the range is divided into two equal sub ranges, one from the starting point of the range to the median and another from the point after the median to the end of the range. The Voronoi function is then called again for each sub range, denoted as VDL and VDR for left and right sub ranges respectively, with the full set of points. Once the voronoi structure for these two are calculated, the merge function is called with VDL and VDR and the function ends as it is not value returning.

If the range is three, the function generates three lines, one for the each pair of the three point. The intercepts of the lines is determined, if it does not exist, an invalid line exists, this line is found and removed. If the intercept does exist, it is found and the lines are clipped at the intercept. The lines are listed and returned with the range and the corresponding convex hull structure made up of all three points.
For a range of two points, the bisecting line is determined as the only line in the structure. The line is placed in a list where it is the only element and along with the point range and the convex hull, only made up of the two points, are returned.
The case of a single point is excluded as the first case of multiple points being divided in two equal or near equal sets and the case of three and two points in a set will make the case of a single point impossible.

2.2.3 Convex Hull

The convex hull of a set of n points, $P = p_1, p_2, ..., p_n$, 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 or 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).

Andrew's monotone chain convex hull algorithm (Andrew, 1979) was used to determine the convex hull of a set of points. The function is first passed a range of points on which to operate. The points must be ordered lexicographically (first by the x coordinate then by y if there is a tie), which they are as this is needed for generating the voronoi structure. A list of zeros, twice the size of the number of points in the range is generated, this list will hold the elements of the chain. The complete convex hull is calculated in two steps, the first finds the upper hull and the second, the lower hull.
The first point (the leftmost) and the second point in the range are added to the list. We then iterate over the rest of the range, adding the next point to our list, if the angle created by these three points is less than π radians, the points are left in the list, the next point in the range is added and the three latest points in the list are analysed for their angle again until we reach the. If the angle is greater than π radians, the two latest points in the list are removed and the process continues. This generates the upper hull of the convex hull.
To generate the lower hull, the same process is used, but the range is iterated over in reverse, starting at the rightmost point and ending at the leftmost.
One the list is populated with the points of the lower and upper hulls, they form the monotone chain. The chain is then iterated over and each point in the chain adds the preceding point to its fifth parameter and the next point to its fourth.

2.2.4 Voronoi Merge Function

The merge begins by finding the upper and lower tangents of the VDL and VDR sets. These tangent lines are defined as connections between the convex hulls of VDL and VDR with a point in each as the endpoints of the line. Starting from the rightmost point in VDL and the leftmost in VDR, it finds the upper tangent by finding the pair of points (one in VDL and one in VDR) who, together with their respective adjacent points, both generate an angle less than π radians, the order and the adjacent point being used is counter-clockwise for VDL and clockwise for VDR. To find the lower tangent, the order is reversed with VDL stepping clockwise and VDR counter-clockwise.
The upper tangent is used to find the starting point of the polygonal chain used to merge the voronoi. Bisector of the endpoints of the line, which both lie on convex hulls, are used as the first line segment in the chain which is appended to a list of lines called HP.

The uppermost of the two points in the upper tangent is determined and a new tangent is generated with the lower point and the uppermost point's neighbour, not necessarily in the convex hull, so long as it is not a convex hull neighbour of the lower point and it's related line to the upper point is both available and intercepts the bisector of the upper tangent. The bisecting line of the new point which intercepts the new bisector along with the new bisector itself is added to a list of lines to be clipped, clip_lines. The same set of operations then occur with the new point and the lower of the upper tangent defining the new tangent. This continues until the bisecting line of the lower tangent is determined. Once complete, we are left with a list, HP, which contains the bisecting lines between points in VDL and VDR and a list of bisecting lines from VDL and VDR together with lines from HP, clip_lines. The lines in clip_lines and HP are both 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.

An Example of the Voronoi Tessellation can be seen in Figure 2.2

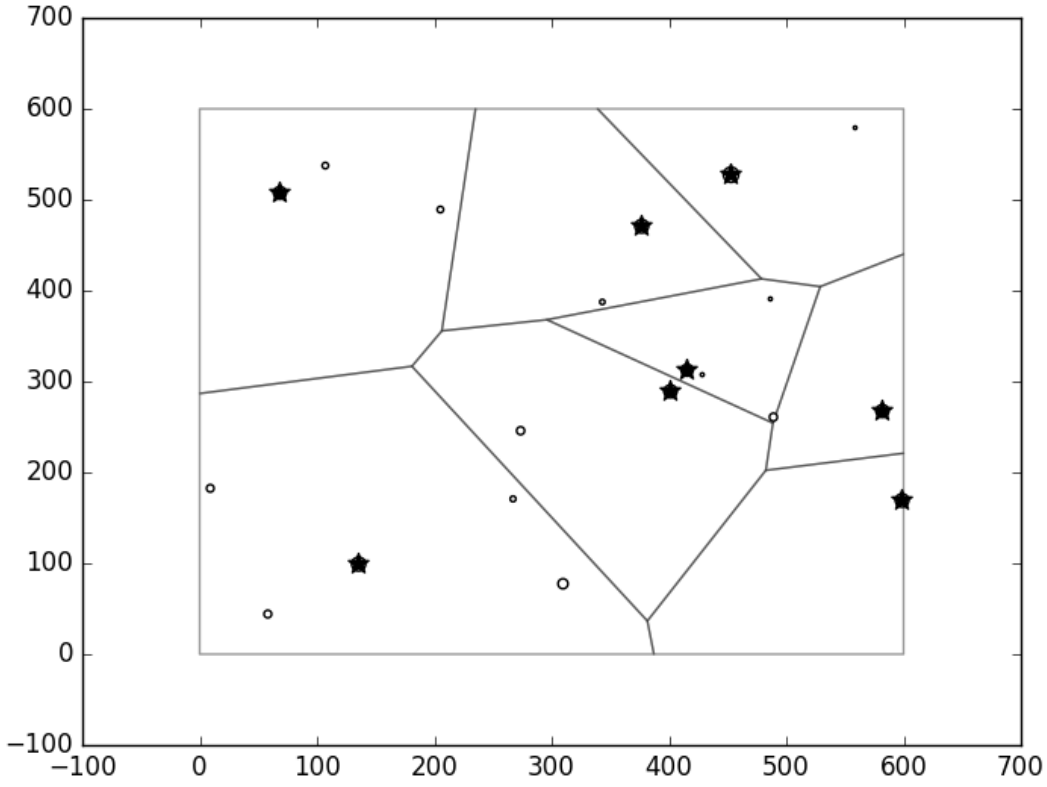


Figure 2.2: A Voronoi tessellation generated by the divide and conquer algorithm.

2.2.5 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 can be seen in equation 2.1 where z_1 and z_2 are the intensities of $\vec{x_1}$ and $\vec{x_2}$ respectively.

$$d'(\vec{x_1}, \vec{x_2}) = \frac{z_1}{z_1 + z_2}\vec{x_1} + \frac{z_2}{z_1 + z_2}\vec{x_2} \tag{2.1}$$

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

$$\begin{aligned} d'(\vec{x_1}, \vec{x_2}) &= \frac{z_1}{z_1 + z_2}\vec{x_1} + \frac{z_2}{z_1 + z_2}\vec{x_2} \\ d'(\vec{x_1}, \vec{x_2}) &= \frac{1}{2}\vec{x_1} + \frac{1}{2}\vec{x_2} \\ d'(\vec{x_1}, \vec{x_2}) &= \frac{\vec{x_1} + \vec{x_2}}{2} \end{aligned}$$

Some complications were found in this, namely on the convex hull. Weighted centres which are not on the hull but due to their larger weighting still have cells which dominate areas of the hull. This leads to them not being included in the merging process and their line segments not being clipped or deactivated. Another problem with this is that it generates undefined regions in the space, regions where domains overlap due conflicting weightings, an example of this can be seen in Figure 2.3.

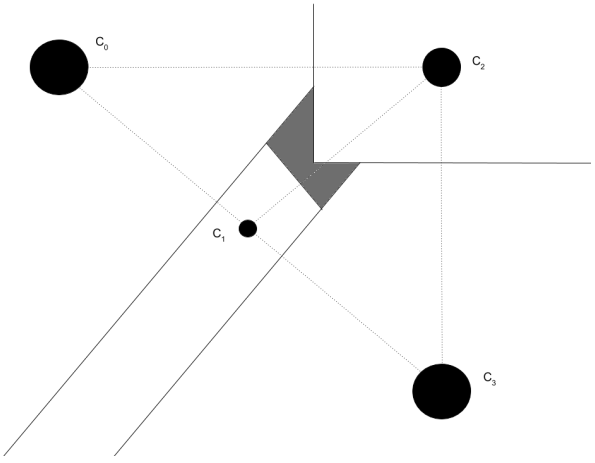


Figure 2.3: An unclassified area (grey) generated by a weighting conflict in c_1 and c_2.

It shows a conflicting area between c_1 and c_2, it should be expected that c_2, with its higher

intensity, should claim the area, but in doing so it will cross into what then 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 generated was abandoned and that the intensities would rather be used for calculating the error and calculating cell merges. An example of a failed weighted Voronoi tessellation can be seen in Figure 2.4.

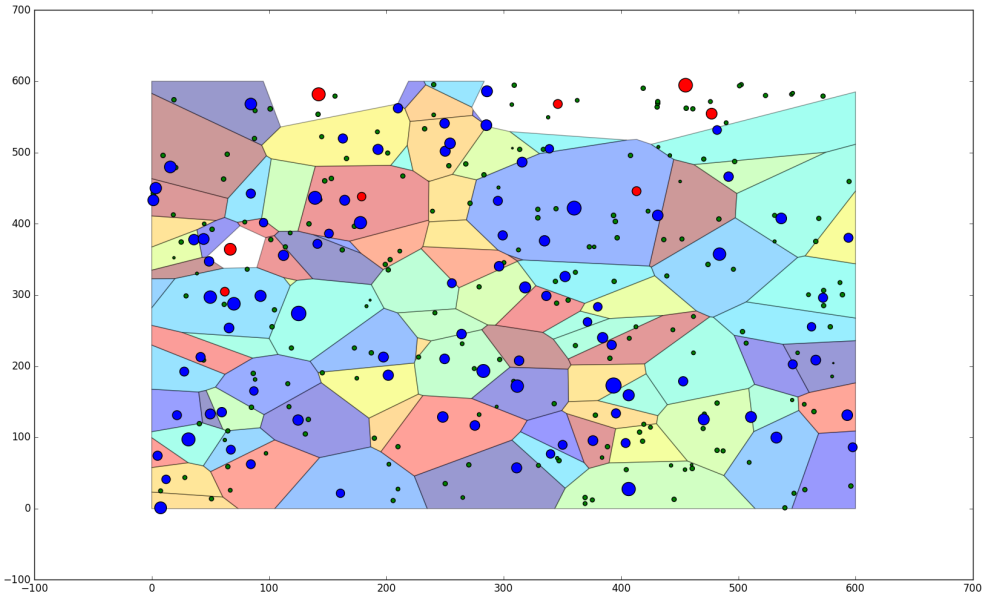


Figure 2.4: A failed visualisation of a weighted Voronoi tessellation.

2.3 Cell Error

Once the Voronoi tessellation is 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:

$$d = \sqrt{(x_i - x_c)^2 + (y_i - y_c)^2}, \tag{2.2}$$

where (x_i, y_i) is the location of a source in the plane and (x_c, y_c) is the location of a centre. The closest centre is such that d is minimum. This is done to add the influence of weaker sources in overall correction. This 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

$$\epsilon = \sum_{i=0}^{N} z_i |\vec{r_i} - \vec{r_c}|^2, \tag{2.3}$$

with $\vec{r_i} = (x_i, y_i)$ is the location and z_i is the intensity of some source in the cell and $\vec{r_c}$ as the location of a new centre.
This error function will have a local minima at the point where its derivative with regards to $\vec{r_c}$ is zero, or

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

or

$$2\sum_{i=0}^{N} z_i \vec{r_i} = 2\sum_{i=0}^{N} z_i \vec{r_c}$$

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

$$\vec{r_c} = \frac{\sum_{i=0}^{N} z_i \vec{r_i}}{\sum_{i=0}^{N} z_i} \tag{2.4}$$

From this, the new centre is determined, since it is required for the cell merge, the new centre retains the old centres intensity value i.e. the highest intensity from a source in the cell. Once the cell's new centre is obtained, its error is calculated using Equation 2.3. An example of centre correction can be seen in the difference between Figure 2.5 and Figure 2.6.

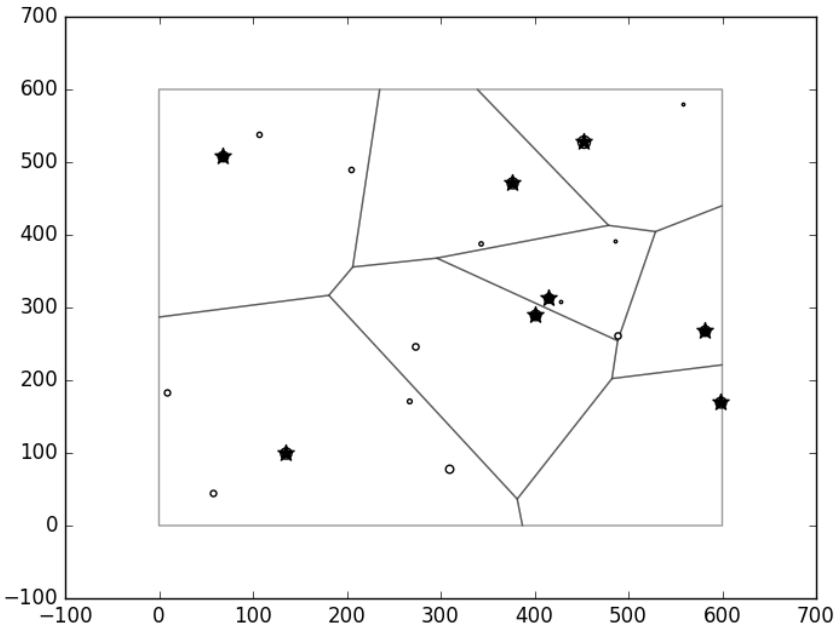


Figure 2.5: Tessellation with high intensity points as centres.

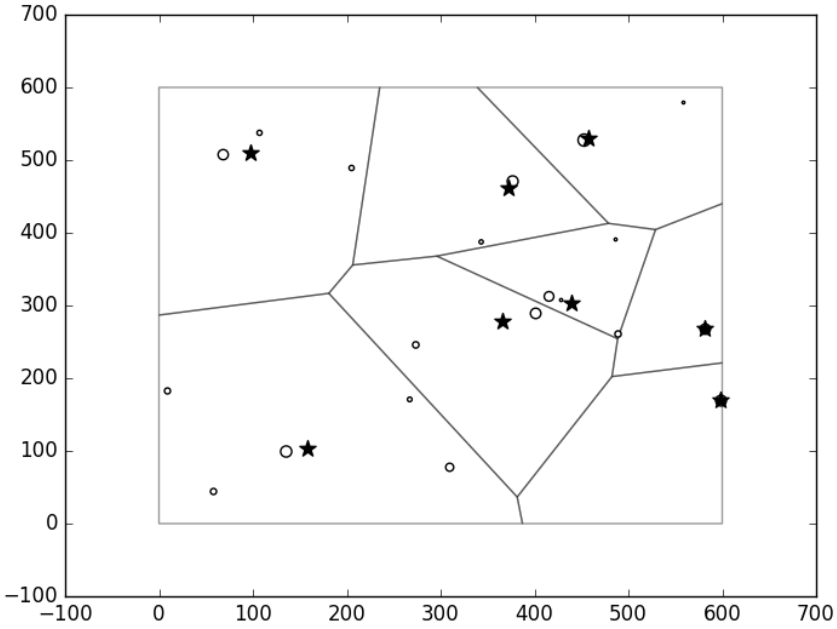


Figure 2.6: Tessellation with the weighted average of the points in the cells as their centres.

2.4 Cell Merge

The merge process is iterative and runs dependant on the total sum of the error of the tessellation. Initially, this error is relatively low as a multiple cells are generated with one or very few sources contained in it. In order 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, the size of the plane and the number of sources in the plane. 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.

2.4.1 Obtaining the best Merge

The best merge is obtained by iterating over the list of points and, for each point, testing its active neighbouring points.

The merge test works by determining a new centre, determined by two existing centres, $\vec{p_1}$ and $\vec{p_2}$, with intensities z_1 and z_2 as

$$P = t\vec{p_1} + (1-t)\vec{p_2} \text{ with } t = \frac{z_1}{z_1 + z_2} \tag{2.5}$$

the new weight for the merged centre is now defined as

$$Z = z_1 + z_2 \tag{2.6}$$

Since p_1 and p_2 are centred sums of the positions of the sources in their cells, by expanding them to their original forms

$$\vec{p_1} = \frac{\sum_{i=0}^{N} z_{1i}\vec{x_{1i}}}{\sum_{i=0}^{N} z_{1i}} \text{ and } \vec{p_2} = \frac{\sum_{i=0}^{M} z_{2i}\vec{x_{2i}}}{\sum_{i=0}^{M} z_{2i}}$$

and by noting that

$$z_j = \sum_{i=0}^{N} z_{ji}$$

Substituting these into equation 2.5, we obtain

$$\begin{aligned}
P &= t\vec{p_1} + (1-t)\vec{p_2} \\
&= \frac{z_1}{z_1+z_2}\frac{\sum_{i=0}^{N} z_{1i}\vec{x_{1i}}}{z_1} + (\frac{z_1+z_2}{z_1+z_2} - \frac{z_1}{z_1+z_2})\frac{\sum_{i=0}^{M} z_{2i}\vec{x_{2i}}}{z_2} \\
&= \frac{\sum_{i=0}^{N} z_{1i}\vec{x_{1i}}}{z_1+z_2} + \frac{\sum_{i=0}^{M} z_{2i}\vec{x_{2i}}}{z_1+z_2} \\
&= \frac{\sum_{i=0}^{N} z_{1i}\vec{x_{1i}} + \sum_{i=0}^{M} z_{2i}\vec{x_{2i}}}{z_1+z_2} \\
&= \frac{\sum_{i=0}^{N} z_{1i}\vec{x_{1i}} + \sum_{i=0}^{M} z_{2i}\vec{x_{2i}}}{\sum_{i=0}^{N} z_{1i} + \sum_{i=0}^{M} z_{2i}}
\end{aligned}$$

This shows that equation 2.5 is equivalent to finding the centre of all the sources in both p_1 and p_2.

Once the new centre is found, the error must be determined, this is again found 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 which 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. If it is lower, it is then stored along with the new coordinates and the centres used to generate it.

Once the best merge is found, it is determined if the new sum of errors will exceed the threshold, if it does, the merge process is halted and the structure returned as the best possible merge. 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 is set too high for the system and the structure is returned as only the set of points with no active lines left and so no merged Voronoi can be generated. If the system finds a valid merge which is still under the threshold, it adds the difference in the merge error to the total error and executes the merge.

2.4.2 Executing the Merge

The merge execution algorithm takes in the new coordinates and intensity, P, the new error, as well as the centre, p_1, and its related centre with which it is to be merged, p_2.

It starts by setting the coordinates and intensity of p_1 to those of P. It then sets active status the line which relates p_1 to p_2 to false and appends the list of sources in p_2 to that of p_1. The error of p_1 is then set to that of the new error and p_2 and it's list of consumed centres is added to the list of consumed centres of p_1. Finally, the list of centres is iterated over and any centre which references p_2 is changed to reference p_1 and all lines which relate other centres to p_2 are changed to relate to p_1 instead. Once this is complete, the process of finding the best merge begins again until the threshold is reached.

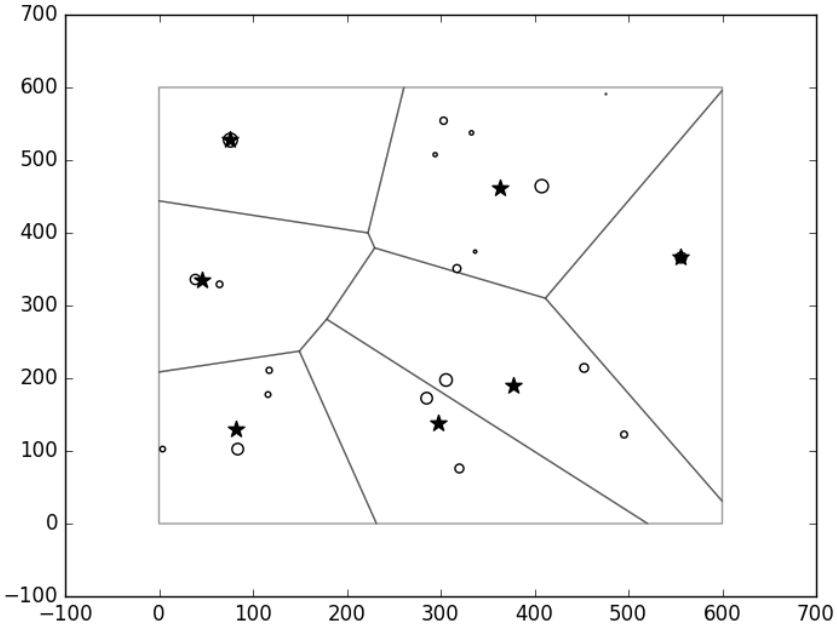


Figure 2.7: Re-centred Voronoi before merge.

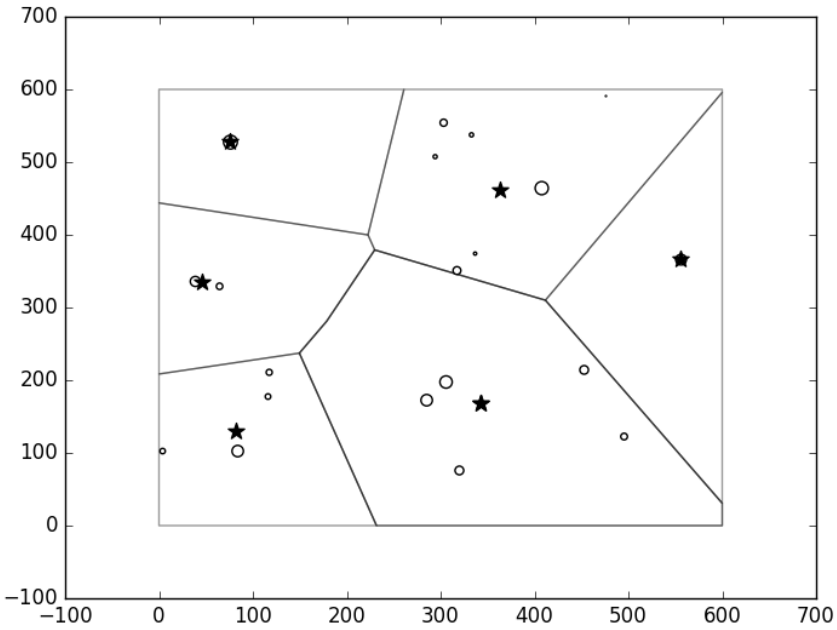


Figure 2.8: Re-centred Voronoi after single merge.

The transition from Figure 2.7 to Figure 2.8 shows the effects of a single merge. As shown, the weighted distance between the merged centres is less than those of any other neighbouring centre pairs.

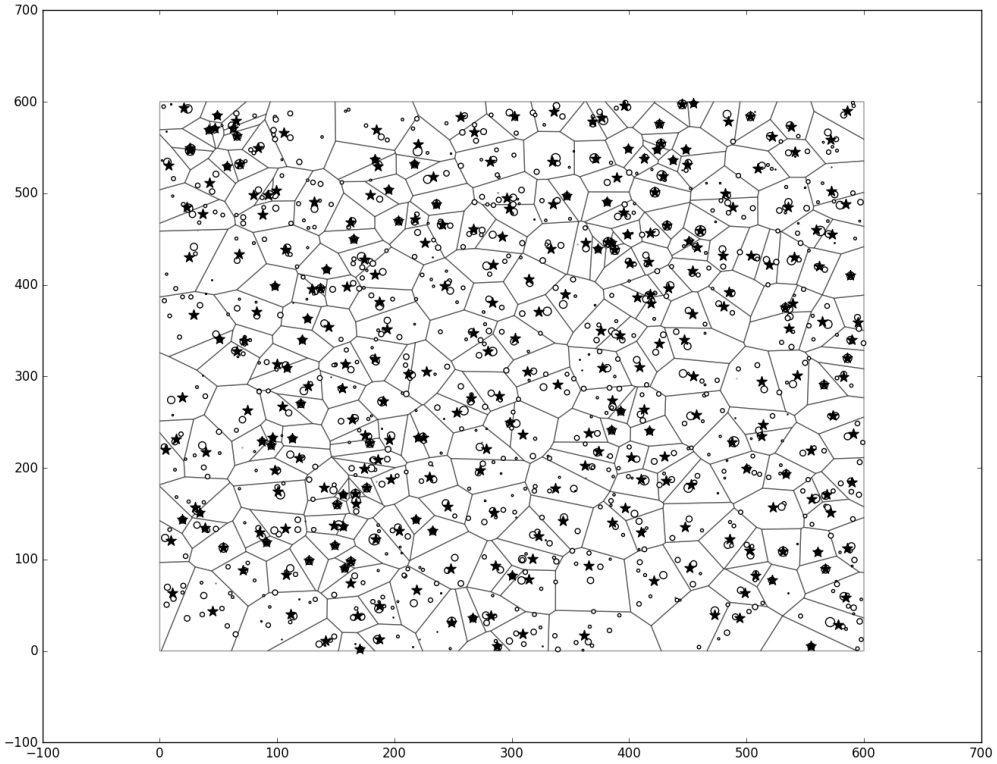


Figure 2.9: Re-centred Voronoi with 1000 sources and 301 centres before merge.

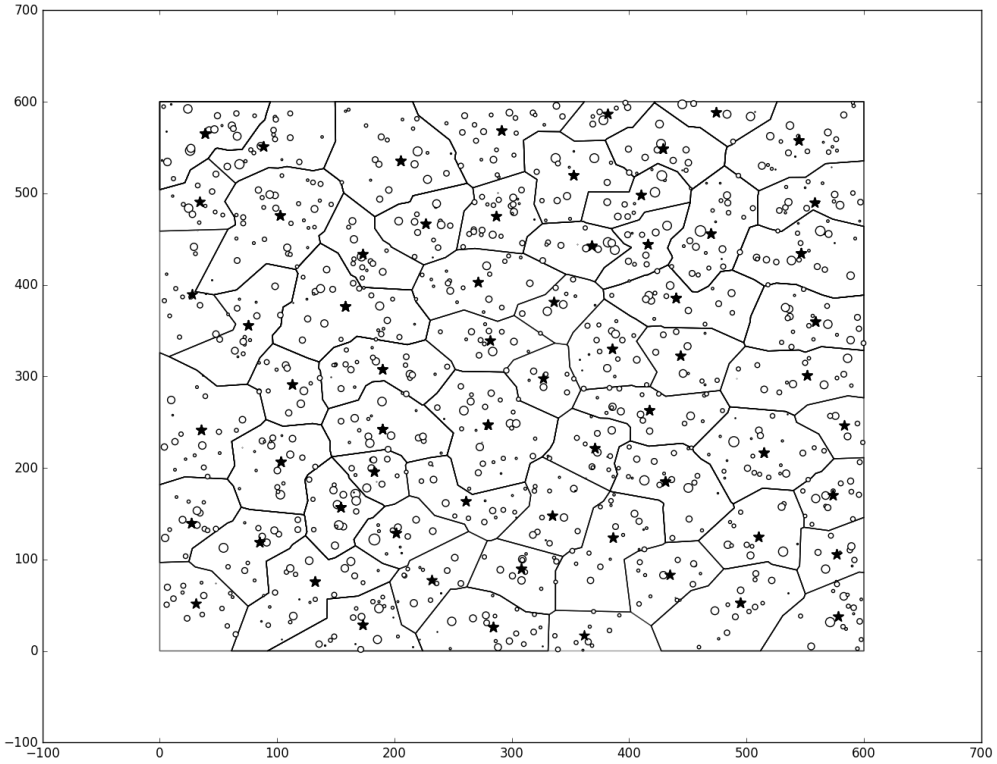


Figure 2.10: Re-centred Voronoi with 1000 sources and 64 centres after merge.

Figure 2.9 shows a Voronoi Tessellation and Figure 2.10 shows its completed merged structure, it shows larger, more centralised cell structures.

2.5 GPU Implimentation

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 inculde 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 callsNVIDIA (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 diagram 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 paper, we will only look at parallel computation of finding the best merge.

References

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.

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.

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.