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: Literature Review

Antonio Bradley Peters

supervised by
Prof. Karen Bradshaw
Prof. Denis Pollney

project originated by
Dr. Cyril Tasse & Prof. Oleg Smirnov

August 18, 2016

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 will also be discussed.
Lastly we look at parallelism, 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.

2 Radio Astronomy

2.1 Introduction to Radio Astronomy

Radio astronomy is the study of inter- and extragalactic object 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 as to what lies in it's center. Radio frequency radiation is also emitted from cold sources, allowing us to view extragalactic bodies with greater quality and to better precision[1].

2.2 Radio Telescopes

2.2.1 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. 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 +70m) 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 a more accurate signal detection. These arrays do so through radio interferometry (**?**).

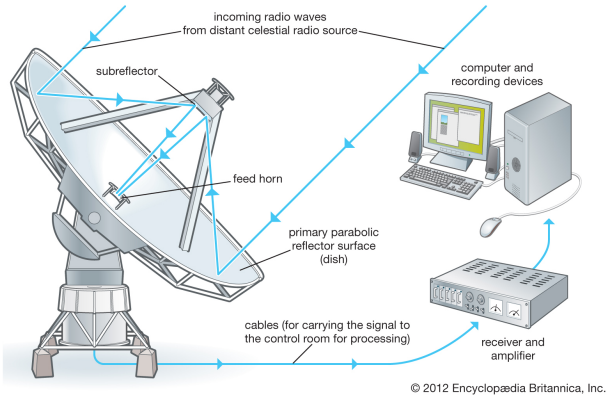


Figure 1: Parabolic reflector antenna design[2].

2.2.2 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} meters (**?**). The interferometers finds the source of these waves by detecting correlations in the parallel ray signal transmitted by the radiating source (source) and collected by multiple antennas in order to determine the delay as well as the amplitude and frequency of the source to calculate the position, size and intensity of the source (**?**).

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

2.2.3 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. Altazimuth mounts rotate 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 2.3.3) than an altazimuth mount. Altazimuth mounts are still more common as they are relatively cheaper and easier to build than an equatorial mount (**?**).

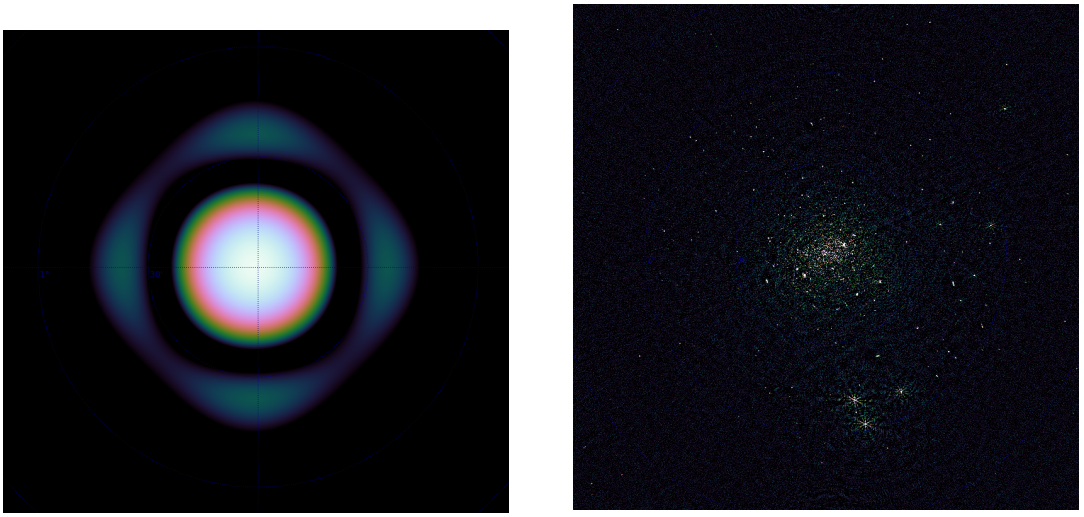
2.3 Image Capturing and Processing

2.3.1 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 is Fourier transformed from the frequency domain to that of the spatial domain, to give a two dimensional image (**?**).

2.3.2 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 can be seen in Figure 2a. The circular sensitivity present in Figure 2a can be seen affecting the uncorrected image present in Figure 2b (**?**).



(a) Sensitivity pattern

(b) Resulting uncorrected image produced through radio interferometry.

Figure 2: Primary beam and its effects.

2.3.3 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 (**?**). 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 2.2.3). 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 $\boldsymbol{x}_i$ is dependent on the intensity at the point, $\boldsymbol{I}_i$, and the distortion at the point relative to ξ, $D(\boldsymbol{x}_i, \xi)$. Therefore, to minimize this error, every point can be made a correction seed and the image can be broken up by these points and reassembled to form an image with little to no error. However, this is computationally ineffective as there are hundreds of sources per image and also due to the fact that the image is sparely populated. We therefore seek a method which optimally compromises computational feasibility and error reduction (**?**).

2.4 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 lies in the fact that either the sub-image is void and has no definite points; if ξ is set at the center, it could be far from every other point and has 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 (**?**). An example can be seen in Figure 3.

Figure 3: Figure 2b corrected on a 23×23 grid.

3 Models and Algorithms

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 (**?**). An example of a Voronoi Diagram can be seen in Figure 4.

3.1.1 Weighted Voronoi Tessellations

The basic form of the Voronoi tessellation has the seed points as being indistinguishable from one another, other than their different positions 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 **?** include multiplicative, additive, compound and power Voronoi diagrams. These diagrams have distance operators described as d_M, d_A, d_C, and d_P respectively. $\mathrm{d}_M(s, p_i) = \frac{1}{w_i} d$
$d_A(s, p_i) = d - w_i$
$d_C(s, p_i) = \frac{1}{w_{i1}} d - w_{i2}$
$d_P(s, p_i) = d^2 - w_i$
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) (**?**). 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

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

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 great 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 (**?**).

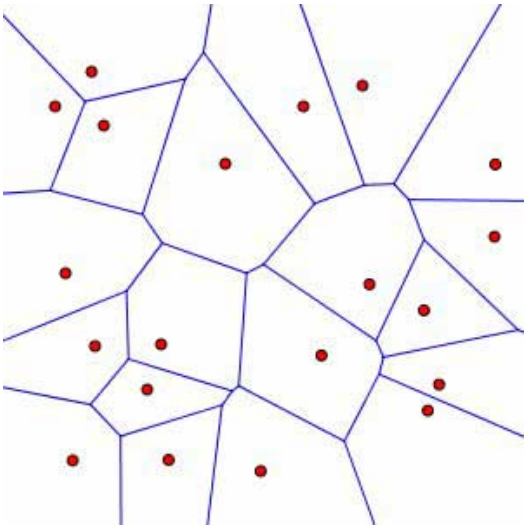


Figure 4: Voronoi diagram[3]

3.2 Voronoi Tessellation Generation Algorithms

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

3.2.1 Incremental Algorithm

The most simplistic of the generation algorithms, the Incremental is an iterative algorithm as described below (**?**) (**?**):

1. Starting from $i = 0$ and 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 $p_f = p_{nn}$ again.
9. Set $i = i + 1$ and repeat from step 2 until $i = n$

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

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

3.2.2 Divide and Conquer Algorithm

The Divide and Conquer algorithm was first proposed by **?** and also described in **?**. It is a recursive algorithm that improves on the Incremental algorithm by having a construction 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.

3.2.3 Fortune's Algorithm (Sweep-Line Method)

? 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 $(x(s), y(s))$,

$$\phi(x(s), y(s)) = (x(s) + r(s), y(s)), \tag{2}$$

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 **?** goes 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. The half-hyperbola intercept(s) with any other hyperbolas are found, if they exist, and are 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.

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.

3.3.1 K-Means Algorithm

K-means clustering is an iterative process where an initial guess at the center of a cluster, c, is made and 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 value of all points, $\frac{1}{m}\sum_{j=1}^{m} o_j^i$, in C_i. **?** 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 help drastically reduce the runtime of the algorithm. The clusters produced are also dependent on the initial placement of the mean points (**?**). Other methods of improving the runtime include probabilistic choices of starting mean points (**?**) and constraining the distance and using the triangle inequality (**?**).

3.3.2 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 **?**, goes 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.

3.3.3 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. **?** 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-mean. 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.

3.4 Related Work

3.4.1 Standard Voronoi Faceting

In **?**, **?** and **?**, 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 5 where the facets are superimposed over the source image from which they are derived.

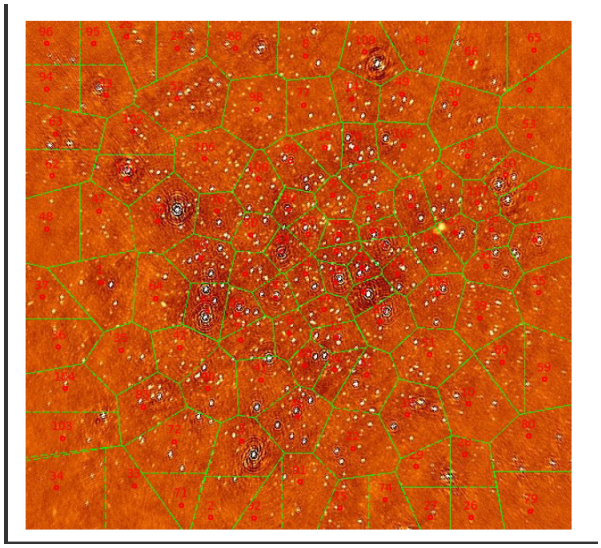


Figure 5: Example of Voronoi faceting to group extragalactic points for DD-calibration

4 GPU Architecture and Concepts

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 (**?**). 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 (**?**).

The increased need for parallelism came about in 2005, when CPU frequency peaked at 4 GHz due to heat dissipation issues. However, Moore's Law (**?**) 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 (**?**).

4.2 GPU Execution

Graphical Processing Units (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 common place as a way to accelerate data processing through data parallelism and task parallelism through the simultaneous execution of similar tasks (**?**).

The power of a GPU comes from its architecture which is optimized for a special case of SIMD (single instruction multiple data) processing known as SIMT (single instruction multiple threads). 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 4.3.2) 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 a CPU (**?**).

4.3 The NVIDIA GeForce GTX 750 Ti

4.3.1 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 6 (**?**), (**?**).

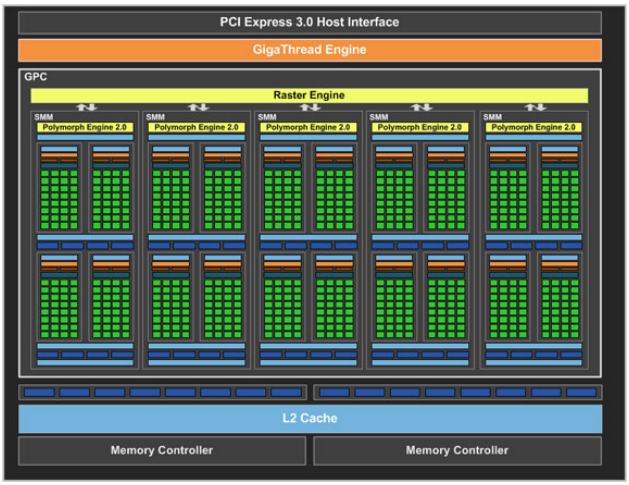


Figure 6: NVIDIA Maxwell Architecture[4]

4.3.2 Streaming Multiprocessors

The GPC is further broken down into five streaming multiprocessors (SMs) which are further 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 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 (**?**).

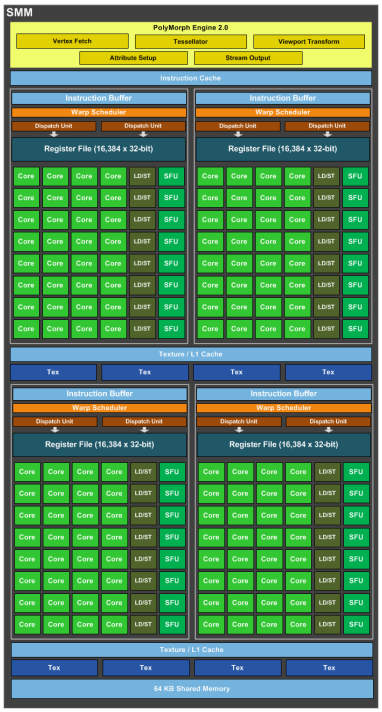


Figure 7: Maxwell Streaming Multiprocessor[5]

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

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:

```
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) (**?**).

4.4.1 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 be easily created, run its small task and be destroyed to make place for the next thread. Threads are arranged into three dimensional blocks with each thread having a unique 3 dimensional 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 (**?**).

4.4.2 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, three dimensional grid structure. Similarly to threads, blocks have a unique three dimensional 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) (**?**).

4.4.3 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 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 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

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

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. local memory lies in the L1 and L2 cache 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 (**?**).

4.5 CUDA Optimization

As stated in Section 4.2, GPUs are designed for parallel computing to speed up the execution of a process when compared to it running 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.

4.5.1 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 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 transfered 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, which 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 4.4.3 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 (**?**).

4.5.2 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 concurrency to reduce the idle time of each warp. Another means of this 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 (**?**).

4.5.3 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 are 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 **?**. 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 (**?**).

4.6 Related Work

4.6.1 Jump Flooding

? 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 $n/2$.

5 Summary

In retrospect 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 in order 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 were 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 processes 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.

6 Project Time-line

Objective	Time to complete	Expected Completion Date	Details
Coursework	2 weeks	late June	Completing existing coursework for the semester.
Exam	1 week	mid June	Studying for and writing GPU exam.
Basic Voronoi	1 week	early July	Writing a basic Voronoi algorithm for testing.
Power Diagram	2 weeks	mid July	Extending the Voronoi algorithm to account for weighted points and solving edge case problems.
Coursework	4 weeks	mid August	Time will be dedicated to completing coursework.
Non-Euclidean	2 weeks	late August	A non-Euclidean distance measurement will be implemented to account for spatial warping.
GPU implementation	3 weeks	mid-late September	Completed Voronoi algorithm will be coded for GPU.
GPU optimization	to end	end October	The GPU code will be improved on until final hand in date.
Thesis	entire duration	end October	At every stage, writing will be done on the project, with an expected minimum of one page per week.