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 the SKA: Literature Review

Antonio Bradley Peters

supervised by
Prof. Karen Bradshaw
Prof. Denis Pollney

project originated by
Prof. Oleg Smirnov

May 11, 2016

1 Introduction

1.1 The SKA

The SKA project was started in order to create the world's largest array of radio telescopes. This will be achieved by having 197 radio telescopes situated in South Africa and Australia working together and covering an area close to one square kilometre. The array is set to have a resolution of over 50 times that of the Hubble Space Telescope while still covering massive areas of the sky (SKA).

1.2 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 centre of the direction in which it is facing, with fringes of sensitivity radiating out as can be seen in Figure 1.4. The circular sensitivity present in Figure 1.4 is also due to the fact that the telescope rotates in order to keep the centre of the beam focused on the same area of the sky. The errors produced by the lack of sensitivity in certain areas can be compensated for, but that lies outside the scope of this paper (Smirnov).

1.3 Image Capturing

1.4 Errors in the Image

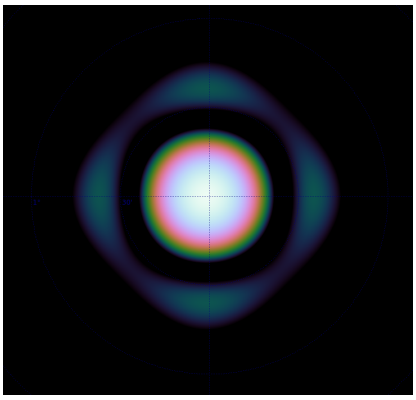

Figure 1: Primary Beam Focus Pattern(Smirnov)

2 Models and Algorithms

2.1 Voronoi Tessellations

A Voronoi Diagram is a partitioning of a space S by a set of points. Given n points (seed points) the the space, $P = \{p_0, p_2, ..., p_{n-1}\}, P \subset S$, is partitioned into n regions, known as Voronoi Regions or Voronoi Cell, 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 and Chiu [2009]). An example of a Voronoi Diagram can be seen in Figure 2.1.2.

2.1.1 Weighted Voronoi Tessellations

2.1.2 Voronoi Tessellations in Non-Euclidean Spaces

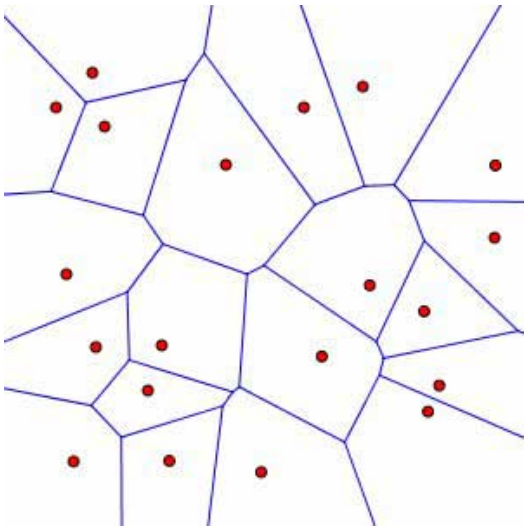

Figure 2: Voronoi Diagram(vor)

2.2 Voronoi Tessellation Generation Algorithms

Although Voronoi Tessellations extend to multiple dimensions, for the sake of these algorithms we will only discuss those in a two dimensional plane.

2.2.1 Incremental Algorithm

The most simplistic of the generation algorithms, the Incremental is an iterative algorithm as described by Green and Sibson [1978] and Okabe and Chiu [2009].

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 it's most naive form, this algorithm achieves and efficiency of $O(n^2)$.

2.2.2 Divide and Conquer Algorithm

The Divide and Conquer algorithm was first proposed by Shamos and Hoey [1975] and also described in Okabe and Chiu [2009]. It is a recursive algorithm which improves on the Incremental algorithm by having a construction time of $O(n \log n)$.

1. If the contains only one point, return it with the entire plane as it's voronoi region.
2. Divide the space, S containing the set of seed points, P with n points, 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 intercepting edge of the 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.

2.2.3 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 $(x(s), y(s))$,

$$\phi(x(s), y(s)) = (x(s) + r(s), y(s)) \tag{1}$$

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 essential for the algorithm. It is important to note that the perpendicular bisectors of seed points in S, through the transform, become hyperbola 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 parts 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 hyperbolic half-edges will be added as they are found. The algorithm, as described by Okabe and Chiu [2009] goes as follows:

1. Add P to Q

2. Choose and delete the leftmost seed point from Q

3. While Q is not empty repeat steps 3a, 3b and 3c.

   (a) Choose and delete the leftmost element, w of Q

   (b) If w is a seed point:

      i. Set $w = p_i$

   (c) If w is a half-edge:

      i.

2.3 Clustering Algorithms

2.3.1 K-Means Algorithm

3 GPU Architecture and Concepts

3.1 Parallelism

One of the main means of optimizing processing 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 (Subhlok and Gross [1993]).

The increased need for parallelism came in about 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 (Rajan [2013]).

3.2 GPU Optimization

GPU's were originally designed for rendering pixels and vectors in games. They were especially designed for this since CPU's 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 task (NVIDIA [a]).

The power of GPU's come 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 thread 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 in faster time than that of a CPU (Vuduc and Choi [2013]).

3.3 The NVIDIA GeForce GTX 750 Ti

3.3.1 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 3 (NVIDIA [b], NVIDIA [c], NVIDIA [d]).

3.3.2 Streaming Multiprocessors

The GPC is further broken down into five Streaming Multiprocessors(SM) which are further divided into four processing blocks. The processing blocks each contain an instruction buffer, a scheduler and 32 CUDA cores as seen in Figure 4 (Nathan Kirsch). 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 variable.(NVIDIA [e]

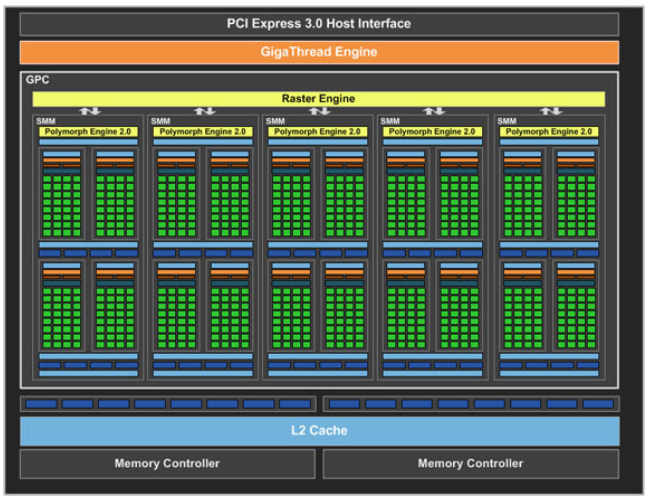

Figure 3: NVIDIA Maxwell Architecture(George Cella)

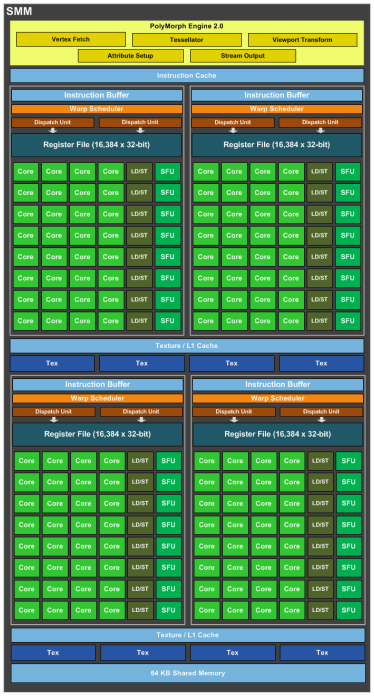

Figure 4: Maxwell Streaming Multiprocessor(Nathan Kirsch)

3.4 GPU Memory

3.4.1 Registry Memory

3.4.2 Cache

3.4.3 Global Memory

3.4.4 Constant Memory

4 Software

4.1 CUDA

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

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

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

4.1.1 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 it's small task and be destroyed to make place for th 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 (NVIDIA [e]).

4.1.2 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) (NVIDIA [e]).

4.2 Python

5 Related Works

5.1 Naive Method

5.2 Standard Voronoi Model

6 Summary

References

SKA. The SKA project. http://www.ska.ac.za/about/project.php. Accessed on: 21/02/2016.

Oleg Smirnov. Personal Communication. SKA Chair at Rhodes Centre for Radio Astronomy Techniques & Technologies.

Sugihara Okabe, Boots and Chiu. *Spatial tessellations: concepts and applications of Voronoi diagrams*, volume 501. John Wiley & Sons, 2009.

Furthest point voronoi diagrams: Voronoi diagrams. http://www.ams.org/featurecolumn/images/august2006/diagramintro.1.jpg. Accessed on: 26/02/2016.

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

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

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

O'hallaron Subhlok, Stichnoth and Gross. Exploiting task and data parallelism on a multicomputer. In *ACM SIGPLAN Notices*, volume 28, pages 13–22. ACM, 1993.

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

NVIDIA. CUDA. http://www.nvidia.com/object/cuda_home_new.html, a. Accessed on: 22/02/2016.

Richard Vuduc and Jee Choi. A brief history and introduction to gpgpu. In *Modern Accelerator Technologies for Geographic Information Science*, pages 9–23. Springer, 2013.

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

NVIDIA. GeForce GTX 750 Ti Specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-750-ti/specifications, c. Accessed on: 26/04/2016.

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

Nathan Kirsch. NVIDIA GeForce GTX 750 Ti 2GB Video Card Review. http://www.legitreviews.com/nvidia-geforce-gtx-750-ti-2gb-video-card-review_135752/2. Accessed on: 27/04/2016.

NVIDIA. CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz47aOWTI3j, e. Accessed on: 03/05/2016.

George Cella. MSI GTX 750 Ti Gaming Video Card Review. http://www.hitechlegion.com/reviews/graphics/38752-msi-gtx-750-ti-gaming-video-card-review?start=2. Accessed on: 27/04/2016.