Incremental Delaunay Triangulation on GPUs

Aditya Acharya
Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore, India
adiarya1a1@gmail.com

***Abstract*—Delaunay Triangulation of points on a plane is an essential and indispensible technique in numerous fields of computing from computational geometry and finite elements method to wireless networks. In this project I implemented a parallelized extrapolation based incremental algorithm for the exact delaunay triangulation of large point sets (in 2D) on GPUs using CUDA. What makes this algorithm unique is the fact that it can be applied to an online setting in which the point are discovered radially. The algorithm achieves a reasonable degree of parallelism, by expanding the triangulations from certain seed values in the domain on the GPU and by allowing rare incomplete triangulations to be completed on CPU. Subtle tricks like data replication and bucketing were used although the algorithm has a lot of scope for future improvements.**

I. Introduction

With the advent of massively parallel architectures like GPUs and programing paradigms such as CUDA there has been a tremendous development in parallelizing geometric and computatonal algorithms. However most of them involve irregular structures which makes it a challenge to implement them on GPUs. Delaunay triangulation is one such example which has many applications in fields of computing, especiallly in computer graphics , compuational geometry , finite element methods and even in wireless networks. The incremental algorithms for such a triangulations are popular due to their capability of handling dynamical insertion of points.

The primary aim of the project was to implement an incremental class algorithm for exact delaunay triangulation of a 2D pointset on GPU using CUDA.The particular algorithm followed in this project allows for outward growth of triangulations without the need of modifying existing triangulations. It is especially useful when sites or points are discovered radially from source nodes for example by a radar. Implementation was carried out on Nvidia Tesla cluster using CUDA programming language for diffrent distribution of points over the domain, resulting in reasonable speedups over the serial implemenation and with comparison to popular softwares for serial Delaunay Triangulation as well

A. Basics & Definitions

In mathematics and computational geometry, a Delaunay triangulation for a set P of points in a plane is a triangulation $\mathcal{DT}(P)$ such that no point in P is inside the circumcircle of any triangle in $\mathcal{DT}(P)$. Delaunay triangulations maximize the minimum angle of all the angles of the triangles in the triangulation.

So formally P is a set of points in $\mathbb{R}^2$, sometimes called sites. $\mathcal{DT}(P)$ is a partition of the convex hull of P into triangles such that the circumcircle of every such triangle is empty. Let N denote the number of sites and k the number of triangles in $\mathcal{DT}(P)$.

II. Related Work

Even before the advent of GPUs there were few attempts at triangulating a point set in parallel mostly using divide and conquer paradigm. DeWall [1]was one such algorithm proposed which was subsequently parallelized as well. there were some other parallel schemes for Delaunay Triangulations on distributed memory systems as well for e.g. [2].

Incremental class of algorithms are one of the most popular schemes for Delaunay triangulations considering their widespread use in adaptive refinement of meshes. Morever they serve as an excellent solution for triangulation in an online setting where points and sites are added dynamically. The straightforward incremental way of efficiently computing the Delaunay triangulation is to repeatedly add one vertex at a time, retriangulating the affected parts of the graph by edge and triangle flips. Another approach known as the 'Bowyer-Watson' algorithm [3] [4], removes the triangle containing the newly inserted point and retriangulates the star shaped polygon left behind.

As far as GPUs are concerned there have been several papers in the recent years. For example [5] finds a discrete version of the voronoi diagram (dual of delaunay triangulation) on GPU to approximate the triangulation followed by correction on CPU. Another recent approach has been found in [6] which restores an arbitrary triangulation to $\mathcal{DT}$, but requires an appropriate initial triangulation.

None of the prior attempts have followed the method of incremental triangulation on a GPU. The incremental algorithm chosen for this particular project was described first in [7] (long before GPUs were popular), in which the triangulations grow outwards from certain seed values .As the newer points lie outside the current triangles none of the prior triangulations are affected. While it is very unclear how to paralellize the earlier mentioned incremental algorithms on GPU, this simple yet effective incremental algorithm is worth exploring on massively parallel GPU systems.

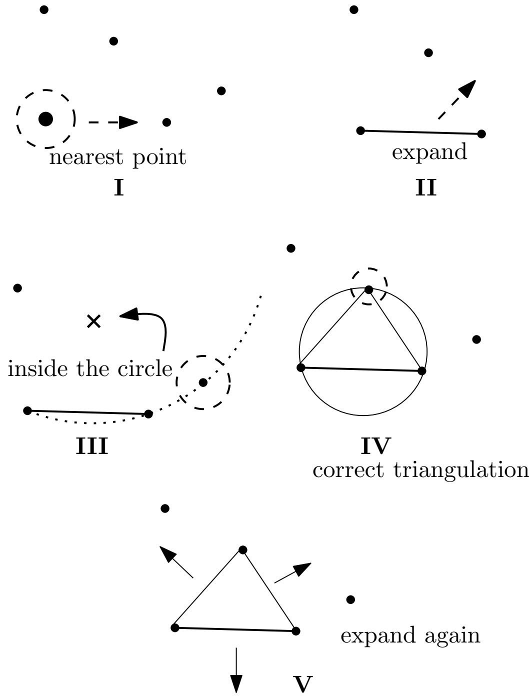

Figure III.1. Triangulation by incremental extrapolation

III. Methodology

A. Sequential incremental extrapolation

The extrapolating algorithm starts out by taking certain seed triangles (starting triangles) and subsequently adding an exterior point to one of the edges of the present triangles, to append a new triangle to the list of present triangles. We define AE as the list of active edges (that can be expanded into another triangle) and T as the list of triangles in $\mathcal{DT}$ at some iteration.

As shown in FigIII.1 the sequential algorithm starts from a particular seed site. next it expands by joining itself with the nearest site to form a line segment. this line segment is the first member of the AE , which is expanded by looking at points in increasing order of their distance from the edges. In step III we have a point within the circumcircle (marked as x) , hence the set of points is not a valid triangulation. Moving on to the next point we observe the circumcircle is empty resulting in a correct triangulation. Finally we observe that all the three edges of the triangle at the IV step constitute the AE which can be further expanded into triangles by looking at points lying near them.

B. Parallel Algorithm

The basic parallel algorithm starts from a number of seed sites and is presented in Algorithm1. Since all the active edge expansions can be done independently, this step is done in parallel. Furthermore, instead of starting the construction from a single seed point and extrapolating outward, we can start the construction from number of seed points.

To find if a particular point D lies within the circumcircle of $\triangle ABC$ with A, B, C in counterclockwise order we check if the following condition is satisfied. If it is true D lies within $\triangle ABC$'s circumcircle.

Algorithm 1 Parallel Incremental Delaunay Triangulation

Input: Point Set P , Output: T list of delaunay triangles

1) Locate various seed vertices to form the set S.
2) For $\forall s \in S$ in parallel : Join the nearest vertex in P with s and add all the edges to the active edge set AE
3) While $AE \neq \phi$
   a) For $\forall e \in AE$ in parallel : expand e by finding a point x which induces an empty (or smallest) circumcircle on $e \cup x$.
   b) Add $e \cup x$ to T
   c) Add the deges :$e \times x$ to AE . Remove e from AE

$$\begin{vmatrix} A_x - D_x, & A_y - D_y, & (A_x^2 - D_x^2) + (A_y^2 - D_y^2) \\ B_x - D_x, & B_y - D_y, & (B_x^2 - D_x^2) + (B_y^2 - D_y^2) \\ C_x - D_x, & C_y - D_y, & (C_x^2 - D_x^2) + (C_y^2 - D_y^2) \end{vmatrix} > 0$$

The algorithm can be visualized as breadth-first-search of triangles (rather than vertices) through circles (rather than edges). Hence it terminates in fixed number of iterations. Since the final graph is planar the number of edges and hence the number of triangles is $O(n)$. Hence there are $O(n)$ expansions and each expansion requires point search .Although the naive sequential version would take $O(n^2)$ time implementing the point searches through efficient data structures like quad trees reduces its running time $O(n \log n)$.The performance of the parallel algorithm will heavily depend upon the starting seed triangles. Further, as apparent from the algorithm the maximum level of parallelism is maximum size of $|AE|$

C. Implementation details

The data-parallel approach seems particularly suited for the GPU architecture. In general, we can assign one thread per active edge . For large values of N, this allows us to fully saturate the GPU with a large number of threads.

For a roughly uniform distribution of points across our domain , we divide the domain into a 16×16 mesh, resulting in 256 blocks (which is similar to the number of SMs available on the cluster, 240). each block starts out with a seed point and expands its triangulation parallely.

Step 3 in Algorithm1 is implemented as a series of CUDA kernels which are then typically run for a small number of iterations until the AE is empty and the triangulation is complete. The number of iterations can be approximated as $\log_2 \frac{N}{256}$, which can be easily seen from the fact that a complete delaunay triangulation of a block ,being planar will have $O\left(\frac{N}{256}\right)$ edges and since every iteration the number of edges contributing to the delaunay triangulation doubles we have the afforementioned number. Similarly the optimal number of threads can be approximated as $T_{opt} \sim \sqrt{\frac{N}{256}}$ as it is the maximum possible size of AE. Again it can be verified from the fact that since size of AE steps through 1

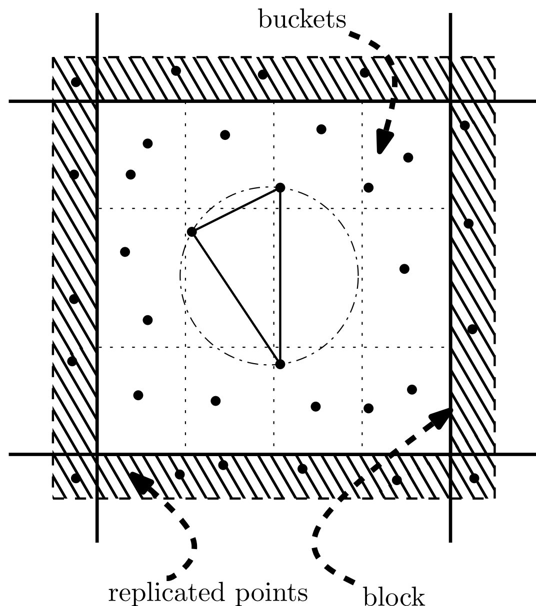

Figure III.2. Distribution of points

to its maximum size (say m) $1+2+\cdots m \sim \frac{N}{256}$. and hence $m \sim \sqrt{\frac{N}{256}}$.

To accelerate the search of points , each block was further divided into 16×16 buckets and the points stored in corresponding buckets. This enables faster search of points in the buckets that are nearer to a particular active edge which are most likely to be part of the triangle involving the edge.

After the blocks carry out their individual triangulations, these can be combined on CPU although this would lead to very poor performance. Another clever approach would be to exploit the uniform distribution of points. A fraction of the points from the neighbouring blocks can be copied into a particular block , resulting in a small replication but also enabling the triangulations extending aross block boundaries. A small caveat is that this technique might lead to inconsistencies which happens only when some 4 of the points are co-circular. To avoid such a case a small noise is added to every point so that eventually no four of them are co-circular and the the delaunay triangulation is unique. Lastly some very rare incomplete triangulations can be completed by the CPU (which does not occur if N is large enough)

FigIII.2 illustrates the distribution of the points using the above techniques.

IV. Experiments and Results

A. Experiment Setup

Initially the optimum number of threads was found out as a proof of the concept experiment for varying sizes of uniformly distributed points. Next the speedups with the sequential version as well as with $Triangle^{TM}$ [8](a popular delaunay triangulation software) was found out for sizes ranging from 32^2 to 4096^2. A nonuniformly distributed pointset involving two clusters of gaussian distribted points was also taken. To deal with the load imabalance that goes hand in hand with such distribution an ORB scheme was used to distribute the points across blocks but the triangulations carried out by individual blocks had to be combined on CPU to give the eventual triangulation.

B. Results

1) Optimum no. of threads: The relative time for various no. of threads over varying size of poinsets is shown in FigIV.1. It can be seen that the optimal number of threads depending on the data size $T_{opt} \sim \sqrt{\frac{N}{256}}$

2) Speedups: FigIV.2 shows the speedup of the parallel version on uniformly distributed points vs sequential and $Triangle^{TM}$. For 4096^2 points while the speedup is about $16\times$ vs sequential it is about 6 against $Triangle$. $Triangle$ is one of the fastest triangulator but uses fundamentally very different algorithms than the ones considered in the project. The overall tad discouraging results against the sequential version can be attributed to the fact that while the serial version is optimized by structures such as quad tree, but the parallel version uses a simple bucketing scheme as implementation of complicated structures such as quad tree or k-d tree on GPUs is very challenging.

3) Non-Uniform distributed points: The speedup values for a certain non-uniformly distributed pointset is shown in FigIV.3. Because of the large number of serial combination of the incomplete triangles on CPU, the speed up values take a hitting. It drops to around 7 for 4096^2 points

4) Sample triangulations: Sample triangulations result verifiable by the eye are shown in FigIV.4 & IV.5. In the latter the long skinnny triangles are the result of the triangulation carried out on CPU while the closely clustered triangles are mostly constructed by GPU. In th euniform case it can be seen none of the triangles are too thin thereby proving the fact that replicating a small fraction of neighbourhood points is effective.

V. Conclusions

The digital Delaunay triangulation on GPU [5] is about 2-3 times faster than our algorithm. Hence it seems to be the natural choice for delaunay triangulations on GPU, given that our implemenaton does not fare well against non-uniform distributed point sets. But the implementaion presented here is far from the algorithm's most effective implemenation. Infact the reasonable speedups obtained show the parallelizable potential of the incremental method. To be truly compared to its real world alternatives, many possible improvements shuld be considered for example looking at coalesced and minimal access of memory , reducing the number of kernel calls among others. Morever an efficient search space such as K-d tree on GPU can provide a much needed boost to the performance of our implementation. Nevertheless to truly realize the potential

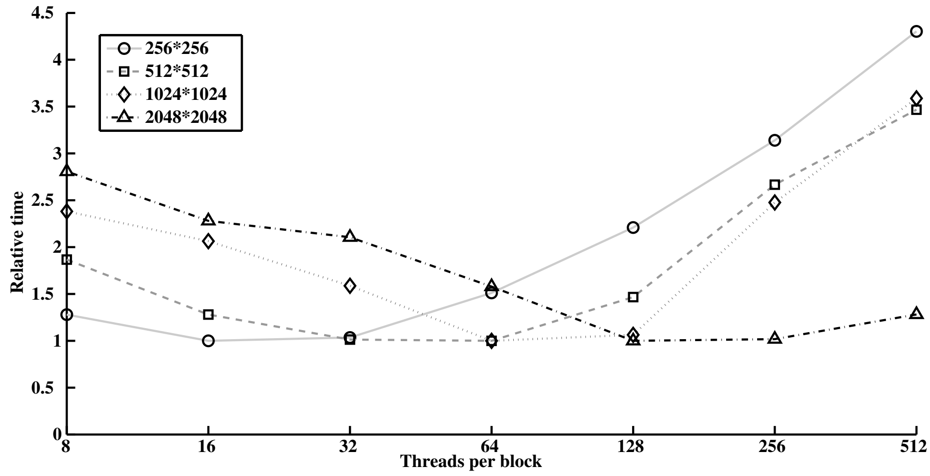

Figure IV.1. Varying threads per block for various sizes of input sets

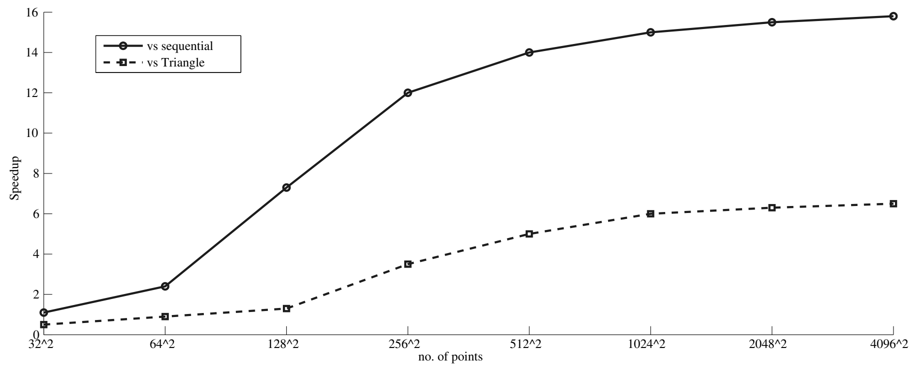

Figure IV.2. Speedup , vs Sequential and $Triangle^{TM}$

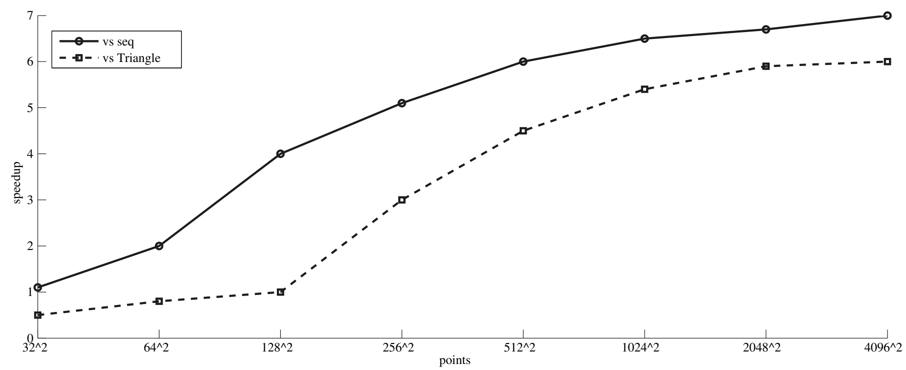

Figure IV.3. Speedup for non-uniform distribution

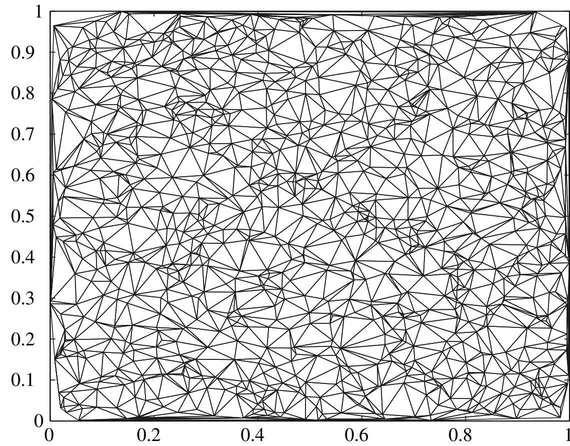

Figure IV.4. Triangulation of uniform distribution of points $N = 32^2$

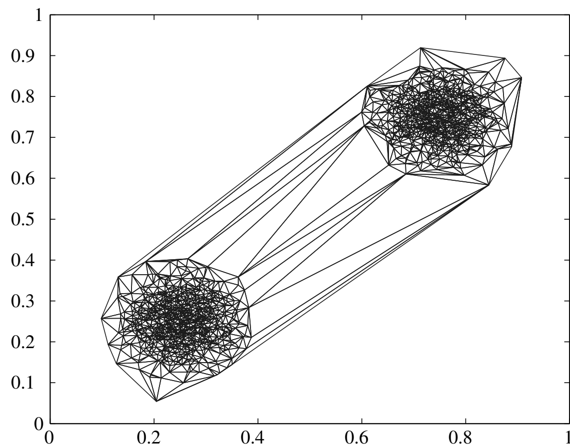

Figure IV.5. Triangulation for gaussian distributed points $N = 32^2$

of this method an online setting of points should be considered ,simulating which was beyond the scope of the project.

Some of the lateral or future directions are presented below.

A. Future Directions

- Handling points having high resolution coordinates (leading to lots of inconsistency and degeneracy) through GMP library
- Constrained Delaunay triangulations in which certain edges or triangles are bound to be present in a valid triangulation
- Extending the incremental 2D implementation to higher dimensions etc.

Acknowledgement

I would like to thank Prof. Sathish Vadhiyar for his encouragement and motivation and the department of SERC for its computing facilities

References

[1] P. Cignoni, C. Montani, and R. Scopigno, "Dewall: A fast divide and conquer delaunay triangulation algorithm in e^d," *Computer-Aided Design*, vol. 30, no. 5, pp. 333–341, 1998.

[2] S. Lee, C.-I. Park, and C.-M. Park, "An improved parallel algorithm for delaunay triangulation on distributed memory parallel computers," *Parallel Processing Letters*, vol. 11, no. 02n03, pp. 341–352, 2001.

[3] A. Bowyer, "Computing dirichlet tessellations," *The Computer Journal*, vol. 24, no. 2, pp. 162–166, 1981.

[4] D. F. Watson, "Computing the n-dimensional delaunay tessellation with application to voronoi polytopes," *The computer journal*, vol. 24, no. 2, pp. 167–172, 1981.

[5] G. Rong, T.-S. Tan, T.-T. Cao *et al.*, "Computing two-dimensional delaunay triangulation using graphics hardware," in *Proceedings of the 2008 symposium on Interactive 3D graphics and games*. ACM, 2008, pp. 89–97.

[6] C. A. Navarro, N. Hitschfeld-Kahler, and E. Scheihing, "A parallel gpu-based algorithm for delaunay edge-flips," in *The 27th European Workshop on Computational Geometry, EuroCG*, vol. 11, 2011.

[7] Y. A. Teng, F. Sullivan, I. Beichl, and E. Puppo, "A data-parallel algorithm for three-dimensional delaunay triangulation and its implementation," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. ACM, 1993, pp. 112–121.

[8] J. R. Shewchuk, "Triangle: Engineering a 2d quality mesh generator and delaunay triangulator," in *Applied computational geometry towards geometric engineering*. Springer, 1996, pp. 203–222.