Efficient Parallel Algorithm To Compute A Sub-complex Of The Weighted Delaunay Triangulation For Molecular Data

A Thesis

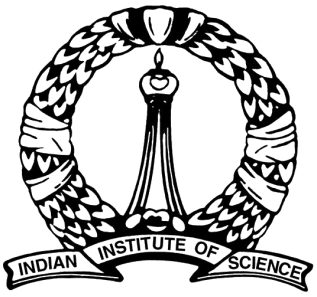
Submitted for the Degree of

Master of Technology

in the **Faculty of Engineering**

by

Ranjanikar Nikhil Prabhakarrao



Department of Computational and Data Sciences
Indian Institute of Science
Bangalore – 560 012 (INDIA)

JUNE 2016

DEDICATED TO

My Parents and Sister

Signature of the Author: .....

Ranjanikar Nikhil Prabhakarrao
Dept. of Computational and Data Sciences
Indian Institute of Science, Bangalore

Signature of the Thesis Supervisor: .....

Vijay Natarajan , Sathish Vadhiyar
Professor
Dept. of Computational and Data Sciences
Indian Institute of Science, Bangalore

Acknowledgements

Though only my name appears on the cover of this dissertation, a great many people have contributed to its production. I owe my gratitude to all those people who have made this dissertation possible and because of whom my graduate experience has been one that I will cherish forever.

I am deeply indebted to my thesis advisors Dr. Vijay Natarajan and Dr. Satish Vadhiyar for their guidance and council. They have been always there to listen and give advice. I am deeply grateful to them for the long discussions that helped me sort out the technical details of my work. I am also thankful to them for encouraging the use of correct grammar and consistent notation in my writings and for carefully reading and commenting on countless revisions of this manuscript. Learning is an essential life process that is often accidental and unforeseen. I firmly believe that I have learned more unquantifiably and intangibly from them than intentionally and directly. Their emphasis on critical thinking and academic rigor have deeply influenced my approach to both, academic and non-academic pursuits.

IISc today preserves a small piece of Bangalore of yesteryear, in which I felt privileged to find calm and peace during my graduate work. Its people and activities have played an important role during my time here. Labmates have been a constant source coffee and conversation, both academic and non-academic. Many friends and my classmates have helped me stay sane through these difficult years. Their support and care helped me overcome setbacks and stay focused on my graduate study. I greatly value their friendship and I deeply appreciate their belief in me. I am also grateful to the following former or current staff at Department of Computational and Data Sciences, for their various forms of support during my graduate study.

Most importantly, none of this would have been possible without the love and patience of my family. My parents and sister to whom this dissertation is dedicated to, have been a constant source of love, concern, support and strength all these years. I would like to express my heart-felt gratitude to my family.

Abstract

In the field of bio-molecules, it is utmost important to study the behaviour of a molecule while interacting with another. This interaction decides the functionality of the molecule. It is widely accepted fact that the geometrical shape of bio-molecules determines their functionality. Using alpha complex and regular triangulation, one can easily calculate most of the geometric parameters of a molecule. Volume of a molecule and many other things can be computed using alpha complex. In most of the applications, alpha complex corresponding to alpha values 0 and 1.4, is required. Here, the application requires a small sub-set of the complete triangulation. However, to get that small sub-set, we have to bear additional over head of computation of the whole triangulation. Hence, we propose a novel approach which can directly compute the sub-complex or sub-set of the triangulation without generating the whole triangulation.

We propose an algorithm to compute the sub-complex of a regular triangulation(RT) for 3D molecular data using localized properties. In this paper, we have implemented an incremental algorithmic approach, which builds a smaller tetrahedra first and then a larger one, to compute sub-complex of RT that is parametrized by a real value α. We have used edge and alpha optimization to reduce time complexity of algorithm. Our algorithm can exploit massive parallelism supported by GPUs in order to construct RT. For getting maximum advantage at an architectural level, we used three optimizations. Pinned memory and CUDA streams are used to reduce the data transfer time while texture memory is used to reduce memory access time.

All of these optimizations reduce the execution time of our algorithm by 27 %. We obtain upto 88 % improvement in execution time for smaller alpha value, in computing a sub-complex, over an existing state-of-art method gReg3d, which computes complete triangulation. We obtain speedup upto 9x over best sequential CPU implementation, CGAL for zero alpha value.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Delaunay triangulation (DT) is a particular triangulation that satisfies empty-circle property. In other words, it is a triangulation in which no point in the given point set lies inside the circumcircle of any triangle in the triangulation. Regular triangulation is similar to delaunay triangulation. Only difference between them is that DT is done for unweighted points and uses euclidean distance while regular triangulation is done for weighted points and uses power distance. Delaunay triangulation is a dual of voronoi diagram.

Let T be the finite set of points in 3D euclidean space. The voronoi region V_r of point p ϵ T is a set of all points in the space that are closer to p than any other point in T. This partitioning of the space is called voronoi diagram. Let w(p) be the weight of point p ϵ T and power distance pd(x,p) is given by pd(x,p) = $\|x - p\|^2$, where x ϵ T. [1]. Figure 1.1 shows the voronoi diagram and delaunay triangulation for unweighted points.

Molecules are represented using the union of balls model where each atom is represented by a ball and has a weight equal to the square of its van der Waals radius. The contribution from each atom p is equal to the intersection between the ball corresponding to the atom and the weighted Voronoi cell of p. The corresponding dual structure is called the dual complex. The dual complex is a sub-complex of the weighted Delaunay triangulation. If radii r of balls are increased to $\sqrt{r^2 + \alpha}$ then a dual complex of the corresponding set of balls is called the alpha complex. Figure 1.2 shows a weighted point set, corresponding weighted DT and the alpha complex. [13].

Alpha complex is a subset of regular triangulation. Delaunay triangulation along with alpha complex are the important geometric structures and have numerous applications in different fields such as computational geometry, computer graphics, numerical analysis, mesh generation in finite element methods, scientific visualization, robotics, image synthesis as well as mathematics and natural sciences.

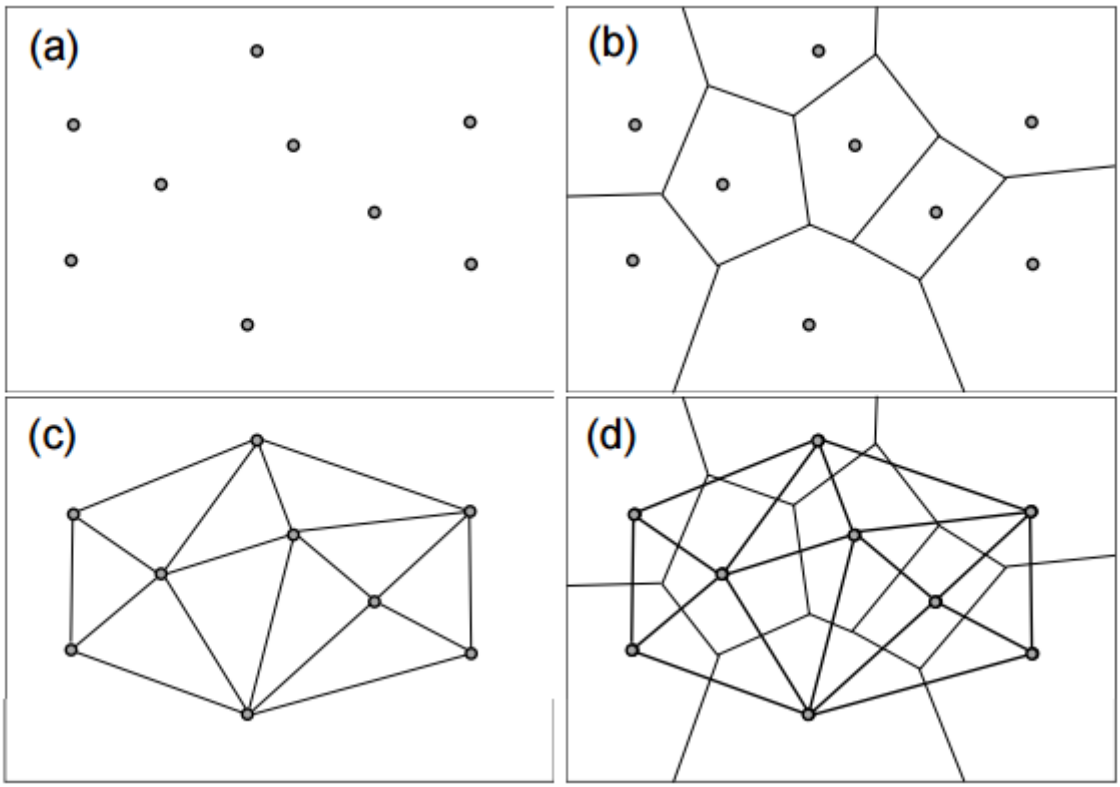


Figure 1.1: (a). Point Set. (b). Voronoi Diagram. (c). Delaunay Triangulation. (d). Voronoi and Delaunay complex overlayed

[17]

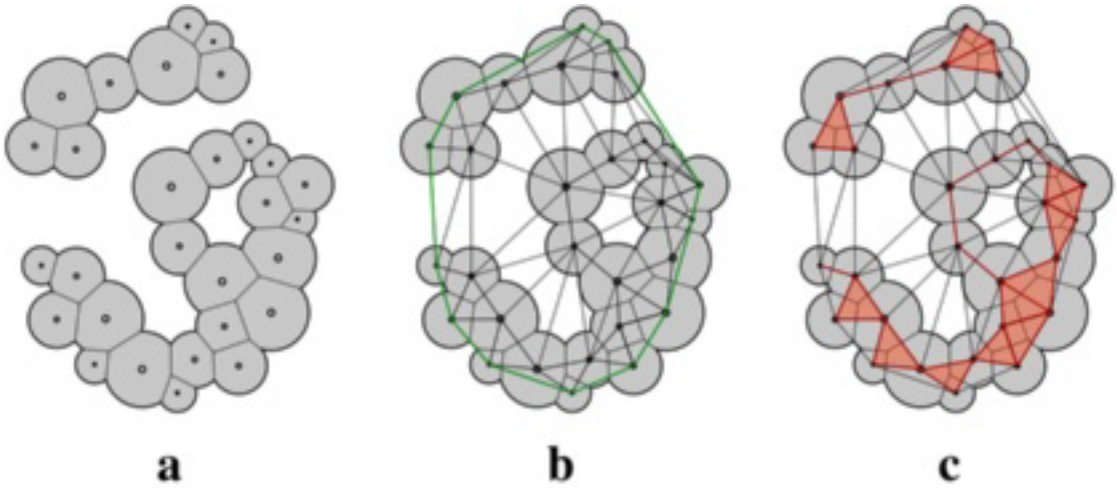


Figure 1.2: a)A set of weighted points in the plane b) RT of points in the plane c)Alpha Complex of points in the plane

[13]

In the field of bio-molecules, it is utmost important to study the behaviour of a molecule while interacting with another. This interaction decides the functionality of the molecule. It is widely accepted fact that the geometrical shape of bio-molecules determines their functionality. Hence, the geometrical shape of bio-molecule, determined in terms of its cavities, protrusions, dynamics and energetics, determines how it interacts with another bio-molecule. Using regular triangulation (RT) or weighted delaunay triangulation and alpha complex of atoms in the molecule, we can compute cavities and protrusions. The construction of the alpha complex plays an important role in finding out the volume of a molecule, which is one of the important parameters of geometric shape. Hence, it is of utmost important to construct RT and alpha complex for molecular data in order to study its behaviour [18].

In most of these fields, the computation of delaunay triangulation or RT is a bottleneck. Due to this, there is a need to develop a fast algorithm that can compute delaunay triangulation accurately and efficiently. There are many serial algorithms, but all are time and memory intensive. Hence, we require parallel implementation. We exploit fine grain parallelism supported by GPUs, which uses a massively parallel architecture with hundreds to thousands of processing elements, to efficiently compute a sub-complex of DT. As DT involves an irregular structure, very few serial algorithms can be parallelized. As a consequence, it becomes challenging to implement DT on GPUs.

Generally DT is computed first and then its sub-complexes such as alpha complexes are constructed. We have seen that RT and its sub-complexes are used in different fields and to generate sub-complex we have to generate whole DT, which is a bottleneck. However, in many applications, only a small sub-set like, alpha complex corresponding to 0 and 1.4, are required. To compute such small sub-set, we have to incur an overhead of computation of whole triangulation. Hence, we propose an algorithm which eliminates the bottleneck step of computing the whole triangulation and can compute the sub-set or sub-complex directly.

Hence, we propose a novel approach to compute sub-complex directly for a given alpha value without computing complete RT. The notion behind this approach is that the efficient computation of the sub-complex may lead to an efficient parallel computation of the RT. Given a 3D point set, which contains x, y, z coordinates and radius of each atom in a given molecule and some alpha value, our algorithm constructs sub-complex of a regular triangulation of a point set P that consists exactly the tetrahedra in the alpha complex. We aim to develop an efficient algorithm for the particular case of the point set representing the atoms in a molecule. The atom spheres (or balls) in the model of molecule have some properties that we can exploit to get an efficient algorithm.

1. Although two balls may interpenetrate, their centers can not get too close because of the

van der Waals forces. Hence, we can assume that the center of a ball can not lie in the another ball.

2. Their radii range is fairly restricted. By looking at van der Waals atom radii table, we can say that the ratio of the largest atom radius to the smallest atom radius is always less than or equal to three.[4]

3. The distribution of atoms is nearly uniform.

4. Fourth property is a proven theorem. It states that for each atom in a molecule, the maximum number of atoms in the molecule that it can intersect is bounded by a constant.[12]

Our algorithm consists of three stages. The first stage is the formation of the grid. This is useful as we can take advantage of the grid in the next step for neighbor search. This stage is performed on CPU. The second stage involves generation the possible tetrahedra list. Here, we perform neighbor search and then generate neighbor list. From this neighbor list, the possible tetrahedra list is generated. This is costlier than any other sub-stage. Hence, to reduce the time complexity of this stage, we use two optimizations namely edge optimization and alpha optimization. The third stage is checking the validity of tetrahedra. Here, we check whether the tetrahedron generated follows the property for alpha complex or not. All of these stages are performed on GPU. Moreover, for getting maximum advantage at an architectural level, we used three optimizations. Pinned memory and CUDA streams are used to reduce the data transfer time while texture memory is used to reduce memory access time.

All of these optimizations reduce the execution time of our algorithm by 27 %. Our algorithm gives better performance for small alpha values in computing a sub-complex than the state of the art method gReg3d, which computes complete triangulation. We obtain 15-88% improvement over the gReg3d for alpha values between 0-3. We obtain upto 9x speedup over the best sequential CPU implementation, CGAL.

In related work chapter, different algorithms to compute regular triangulation are discussed. Methodology chapter explains two different approaches. First part of this section explains how alpha complex can be used to generate RT and also explains the practicality of this approach. The second part discusses the approach of building RT from sub-complexes containing only tetrahedra. The main idea here is to incrementally construct larger tetrahedra from smaller tetrahedra. This part also contains an algorithm, data structure optimizations related to algorithm and optimization related to GPUs. To the best of our knowledge, this approach has not been tried yet. Experiments and result chapter gives in-depth insight about the effect of optimizations on the performance. It also gives the comparison between the result generated

by our algorithm with the fastest GPU algorithm (gReg3d) till now and also with the serial version. Future work chapter throws a light on the prospective future work that can be done in order to improve its performance.

Chapter 2

Related Work

Triangulations have been studied since 1934. There are several sequential algorithms which can be categorized under the following three classes.

2.1 Sequential Algorithms

2.1.1 Incremental Insertion

In this method, initially a simplex is generated which can encompass all the points, then the points in the point set are inserted one at a time. Due to the addition of new point, the earlier simplex is partitioned into sub-simplices. All the sub-simplices are then recursively checked for empty circumcircle property and flipping is performed to preserve the property of DT. Edge flipping is shown in figure 2.1. This method can also be extended to higher dimensions. For d-dimensional point set, the time complexity of this algorithm is O($nlog(n) + n^{\lceil (d/2) \rceil}$) [11].

In Bowyer-Watson algorithm[5] [20], points are added one at a time. After every insertion, triangles whose circumcircles contain the new point are deleted, leaving a star-shaped polygonal hole which is then re-triangulated using the new point. By using the connectivity of the triangulation to efficiently locate the triangles to remove, the algorithm can take O(nlogn) operations to triangulate n points in 2D.

2.1.2 Higher Dimensional Embedding

In this method, points are transformed to a D+1 dimensional space, and a convex hull of those uplifted points are constructed. By projecting the convex hull back to a D-dimensional space, we get the delaunay triangulation. For 2D point set, the time complexity of this algorithm is O(nlogn) [2].

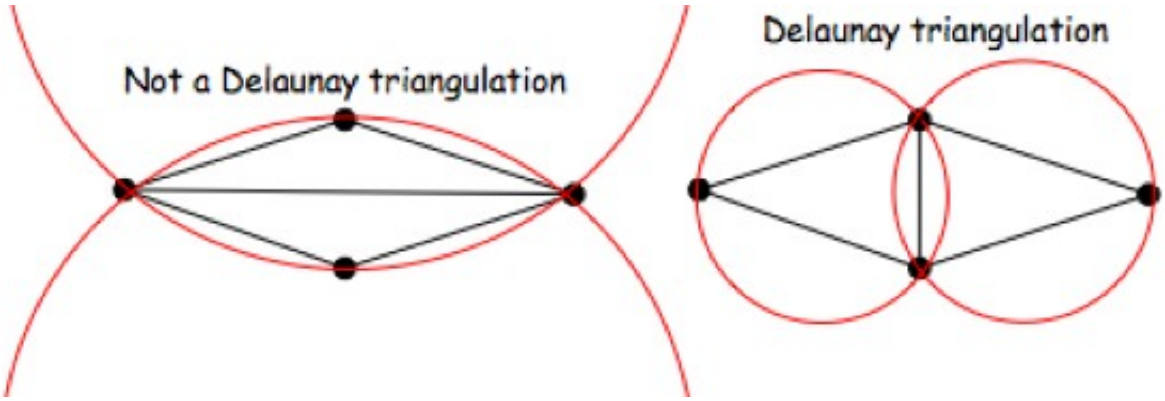


Figure 2.1: Flipping of an Edge

2.1.3 Divide and Conquer

These methods are based on recursive partitioning and local triangulation of point sets, and in the merging phase the resulting triangulation is obtained by cleverly combining two triangulations. These algorithms are proven to be optimal in 2D space, both in the average and the worst case. Explicit ordering of edges incident on a vertex makes it an optimal process in 2D. However, for a higher dimension, merging phase becomes time consuming.

2.2 Parallel Algorithms

Even before the advent of the GPU, there were a few attempts to triangulate the point set parallely, mainly using divide and conquer and higher dimensional embedding paradigm.

One of such successful attempts was done by Blelloch et al. [3]. The higher dimensional embedding technique is used to triangulate 2D point set. They modified the convex hull algorithm. After projecting points in a higher dimension, points are projected again on a plane, passing through the median and perpendicular to an original 2D plane. They proved that concatenating the results of the left half and the right half with the lower convex hull of these projected points will give the DT. They used MPI and achieved speedup upto 5x for 128k points on the Cray T3D with 64 processors. However, this approach is only applicable for 2D unweighted points and not for 3D weighted points.

De-Wall algorithm [9] uses a divide and conquer strategy. In this algorithm, points are divided into two halves and while dividing the points, triangulation of border points are constructed. This border triangulation (wall) will not be dismantled afterwards. After construction of the wall, the point set is divided into two parts and both of these parts can be constructed independently. Hence, each processor constructs its wall and generates two independent halves

which can be processed further by other processors. Initially, parallelism is less, but as algorithm progresses parallelism increases. For 2000 points and 64 processors, they achieve a speedup up to 19x [8].

Third algorithm (gReg3d) [6] is based on incremental insertion and parallelized on the GPU. There are two phases. The first phase is to perform parallel flipping insertion based algorithm to compute approximate DT. To avoid getting stuck while flipping, the author has suggested to perform flips after each iteration of point insertion. In the second phase, star-splaying approach is used to convert locally non-delaunay facets into locally delaunay facets. They used NVIDIA GTX 580 Fermi graphics card and obtained upto 10x speedup for 3M unweighted points in 3D. This algorithm is for unweighted point, but can be converted into the weighted point algorithm.

Chapter 3

Methodology

We have implemented two different incremental approaches

3.1 Part I : using alpha complex

The basic idea of this approach is that alpha complex is a subset of the RT and if we increase alpha to a large enough value, then the alpha complex will essentially be equal to the RT. Figure 4.1 shows the flowchart of algorithm 1.

Algorithm 1 has two stages. The first stage is dividing the space into a grid of proper size. The second stage is to generate the alpha complex. The first stage can be done on the CPU and the whole data is then transferred to the GPU. In the second stage, we invoke three kernels to compute different simplices and each thread will work on a point in point set. The first kernel generates the edges, which are not part of the triangles and the tetrahedra, the second kernel generates triangles and the third generates tetrahedra.

However, this algorithm will not give correct results when four or more atoms intersect with each other. For example, consider four atoms A, B, C, D in the 2d plane as shown in figure 4.2. All of the four atoms are intersecting with each other. Our algorithm gives all edges i.e. AB, BC, CD, DA, AC, BD. However, the edge BD is invalid because it does not follow the property of RT. The problem here is we can not decide whether an edge belongs to alpha complex just by focusing on two points. We have to construct a triangle in 2d in order to decide that. The problem with this strategy is that we do not know, in advance, at what alpha value this triangle will form. Similarly, we can not decide which edge or a triangle is a part of the alpha complex based only on the local properties at that alpha value in 3d. However, there are theorems for tetrahedra in 3d which are useful in deciding the validity of tetrahedra at that alpha value. Hence, we switch to the second approach.

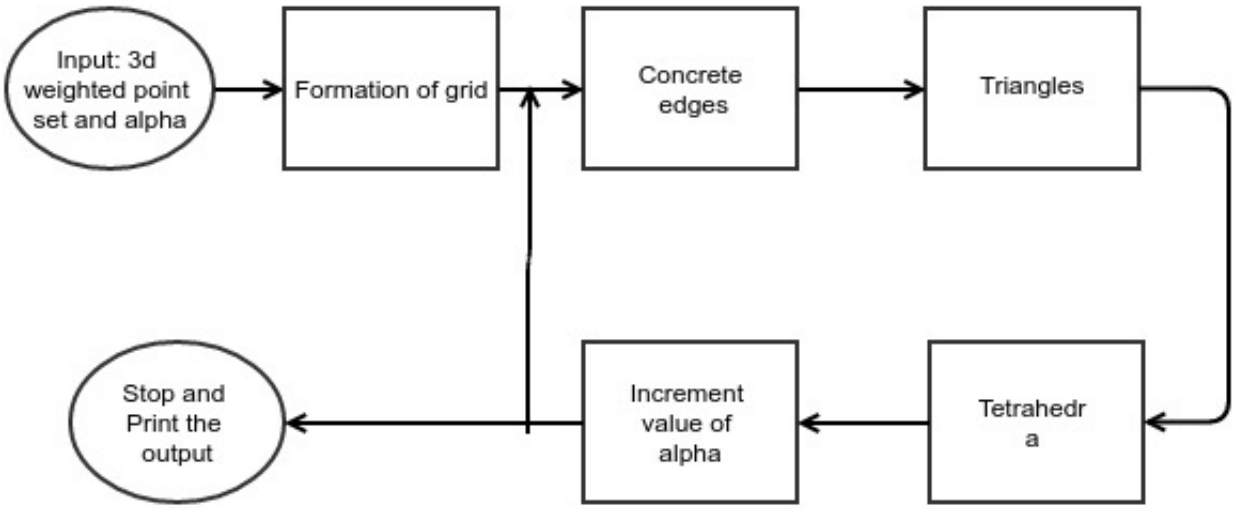


Figure 3.1: Flowchart for algorithm 1

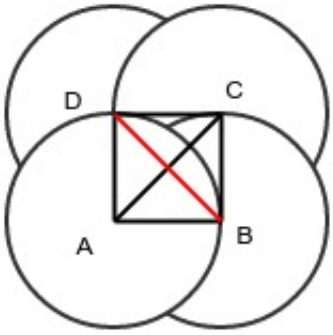


Figure 3.2: Example that will contradict Part 1 algorithm

Algorithm 1 Algorithm for part I

```
1: procedure
2:     Formation of grid -Make 100*100*100 size grid
3:     loop:
4:     3 GPU Kernel calls
5:     Kernel I:Construct Concrete Edges
6:     Kernel II:Construct Triangles
7:     for Every tetrahedron in possible tetrahedra list do
8:         if Satisfies orthogonal property then
9:             Add the point to final tetrahedra list.
10:        end if
11:    end for                                  ▷ // Kernel III:Construct tetrahedra
12:    Increment alpha value by width of grid.
13:    Go to loop until every point gets saturated.
14: end procedure
```

3.2 Part II : using sub-complex containing only tetrahedra

Algorithm 2 Algorithm for part II

```
1: procedure
2:     Formation of grid -Make 100*100*100 size grid
3:     3 GPU Kernel calls
4:     Kernel I: Generation of the possible tetrahedra list
5:     Kernel II: Checking the validity of RT properties for non-degenerate cases
6:     Kernel III: Checking the validity of RT properties for degenerate cases
7: end procedure
```

3.2.1 Algorithm

As there are proved theorems about the tetrahedron in 3d, we can compute tetrahedron based on local properties and triangulation generated by this method will be unique assuming the points are in general position. Figure 4.3 shows the flowchart of algorithm 2.

Algorithm 2 has two stages. The first stage is dividing the space into a grid of proper size. The second stage is to generate tetrahedra list and check validity of tetrahedra for generating unique triangulation in both degenerate and non-degenerate cases. The first stage is done on the CPU and the second stage is done on the GPU. We invoke three kernels in the second stage. The first kernel is used to generate the possible tetrahedra list. The second and third kernels

Algorithm 3 Kernel I: Generating possible tetrahedra list

```
1: procedure
2:     Search the points in r + r_max neighbourhood in every direction.
3:     for Every tuple in point list do
4:         Find the ortho-center and ortho-radius for the tetrahedron
5:         Compute the α at level
6:         if α given by user >= α at level then
7:             Add the tetrahedron to possible tetrahedra list.
8:         end if
9:     end for
10: end procedure
```

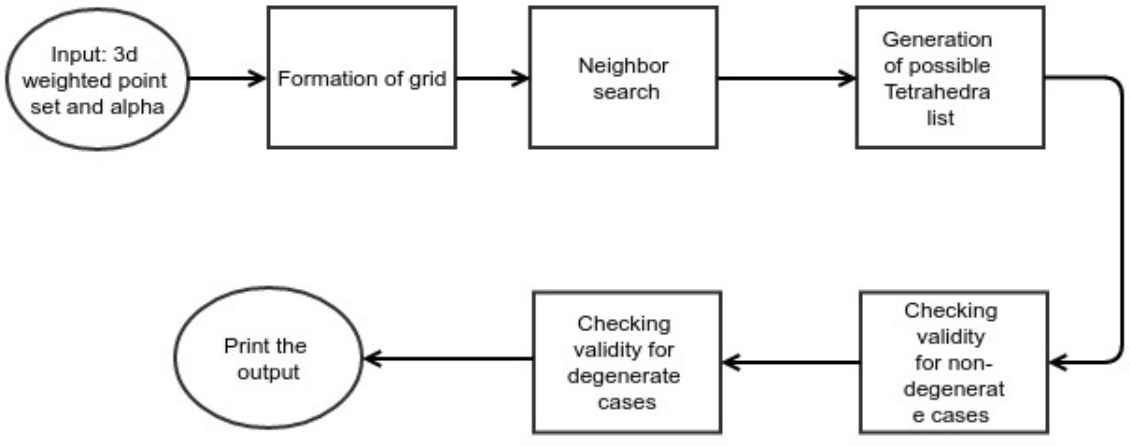


Figure 3.3: Flowchart for algorithm 2

are used to check the validity of the triangulation.

3.2.2 First stage

In this stage, we have divided the 3d molecular space into a grid of size of 100*100*100. This space division is very useful in searching the neighborhood points in second stage. As the neighbor search is restricted to a cube of length equal to twice the addition of the radius of the point or an atom under consideration and largest radius of an atom in a molecule, we have divided the space optimally. Larger the grid size, smaller the number of points in a cell and we have to search more number of cells. As a result, memory complexity increases. However, if we use smaller grid size, there will be a larger number of points in a cell. Though we are required to search a less number of cells, time complexity increases due to a large number of points in a cell. Hence the optimal division of space is important. After doing some experiments, we find out that 100*100*100 is an optimal grid size for our application.

3.2.3 Data Structure

Coalesced memory is an important factor in improving performance on GPUs. Here, each thread will access consecutive memory locations. This actually helps to access all the elements required for 32 threads using single access. This in turn increases the effective bandwidth. To achieve this, we have sorted the data according to $cell_{index}$ and the $cell_{index}$ number is generated for all cells using the following formula

$$cell_{index} = cell_x * 100 * 100 + cell_y * 100 + cell_z$$

All the points, i.e. x, y, z co-ordinates of points and their radii are also arranged based on sorted $cell_{index}$ array. The data structures are set up on the CPU and then the data is transferred to the GPU. As a result, the points having same $cell_z$ co-ordinate will be together. While searching the neighborhood we are generating the $cell_{index}$ in such a way that the points having same $cell_z$ co-ordinates get accessed together. Therefore, our data structure can exploit coalesced memory access and increase the effective bandwidth.

3.2.4 Kernel I: Generating possible tetrahedra list

Algorithm 3 gives the detailed description of this stage. Here, each thread works on a point in the data set. There are two sub stages in this stage. The first one is generating neighborhood point list. For this, we are searching in the cube centered at the focused point and has a length equal to twice the addition of the radius of the focused point and largest radius of a point in the data set. Using assumption 1, we can say that the maximum distance between centers of

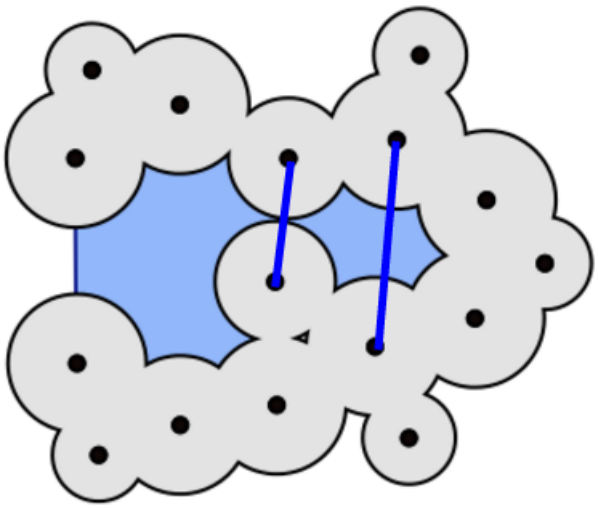


Figure 3.4: Example for edge optimization in 2d

two points, that are just touching each other, is $r + r_{max}$ where r is radius of a point and r_{max} is radius of largest point in a molecule. Hence, this search box or cube gives us all the points that can be touched to a focused point.

The second sub stage is to generate the tuples from the neighborhood point list, which was generated in earlier sub stage. The time complexity of this sub stage is n^3 where n is the number of neighbors. Hence, we used two optimizations.

3.2.5 Edge optimization

The first optimization is called edge optimization. The basic idea is that tetrahedra will be formed only if all of the four points intersect with each other. So, if any of the pair is not intersecting then we can ignore all the tuples having the respective pair. For example, in figure 4.4, blue edges can not be a part of alpha complex at this alpha value. The reason is that balls corresponding with this edges are not intersecting with each other. So, we can ignore these edges.

3.2.6 Alpha optimization

The second optimization is based on alpha hence its called alpha optimization. Here, we are using the basic idea in edge optimization together with the alpha complex concept. There are two facts that will be used in this optimization. The first fact is the basic idea of edge

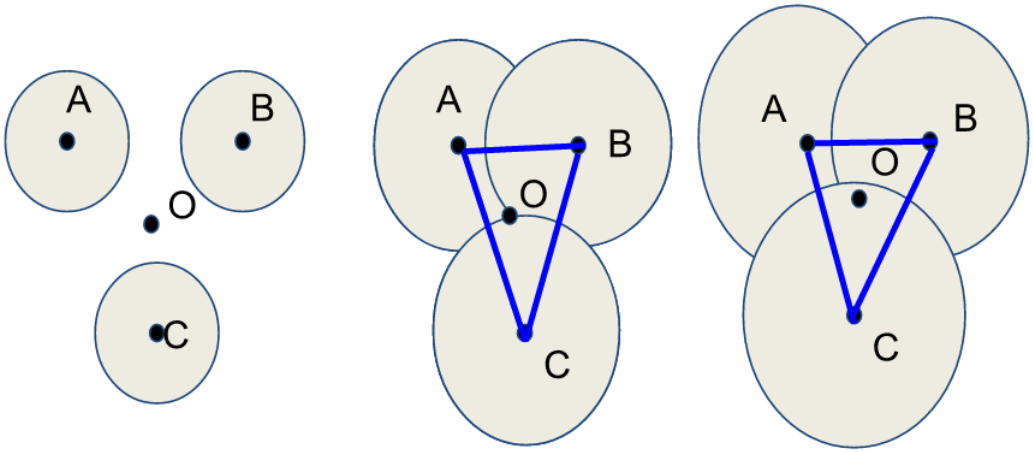


Figure 3.5: Example for alpha optimization in 2d

optimization and the second fact is that orthocenter is equidistant from all four points of tetrahedra. From these two facts, we can say that if all the four balls are grown by alpha (alpha at level) such that for all of them orthocenter lies on the surface, then all the four balls have a point i.e. orthocenter as a common intersection. Hence, alpha at level or greater alpha value than the alpha at level, these balls will always have a common intersection. As a result, those four points will not form a tetrahedra if the alpha value is below the alpha at level. For example, consider three atoms A, B, C in 2d shown in figure 4.5 and point O is orthocenter. According to alpha optimization the triangle will form only if the grown ball encompasses the orthocenter. That's what is shown in figures 4.5.

Tetrahedra, generated after all these steps, are added to the possible tetrahedra list and given to Kernel II as an input.

3.2.7 Kernel II and III

As we do not know, beforehand, how many possible tetrahedra each thread will generate, we have calculated the maximum number of tetrahedra that can be generated from the maximum number of neighbors in neighborhood list. Each thread is allocated a chunk of global memory equal to the maximum number of tetrahedra. However, each thread will not use all the allocated global memory. So, most of the array elements are empty and we require contiguous filled memory which can be easily used by later GPU kernels. Hence, we are using prefix sum to collect all tetrahedra generated by all threads in a single array. For this, we are using prefix sum library provided in the thrust package of CUDA. This array is actually provided as an input to kernel II. In kernel II and III, each thread will work on a tetrahedron.

In both of these kernels, we are checking the validity of the tetrahedron. The tetrahedron is said to be valid if there are no points in the data set which lies in orthosphere generated by the tetrahedron points. For checking validity of the tetrahedron, we have to get orthocenter and orthoradius of orthosphere. For this, we have to solve linear system of equations. We are using LU factorization for this. This is done for every tetrahedron.

The actual implementation uses the floating point types provided by the processor architecture. Even if these data types are used to represent the input, the computation can produce an intermediate or final result which cannot be precisely represented in the native data type. For example, orientation test for positively oriented tetrahedron in 3d can produce false answer due to numerical inaccuracy. To overcome this, we are using exact floating point computation. However, floating point arithmetic is fast but unreliable and exact arithmetic is precise but slow. Hence, we are using a mixed-precision approach where we are calling the exact computations only when the error crosses a threshold epsilon value. Our implementation adapts the exact floating point computation implemented by Shewchuk [19] on the GPU.

Exact floating point computation uses hundreds of registers and memory for stack is limited to 512KB. Additionally, exact computation for insphere test needs local storage of up to 14880 floating point numbers[6] for expansions and other results. Due to these constraints , we are invoking two separate kernels. In kernel II, only points which are at a distance greater than $r + \epsilon$ and less than $r - \epsilon$ are considered and then the tetrahedron is added to the output if it is valid. Here, we are using floating point arithmetic. In kernel III, we are considering the points in between $r + \epsilon$ and $r - \epsilon$. Here, we are using exact computation.

In many computational geometry problems, degenerate cases occur. Consider that the input has four points in 3d that are coplanar. An exact variant of orientation test of these points will report that they are coplanar. These cases are called degenerate cases. The elegant solution to handle such cases is Simulation of Simplicity (SoS) [10]. This general method simulates a perturbation of the input points such that they appear to be in general position. We have adopted the SoS implementation provided in gReg3d[6]. We call Sos implementation only when exact computation in kernel III gives zero value for both orientation and insphere test.

3.2.8 Optimization related to GPUs

We have implemented three optimizations related to GPUs which increase the performance considerably. We found out that memory transfers and computation of kernel I are taking too much time. Hence, we decided to use following optimizations.

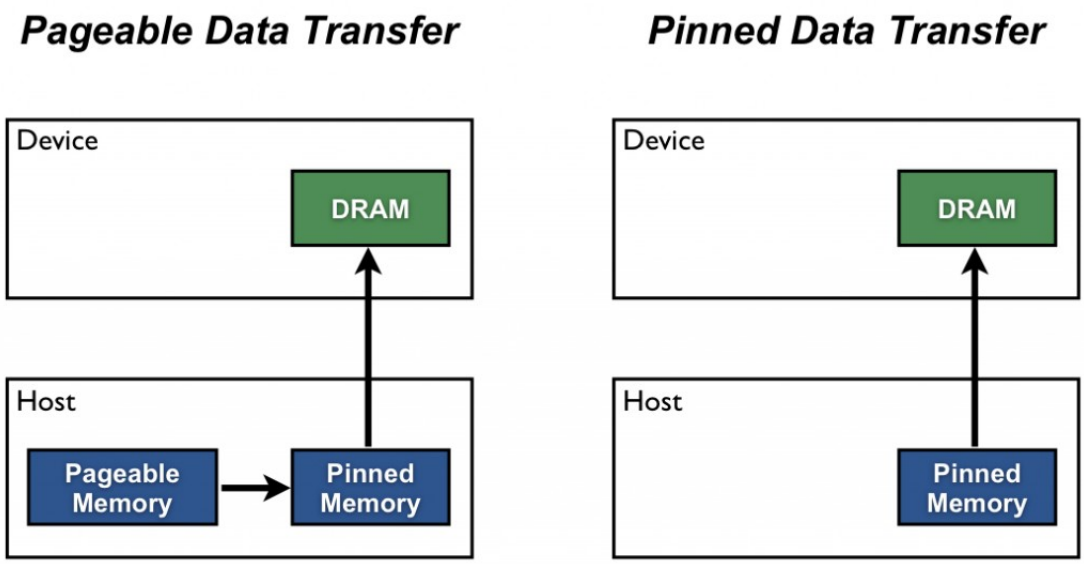


Figure 3.6: Pinned memory

3.2.8.1 Pinned memory

Host (CPU) data allocations are pageable by default. The GPU cannot access data directly from pageable host memory, so when a data transfer from pageable host memory to device memory is invoked, the CUDA driver must first allocate a temporary page-locked, or pinned, host array, copy the host data to the pinned array, and then transfer the data from the pinned array to the device memory. As can be seen in the figure 4.6, pinned memory is used as a staging area for transfers from the device to the host. We can avoid the cost of the transfer between pageable and pinned host arrays by directly allocating our host arrays in the pinned memory. We allocated pinned host memory in CUDA using cudaMallocHost().[14]

3.2.8.2 CUDA streams

A stream in CUDA is a sequence of operations that execute on the device in the order in which they are issued by the host code. All device operations (kernels and data transfers) in CUDA run in a stream. When no stream is specified, the default stream (also called the null stream) is used. The default stream is different from other streams because it is a synchronizing stream with respect to operations on the device: no operation in the default stream will begin until all previously issued operations in any stream on the device have completed, and an operation in the default stream must complete before any other operation (in any stream on the device) will begin. Using streams, we can perform the data transfers parallelly. [16][14]

3.2.8.3 Texture memory

Texture memory is one of the important type of read-only memory that can improve performance and reduce memory traffic when reads have certain access patterns. Although texture memory was originally designed for traditional graphics applications, it can also be used quite effectively in some GPU computing applications. Texture memory is cached on chip, so in some situations it will provide higher effective bandwidth by reducing memory requests to off-chip DRAM. Specifically, texture caches are designed for graphics applications where memory access patterns exhibit a great deal of spatial locality [15]. As it contains spatial type of caching, the memory access time reduces considerably. As a result, texture memory is faster than global memory. [14]

Pinned memory and CUDA streams help in reducing the memory transfer time. However, texture memory helps in reducing the data access time in all kernels.

Chapter 4

Experiments and results

We implement our algorithm using CUDA 5.5. All experiments are performed on a cluster with 5GB of memory and NVIDIA Tesla K20 graphics card. We first compare the performance of our implementation with the fastest sequential 3D RT, CGAL[7], on the CPU and then with state-of-the art method, gReg3d. Subsequently, we analyze the effect of the techniques we propose and effect of the different GPU optimizations on the performance.

Name	No of points	Memcpy	Kernel1	Prefixsum	Kernel 2	Kernel 3
1U71	1505	4547	18195	343	5744	64
3N0H	1509	4248	19812	159	5763	46
3L3R	1520	4297	19470	210	5555	43
3SY7	2969	5150	21982	153	5103	472
3SYB	3157	5287	21624	168	5386	450
1B4V	3848	6365	22237	187	4990	61
4HHB	5306	11792	66362	394	17884	81
2J1N	8142	9417	31165	328	4646	433
3SY9	11297	11015	39154	431	4398	44
2VL0	15156	25452	160451	789	91119	180
1K4C	16068	15091	78586	583	5607	51
2OAU	16647	28403	173204	826	108397	175
2BG9	17923	36833	222422	1047	178382	257
1AON	58674	52569	210387	1974	72464	148

Table 4.1: Data set sizes and splitups of time for different stages for zero alpha

4.1 Speedup over best sequential CPU implementation

We have taken 14 different data sets and calculated their speedup over a sequential CPU implementation. As can be seen from table I, the number of points in data set varies from 1500

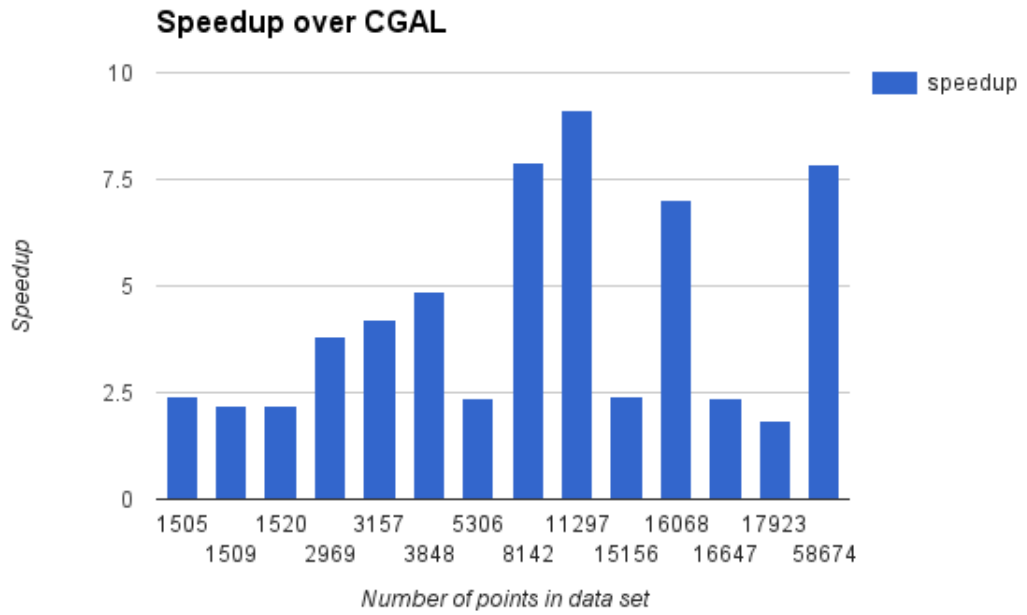


Figure 4.1: Speedup over sequential CPU implementation

to 60000. Figure 4 shows the speedup that we obtain over the best sequential code. We obtain speedup upto 9x. The prominent reason for speedup is that we are using approximately 2500 cores, whereas only one core is used in CPU implementation. All the optimizations, explained above, helps to reduce the execution time, which in turn results in an increase in speedup. For some cases such as 5306, 15156, 16647 and 17923 we are getting very less speedup. Primary reason behind this is that these molecules are more compact than others. Hence, they contain more molecules in their neighborhood. This in turn increases work in generating tuples step and which is also evident from table 5.1.

4.2 Running time comparison

Our algorithm computes a sub-complex of the RT that is parametrized by a real value α whereas gReg3d computes the complete triangulation. Still, we are comparing our algorithm with gReg3d. The reason behind this is that we can compute the subcomplex of the RT that is parametrized by a real value α only after the computation of whole triangulation. So, gReg3d requires a little bit of post processing to get the required sub-complex. Hence, this is a fair comparison. Figures 5.2, 5.3 and 5.4 represent the percentage improvement in execution time for calculating tetrahedra for different alpha value over a gReg3d. It is evident from the figures that, in most of the cases, as the number of points increases the alpha up to which our algorithm

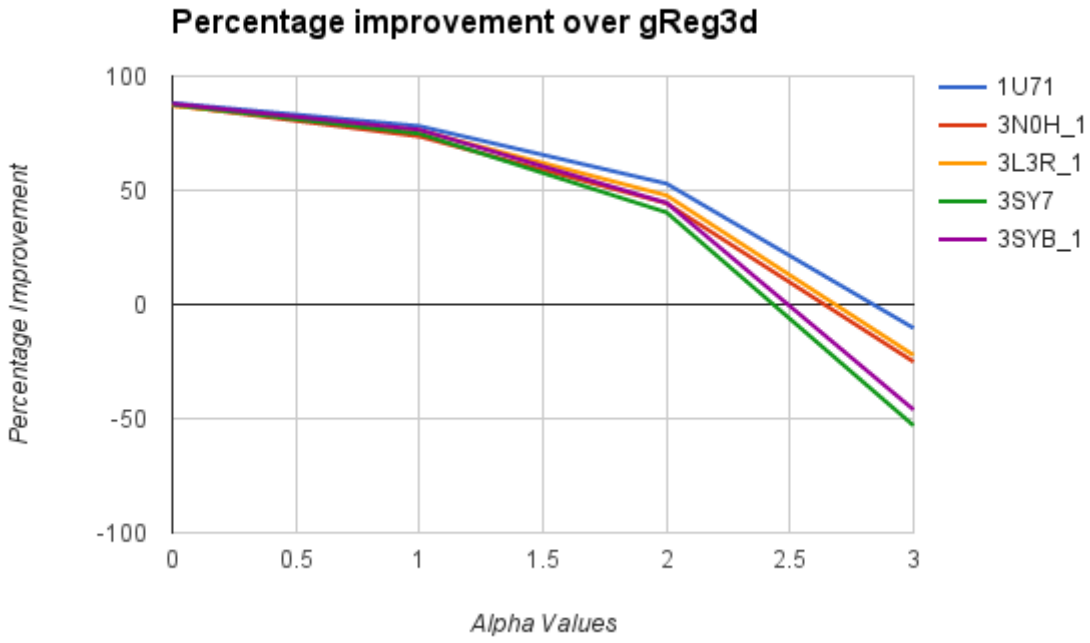


Figure 4.2: percentage improvement over gReg3d

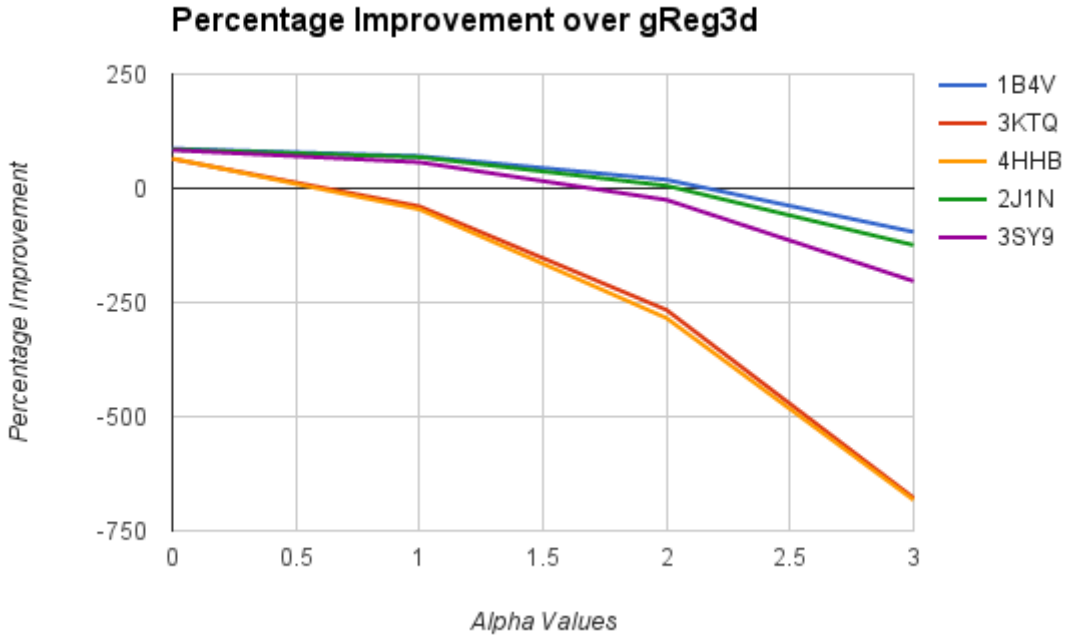


Figure 4.3: percentage improvement over gReg3d

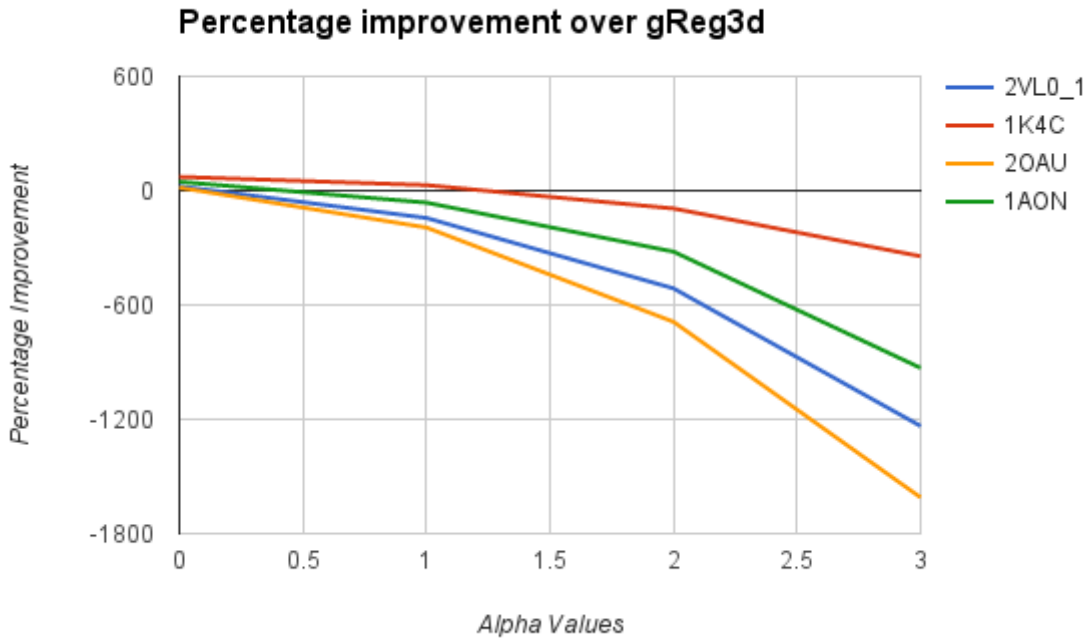


Figure 4.4: percentage improvement over gReg3d

can perform better than gReg3d decreases. The most prominent reason for such behavior is that molecules have a large number of neighbors which in turn increases the number of tetrahedra in the possible tetrahedra list. In most of the applications we require, sub-complex of 0 and 1.4 as alpha value. From figure 5.5, we can see that up to 15000 points, we can compute sub-complex for both of these values. However, after that we can only compute for zero alpha value. The reason for this behavior is same as above. We obtain up to 88 % improvement for calculating 40 % of total tetrahedra in triangulation. In general, we obtain 15-88% improvement over gReg3d for alpha values between 0-3.

Figure 5.7 gives the percentage redution in execution timings with optimization related to GPU turned on and off. Here, we can see that texture memory gives more reduction in time. Texture memory gives upto 23 % reduction in time and with all optimizations enabled, the execution time is reduced by 27 %.

Figure 5.8 shows the percentage breakup of timings for different stages for the optimized version of the code and it is evident that kernel I is still taking the highest time. As it contains generation of neighborhood point list and tuples from it, kernel I should logically take more time and this is what we exactly see in figure 5.8.

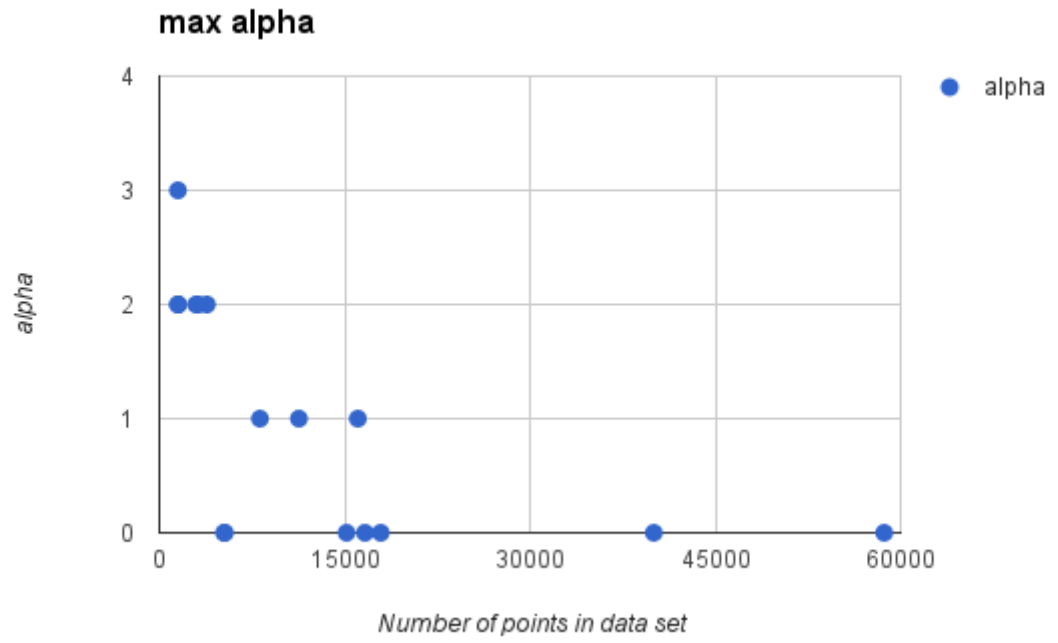


Figure 4.5: Alpha upto which our algorithm gives better performance than gReg3d

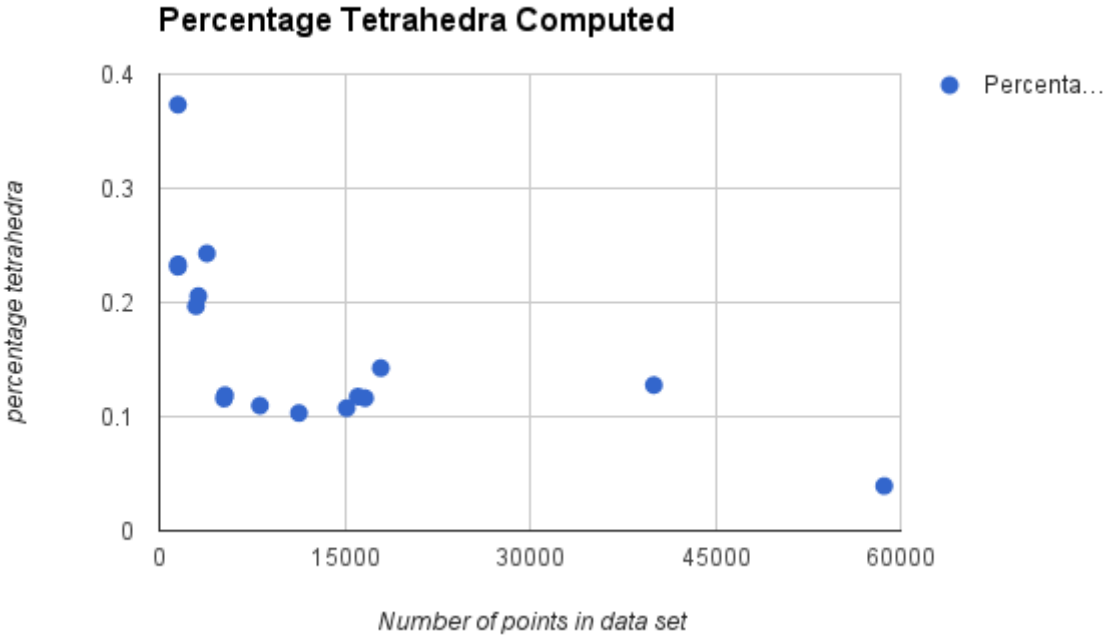


Figure 4.6: Percentage tetrahedra which we are computing over whole triangulation

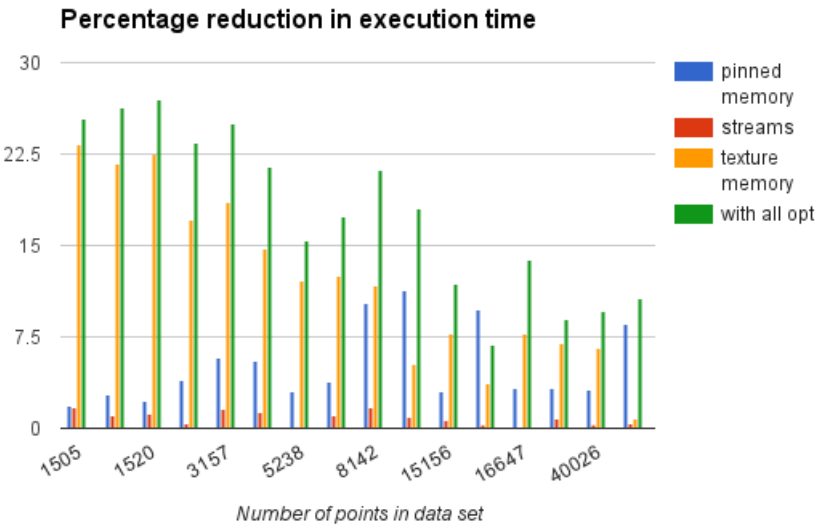


Figure 4.7: Percentage reduction in execution time when different optimizations are turned on and off

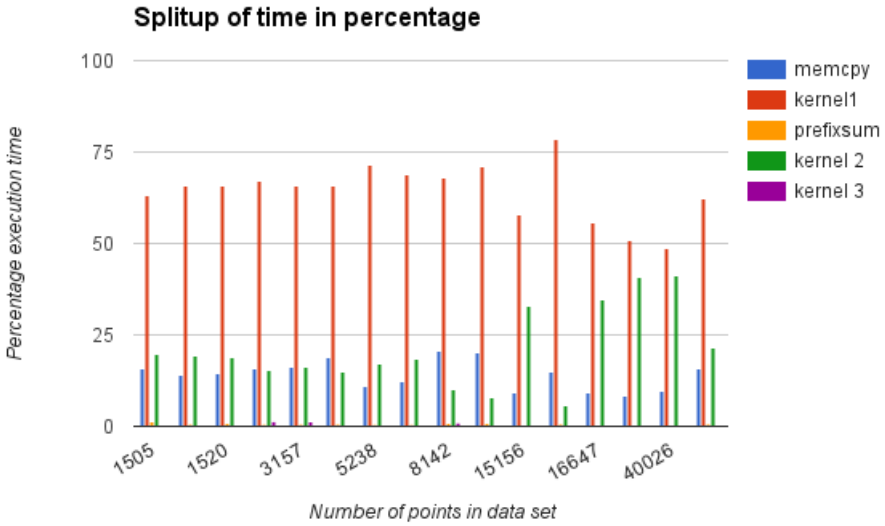


Figure 4.8: Percentage breakup of times for different stages

Chapter 5

Conclusions

The focus of this thesis is to efficiently construct the sub-complex of RT containing only tetrahedra that satisfy the alpha complex properties. In many application, we need only a small sub-set of the triangulation and still, have to bear the overhead of the computation of the whole triangulation. In this paper, we have shown a novel approach to construct the sub-complex of RT. We have shown that it is possible to construct sub-complex solely based on local properties.

In this paper, we described an incremental algorithmic approach, as in building a smaller tetrahedra first and then a larger one, to compute sub-complex of RT that is parametrized by a real value α. We have implemented two optimizations to reduce time complexity of algorithm. Edge optimization reduces the complexity of tuple generation step and alpha optimization helps in pruning the tetrahedra from the generated tetrahedra list. For getting maximum advantage at an architectural level, we used three optimizations. Pinned memory and CUDA streams are used to reduce the data transfer time while texture memory is used to reduce memory access time. Experimental results show that the reduction of 27% is obtained when all the optimizations are turned on. Moreover, texture memory alone contributes upto 23% reduction for the smaller data sizes but pinned memory alone gives upto 11 % reduction for the larger data sizes.

We have shown that our algorithm can exploit massive parallelism supported by GPUs in order to construct RT. Our algorithm gives upto 88 % improvement in execution time over the best implementation till now, gReg3d, for a smaller alpha values. Our algorithm handles degenerate cases also. Our algorithm gives speedup upto 9x over the best sequential CPU implementation, CGAL for zero alpha value.

To the best of our knowledge, this is the first algorithm which computes sub-complex directly. It is the first step towards the efficient parallel computation of the complete regular triangulation by incrementally building the sub-complexes.

Chapter 6

Future Work

There are few limitations of our algorithm. One of the limitations is that it can be used only for molecular data. The second biggest limitation is that it can compute whole RT but takes a lot of time. This happens because our algorithm computes the tetrahedra incrementally based on tetrahedra sizes.It can only compute sub-complex for a given alpha value. A very large alpha value is required to get complete RT. As a result, each atom will have more number of atoms in its neighborhood. This in turn results in an increase in the time complexity of tuple generation step. Hence, it takes a lot of time to compute complete RT.

The possible solution to this problem is to increment alpha value gradually and compute sub complex for each alpha value. However, in our algorithm this strategy will not work. The most prominent reason for such behavior is that, as we increment the alpha value we still have to consider the points from lower alpha value. For example, we compute the sub-complex for zero alpha value and then we increment alpha by one. If we consider points in annular region of two neighborhood search boxes such as box for zero alpha value and box for incremented alpha value, then we will not get correct triangulation. The points in the zero alpha box also play an important role in deciding the triangulation. Hence, for incremented alpha value also, we have to consider all the points in the box which is generated using incremented alpha value. As a result of this, we are actually losing the advantage of an incremental approach.

To circumvent this problem, we can use the star completion approach. The star of a point is said to be complete if all the terahedra incident on that point form a sphere in 3d. This point is also called as saturated point. This strategy will help in our algorithm. After a few increments in alpha value, most of the points will get saturated and then we don't have to consider those saturated points in next alpha values. This will reduce the number of points in neighbor list and also reduces the time complexity of generation of tuple step. Hence, this strategy will work.

Another possibility is to use more than one GPU device. This actually speedups the computation further more by taking advantage of parallelism provided by GPU devices. We can compute RT for larger molecules. It might be possible that this will scale well in spite of communication overhead produced by using more than one GPU device.

For larger data set sizes which can not accommodated in a GPU memory, we can use hybrid approach. In this, data can be divided in CPU and GPU and each will either perform the whole algorithm and then communicate for outputting the result or each will perform a step in the algorithm and communicate after each step. We can use CUDA and OpenMP to do such kind of programming.

Besides, our implementation is currently memory bound, especially on some GPUs with very high computation power, so it will benefit from improvements in the data structure and memory access optimizations.

Hence, there is a scope for the improvement in algorithmic as well as in computational part of the problem.

Bibliography

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

[2] Franz Aurenhammer. Voronoi diagramsa survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, 23(3):345–405, 1991. 6

[3] Guy E Blelloch, Gary L Miller, Jonathan C Hardwick, and Dafna Talmor. Design and implementation of a practical parallel delaunay algorithm. *Algorithmica*, 24(3-4):243–269, 1999. 7

[4] A_ Bondi. van der waals volumes and radii. *The Journal of physical chemistry*, 68(3): 441–451, 1964. 4

[5] Adrian Bowyer. Computing dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981. 6

[6] Thanh-Tung Cao, Ashwin Nanjappa, Mingcen Gao, and Tiow-Seng Tan. A gpu accelerated algorithm for 3d delaunay triangulation. In *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 47–54. ACM, 2014. 8, 16

[7] CGAL. Library. `www.cgal.org`. 19

[8] Paolo Cignoni, Claudio Montani, Raffaele Perego, and Roberto Scopigno. Parallel 3d delaunay triangulation. In *Computer Graphics Forum*, volume 12, pages 129–142. Citeseer, 1993. 8

[9] Paolo Cignoni, Claudio Montani, and Roberto Scopigno. Dewall: A fast divide and conquer delaunay triangulation algorithm in e d. *Computer-Aided Design*, 30(5):333–341, 1998. 7

[10] Herbert Edelsbrunner and Ernst Peter Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics (TOG)*, 9 (1):66–104, 1990. 16

[11] Herbert Edelsbrunner and Nimish R Shah. Incremental topological flipping works for regular triangulations. *Algorithmica*, 15(3):223–241, 1996. 6

[12] Dan Halperin and Mark H Overmars. Spheres, molecules, and hidden surface removal. In *Proceedings of the tenth annual symposium on Computational geometry*, pages 113–122. ACM, 1994. 4

[13] Talha Bin Masood, Sankaran Sandhya, Nagasuma Chandra, and Vijay Natarajan. Chexvis: a tool for molecular channel extraction and visualization. *BMC bioinformatics*, 16(1):1–19, 2015. 1, 2

[14] NVIDIA. Cuda programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/. 17, 18

[15] Cuda Programming. Blog. http://cuda-programming.blogspot.in/2013/02/texture-memory-in-cuda-what-is-texture.html, . 18

[16] Cuda Programming. Blog. https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/, . 17

[17] GS Raghavendra. Identification and quantification of important voids and pockets in proteins. 2013. 2

[18] FM Richards and T Richmond. Area,volumes,packing and protein structure. *Molecular Interactions and Activity in Proteins*, 916:23, 2009. 3

[19] Johnathan Richard Shewchuk. Robust adaptive floating-point geometric predicates. In *Proceedings of the twelfth annual symposium on Computational geometry*, pages 141–150. ACM, 1996. 16

[20] David F Watson. Computing the n-dimensional delaunay tessellation with application to voronoi polytopes. *The computer journal*, 24(2):167–172, 1981. 6