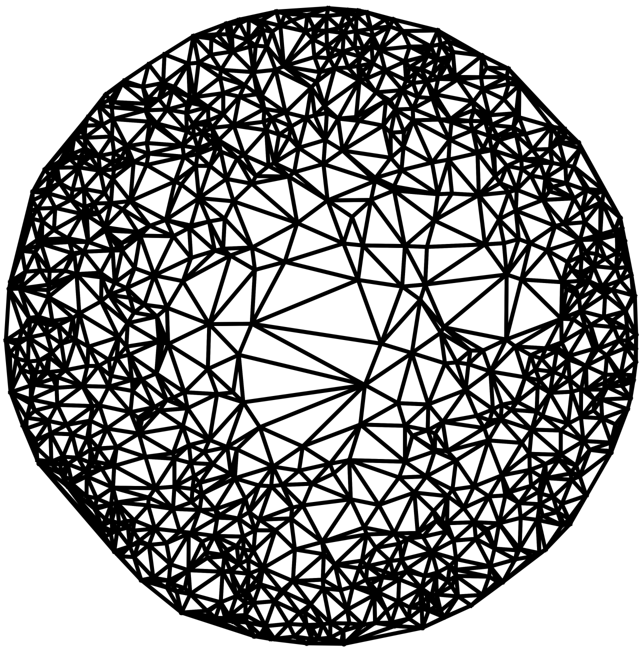
Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Delaunay Triangulations on the GPU

In partial fulfillment of MSc in High-performance Computing in the School of Mathematics



Name:	Patryk Drozd
Student ID:	24333177
Supervision:	Jose Refojo
Date of Submission:	25/09/2025
Project Code:	https://github.com/drozd324/GPU_Delaunay_Triangulation

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Declaration concerning plagiarism

I hereby declare that this thesis is my own work where appropriate citations have been given and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http:www.tcd.ie/calendar.

Name: Patryk Drozd

Student ID: 24333177

Signature:

Date: 25/09/2025

Acknowledgements

Write acknowledgements to your supervisor, classmates, friends, family, partner… anyone who supported you during the MSc. Lorem ipsum.

Contents

Abstract

Triangulations of a set of points are a very useful mathematical construction to describe properties of discretised physical systems, such as modelling terrains, cars and wind turbines which are a commonly uses for simulations such as compulational fluid dynamics or other physical properties, and even have use in video games for rendiring and visualising complex geometries. To paint a picure you may think of a triangulation given a set of points P to be a bunch of line segments connecting each point in P in a way such that the edges are non instersecting. A particulary interesting subset of triangulations are Delaunay triangulations (DT). The Delaunay triangulation is a triangulation which maximises all angles in each triangle of the triangulation. Mathematically this gives us an interesting optimization problem which leads to some rich mathematical properties, at least in 2 dimensions, and for the applied size we have a good way to discretize space for the case of simulations with the aid of methods such as Finite Element and Finite Volume methods. Delaunay triangulatinos in particular are a good candiate for these numerical methods as they provide us with fat triangles, as opposed to skinny triangles, which can serve as good elements in the Finite Element method as they tend to improve accuracy [1].

There are many algorithms which compute Delaunay triangulations (cite some overview paper), however alot of them use the the operation of 'flipping' or originally called an 'exchange' [2]. This is a fundamental property of moving through triangulations of a set of points to with the goal of optaining the optimal Delaunay triangulation. This flipping operation involves a configuration of two triangles sharing an edge, forming a quadrilateral with its boundary. The shared egde between these two triangles will be swapped or flipped from the two points at its end to the other two points on the quadrilateral. The original agorithm motivated by ([2]) is hinted to be us this flipping operation to iterate through different triangulations and eventually arrive at the Delaunay trianglution which we desire.

With the flippig operation being at the core of the algorithm, we can notice that is has the possibility of being parallelized. This is desirable as problems which commonly use the DT are run with large datasets and can benefit from the highly parallelisable nature of this algorithm. If we wish to parallize this idea, and start with some initial triangulation, conflicts would only occur if we chose to flip a configuration of triangles which share a triangle. With some care, this is an avoidable situation leads to a highly scalable algorithm. In our case the hardware of choice will be the GPU which is designed with the SIMT model which is particularly well suited for this algorithm as we are mostly performing the same operations in each iteration of the algorithm in parallel.

The goal of this project was to explore the Delaunay triangulations through both serial and parallel algorithms with the goal of presenting a easy to understand, sufficiently complex parallel algorthm designed with with Nvidia's CUDA programming model for running software on their GPUs.

1. Delaunay triangulations

In this section I aim to introduce triangulations and Delaunay triangulations from a mathematical perspective with the foresight to help present the motivation and inspiration for the key algorithms used in this project. For the entirety of this project we only focus on 2 dimensional Delaunay triangulations.

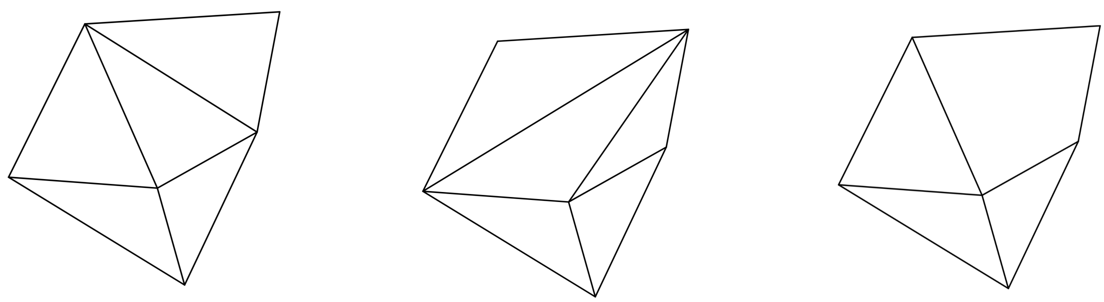
In order to introduce indroduce the Delaunay Traingulation we first must define what we mean mean by a triangultion. In order to create a triangualtion we need a set of points which will make up the vertices of the triangles. But first we want to clarify a possible ambiguity about edges.

Definition 1.1: For a point set P, the term edge is used to indicate any segment that includes precisely two points of S at its endpoints. [3].

Alterantively we could say an edge doesnt contain its enpoints which could be more useful in different contexts. But now we define the triangulation.

Definition 1.2: A *triangulation* of a planar point set P is a subdivision of the plane determined by a maximal set of noncrossing edges whose vertex set is P [3].

This is a somewhat technical but precise definition. The most imortant point in Definition 1.2 is that it is a *maximal* set of noncrossing egdes which for us means that we will not have any other shapes than triangles int this structure.



(a) (b) (c)

Figure 1: Examples of two traingulations (a) (b) on the same set of points. In (c) an illustration of a non maximal set of edges.

A useful fact about triangulations is that we can know how many triangles our triangulation will contain if given a set of points and its convex hull. For our purposes the convex hull will allways be a set which will covering a set of points, in our case the points in our triangulation. This will be usefull when we will be storing triangles as we will allways know the number of triangles that will be created.

Theorem 1.1: Let P be a set of n points in the plane, not all collinear, and let k denote the number of points in P that lie on the boundary of the convex hull of P. Then any triangulation of P has $2n - 2 - k$ triangles and $3n - 3 - k$ edges. [4]

A key feature of all of the Delaunay triangulation theorems we will be considering that no three points from the set of points P which will make up our triangulation will lie on a line and alse that no 4 points like on a circle. Motivation for this defintion will become more apparent in Theorem 1.2 and following. Definition 1.3 lets us imagine that our points are distributed randomly enough so that our algorithms will work with no degeneracies appearing. This leads us to the following definition.

Definition 1.3: A set of points P is in *general position* if no 3 points in P are colinear and that no 4 points are cocircular.

From this point onwards we will allways assume that the point set P from which we obtain our triangulation will be in *general position*. This is neccesary for the definitions and theorems we will define.

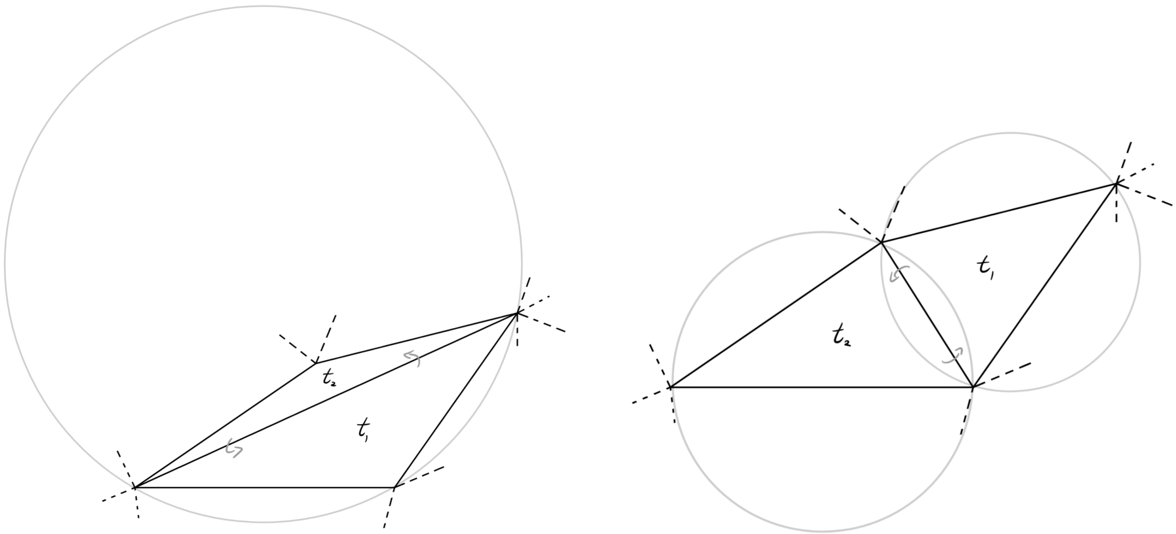
In order to define a Delaunay triangulation we would like to establish the motivation for the definition with another, preliminary definition. A Delaunay triangulation is a type of triangulation which in a sense maximizes smallest angles in a triangulation T. This idea is formalized by defining an *angle sequence* $(\alpha_1, \alpha_2, ..., \alpha_{3n})$ of T which is an ordered list of all angles of T sorted from the smallest to largest. With angle sequences we can now compare two triangulations to eachother. We can say for two triangulations T_1 and T_2 we write $T_1 > T_2$ (T_1 is fatter than T_2) if the angle sequence of T_1 is lexographically greater than T_2. Now we can compare triangulations. And by defining Definition 1.4 are able to define a *Delaunay triangulation.*

Definition 1.4: Let e be an edge of a triangulation T_1 , and let Q be the quadrilateral in T_1 formed by the two triangles having e as their common edge. If Q is convex, let T_2 be the triangulation after flipping edge e in T_1. We say e is a *legal edge* if $T_1 \geq T_2$ and e is an *illegal edge* if $T_1 < T_2$ [3]

Definition 1.5: For a point set P, a *Delaunay triangulation* of P is a triangulation that only has legal edges. [3]

As per Definition 1.5, Delaunay triangulations wish to only contain legal edges and this provides us with a "nice" triangluation with fat triangles.

Theorem 1.2 *(Empty Circle Property)*: Let P be a point set in general position. A triangulation T is a Delaunay triangulation if and only if no point from P is in the interior of any circumcircle of a triangle of T. [3]



(a) (b)

Figure 2: Demonstration of the flipping operation. In (a) A configuration that needs to be flipped illustrated by the circumcircle of t_1 containg the auxillary point of t_2 in its interior. In (b) configuration (a) which has been flipped and no longer needs to be flipped as illustrated by the both circumcirles of t_1 and t_2.

Theorem 1.2 is the key ingredient in the the Delaunay triangulation algorithms we are going to use. This is because instead of having to compare angles, as would be demanded by Definition 1.5, we are allowed to only perform a computation, involving finding a circumcicle and performing one comparison which would involve determining whether the point not shared by triangles circumcircle is contained inside the circumcircle or not. Algorithms such as initially intruduced by Lawson [5] exist which do focus on angle comparisons but are not preferred as they do not introduce desired locality and are more complex.

And finally we present the theorem which guarantees that we will eventually arrive at our desired Delaunay triangluation by stating that we can travel across all possible triangultions of our point set P by using the fliping operation.

Theorem 1.3 *(Lawson)*: Given any two triangulations of a set of points P, T_1 and T_2, there exist a finite sequence of exchanges (flips) by which T_1 can be transformed to T_2. [2]

2. The GPU

The Graphical Processing Unit (GPU) is a type of hardware accelerator originally used to significantly improve running video rendering tasks for example in video games through visualizing the two or three dimensional environments the player would be interacting with or rendering vidoes in movies after the addition of visual effects. Many different hardware accelerators have been tried and tested for more general use, like Intels Xeon Phis, however the more purpose oriented GPU has prevailed in the market and in performance mainly lead by Nvidia in prevoius years, Today, the GPU has gained a more general purporse status with the rise of General Purpose GPU (GPGPU) programming as more and more people have noticed that GPUs are very useful as a general hardware accelerator.

The traditional CPU (based on the Von Neumann architecture) which is built to perform *serial* tasks , the CPU is built to be a general purpose hardware for performing all tasks a user would demand from the computer. In contrast the GPU can't run alone and must be used in conjuction to the CPU. The CPU sends compute intructions for the GPU to perform and data is commonly passes between the CPU and GPU.

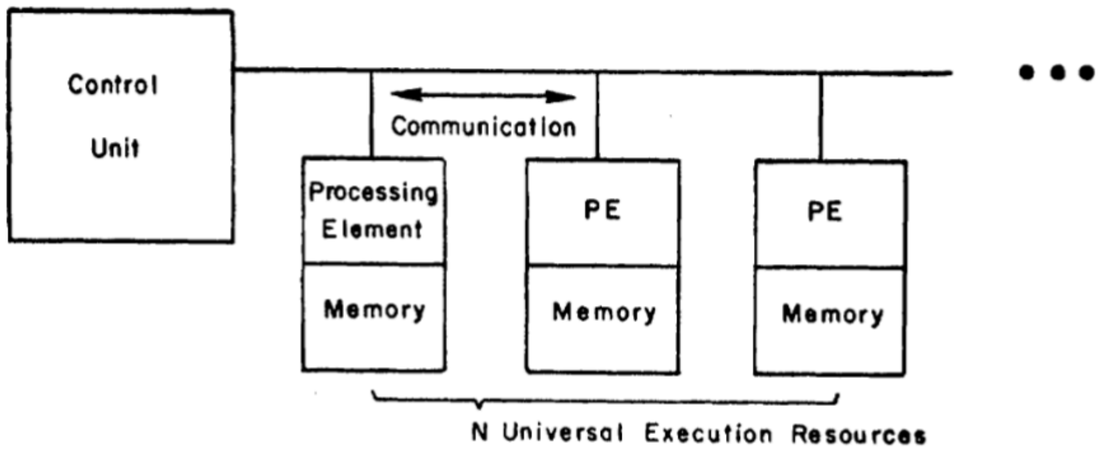


Figure 3: The *Single Instruction Multiple Threads (SIMT)* classification, originally known as an *Array Processor* as illustriated by Michael J. Flynn [6]. The control unit communicates instructions to the N processing element with each processing element having its own memory.

What makes the GPU increadibly usefull in certain usecases (like the one of this report) is its architecture which is build to enable massively parallelisable tasks. In Flynn's Taxonomy [6], the GPUs architecture is based a subcategory of the Single Instruction Multiple Data (SIMD) classification known as Single Instruction Multiple Threads (SIMT) also known as an Array Processor. The SIMD classification allows for many proccessing units to perform the same tasks on a shared dataset with the SIMT classification additionally allowing for each processing unit having its own memory allowing for more diverse proccessing of data.

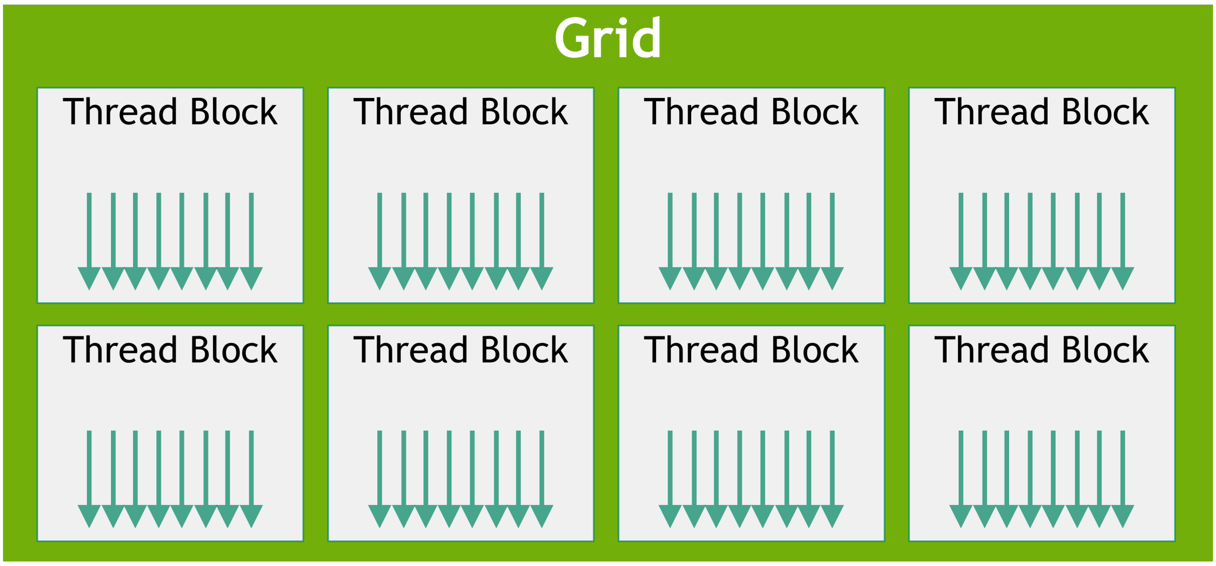


Figure 4: An illustration of the structure of the GPU programming model. As the lowest compute instrcution we have a thread block consisting of a number threads ≤ 1024. The thread blocks are contained in a grid. [7]

Nvida's GPUs take the SIMT model and further develop it. There are three core abstractions which allow Nvidia's GPU model to be succesfull; a hierarchy of thread groups, shared memories and synchronization [7]. The threads, which represent the a theoretical proccesses which encode programmed instrcutions, are launched together in groups of 32 known as *warps*. This is the smallest unit of instructions that is executed on the GPU. The threads are further grouped into *thread blocks* which are used as a way of organising the shared memory to be used by each thread in this thread block. And once more the *thread blocks* grouped into a *grid*.

3. Algorithms

In this section we focus on two types of algorithms, serial and parallel, but with a major focus on the parallel algorithm. Commonly algorithms are first developed with a serialized verion and only later optimized into parallelized versions if possible. This is how I will be presenting my chosen Delanay Triangulation (DT) algorithms in order to portray a chronolgical development of ideas used in all algorithms. And so we first begin by explaining the chosen serial verion of the DT algorithm.

3.1. Serial

The simplest type of DT algorithm can be stated as follows in Algorithm 1

Algorithm 1: Lawson Flip algorithm

Let P be a point set in general position. Initialize T as any trianulation of P. If T has an illegal edge, flip the edge and make it legal. Continue flipping illegal edges, moving through the flip graph of P in any order, until no more illegal edges remain. [3]

This algorithm presents with a bit of ambiguity however I believe its a good algorithm to keep in mind when progressing to more complex algorithms as it presents the most important feature in a DT algorithm, that is, checking if an edge in the triangulation is legal, and if its not, we flip it. Most DT algorithms take this core concept and build a more optimized verions of it with as Algorithm 1 has a complexity of $O(n^2)$ [8].

The next best serial algorithm commonly presented by popular textbooks [4], [9] is the *randomized incrimental point insertion* Algorithm 2. When implemented properly this algorithm should have a complexity of $O(n \log(n))$ [4]. This algorithm is favoured for its relative low complexity and ease of implementaion. The construction this algorithm is a bit mathematically involved however the motivation behind the construction of the algorithm is to perform point instertions, and after each point insertion we perform necessary flips to transform the current triangulation into a DT. This in turn reduces the number of flips we need to perform and this is reflected in the runtume complexity.

Algorithm 2: Randomized incremental point insertion

```
Data: point set P
Out: Delaunay triangulation T
    Initialize T with a triangle enclosing all points in P
    Compute a random permutation of P
    for p ∈ P
    Insert p into the triangle t containing it
    for each edge e ∈ t
        LegalizeEdge(e, t)
        return T
```

A signficant part of this algorithmis the FlipEdge function in Algorithm 3. This function performs the necessary flips, both number of and on the correct edges, for the triangulation in the current iteration of point insertion to become a DT.

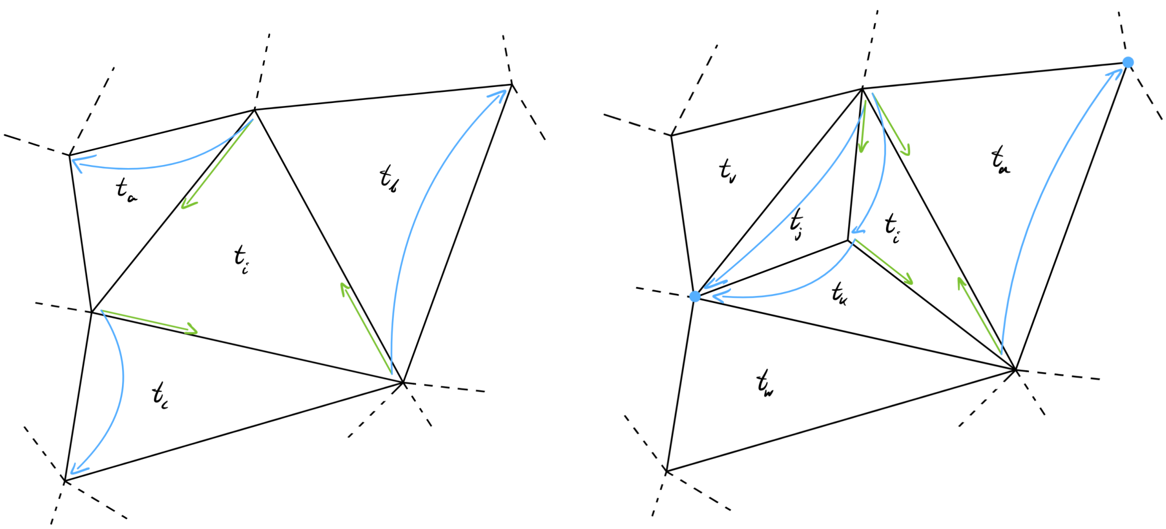
Algorithm 3: LegalizeEdge

```
Data: egde e, triangle t_a
if e is illegal
    flip with triangle t_b across edge e
    let e_1, e_2 be the outward facing egdes of t_b
    LegalizeEdge(e_1, t_b)
    LegalizeEdge(e_2, t_b)
```

The contrustions neccesary to explain wyh the *LegalizeEdge* routine created a DT is again slightly mathematically involved but is discussed in [4]. In the following sections we will discuss the point insertion and flipping steps in more detail.

3.1.1. Point insertion

Point instertion procedure goes as follows. An initial triangulation is neccesary to begin advance in the point insertion procedure. This is commonly done by adding 3 extra points to our triangulation from which we will construct a *supertriangle* which will contain all of the point in the set we wish to construct the DT. These extra 3 points will later be removed. In our triagulation if there is a point not yet insterted we choose to use it to split the existing triangle in which this point lies in into 3 new triangles. This process is repreated untill no more points are left to insert. The point insertion step would be followed by the *LegaiseEdge* procedure. Figure 5 illustrates this process.



(a) Before insertion. (b) After insertion.

Figure 5: An illustration of the point insertion in step 4 of Algorithm 2. In figure (a) the center most triangle t_i will be then triangle in which a point will be chosen for insertion. Triangle t_i knows its neighbours across each edge represented by the green arrows and knows the points opposite each of these edges. After the point it inserted (b), t_i is moved and two new triangles t_j, t_k are created to accomodate the new point. Each new trianlge t_i, t_j, t_k can be fully constructed from the previously existing t_i and each neighbour of t_i in (a) has its neighbours updated to reflect the insertion. The neighbouring triangles opposite points are updated by accesing the opposite point across the edge of the neighbouring triangle and obtaining the index of the edge which has the triangle currently being split. The index of the opposite point will allways be 0 by construction. The neighbouring triangle is also updated similarly but with the appropriate index which will be the one of the triangle who's modifying the neighbouring triangle.

It might be nice to see results from just running the point insertion algorthim by itself, whithout the flipping which would take place inbetween which will be further explored in the next section. In Figure 6 we see the result after the super triangle points and their corresponding triangles have been removed. It is good to note that the point insertion algorithm is in general a triangluation algorithm.

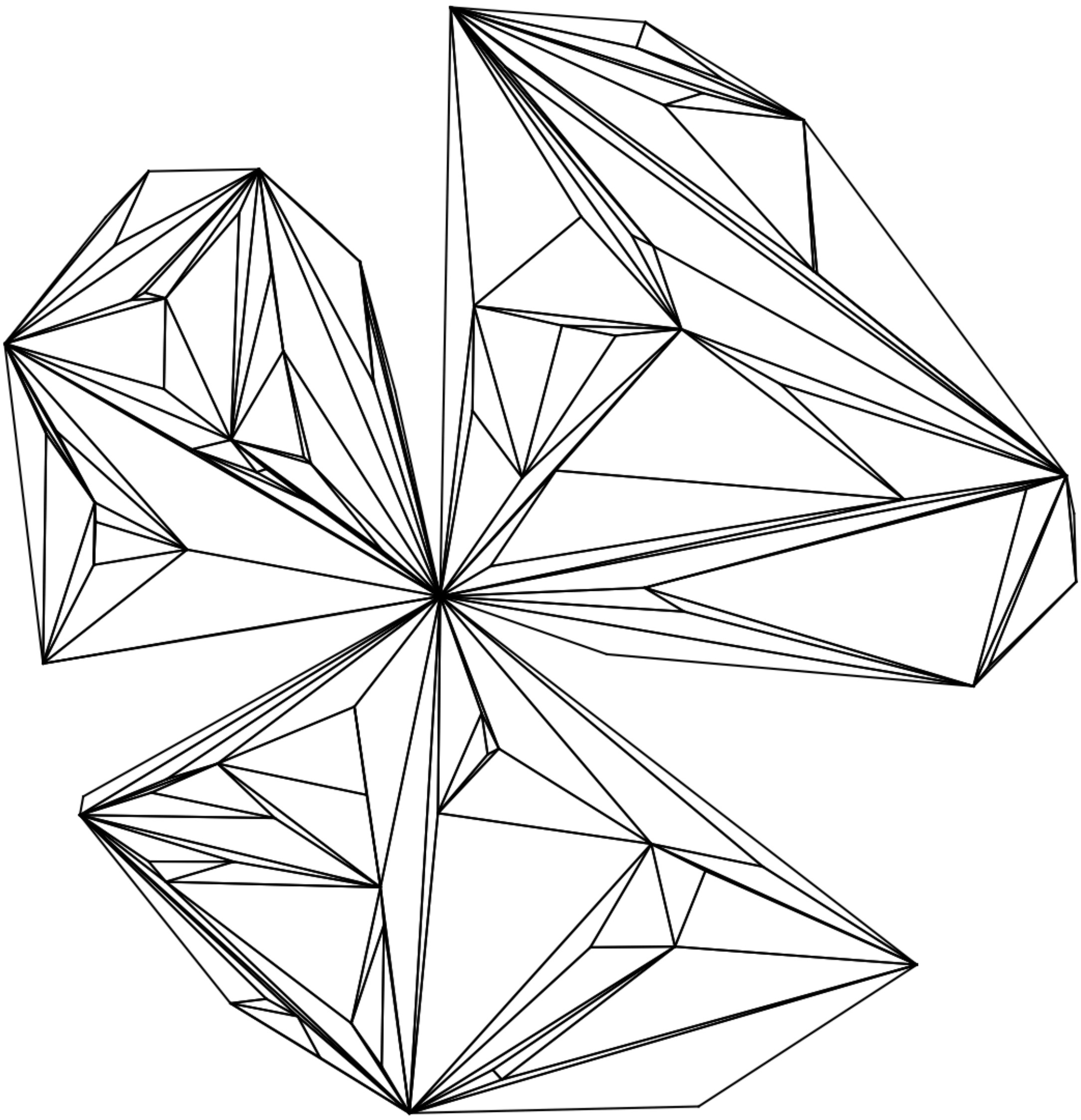
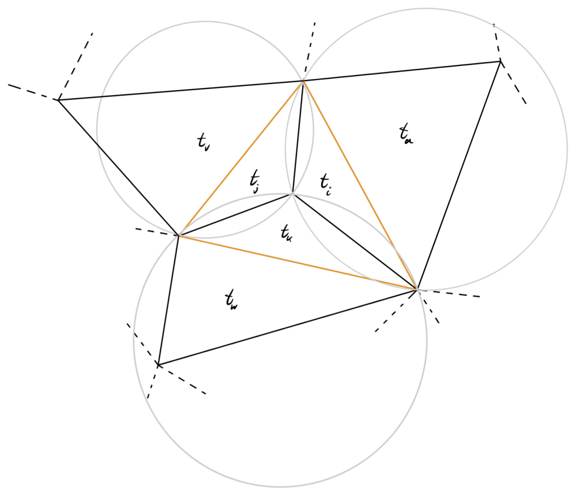


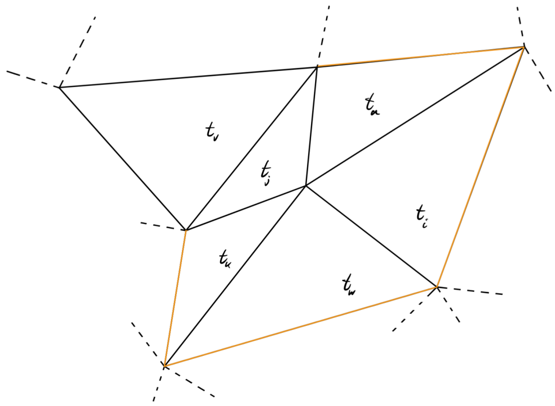
Figure 6: Output from only running the point insertion triangulation algorithm. The additional points added to form the super triangle and triangles containing these points are removed from this uniform distribution of points on a disk.

3.1.2. Flipping

Once a point insertion step is complete, appropriate flipping operations are then performed. Figure 7 illustrates this procedure. One can observe that the new edges introduced by the point insertion do not need to be flipped as they their circumcircles will not contain the points opposite the edge by construction [10] and also would interfere with other triangles if flipped as the configurations are not convex. New edges are chosen to be ones which have not been previously flipped surrounding the point insertion and only need to be checked once.



(a) Before flipping.



(b) After flipping.

Figure 7: Illustrating the flipping operation. In figure (a), point r has just been inserted and the orange edges are have been marked to be checked for flipping. Two of these end edges end up being flipped in (b). The edges inside would not qualify for flipping as any quadrilateral would not form a convex region. In order to perform the edge flipping algorithm we choose to construct the two new triangles which would form after the edge flip and the overwrite the two triangles which should no longer exist. A useful construction for ease of implementation and readability is to create a temporary *quad* data structure which contains the necessary information for constructing the new triangles. The existing edge can be thought of a being rotated counter clockwise which lets us know where the indexes of the previous triangles are being overwritten to in the array of *tri* structures in later described in Listing 1. Most of the triangles can be constructed internally but the neighbouring triangles also need to have their neighour information and points are opposite the neigbouring edge updated.

3.1.3. Implementation

The implementaion was written in C++ and was not written with a large amount of object oriented programming (OOP) techniques for an gentler transition to a CUDA implementaion as CUDA heavily relies on pointer semantics and does not support some of the more convenient OOP feautres. However as CUDA does support OOP features on the host side so the I chose to write a *Delaunay* class which holds most of the important features of the computation as methods which are exectued in the constructor of the *Delaunay* object.

3.1.4. Analysis

The analysis in this section will be brief but I hope succint as the majority of the work done was involved in the paralleliezd verions of this algorithm showcased in the following sections.

In Figure 8 below we can observe the time complexity of the serial algorithm. This algorithm can theoretically achieve a complexity of $O(n \log(n))$ however my naive implementaion does not achieve this and we have a $O(n^2)$ scaling as seen by the straight line in the log plot. Even though this is not the result I have hoped for, this is still a uselfull piece of code to compare the future GPU implementaion with. I believe that a $O(n \log(n))$ complexity can be achived by using a directed acyclic graph structure (DAG) for faster memory access in finding in which triangles points are contained in.

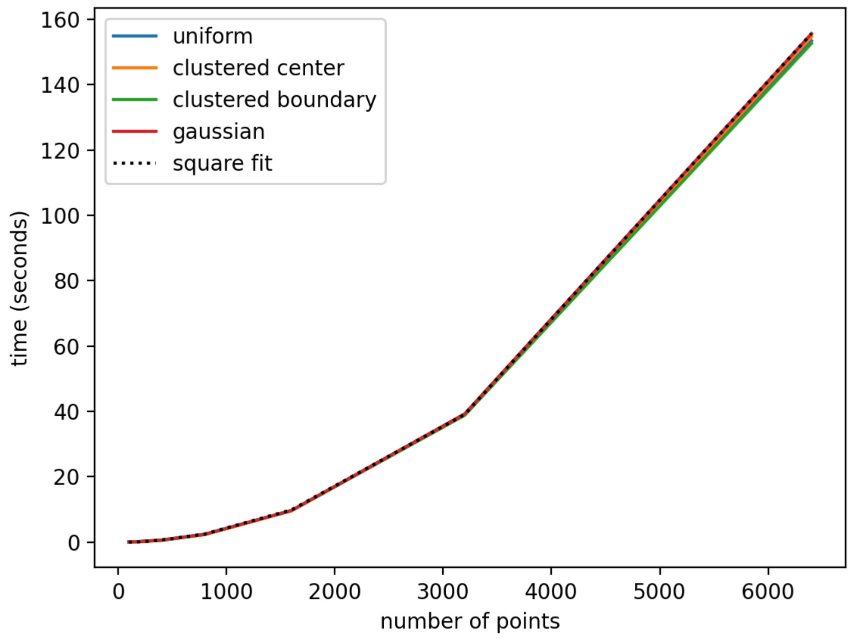


Figure 8: Plot showing the amount of time it took serial code to run with respect to the number of points in the triangulation. This is a loglog plot which shows us the algorithm has a complextiy of $O(n^2)$

3.2. Parallel

The parallelization of the DT is conceptually not very different than its serial counterpart. We will be considering only parallelization with a GPU here which lends itself to algorithms which are created with a GPUs architecture in mind. This means that accesing data will be largely done by accessing global arrays which all threads of execution have access to. Methods akin to divide and conquer [11] would be useful if we consider multi CPU or multi GPU systems but that is not in the scope of this project but would be particulary interesing to see a multi GPU systems implementation for this algorithm made publicly available. An overview of the parallelized algorithm is in Algorithm 4 mostly adapted from [12] which is to my understanding as of this moment the fastest GPU delaunay trianglulation algorithm.

Algorithm 4: Parallel point insertion and flipping

```
Data: A point set P
Out: Delaunay Triangulation T
1  Initialize T with a triangle t enclosing all points in P
2  Initialize locations of p ∈ P to all lie in t
3  while there are p ∈ P to insert
4      for each p ∈ P do in parallel
5          choose p_t ∈ P to insert if any
6      for each t ∈ T with p_t to insert do in parallel
7          split t
8      while there are illegal edges
9          for each triangle t ∈ T do in parallel
10             mark whether it should be flipped
11         for each triangle t ∈ T in a configuration marked to flip do in parallel
12             flip t
13     update locations of p ∈ P
14 return T
```

Algorithm 4 is takes as input a point set P for the triangluation to be constructed from and return the DT from the transformed triangulation T. *(line 1)* The triangulation is initialized as a triangle enclosing all points in P by adding 3 new points to the triangulation and is constructed in a way such that all of the other points lie inside this triangle which is noted in *(line 2)*. These extra three points will later be removed. *(line 3)* Tells us to keep peforming the main work of the algorithm as long as there are points to be inserted into T. *(lines 4-5)* We pick out points in parellel which can be inserted into T by checking in which triangle each point not yet inserted, if any, is closest to the circumcenter of the triangle. This point will be inserted in the *(lines_6-7)* in which for every triangle which has a point inside it to be inserted we split the existing triangle t into 3 new triangles which all contain the inserted point p. Now in *(lines 8-12)* at this point, we have a non Delaunay mesh which needs to be transformed and so we perform neccesary flipping operations in order for this to be a DT. For each triangle we first check whether we should flip with any 3 any of its neighbours by checking if each edge is illegal. If an edge is found to be illegal the first neighbouring triangle is marked to be flipped with. Following this we check whether any triangles marked for flipping would be conlflicting with any other configuration flipping, and if so, it is discarded for this iteration of the while loop. In *(lines 11-12)* we perform the flipping operation for each triangle which wont have any conflicts. At the end of the outermost while loop in *(line 13)* we update our knowlege of where points which have not yet been instered not lie after the chages by the point insertion creating new triangles and flipping changing the triangles themselves.

Algorithm 4 exploits the most parallizable aspects of the point insertion Algorithm 2, which are the point insertion, for which only one triangle is involved in at a time, and the flipping operation, which can be parallized but some book keeping needs to be taken care of in order for confilicting flip to not be performed. With a large point set this parallelization allows for a massively alogorithm as a large number of point insertions and flips can be performed in parallel. Flipping conlficts can happen when two different configurations of neighbouring triangles want to flip and these two configurations share a triangle, as illustrated in Figure 10.

3.2.1. Constructing the super traingle

In order to be able to begin our DT algorithm, a *supertriangle* needs to be constructed. This only needs to be done only once throughout the duration of the algorithm. Two routines in this algorithm deserve to be paralellized, computing the average point and computing the largest distance between two points. Computing the average point involves calculating the total sum of all points by a reduction which is followed by a division in each coordinate by the number of points in the set. When computing the maximum distance bewteen two points a CUDA kernel is lauched which spaws a thread for each point which then compares every other point to it by calculating the distacnce between them and stores the maximum distance within the memory in each thread. Within this computation each point is compared to itself once which is concious decision since compute on the GPU is cheap and otherwise each thread would be recieving different instructions which is not friendly to the SIMT programming model on the GPU. Once these calculations are finished an atomic max opertation is performed to shared memory and then another atomic max to global memory which gives us our final value of the maximum distance. These two quantities are then used to construct a *supertriagle* which will encompass all points in the set of points we provide. The maximum distance is the radius and the average point is center to a circle which will be the incircle of an equilateral triangle which becomes our constructed *supertriagle*. Algorithm 5 outlines this process.

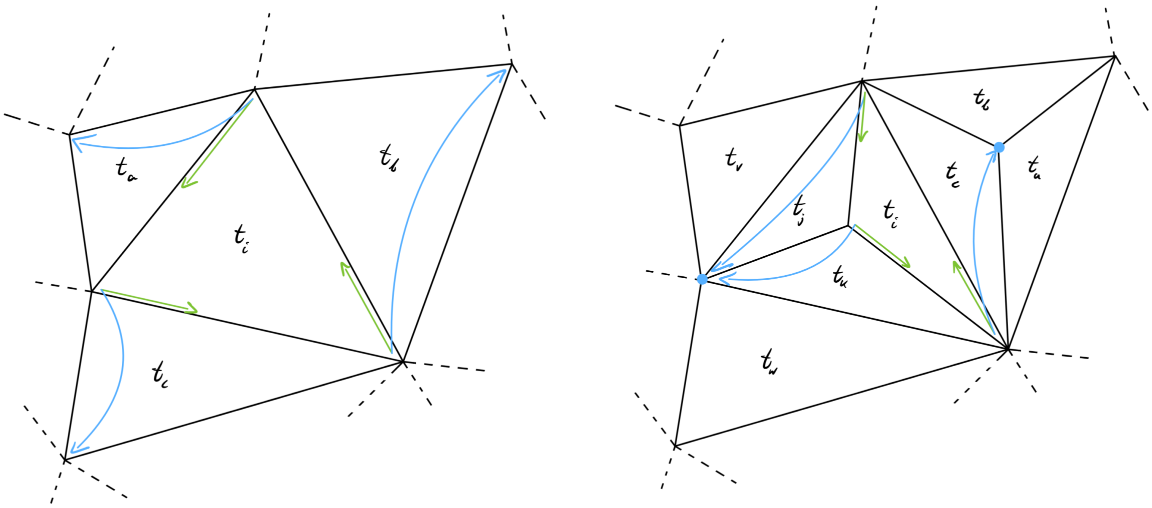
Algorithm 5: Parallel super triangle construction

Data: point set P

```
Compute the average point
Compute the largest distance bewtween two points in P
Set center to average point
Set radius to largest distance
Construct triangle from circle as incircle
```

3.2.2. Insertion

The parellel point insertion step is very well suited for parallelization. Parallel point insertion can be performed with minimal interfernce with their neighbours. This procedure is performed independantly for each triangle with a point to insert. The only complication arises in the updating of neighbouring triangles information about their newly updated neighbours and opposite points. This must be done after all new triangles have been constructed and saved to memory. Only then you can exploit the data structure and traverse the neighbouring triangle to a update the correct triangles appropirate edge.



(a) Before insertion.

(b) After insertion.

Figure 9: Parallel point insertion

Implementation

The implemenation of the parallel point insertion algorithm relies on two steps, preparation of points to be inserted and the insertion of points. If only the point insertion procedure is performed we also need to update point locations which is normally done after the flipping operations needed.

The preparation step ivolves a handful of checks or verifications to find out which point should be inserted into each triangle. In this algorithm we wish to find the most suitable point for each triangle to have inserted into it. We do this by finding out which point, which is not yet inserted into the triangulation lies in which triangle. The point closest to the circumcenter of the triangle is chosen to be inserted. Two CUDA kernels are used in this procedure, one to calculate the distances of each point to their corresponding circumcentres and another to find the minimum distance. This procedure relies on computing the distance twice as compute is cheap on GPU as opposed to copying memory of the triangle structures. In between all of these arrays which contain information about uninserted points *ptsUninsterted* are used throughout in order to not waste resources in the form of threads which would obtain instructions to do nothing. The *ptsUninsterted* array is sorted in order to launch the minimum number of threads needed. A few other kerels are used for book keeping purposes which constist of resetting certain values, for example the smallest distance bewteen two points in each triangle is set to the maximumvalue as there are atomic min opertaions performed for which this is neccesary in the next iteration of point insertion. We also keep an array which holds the indexes of triangles triangles which how hold points to be inserted which again prevents unneccesary thread lauches.

Algorithm 6: prepForInsert

```
Reset index of the point to insert in each triangle
Set counter for number of points uninserted to be 0
Writes uninserted point index to ptsUninserted
Caclualtes and writes the smallest distance to circumcenter of triangle
Finds and writes the index of point with smallelst distance to circumcenter of triangle
Resets counter of the number of poinnts to insert
Counts the number of triangles which are marked for insertion
Sorts the array triWithInsert for efficient thread launches
Resets the value of the distance of point to circumcenter in each triangle
```

Once the preperation step is completed, which makes up the majority of the compute for point insertion procedure Figure 17 we can now actually insert the points which have been pickout out. The logic is mostly consistent as in Figure 5 but needs to be adapted in order for it to be parallelized.

For the creation and rewriting involved in making the 3 new traingles stays the same except two things. First of which the locations in which the new triangles are written in need cannot be simply written to the next unwritten location in the list of triangle structs. A simple map can be created once we know how manya triangles will need to split. We use the index of each thread to identify where we will place each newly created two triangles as we still overwrite the existing triangle with one of the new triangles that it is split into. We can use the following expression to know where to start writing the two new triangles $\mathrm{nTri} + 2 * \mathrm{idx}$ where nTri represents the current number of triangles in the triangulation and idx the index of the thread.

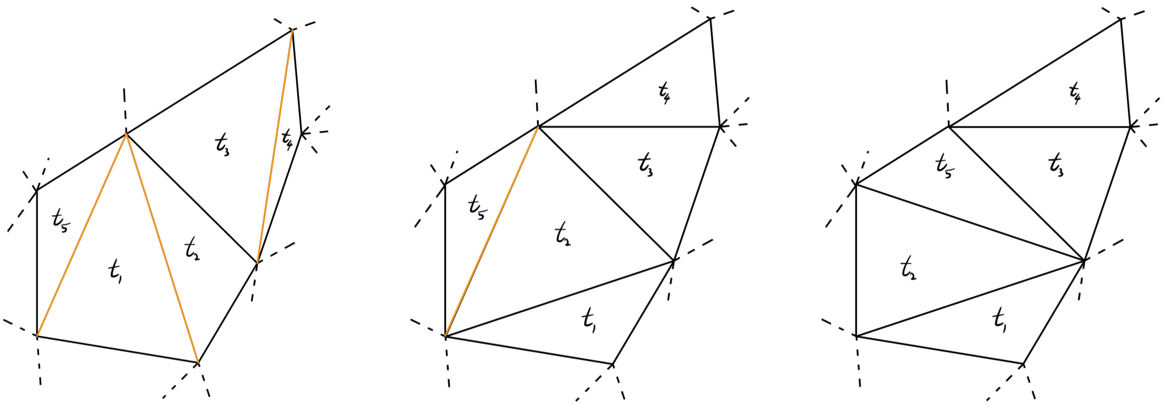
Secondly, the updating of the neighbouring triangles also need some extra care. The splitting or point insertion step is written as two CUDA kernels. One which writes the internal structure of the 3 new triangles and another kernel takes care of updating the relevant neighbours of the 3 new triangles. It is necessary to split up this procedure since if it was not split up the external neighbouring triangles could be overwritten while they are being created. The algorithm relies on the neighbouring triangles already exsiting to find the relevant neighbour to update which is done so by traversing the the split triangles counter clockwise in order to the relevant neighbouring triangle **MAKE FIGURE FOR THIS**. It is also important to note that the nTri variable, should only be updated after the parallel point insertion procedure is complete as the updating it during this process have consequences on the locations of the newly created triangles storage location.

Algorithm 7: Parallel insert

```
Insert point in marked triangles
Update neighbours
Update number of triangles and number of points inserted
Reset triWithInsert for next iteraiton
```

3.2.3. Flipping

As briefly mentiond earlier, flipping can be performed in a highly parallel manner however some book keeping needs to be taken care of. The logic within the flippig operation is split up into three main steps. The first one is the writing of triangles to be flipped each configuraion into a *Quad* Listing 2 data structure which here is mainly created for the purpose of keeping steps in the whole procedure to be non conflicing more imortantly stores relevant information about the previous state of the triangulation. This *Quad* struct will aid us in constructing the flipped cofiguartion. The the two new triagles created from the flip are the written by one kernel and appropriate neighbours are then updated in a separate kerel. Splitting the writing of the new flipped triangles is once again important as updating the neighbours relies on writing to the correct index of triangle since nighbouring triangles could also be involved in a flip. Figure 10 showcases the parallel flipping procedure.



(a) (b) (c)

Figure 10: Illustration of parallel flipping while accounting for flipping non conflicting configurations. Edges colored orange are marked for flipping. For each configuration marked for flipping by each orange edge the triangle with the smallest index will be the one performing the flipping operation, and the configuration with the smallest index (min of both indexes of triangles the configuration) will have priority to flip first in each round of parallel flipping. In the first figure (a) 3 edges are marked for flipping. Only configurations of triangles $t_1 t_2$ and $t_3 t_4$, with configuration indexes 1 and 3 respectively, will flip. Configuration t_5, t_1 with a configuration index of 1 will not flip in the first parallel flipping iteration (b) as it is not the minimum index in its configuration. (c) Showcases the final outcome of the parallel flipping.

However before the we can perform our parallel flipping we need to know which triangles need to be flipped and which triangles should be flipped in order for there to be no conflicts bewteen flips. In order to know which triangles should be flipped a kernel is launched to perform an *incircle* test on each edge of each triangle currently in the triangulation. The *incircle* test whether the point opposite each edge of each triangle is contained inside the circumcircle created by the triangle associated with the thread of computation. This test directly follows from Theorem 1.2. Following this test, some configurtaions of triangles may have been marked in a way that two configurations will share a triangle they want to flip with. In order to avoid we give each configuration of triangles a configuration index obtained by using the mininum index of both triangles and we write this to both triangles using an atomic min operation given a single triangles can be involved in more than one configuration. This is done by one CUDA kernel and is followed by another kernel which stores indexes of triangles which should perform a flipping operation into an auxillary array. Only triangles which are the smallest index of triangles which will be involved in a flip and whose neighbour and itself both still hold the same configuration index are allowed to flip in a given parallel flipping pass. Once this performing and *incircle* test and making sure none of our flips will conflict with eachother we can proceed to the parallel flipping procedure desribed previously.

Algorithm 8: Parallel flipping

```
Set array of triangles which should be flipped to −1
Perform incircle checks on all triangles and mark sucessful triagles for flipping
Check for possible flip conflicts and mark sucessful triagles for flipping
while there are configurations to flip
    Write relevant quadrilaterals
    Overwrites new triangles internal structure
    Updates neighbours information
    Perform incircle checks on all triangles and mark sucessful triagles for flipping
    Check for possible flip conflicts and mark sucessful triagles for flipping
Reset mark for flipping in tri struct
```

3.3. Data Structures

The core data structure that is needed in this algorithm is one to represent a the triangulation itself. There are a handful of different approaches to this problem inculding representing edges by the qaud edge data structure [10] however we choose to represent the triangles in our triangulation by explicit triangle structures [13] which hold neccesary information about their neighbours for the construction of the trianulation and for performing point insertion and flipping operations.

```
struct __align__(64)  Tri {
  int p[3]; // indexes of points in pts list
  int n[3]; // idx to Tri neighbours of this triangle
  int o[3]; // index in neigbouring tri of point opposite the egde

  // takes values 0 or 1 for marking if it shouldn't or should be inserted into
  int insert;
  // the index of the point to insert
  int insertPt;
  // entry for the minimum distance between point and circumcenter
  REAL insertPt_dist;
  // marks an edge to flip 0,1 or 2
  int flip;
  // mark whether this triangle should flip in the current iteration of flipping
  int flipThisIter;
  // the minimum index for both triangles which could be involved in a flip
  int configIdx;
};
```

Listing 1: Data structure needed for Point instertion algorithm. Its main features are that it holds a pointer to an array of points which will be used for the triangulation, the index of those points as ints which form this triangle, its daughter triangles which are represented as ints which belong to an array of all triangle elements and whether this triangle is used in the trianglulation constructed so far. Aligned to 64 bytes for more efficient accesing of memory.

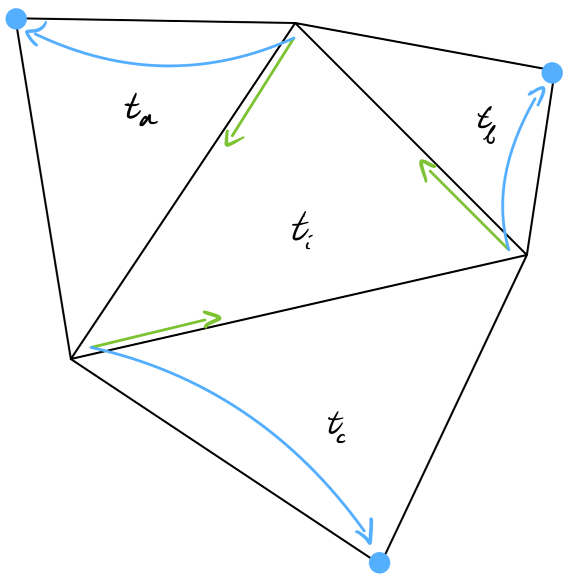


Figure 11: An illustration of the *Tri* data structures main features. We describe the triangle t_i int the figure. Oriented counter clockwise points are stored as indexes an array containing two dimensional coordinate represeting the point. The neighbours are assigned by using the right hand side of each edge using and index of the point as the start of the edge and following the edge in the CCW direction. The neighbours index will by written in the corresponding entry in the structure.

This data structure was chosen for the ease of implementation and as whenever we want to read a traigle we will be a significant amount of data about it and this locality theoreitcally helps with memory reads, as opposed to storing separate parts of date about the triangle in different structures, ie separating point and neighbour information into two different structs.

The Listing 2 below is used in the flipping step of the algorithm and is only used as an intermediate representation of the triangles which will be created and the data needed to update its neighbours

```
struct __align__(64) Quad  {
  int p[4]; // indexes of points in pts list
  int n[4]; // idx to Tri neighbours across the edge
  int o[4]; // index in neigbouring tri of point opposite the egde
};
```

Listing 2: Data structure used in the flipping algorithm. This qualrilateral data structure holds information about the intermediate state of two triangles involved in a configuration currently being flipped. This struct is used in the construction of the two new triangles created and in the updating of neighbouring triangles data. Aligned to 64 bytes for more efficient accesing of memory.

3.3.1. Analysis

In this section we will analyze and visulaize some results and which we have produces for our DT algorithm. All tests were run with a *NVIDIA GeForce RTX 3090* as the GPU alongside an *AMD Ryzen Threadripper 3960X 24-Core Processor* CPU, with the execption of some results in Figure 21.

We shall begin with some visualization of the algorithm. Figure 12 displays the raw evolution of the algorithm. We can follow the figures from left to right in alphabetical order to see the history of the procedure.

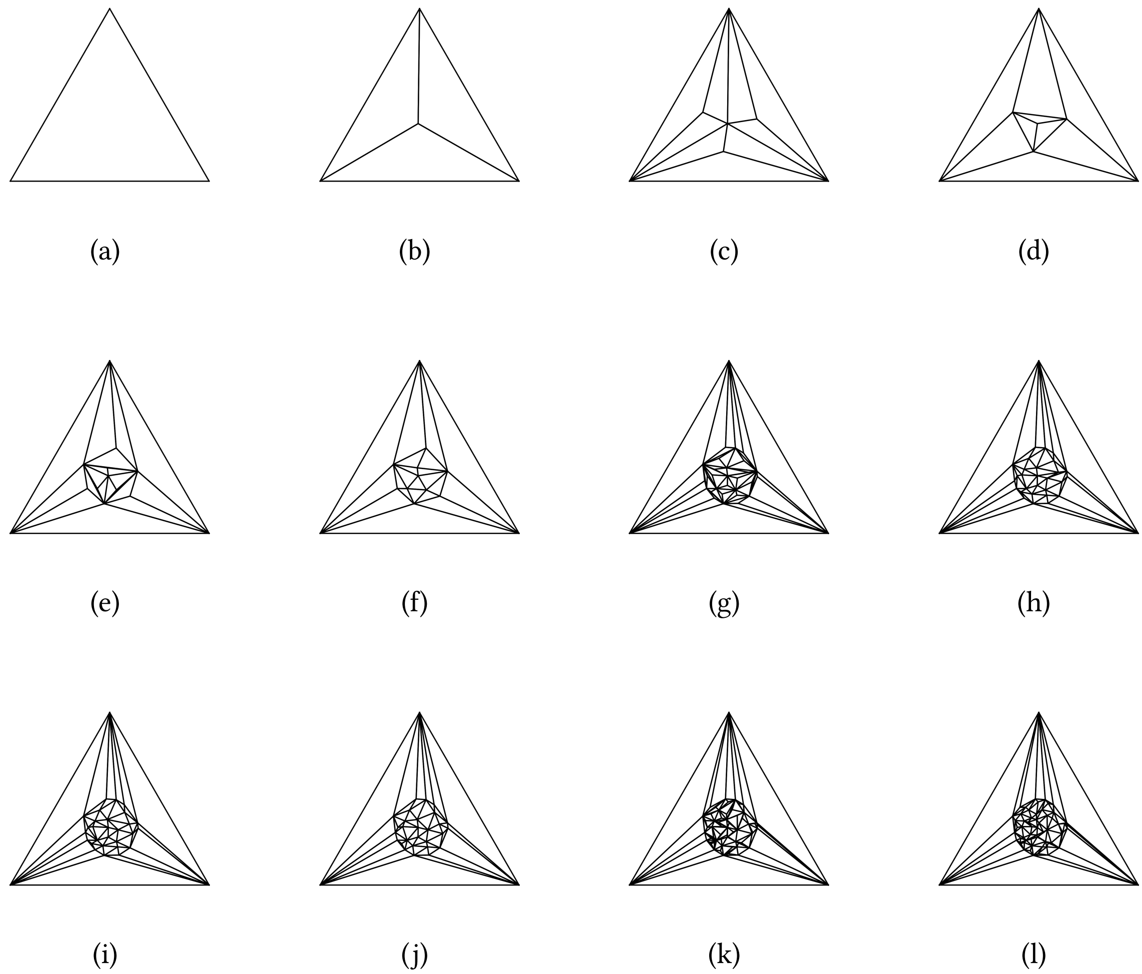


Figure 12: These figures show the history of the DT algorithm. The algorithm begins by initializing a super triangle (a) which is constructed to contain each point desired by the user. Here a uniform point distribution on a unit disk is used. In (b) and (c) a point insertion is performed and in (c) certain edges are marked for parallel flipping for which the result is displayed in (d). The algorithm proceeds in following subfigures with a series of point insertion followed by the required number of parallel flipping operations. In the final result, triangles which contain the initialized supertriangle points are removed and we are left with the desired triangulation as can be seen in Figure 22.

In Figure 13 we see the total runtime of the main compute in our algorithm, this excludes the construction of the supertriangle as it is performed only one and does not contribute to the signifiant parts of the algorithm.

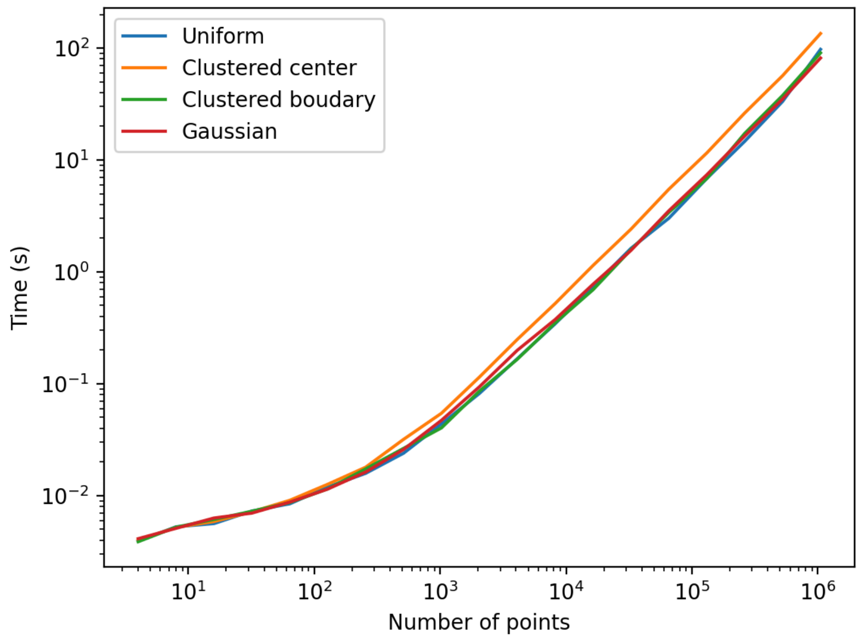


Figure 13: Plot showing the amount of time it took the GPU code to run with respect to the number of points in the triangulation. Different line colors show the code run with a different underlying point distribution.

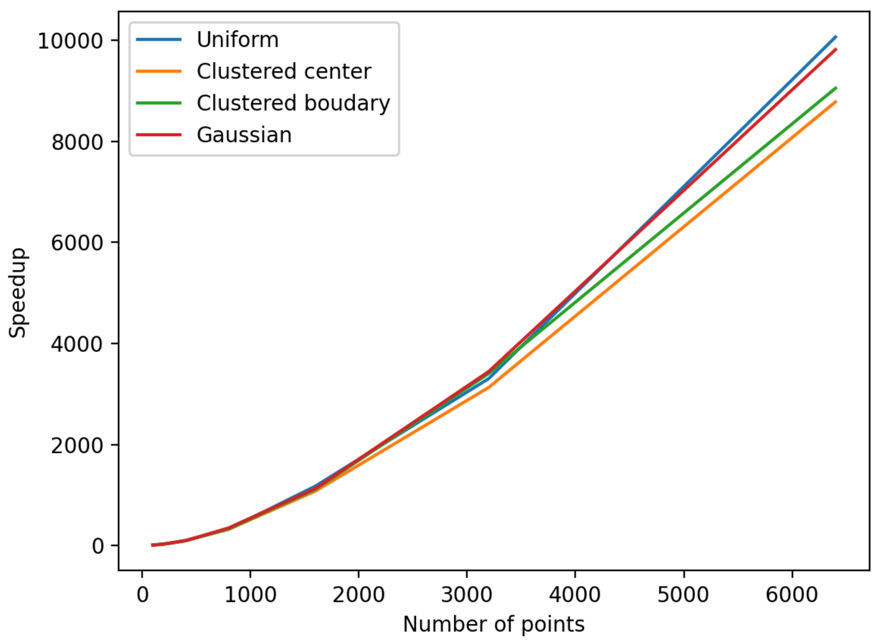
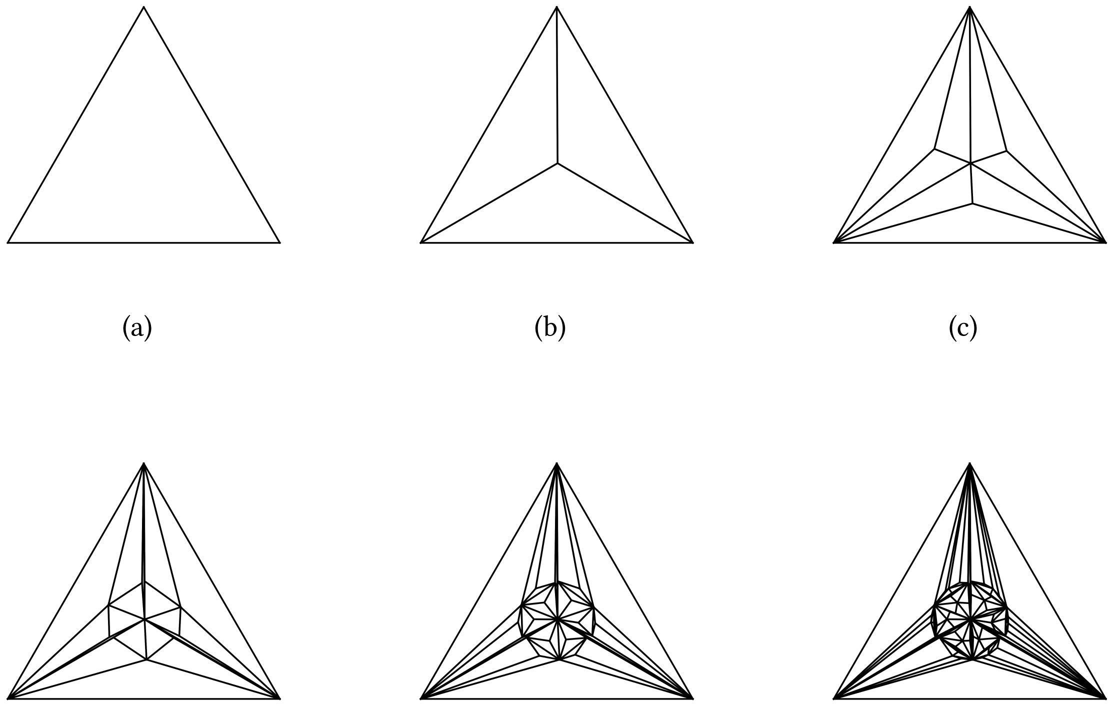


Figure 14: Plot showing speedup of the GPU code with respect to the serial implemenation of the incremental point insertion Algorithm 2. The speedup here is comparing the runtime of the serial code with for a given number of points and with the runtime of the GPU code with the same number of points. Both implementaions are run with single precision floating point arithmetic. Speedup here is defined as the ratio $\frac{\text{timeCPU}}{\text{timeGPU}}$.

Figure 14 displays the speedup by comparing the serial implementation with our GPU implemention. This comparison is quite unfair to the serial implementaion as we are not comparing the same algorithms exactly. The GPU algorithm needed to be rewritten with a deep understanding of the GPU programming model. By the end of the rewrite it is not the same algorithms started with. It is still a useful benchmark since it does show us that with a bit of work converting a simple implementation into a highly parallelized version can give immense amounts of speedup.



(a) (b) (c)

(d) (e) (f)

Figure 15: These figures show the evolution of the only the point insertion algorithm. The point insertion proceeds in alphabetical order noting the labels of each subfigure. During the computation points closest to the circumcenter of each triangle are chosen to be inserted and split each existing triangle with a point to insert. These figures use a uniform point distribution on a unit disk. This figure aids to portray the DT as the angle maximizing triangulation which lacktherof can be seen here in the last few figures with lines drawn which appear to be thick. The triangles in these figures also appear, for the most part, alot more narrow than their counterparts in Figure 12.

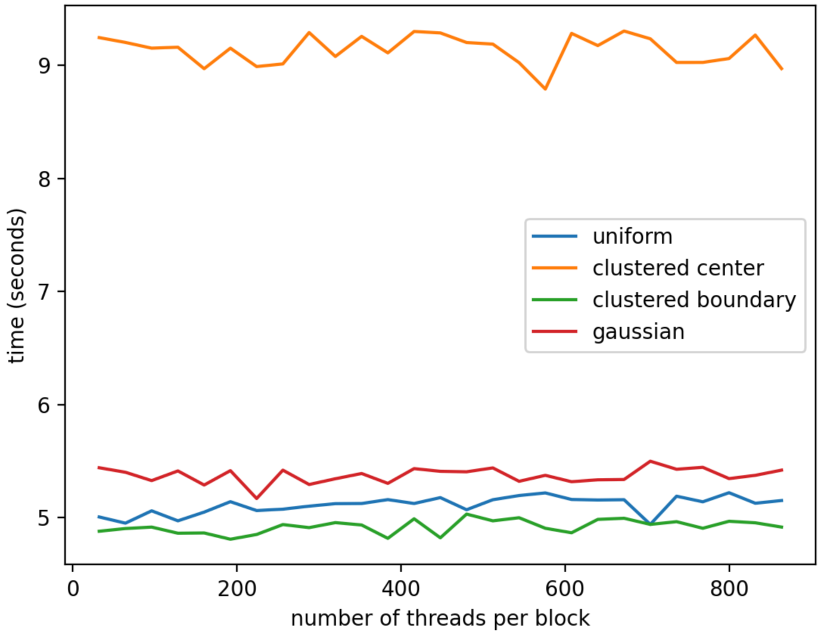


Figure 16: Showing the time it took for the GPU DT code to run with 10^5 points while varying the number of threads per block also know as the block size. This is a rather naiive way of finding the optimal number of threads per block as a better analysis **TODO** would involve logically similar block of code to have their own block size. Currently the block size doesn't rationally affect the runtime.

In order to profile the code we dice to measure how long each significant logical part of the algorithm takes to complete in each pass of insertion and flipping Figure 17. We add these values to obtain the amount of time it took to run each logical chunk over the total runtime of the algorithm. By a quick glance we can see that the flipping proceducures take up the majority of the runtime. This is due to the fact that multiple iterations of flips are performed after each point insertion. What is costly is that after the first two iterations of flips we see that there is an incredibly small amount of configurations which need to be flipped Figure 19. These are configurations which would otherwise have conflicted with other configurations or were only possible after. What really harms the speed of the algorithm is that each flipping iteration within the parallel flipping procedure takes roughly the same amount of time to process, even if it is flipping a relatively small number of configurations.

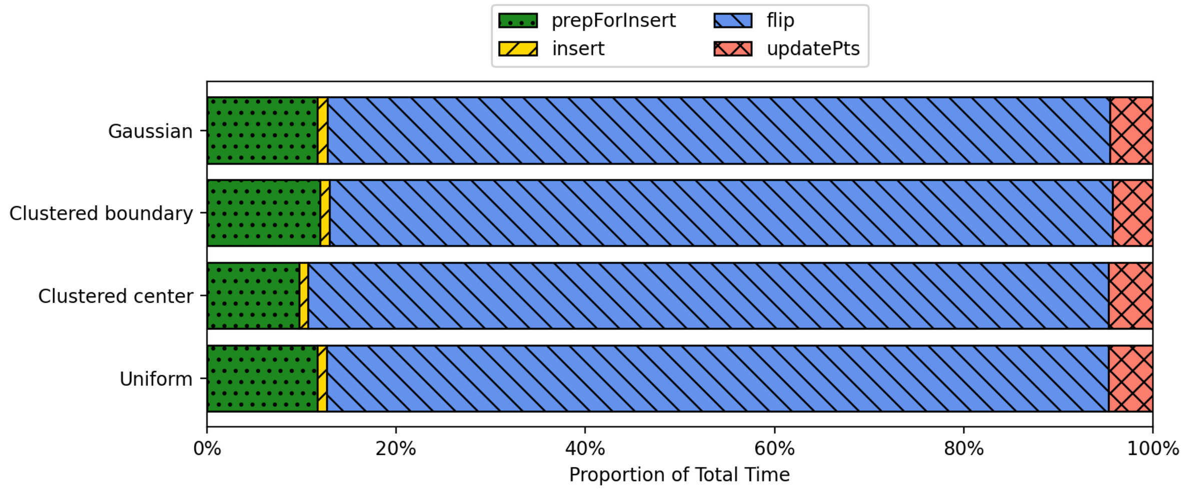


Figure 17: Showing the proportions of time each function took as a percentage of the total runtime. Each color represents a different set opteration which perform a task.

The profling in Figure 17 gives us only an impression of how the code as a whole performs in terms of time spent. This includes both the host and device runtimes in the respective function calls. Another

Depending on the desired application of this algorithm one may use this in

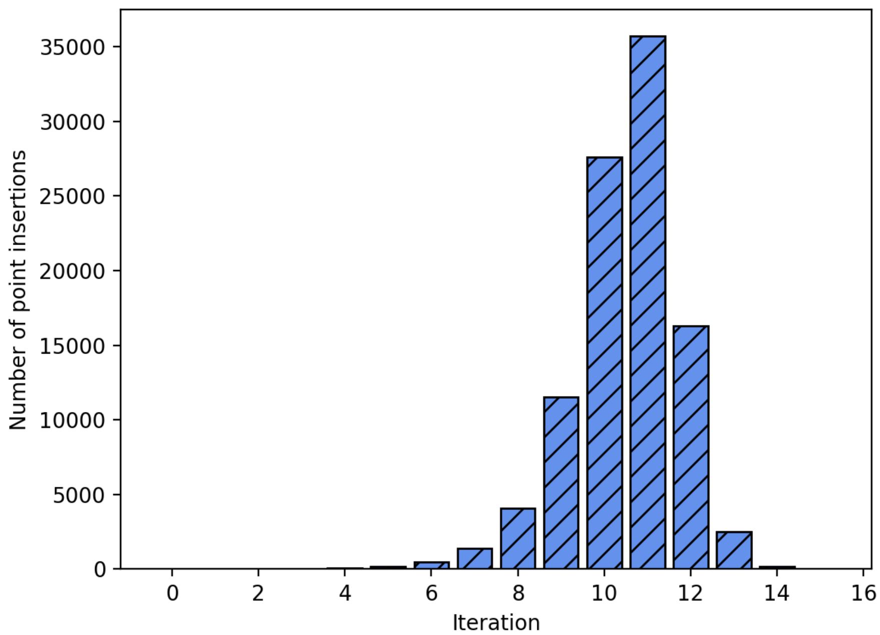


Figure 18: This figure shows the number of points inserted into the existing triangulation during each pass of the algorithm. Algorithm performed on 10^5 points and a uniform distribution of points.

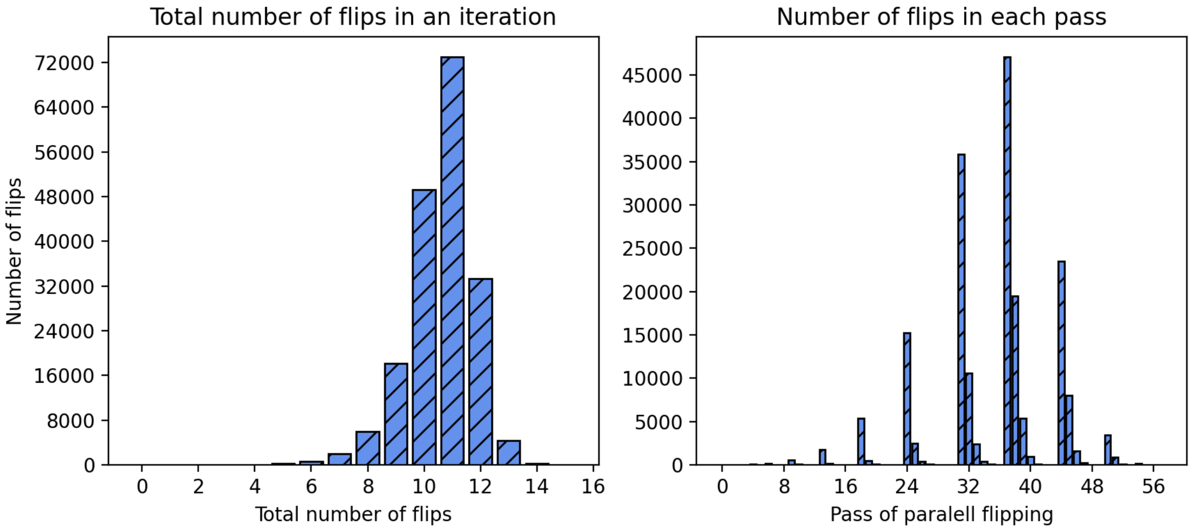


Figure 19: Figures showing numbers of flips performed during during each call to the parallel flipping function *flip()* on the left and on the right the number of flips performed during each pass of parallel flipping performed within the *flip()* function. Algorithm performed on 10^5 points and a uniform distribution of points.

When reaching sizes of around 10^6 points, our DT algorithm begins to get stuck in flipping opertions. This is due to the single precision floating point arithmetic used. This flaw is amended by tracking if the algorithm is repeating the same flipping operations but this leaves us without being certain that what we create is indeed a Delaunay triangulation. Hence the need for double precision arithmetic. Other approches is adaptive methods to change the precision of the incircle checks when needed [12]. We implemented a way of changing the precision of the whole algorithm which allows the user to choose bewteen calculating in single or double precision. In Figure 20 we compare the runtime of single and double precision codes with the number of points which construct the triangulation. Unsuprisingly double precsion arithmetic takes longer than single precision however it could be advantageous to run with double with a larger number of points if the precise nature of the Delaunay triangulation is desired.

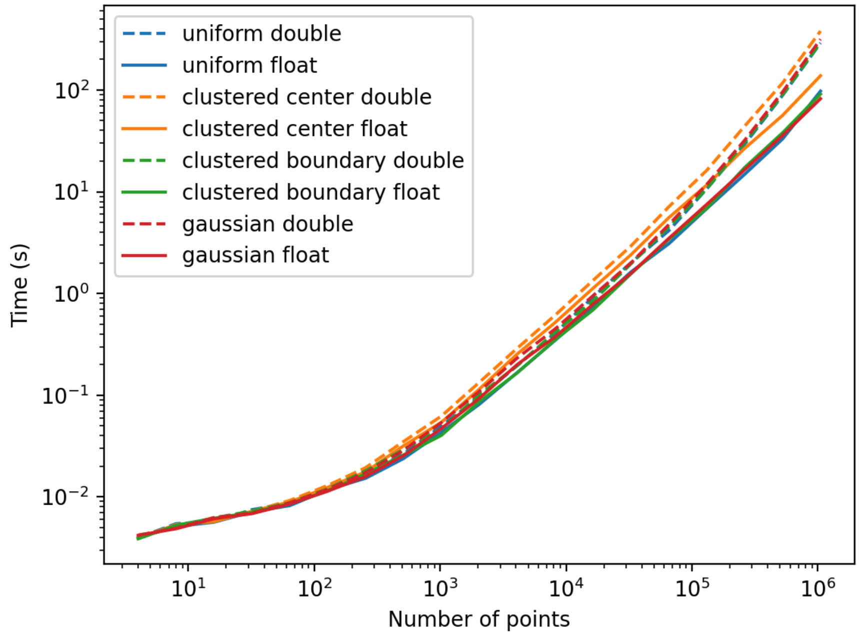


Figure 20: This figure displays the difference in runtime between the same GPU code in single and in double precision. Solid lines show the run time with their respective point distribution in single precision and dashed lines of the same color show the run time of the same distribtion but in double precision instead.

When comparing how scalable an algorithm is in the world of parallel CPU programming, with concepts such as strong and weak scaling, there is no standardized way of doing so for a single GPU code. The strong and weak scaling approches of analysis can be usefull for GPUs when we have a multi GPU code however we have not created a multi GPU code. The next best approach we found, used by [12], is to instead compare run time on different GPUs. Alongside the run time we also calculate the a normalized run time defined by the run time divided by the product of the numer of cores and the base clock frequency of the respective GPU. The normalized runtime is a reasonable metric to consider as the divisor is a measure of how often a computation is performed.

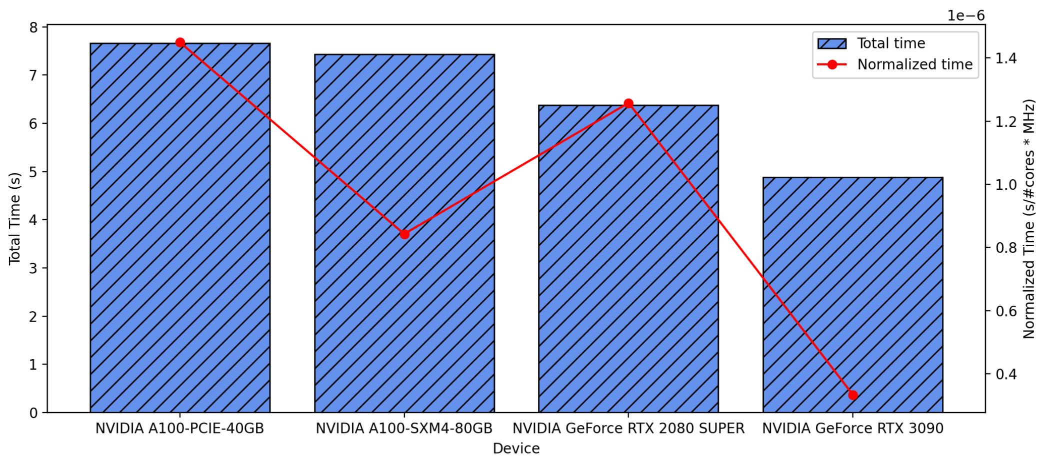


Figure 21: A comparison of the algorithm running on a variety of NVIDIA GPUs. This benchmark is performed by averaging 5 runs of the DT algorithm on a unfiform set of 10^5 points.

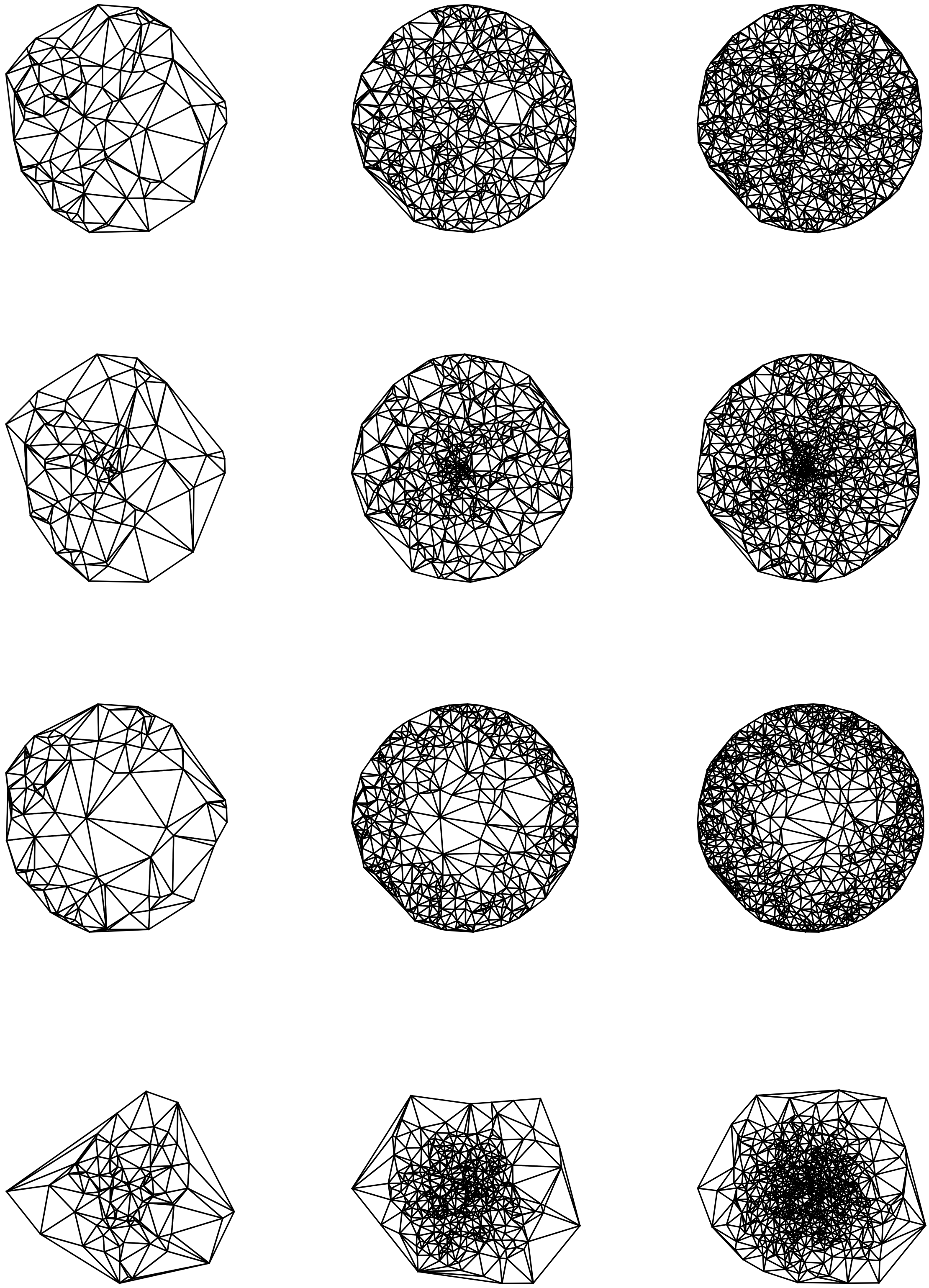


Figure 22: Visualisations of Delaunay triangluations of various point distributions. The grid should be read as follows. Along the horizontal the number of points involved increases gradually and with 100, 500, 1000 points in the the first second and third column respectively. In each row we draw from different point distributions. The rows draw from a uniform unit disk distribution, a distribution on a disk with points clustered in the center, a distribution on a disk with points clustered near the boundary and a gaussiann distribution with mean 0 and variance 1, in rows 1, 2, 3 and 4 respectively.

3.4. User Guide

A quick demo of how to use the object is given below.

```
#include "delaunay.h"
#include "point.h"
#include "ran.h"

int main(int argc, char *argv[]) {

    int n = 100;
    int seed = 69420;

    Point* points = (Point*) malloc(n * sizeof(Point));

    Ran ran(seed);
    for (int i=0; i<n; ++i) {
      points[i].x[0] = ran.doub();
      points[i].x[1] = ran.doub();
    }

    Delaunay delaunay(points, n);

    free(points);
    return 0;
}
```

Listing 3: A quick illustration of how the Delaunay class is called. This code and other relevant source is located in *main/serialIncPtInsertion/src*. Construct an array of points and pass the pointer and the number of pointes generated as arguments to the Delaunay object. A file is created *main/serialIncPtInsertion/data/tri.txt* with the history of and final result of the algorithm.

4. Improvements

5. Conclusion

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.

Bibliography

[1] L. Chen, “Mesh Smoothing Schemes Based on Optimal Delaunay Triangulations.,” 2004, pp. 109–120.

[2] C. L. Lawson, “Transforming triangulations,” *Discrete Math.*, vol. 3, no. 4, pp. 365–372, Jan. 1972, doi: 10.1016/0012-365X(72)90093-3.

[3] S. L. Devadoss and J. O'Rourke, *Discrete and Computational Geometry, 1st Edition*. Princeton University Press, 2011.

[4] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational geometry: algorithms and applications*. Berlin, Heidelberg: Springer-Verlag, 1997.

[5] C. Lawson, “Software for C1 Surface Interpolation.” [Online]. Available: https://www.sciencedirect.com/science/article/pii/B978012587260750011X

[6] M. J. Flynn, “Some computer organizations and their effectiveness,” *IEEE Trans. Comput.*, vol. 21, no. 9, pp. 948–960, Sep. 1972, doi: 10.1109/TC.1972.5009071.

[7] “CUDA C++ Programming Guide.” [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide

[8] “Delaunay Triangulations.” [Online]. Available: https://ti.inf.ethz.ch/ew/courses/Geo23/lecture/gca23-6.pdf

[9] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3rd ed. USA: Cambridge University Press, 2007.

[10] L. Guibas and J. Stolfi, “Primitives for the manipulation of general subdivisions and the computation of Voronoi,” *ACM Trans. Graph.*, vol. 4, no. 2, pp. 74–123, Apr. 1985, doi: 10.1145/282918.282923.

[11] P. Cignoni, C. Montani, and R. Scopigno, “DeWall: A fast divide and conquer Delaunay triangulation algorithm in E d,” *Computer-Aided Design*, vol. 30, pp. 333–341, 1998, doi: 10.1016/S0010-4485(97)00082-1.

[12] T.-T. Cao, A. Nanjappa, M. Gao, and T.-S. 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*, in I3D '14. San Francisco, California: Association for Computing Machinery, 2014, pp. 47–54. doi: 10.1145/2556700.2556710.

[13] A. Nanjappa, “Delaunay triangulation in R3 on the GPU,” 2012.