A GPU accelerated algorithm for 3D Delaunay triangulation*

Thanh-Tung Cao
National University of Singapore

Ashwin Nanjappa
Bioinformatics Institute Singapore

Mingcen Gao
National University of Singapore

Tiow-Seng Tan
National University of Singapore

Abstract

We propose the first algorithm to compute the 3D Delaunay triangulation (DT) on the GPU. Our algorithm uses massively parallel point insertion followed by bilateral flipping, a powerful local operation in computational geometry. Although a flipping algorithm is very amenable to parallel processing and has been employed to construct the 2D DT and the 3D convex hull on the GPU, to our knowledge there is no such successful attempt for constructing the 3D DT. This is because in 3D when many points are inserted in parallel, flipping gets stuck long before reaching the DT, and thus any further correction to obtain the DT is costly. In contrast, we show that by alternating between parallel point insertion and flipping, together with picking an appropriate point insertion order, one can still obtain a triangulation very close to Delaunay. We further propose an adaptive star splaying approach to subsequently transform this result into the 3D DT efficiently. In addition, we introduce several GPU speedup techniques for our implementation, which are also useful for general computational geometry algorithms. On the whole, our hybrid approach, with the GPU accelerating the main work of constructing a near-Delaunay structure and the CPU transforming that into the 3D DT, outperforms all existing sequential CPU algorithms by up to an order of magnitude, in both synthetic and real-world inputs. We also adapt our approach to the 2D DT problem and obtain similar speedup over the best sequential CPU algorithms, and up to 2 times over previous GPU algorithms.

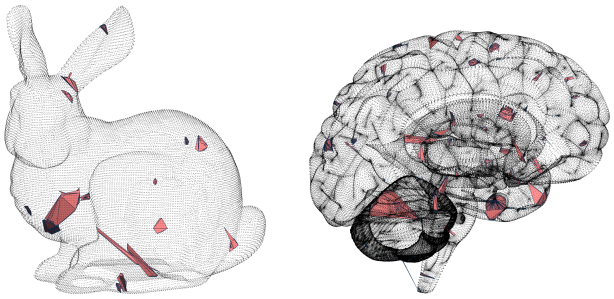
CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modelling—Geometric algorithms I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors

Keywords: Delaunay triangulation, incremental insertion, bilateral flipping, star splaying, GPGPU

1 Introduction

The Delaunay triangulation (DT) has many desirable qualities that make it useful in practical applications. Particularly, the DT is often used to build quality meshes for the finite element method [Huebner et al. 2001]. In $\mathbb{R}^2$, the DT avoids skinny triangles, while in $\mathbb{R}^3$ it minimizes the maximum radius of the minimum containment spheres of the tetrahedra. These are useful properties for the starting point of mesh generation. Therefore, many sequential algorithms have been proposed to construct the DT in $\mathbb{R}^2$ and $\mathbb{R}^3$.

*The research is supported by the National University of Singapore under grant R-252-000-337-112. Project website: http://www.comp.nus.edu.sg/~tants/gDel3D.html. Emails: {caothanh | mingcen | tants}@comp.nus.edu.sg, ashwinn@bii.a-star.edu.sg.



(a) *Bunny – 36K points* **(b)** *Brain – 294K points*

Figure 1: *At the end of the point insertion and flipping phase of our algorithm, less than 0.05% of the facets, shaded in the figure, are locally non-Delaunay .*

To achieve higher performance, many parallel algorithms have been designed, among which one popular approach is to use parallel incremental insertion [Batista et al. 2010]. Starting from an initial DT constructed from a subset of the input points, the rest of the points are inserted and processed in parallel. To ensure correctness, locking strategies are used, and more importantly when insertions conflict with each other, all but one of them must roll back completely and try again later.

There are also attempts to use the GPU to speedup the construction of the DT in particular, and of other fundamental geometric structures in general. The GPU uses a massively parallel architecture with hundreds to thousands of processing elements to execute millions of threads simultaneously. Traditional parallel algorithms, including the one mentioned in the previous paragraph, do not work very well on this programming model, and instead a fine-grained parallel algorithm with regularized work on localized data is preferred. Particularly, locking becomes very inefficient on the GPU, while conflicts during point insertions become uncontrollable. A recent work by Gao *et al.* [2013] uses massive parallel point insertion followed by parallel flipping to obtain a data parallel algorithm, but it only works for the 2D DT and the 3D convex hull problems. As for the 3D DT, if many points are inserted before flipping, there are two difficulties. First, there is no known approach to perform flipping in 3D without getting stuck, or terminating with not too many locally non-Delaunay and unflippable facets. Second, even if only a small number of such facets remain, transforming the result into the 3D DT is costly [Beyer and Meyer-Hermann 2006].

In this paper we show, to our knowledge, the first successful attempt to address the two difficulties mentioned above to compute the DT of a point set in $\mathbb{R}^3$ using the GPU. Our algorithm consists of two phases. In Phase 1, we perform parallel incremental insertion and parallel flipping on the GPU to obtain a triangulation with very few locally non-Delaunay facets; see Figure 1 for some examples. In Phase 2, we adapt the star splaying algorithm by Shewchuk [2005] on the CPU to obtain the DT. Our contributions are as follows:

- An approach of performing parallel point insertion and flipping alternately, plus picking points nearest to the circumcenters of the tetrahedra to insert, to significantly reduce the number of remaining locally non-Delaunay facets. This makes further repairing practical.

- An adaptive approach for the star splaying algorithm so that the work performed is proportional to the amount of modifications needed in getting to the DT.
- Several key GPU techniques, such as handling exact computation and point location, to accelerate our implementation. These are also of independent interest to implementing other computational geometry algorithms on the GPU.

The implementation of our hybrid GPU-CPU algorithm outperforms all existing 3D DT implementations on the CPU by up to an order of magnitude, for both synthetic and real-world data. By adapting the approach of Phase 1 to solve the 2D DT problem on the GPU, we also observe up to an order of magnitude speedup over existing 2D DT implementations on the CPU, and up to 2–5 times speedup over other GPU implementations.

In the following section, we first introduce some basic terminologies and a few related works. Section 3 and Section 4 detail our algorithm and some implementation techniques. In Section 5 we present the experimental analysis of our implementation, before concluding the paper with the limitations and possible future works in Section 6.

2 Preliminaries

In $\mathbb{R}^3$, given a set S of points, the *Delaunay triangulation* (DT) $\mathcal{D}(S)$ of S is a triangulation of S such that the circumsphere of any tetrahedron in $\mathcal{D}(S)$ does not contain any other point in S. Given a triangulation $\mathcal{T}$ of S, a triangle, or *facet*, $f \in \mathcal{T}$ is said to be *locally Delaunay* if and only if it has only one link vertex, or each circumsphere of the tetrahedron formed by f and one of its link vertices does not contain the other vertex. Otherwise, the facet is *locally non-Delaunay*. If every facet of $\mathcal{T}$ is locally Delaunay, then $\mathcal{T} \equiv \mathcal{D}(\mathcal{S})$ [Lawson 1987].

The bilateral flip in 3D is a generalization of the edge flip in 2D [Lawson 1977]; see Figure 2a. A 2-3 flip transforms two tetrahedra $\{acde, bcde\}$ to three tetrahedra $\{abcd, abce, abde\}$ while a 3-2 flip performs the reverse operation. We say that a configuration is *unflippable* if the underlying space of the resulting tetrahedra is larger than that of the original tetrahedra; see Figure 2b. Flipping a locally non-Delaunay facet creates facets that are locally Delaunay.

There are several sequential approaches to construct the DT of a given point set; among which the incremental insertion approach [Bowyer 1981; Watson 1981; Joe 1991] is the most popular one since it has optimal time complexity, is easy to implement and is extensible to higher dimensions. Most parallel DT algorithms, particularly for multi-core systems, also follow this approach [Kohout et al. 2005; Batista et al. 2010; Foteinos and Chrisochoides 2012]. For the GPU, there are also a few recent works to compute the 2D DT [Qi et al. 2012] and the 3D convex hull [Gao et al. 2013]. However, the approach of using the digital Voronoi diagram in Qi *et al.*'s work is not extensible to 3D due to the dualization being an invalid triangulation. The flipping approach used by Gao *et al.* also does not work since flipping in 3D can get stuck as shown by Joe [1989].

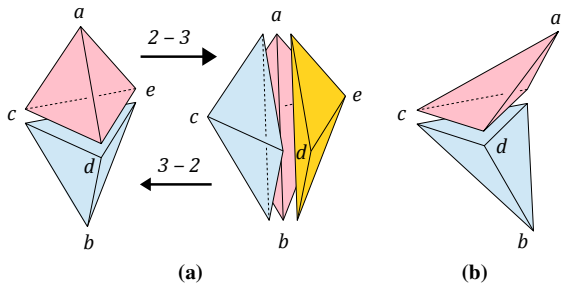


Figure 2: *(a) Bilateral flips in 3D and (b) an unflippable configuration.*

Guibas and Russel [2004] show that in practice, if vertices in a DT move slightly, flipping to get back the Delaunay property seldom gets stuck. On the other hand, flipping from a triangulation constructed by first incrementally inserting all the input points usually leads to a stuck configuration with a lot of locally non-Delaunay facets. To solve the stuck configuration, Beyer and Meyer-Hermann [2006] use an incremental destruction approach to remove all the non-Delaunay tetrahedra, and use incremental construction to patch the cavities. This requires a significant overhead to build an acceleration structure for point search, and the efficiency is also dependent on the point distribution. Another way to fix a stuck configuration of flipping is to use the star splaying algorithm [Shewchuk 2005]. Unfortunately, directly using this approach leads to a lot of time spent constructing the convex stars of all the vertices and checking for their consistencies.

3 Algorithm

From this section onward, point insertion refers to splitting the tetrahedron containing the point, without any further modification to the triangulation. Our algorithm for constructing the 3D DT consists of two phases. In Phase 1, points are inserted in parallel in batches, and parallel flipping is used to transform the triangulation into a near-Delaunay triangulation. This phase is completely performed on the GPU. In Phase 2, our adaptive star splaying algorithm transforms the result of Phase 1 into the DT. This phase is performed on the CPU since by using our algorithm, the work is small.

3.1 Phase 1 – Parallel point insertion and flipping

Our algorithm constructs a triangulation of the point set S using incremental insertion. A simple approach is to insert points in parallel in multiple iterations. In each iteration, each tetrahedron with points inside picks one of them to insert. After all the point insertions, the facets in the triangulation are checked and locally non-Delaunay facets that are flippable are flipped. This simple approach is similar to that used by Gao *et al.* [2013] to construct the 3D convex hull.

This simple approach, however, leads to many locally non-Delaunay facets in the triangulation after flipping, to the extent that it is not practical to be corrected in Phase 2. Instead, we propose to apply flipping after each iteration of point insertion. This has two benefits. First, the earlier we flip, the easier to resolve the locally non-Delaunay facets, and thus less unflippable facets remain when we get stuck, as shown in our experiment. Second, flipping increases the number of tetrahedra significantly before the next iteration of point insertion. If no flipping is performed, before each flipping iteration the number of tetrahedra in $\mathcal{T}$ is $3m$ where m is the number of points inserted so far. By applying flipping after each insertion iteration, the number of tetrahedra in $\mathcal{T}$ approaches the expected value of $6.67m$, the number of tetrahedra in the DT of a uniformly distributed point set [Dwyer 1991]. This means that in the next iteration we can insert more points in parallel, and thus less iterations are needed. Besides, if we look at this problem as computing the convex hull in lifted space, flipping in earlier iterations increases the volume faster, so we expect that less flips are required.

Our proposed approach is detailed in Algorithm 1. In each itera-

Algorithm 1: Incremental insertion and flipping.

Data: A point set S
Result: $\mathcal{D}(S)$
1 initialize $\mathcal{T}$ with a large enough tetrahedron t
2 **for each** $p \in S$ **do in parallel** location[p] $\leftarrow t$
3 **while** $\neg$`Empty(`S`)` **do**
4 | **for each** $p \in S$ **do in parallel** insert[location[p]] $\leftarrow p$
5 | **for each** $t \in \mathcal{T}$ *with* insert[t] $\neq$ null **do in parallel**
6 | | split t using insert[t] and remove insert[t] from S
7 | | label all new facets to be checked
8 | **while** *there are facets to be checked* **do**
9 | | **for each** *facet* f *that needs to be checked* **do in parallel**
10 | | | **if** f *is locally non-Delaunay and flippable* **then**
11 | | | | flip f
12 | | | | label all updated facets to be checked
13 | **end**
14 | update the location of points in S
15 **end**
16 **return** $\mathcal{T}$

tion, we first pick for each tetrahedron a point inside it to insert, if any (line 4). Then we split the tetrahedron and update its neighbors (line 5–7). We also label the new facets so that they are checked in the subsequent flipping. The flipping is performed right after a batch of points is inserted (line 8–13). We repeatedly check the facets in $\mathcal{T}$ to find those locally non-Delaunay facets that are flippable, flip them, and update the neighboring information. Finally, we update the location of the points that are left in S if its old location was split (line 14), to prepare for the next round of point insertion.

In each iteration, when there are tetrahedra containing multiple points, we need to pick one of them to insert. Typical options are choosing randomly, or choosing one near the centroid of the tetrahedron. Instead, we propose to insert the point that is nearest to the circumcenter of that tetrahedron. The motivation is as follows. Constructing the DT is equivalent to constructing the lower hull when the input points are lifted to $\mathbb{R}^4$, i.e. the point (x, y, z) is transformed into the point $(x, y, z, x^2+y^2+z^2)$. Inserting the point nearest to the circumcenter of the tetrahedron is the same as inserting the point furthest to the lifted face. In doing so, the volume of the hull grows the most and is closer to that of the convex hull, thus the number of flipping in the next phase can be reduced. Another reason is that the point nearest to the circumcenter is also the furthest from the vertices of the tetrahedron, i.e. the minimum distance is maximum. Therefore, the facets created during the insertion are of better quality and thus it is also easier for the subsequent flipping. We show in our experiment that by applying this heuristic, not only does the number of flips decrease, but the number of locally non-Delaunay facets at the end of Phase 1 also decreases, thus reducing the effort needed in Phase 2.

The result of this phase is a triangulation that is very close to the DT. There are still facets that are not locally Delaunay but are all unflippable. Using our strategy, this number is usually very small. Figure 1 shows the locally non-Delaunay facets at the end of Phase 1 during the DT construction of two 3D models.

3.2 Phase 2 – Adaptive star splaying

The star splaying algorithm used in this section to transform the result of Phase 1 into the DT is adaptive, and is done sequentially on the CPU. As we show in our experiment, the work needed here is small and thus does not affect the performance of our algorithm.

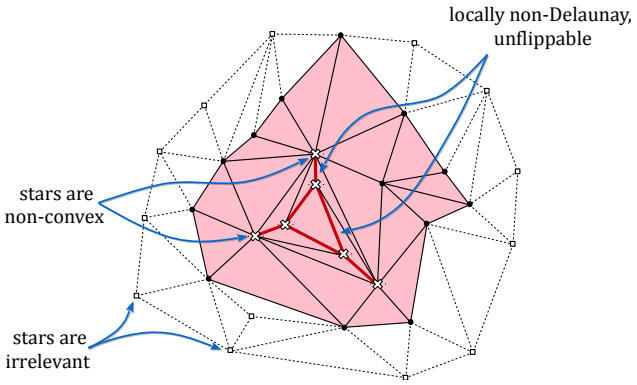


Figure 3: *2D illustration of adaptive star splaying. Only the shaded region is modified.*

There are three steps in our algorithm. Step 1 is to construct the convex stars in $\mathbb{R}^4$. Step 2 is to make the stars consistent by splaying. Step 3 is to convert the stars into a triangulation. These three steps are performed adaptively, with the goal that the work done should be proportional to the actual changes needed.

The three key ideas of our adaptive star splaying algorithm are as follows. First, consider a vertex $s \in \mathcal{T}$ and its star. If s is not incident to any locally non-Delaunay facet, i.e. its star facets are all locally Delaunay, then when lifted to $\mathbb{R}^4$, its star is already convex. Therefore, only stars of the vertices that are incident to some locally non-Delaunay facets are not yet convex and need to be corrected. Other stars can be derived directly from $\mathcal{T}$ without any modification. Second, inside each star that is reconstructed, only the tetrahedra that do not appear in $\mathcal{T}$ need to be checked for consistency, since those that are in $\mathcal{T}$ should already be consistent. Third, only the stars that are modified need to be converted and patched back into $\mathcal{T}$. These ideas are illustrated in Figure 3. Applied correctly, these three ideas help us achieve the goal mentioned earlier.

3.2.1 Building the initial stars

We only rebuild the stars of the vertices that are incident to some locally non-Delaunay facets. These vertices are called *failed vertices*. Since the locally Delaunay checks on all facets have been done on the GPU during the flipping, we use the GPU to collect those failed vertices and pass it to the CPU.

A naïve approach to construct the star of a vertex s is to find all its neighbors in $\mathcal{T}$ and insert them one by one using the Beneath-Beyond method. This, however, is very costly because a vertex may have many neighbors. Also, it is wasteful since the star of s in $\mathcal{T}$ should already be very close to be convex. Now consider the star of $\mathcal{T}$ lifted to $\mathbb{R}^4$. Since any vertex s is an extreme vertex, the star of s is contained in a half-space. Thus, there exists a hyperplane $\mathcal{H}$ near s in $\mathbb{R}^4$ that cuts through the star of s. This intersection is a polyhedron $\mathcal{P}$, and the problem of constructing the convex star for s is equivalent to making this polyhedron the convex hull. Figure 4 illustrates this in $\mathbb{R}^2$ lifted to $\mathbb{R}^3$. In the following discussion, the term "vertical" means along the lifting direction.

Claim 1. *The polyhedron $\mathcal{P}$ on $\mathcal{H}$ is star-shaped w.r.t the intersection of $\mathcal{H}$ and the vertical line through s.*

Proof. We sweep each tetrahedron t in the star of s in $\mathbb{R}^3$ along the vertical direction, forming non-overlapping wedges in $\mathbb{R}^4$. Note that this is the same as sweeping the tetrahedra in the lifted star of s.

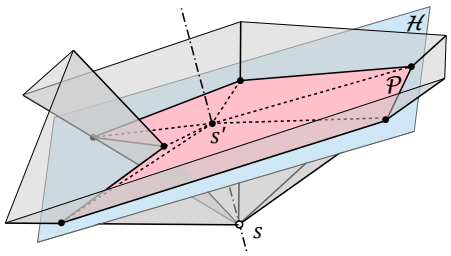


Figure 4: *Constructing the convex star of s in $\mathbb{R}^2$ lifted to $\mathbb{R}^3$.*

Thus, these wedges intersect $\mathcal{P}$ at some non-overlapping tetrahedra, while the vertical line through s intersects $\mathcal{H}$ at a point s' that is inside $\mathcal{P}$ and is a vertex of all these tetrahedra. The boundary of the polyhedron $\mathcal{P}$ is actually the link of s'. Therefore, $\mathcal{P}$ is star-shaped w.r.t to s' on $\mathcal{H}$. □

This claim allows us to use the Flip-Flop algorithm [Gao et al. 2013] to compute the convex hull of $\mathcal{P}$, which is equivalent to computing the convex star of s. That is much more efficient than constructing the convex stars from scratch. We retrieve the link triangulation of s from $\mathcal{T}$, and apply Flip-Flop to transform it into a convex star in $\mathbb{R}^4$. The hyperplane $\mathcal{H}$ is used for explanation only, and it needs not be explicitly computed. The actual orientation tests can be done directly in $\mathbb{R}^4$ with respect to the point s.

3.2.2 Adaptive splaying

The splaying step is done as described by Shewchuk [2005]. We repeatedly check for each tetrahedron $abcd$ in the star of a whether it exists in the star of b (and similarly c and d). If not, we insert a, c and d into b's star using the traditional beneath-beyond method, in an attempt to splay it wider to include the tetrahedron. If any insertion fails, it implies that the corresponding point is enclosed by the star of b, and some vertices on the link of b is inserted into the star of a to splay that star further, thus removing tetrahedron $abcd$. During this step, we may need to access some stars which were not constructed in the previous step. We simply retrieve these stars from $\mathcal{T}$, since they are already convex.

If we check all the tetrahedra created in the previous step for consistency, we need to pull in the stars of all the vertices incident to the failed vertices. This might turn out to be unnecessary if these stars are still consistent. We observe that if a tetrahedron already exists in $\mathcal{T}$ then it need not be checked since any three of its vertices should have already be on or inside the convex star of the fourth one. Thus, during the Flip-Flop in the previous step, we only label the tetrahedra that are modified, from which we start the consistency check in this step. This reduces the number of checks as well as the number of stars that need to be retrieved from $\mathcal{T}$.

3.2.3 Patch the triangulation

After the stars are consistent, $\mathcal{T}$ needs to be updated. Consider the set T_p of all tetrahedra in $\mathcal{T}$ we have previously used to build the stars, and let T_n be the set of tetrahedra we can derive from the set of new stars after splaying. During the star construction, we keep a map between the tetrahedra in the stars and the corresponding ones in $\mathcal{T}$. From that, we find the set of newly created tetrahedra $T^+ = T_n \setminus T_p$ and the set of deleted tetrahedra $T^- = T_p \setminus T_n$. We remove the tetrahedra in T^- from $\mathcal{T}$, and add those in T^+ to $\mathcal{T}$.

Next we update the connectivity between the tetrahedra. We do not need to process those in $T_p \setminus T^-$, which are the old tetrahedra that still survive. Instead, for each tetrahedron t in T^+, we set its 4 neighbors and also update the neighbors to point to t using the connectivity from the stars. This way, for the tetrahedra in $T_p \setminus T^-$ that are adjacent to some new tetrahedra, the connectivity is updated correctly. As a result, we only update the portions of $\mathcal{T}$ that are changed. After this step, the resulting triangulation is the DT.

4 Implementation details

In this section, we highlight some implementation techniques for our proposed algorithm. The discussion includes the following topics: updating the location of each point during flipping, performing each flipping iteration efficiently on the GPU, improving the memory access performance, and handling exact arithmetic.

4.1 Point location

The location of the points that are not yet inserted is updated at two places: after we insert one batch of points, and after we perform the flipping. The first case is simple, since each tetrahedron being inserted is split into four tetrahedra, and points inside are relocated into these new tetrahedra using the orientation predicate. This is done right after point insertion.

For the update after flipping, a simple approach is to update after each flipping iteration. This, however, is not GPU friendly, since all the points need to participate in the relocation step but only few of them are affected by the flips in this iteration. Instead, we record all the flips done in the flipping loop into a directed acyclic graph (DAG), and use this data structure to relocate the points; see Figure 5. This history DAG stores the evolution of the triangulation during the flipping. Each node represents a flip, containing the indices of the 5 vertices and the three tetrahedra involved. Note that we reuse the tetrahedra indices, so a 2-3 flip transforms $\{t_1, t_2\}$ to $\{t_1, t_2, t_3\}$, and vice versa. Each node has up to three pointers $\{n_1, n_2, n_3\}$ that point to the nodes corresponding to the future flips that modify the tetrahedra created in this flip.

The history DAG is constructed as follows. During the flipping, we record all the flips as nodes in the DAG, without pointing them to each other. After that, we build the connectivity by processing the flipping iterations bottom up; see Algorithm 2. We use last[t] to store the last flip node that modifies t. From bottom up, the flips in each iteration are processed in parallel. For each flip f creating tetrahedra t_1, t_2 (and possibly t_3), we update the corresponding node in the DAG. We point that node to the two (or three) nodes that correspond to the future flips that modify its tetrahedra, using the last array (line 6, 8). Then, we update the last array accordingly (line 7, 9). By processing the flipping iterations from bottom up, setting the pointers are coherent memory writes.

To update the point locations using the history DAG, each thread processing a remaining point s starts from its location t and fol-

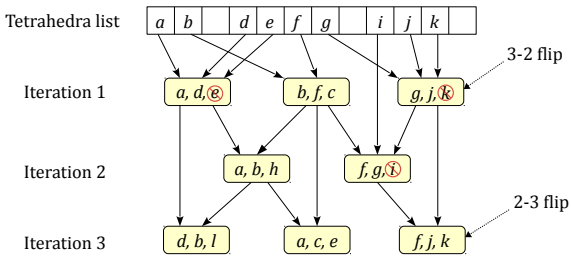


Figure 5: *A history DAG of flipping in 3D.*

Algorithm 2: Construct the history DAG from the list of flips.

```
1  let k be the number of flip iterations performed
2  initialize last[t] = null for all tetrahedra t
3  for i = k to 1 do
4      for each flip f performed in iteration i do in parallel
5          Let x be the corresponding node in the DAG
6          x.n1 = last[x.t1], x.n2 = last[x.t2]
7          last[x.t1] = last[x.t2] = x
8          if f is a 2-3 flip then x.n3 = last[x.t3]
9          else last[x.t3] = x
10 end
```

lows the nodes in the history DAG from last[t] till it reaches a null pointer. At each flip node we use one (or two) orientation tests to determine the new location after that flip. The last tetrahedron recorded is the new location of s.

4.2 Flipping: Compaction or collection

Consider the flipping loop at line 8–13 of Algorithm 1. We assign one thread per tetrahedron, checking its four facets if they are labeled. Any duplicate checks caused by two tetrahedra sharing a facet can be avoided by comparing their indices. The problem is that after a few iterations, many tetrahedra have no more facets to be checked, so many threads are idle thus reducing the efficiency due to thread divergence. Two solutions are possible.

The first solution is to compact the list of tetrahedra after each iteration. We label the tetrahedra that need to be checked in this iteration; they are called *active tetrahedra*. We use parallel stream compaction to compact all the active tetrahedra before launching the kernel to check them. This is costly near the end of the flipping where few tetrahedra are involved.

The second solution is to collect the tetrahedra to be checked in the next iteration during the flipping. Each flip modifies at most 3 tetrahedra, so we pre-allocate an array of size 3 times the number of active tetrahedra in this flipping iteration. Each flip writes down the tetrahedra that it modified into this array. The array is then compacted and used in the next flipping iteration. This strategy scales well with the number of active tetrahedra in each iteration, but it also shuffles the order of the tetrahedra to be checked, and thus the memory access is not in order.

In our implementation, we combine these two approaches. In the first few iterations when the number of active tetrahedra is still very large, we use the first strategy. When this number drops to below a certain threshold, we switch to the second strategy.

4.3 Memory access optimization

We use two approaches to rearrange the data so that the GPU cache is better utilized during point insertion, point relocation, and flipping. First, we observe that during point insertion and relocation, each thread processing a point needs to access the tetrahedron containing that point, including the coordinates. We sort the input points along the Hilbert space filling curve. As a result, points in the same tetrahedron tend to stay near each other in the point list and are processed by adjacent threads, so the access by these threads are cached. The same benefit applies to the point relocation, where nearby threads tend to travel the same path in the history DAG.

Second, during flipping, each thread processing a tetrahedron also accesses its neighbors. Ideally we also want the tetrahedra to be spatially sorted. However, flipping modifies the tetrahedra, and to sort again after each iteration is too costly. Instead, we sort the tetrahedra after each point insertion iteration to benefit the next few flipping iterations which are the most expensive ones. The sorting is done using for each tetrahedron the smallest index of its vertices.

4.4 Exact arithmetic and robustness

Our algorithm relies on two *predicates*, the orientation predicate and the insphere predicate, to make decisions. To deal with numerical error, we adapt Shewchuk's implementation of exact predicates [Shewchuk 1996b] on the GPU. Each predicate consists of two parts: a fast check, which uses floating point arithmetic, and an exact check, which uses floating point expansion. Most of the threads do not need to go into exact computation, so this causes thread divergence. Besides, each exact computation requires a lot more temporary memory, and hence the number of threads we can launch is much smaller. If both checks are done in the same kernel, the threads performing the fast checks will be slowed down. Furthermore, we use the simulation of simplicity method [Edelsbrunner and Mücke 1990] to deal with degenerate input, and this causes further load unbalance among the threads.

To handle this divergence, we split each kernel that performs a predicate into two and launch one after the other. In the first kernel, only fast checks are used, and threads that need exact checks are marked. In the second kernel, the marked threads perform the exact check. Moreover, we only use exact computation for flipping iterations near the end of the algorithm, effectively pushing all the flips that involve degenerate configurations to be processed together.

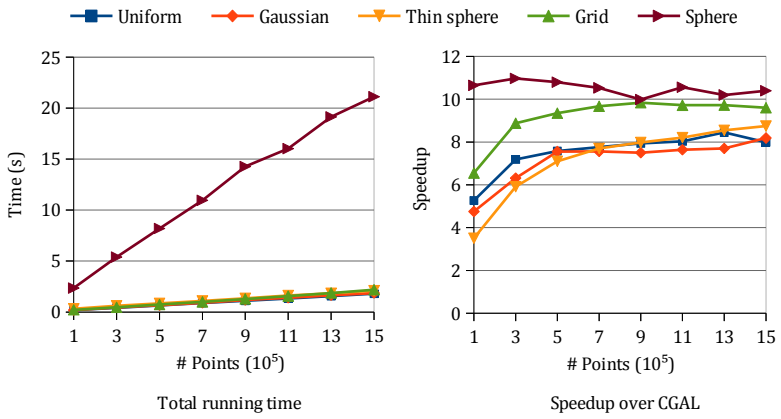
When launching the kernel to perform the exact checks, very few threads actually have work to do. Instead of using stream compaction after the kernel performing the fast checks, we use on-the-fly compaction on shared memory right in this kernel. Threads in the same block do the compaction on shared memory, and then one thread performs an atomicAdd to get a global offset to output the compacted result to global memory. Since the number of exact checks needed is usually not very large, global memory access is reduced.

5 Experimental results

We implement our algorithm using the CUDA programming model by NVIDIA [Nickolls et al. 2008]. All the experiments are conducted on a PC with an Intel i7 2600K 3.4GHz CPU, 16GB of DDR3 RAM and an NVIDIA GTX 580 Fermi graphics card with 3GB of video memory, unless otherwise stated. We first compare the performance of our implementation, termed gDel3D, with CGAL 4.2, the fastest sequential 3D DT library on the CPU. Subsequently we analyze the effect of the techniques we propose. All implementations used are compiled with optimization enabled.

5.1 Running time comparison

We create synthetic data by generating points randomly in the uniform, Gaussian, and three other distributions. In the thin sphere distribution, points lie in between the surface of two balls of slightly different radius. In the grid distribution, points are on a grid of size 512^3. In the sphere distribution, points are cospherical. The latter two distributions are degenerate cases. We also use points from several models of real-world objects obtained from the Stanford 3D Scanning Repository, the Georgia Tech Large Geometric Models Archive, and the Princeton Suggestive Contour Library. In these models, points are usually not well distributed, and the amount of degeneracy ranges from moderate to high.



(a) *Synthetic data*

Model	# Points	gDel3D time (s)	Speedup over CGAL
Armadillo	172,974	0.4	6.1
Angel	237,018	0.5	6.7
Brain	294,012	0.5	7.6
Dragon	437,645	0.9	7.1
Happy Buddha	543,652	1.1	7.2
Blade	882,954	1.8	6.7
Asian Dragon	3,609,600	5.4	9.2

(b) *Real-world data*

Figure 6: *The running time and speedup of gDel3D compared to CGAL on (a) synthetic point distributions and (b) real-world data.*

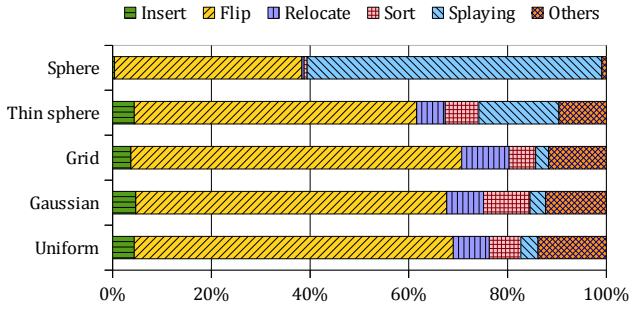


Figure 7: *Time breakdown of gDel3D with different point distributions.*

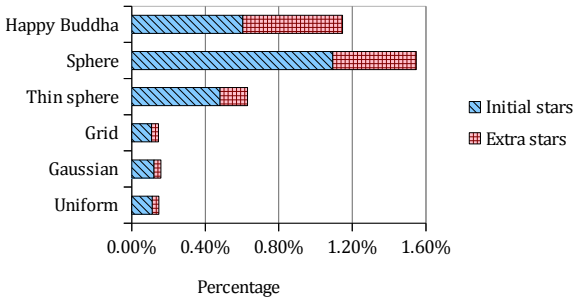


Figure 8: *The number of stars (in percentage of the number of points) constructed in Phase 2.*

Synthetic data

Figure 6a shows the running time and speedup of gDel3D over CGAL on different point distributions, with the input size ranging from 10^5 to 15×10^5. The running time increases linearly with the input sizes, with the performance on the thin sphere and the grid distribution being very close to that of the uniform and the Gaussian distribution. This is the result of our careful handling of exact arithmetic on the GPU. This also means that gDel3D is not much affected by the distribution of points. The sphere distribution needs more time since the amount of exact computation required is very high. Nevertheless, for all distributions, the speedup compared to CGAL starts at 4–6 times, and quickly rises to 8–10 times when the number of points increases. Particularly for the two pathological distributions, not only that gDel3D can handle them robustly, but the speedup over CGAL is even higher.

Real-world data

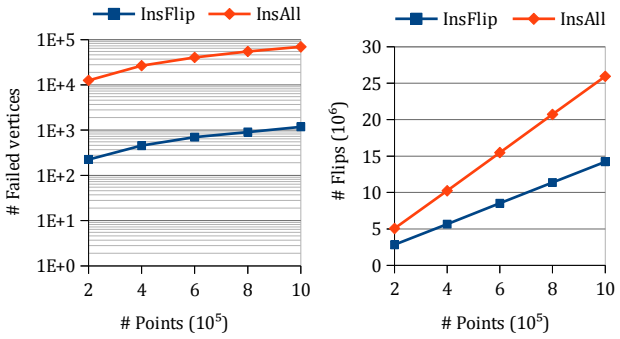
Figure 6b shows the running time of gDel3D on some 3D models and a comparison with that of CGAL. Clearly, even when the points are non-uniformly distributed, gDel3D can handle them with ease. The speedup ranges from 6 to 9 times, even for models that contain a lot of degeneracies such as the Blade model.

5.2 Detailed analysis

Figure 7 shows the percentage of running time taken by different tasks in our implementation, including Phase 1 and Phase 2, on different point distributions with 10^6 points. These tasks are inserting points, flipping, relocating points, sorting, and splaying. As we expect, in the uniform, the Gaussian and the grid distribution, majority of the running time is spent in flipping. Also, by using the history DAG, although the number of flips is so high, the point relocation task still takes quite a moderate amount of time. It would take nearly two times longer if the DAG is not used.

In the thin sphere distribution we start seeing some difference. Being a very non-uniform point distribution, the flipping has more difficulty reaching the DT, and thus more work remains to be done. Therefore, the star splaying takes a bigger portion of the running time. In the pathological case with points being cospherical, the adaptive star splaying time dominates the total running time.

To further understand the behavior of our algorithm, we look at the number of stars participating in the adaptive star splaying in Phase 2. Figure 8 shows the number of stars constructed for the failed vertices, as well as the number of additional stars taken from the triangulation during the splaying. In this experiment we use 5×10^5 points for the synthetic inputs, and the Happy Buddha model, which has approximately the same amount of points, as a real-world input for comparison. As can be seen, on the first three distributions, the number of stars initially constructed is very small, only about 500 stars, and less than 200 additional stars are involved in the splaying. On the thin sphere and the sphere distributions, significantly more stars are involved, but still less than 1.5% of the number of input points in total. The same applies to the real-world test case. Note that the sphere distribution is an extreme case.



(a) *# Failed vertices* **(b)** *# Flips performed*

Figure 9: *Comparison between the InsFlip and the InsAll strategy.*

5.3 InsFlip or InsAll

We analyze the advantage of using our strategy of alternating between point insertion and flipping (termed InsFlip) to the naïve approach of inserting all points before flipping (termed InsAll). Points are randomly generated from the uniform distribution. Instead of focusing on the number of locally non-Delaunay facets remaining after Phase 1, we analyze the number of failed vertices, whose stars need to be constructed in Phase 2. This number directly affects the running time of Phase 2. The chart in Figure 9a uses log-scale because the difference is too large. The InsFlip strategy significantly reduces the number of unflippable facets at the end of Phase 1, and thus the number of failed vertices reduces by nearly two orders of magnitude. In addition, the number of flips is reduced by about 40%, as shown in Figure 9b. This is the main reason why our hybrid approach is practical, as otherwise Phase 2 would take too long.

5.4 Point insertion

We also compare our choice of picking the point near the circumcenter of the tetrahedron to insert, with two strategies that are more common: picking the point near the centroid of the tetrahedron, and picking randomly. Our experiment shows that for points in a uniform distribution, inserting near the circumcenter is similar to inserting near the centroid, and it reduces the number of failed vertices by more than 2 times compared to picking randomly. Whereas, for points in the real-world models, our strategy reduces this number 2–4 times compared to both inserting near the centroid and inserting randomly. It also reduces the number of flips by 20%. Therefore, for these test cases, the total running time of gDel3D decreases by around 2 times when using our strategy.

5.5 Scalability with different GPUs

We run gDel3D on several different GPUs to demonstrate its scalability to the computation power of the hardware. Figure 10 shows the running time with 10^6 input points in uniform distribution. Clearly, gDel3D runs significantly faster on a better GPU, with a speed up of 3x going from the GTS 450 to the GTX 580. When the running time is normalized by the number of cores and their frequencies, we notice similar performance for the GTS 450 and the GTX 460, and slightly higher performance for the GTX 470 and the GTX 580, which has a slightly different architecture. However, the normalized time of the GTX Titan is significantly higher. This is because on the Titan with very high computation power, gDel3D is actually memory bound, hence there is only about 30% speedup over the GTX 580.

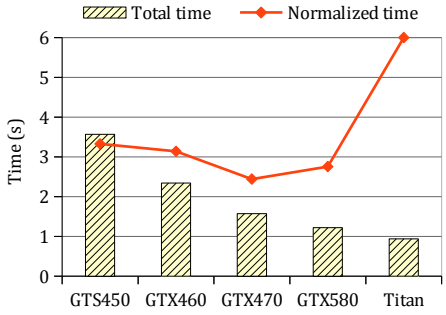
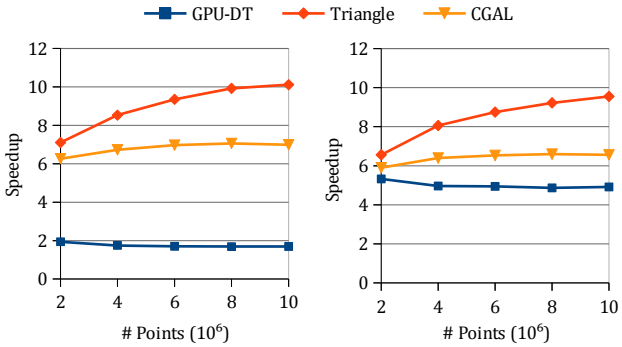


Figure 10: *The running time of gDel3D on different GPUs.*



(a) *Uniform* **(b)** *Thin circle*

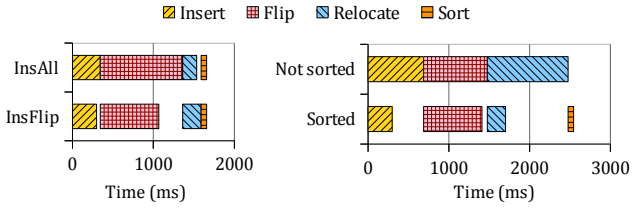
Figure 11: *Speedup of gDel2D over GPU-DT, Triangle and CGAL.*

5.6 2D Delaunay triangulation

Our algorithm can easily be adapted to compute the 2D DT. We only need to use Phase 1 because flipping never gets stuck in $\mathbb{R}^2$ [Lawson 1977]. We compare our implementation, termed gDel2D, to CGAL, *Triangle* [Shewchuk 1996a] and GPU-DT [Qi et al. 2012]. Figure 11a shows the speedup of gDel2D on the uniform point distribution. In general, gDel2D outperforms Triangle by up to 10 times, CGAL by more than 6 times, and this speedup increases as the number of points increases. Also, it is 2 times faster than GPU-DT. Similar behavior is observed with the Gaussian distribution.

We also look at the performance on the thin circle distribution, which is similar to the thin sphere distribution; see Figure 11b. This experiment highlights the advantage of gDel2D over GPU-DT. In this distribution, it is very difficult for the digital Voronoi diagram to accurately approximate the continuous one, thus GPU-DT is significantly slower. Whereas, similar to the 3D case, gDel2D achieves similar speedup over CGAL and Triangle, and is more than 5 times faster than GPU-DT, even when GPU-DT uses a large grid of size 8192^2 to compute the digital Voronoi diagram.

Our algorithm in the 2D case is similar to the algorithm presented in the work of Gao *et al.*, except some techniques such as interleaving the point insertion and the flipping and sorting the input points and triangle list. Figure 12a compares the time breakdown of gDel2D when using InsAll and InsFlip strategies respectively. The InsFlip strategy reduces the flipping time by about 30%, with negligible overhead in the point insertion and relocation. Furthermore, by using sorting, the performance of the point insertion increases by more than 2 times, while that of the point relocation increases nearly 4 times, as show in Figure 12b. The extra cost of sorting is insignificant compared to the benefit.



(a) *InsAll and InsFlip* **(b)** *With and without sorting*

Figure 12: *Time breakdown of gDel2D.*

6 Conclusion

In this paper, we show that parallel bilateral flipping is practical to construct the DT in 3D on the GPU. We propose two techniques, alternating between parallel point insertion and flipping, and picking points near the circumcenter of the tetrahedra to insert, to reduce the number of locally non-Delaunay facets when flipping gets stuck. This allows us to get very close to the DT while still enjoying the benefit of a flipping algorithm being very GPU friendly. From the result after flipping, we introduce a modification of the star splaying algorithm to adaptively transform the triangulation into the DT. Furthermore, we present several GPU implementation techniques, such as using shared memory compaction when handling exact computation and constructing the history DAG for parallel point location after flipping. Our implementation outperforms the best 3D DT implementation on the CPU by up to an order of magnitude in both synthetic and real-world data. It is also robust to degenerated inputs, as shown in our experiment.

We also adapt our approach to the 2D problem, and our experiment shows that it is much better than the previous works, especially when handling non-uniform point distributions. We believe our approach can also be used when moving to higher dimensions, as well as when handling point set with weights (i.e. regular triangulations). It is our future work to experiment with these problems.

One limitation of our algorithm is that for some pathological cases such as when input points lie on two non-intersecting lines in 3D, or on a circle in 2D, very few flips can be performed in parallel. This is because in many places, one flip needs to be done before another one can be discovered. This situation, however, rarely happens in practice. 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 optimization.

References

BATISTA, V. H., MILLMAN, D. L., PION, S., AND SINGLER, J. 2010. Parallel geometric algorithms for multi-core computers. *Computational Geometry 43*, 8, 663–677.

BEYER, T., AND MEYER-HERMANN, M. 2006. Recover from non-flippable configurations in parallel approaches to three dimensional kinetic regular triangulations. In *Proc. 8th WSEAS Int. Conf. Automatic control, modeling & simulation*, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, ACMOS'06, 399–404.

BOWYER, A. 1981. Computing Dirichlet tessellations. *The Computer J. 24*, 2, 162–166.

DWYER, R. 1991. Higher-dimensional Voronoi diagrams in linear expected time. *Discrete & Computational Geometry 6*, 1, 343–367.

EDELSBRUNNER, H., AND MÜCKE, E. P. 1990. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics 9*, 66–104.

FOTEINOS, P., AND CHRISOCHOIDES, N. 2012. Dynamic parallel 3D Delaunay triangulation. In *Proc. 20th Int. Meshing Roundtable*, Springer Berlin Heidelberg, Berlin, Heidelberg, W. R. Quadros, Ed., 9–26.

GAO, M., CAO, T.-T., TAN, T.-S., AND HUANG, Z. 2013. Flip-flop: convex hull construction via star-shaped polyhedron in 3D. In *I3D '13: Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics and Games*, ACM, New York, NY, USA, 45–54.

GUIBAS, L. J., AND RUSSEL, D. 2004. An empirical comparison of techniques for updating delaunay triangulations. In *Proc. 22nd Symp. Computational geometry*, ACM, New York, NY, USA, SoCG '04, 170–179.

HUEBNER, K. H., DEWHIRST, D. L., SMITH, D. E., AND BYROM, T. G. 2001. *The Finite Element Method for Engineers*. Wiley, New York, NY, USA.

JOE, B. 1989. Three-dimensional triangulations from local transformations. *SIAM J. on Scientific and Statistical Computing 10*, 4, 718.

JOE, B. 1991. Construction of three-dimensional Delaunay triangulations using local transformations. *Computer Aided Geometric Design 8*, 2, 123–142.

KOHOUT, J., KOLINGEROVÁ, I., AND ŽÁRA, J. 2005. Parallel Delaunay triangulation in E2 and E3 for computers with shared memory. *Parallel Computing 31*, 5, 491–522.

LAWSON, C. L. 1977. Software for C1 surface interpolation. In *Mathematical Software III*, J. R. Rice, Ed. Academic Press, 161–194.

LAWSON, C. L. 1987. Properties of n-dimensional triangulations. *Computer Aided Geometric Design 3*, 4, 231–246.

NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. 2008. Scalable parallel programming with CUDA. *Queue 6*, 2, 40–53.

QI, M., CAO, T.-T., AND TAN, T.-S. 2012. Computing 2D constrained Delaunay triangulation using the GPU. In *I3D '12: Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics and Games*, ACM Press, New York, New York, USA, 39–46.

SHEWCHUK, J. R. 1996. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In *Applied Computational Geometry towards Geometric Engineering*, M. Lin and D. Manocha, Eds., vol. 1148 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 203–222.

SHEWCHUK, J. R. 1996. Robust adaptive floating-point geometric predicates. In *Proc. 12th Symp. Computational geometry*, ACM, New York, NY, USA, SoCG '96, 141–150.

SHEWCHUK, J. R. 2005. Star splaying: an algorithm for repairing Delaunay triangulations and convex hulls. In *Proc. 21st Symp. on Computational Geometry*, ACM, New York, NY, USA, SoCG '05, 237–246.

WATSON, D. F. 1981. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *The Computer J. 24*, 2, 167–172.